Prefetching Query Results and its Impact on Search Engines

Simon Jonassen
Norwegian University of Science and Technology
Trondheim, Norway
simonj@idi.ntnu.no

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

Fabrizio Silvestri
ISTI - CNR, Italy; and
Yahoo! Research, Barcelona
f.silvestri@isti.cnr.it

ABSTRACT

We investigate the impact of query result prefetching on the efficiency and effectiveness of web search engines. We propose offline and online strategies for selecting and ordering queries whose results are to be prefetched. The offline strategies rely on query log analysis and the queries are selected from the queries issued on the previous day. The online strategies select the queries from the result cache, relying on a machine learning model that estimates the arrival times of queries. We carefully evaluate the proposed prefetching techniques via simulation on a query log obtained from Yahoo! web search. We demonstrate that our strategies are able to improve various performance metrics, including the hit rate, query response time, result freshness, and query degradation rate, relative to a state-of-the-art baseline.

Categories and Subject Descriptors

H.3.3 [**Information Storage Systems**]: Information Retrieval Systems

General Terms

Design, Performance, Experimentation

Keywords

Web search engine, result caching, prefetching

1. INTRODUCTION

Commercial web search engines are expected to process user queries under tight response time constraints and be able to operate under heavy query traffic loads. Operating under these conditions requires building a very large infrastructure involving thousands of computers and making continuous investment to maintain this infrastructure [7]. Optimizing the efficiency of the web search systems is important to reduce the infrastructure costs. Even small improvements may immediately translate into significant financial savings for the search engine.

SIGIR'12, August 12–16, 2012, Portland, Oregon, USA.

Recent research has shown that result caching [5, 15] is a viable technique to enhance the overall efficiency of search engines. The main idea in result caching is to store the results of frequently or recently processed queries in a large cache and readily serve subsequent occurrences of queries by the cache. This way, significantly more expensive computations at the backend query processing system are avoided, leading to important efficiency gains in the form of reduction in query response latencies and backend query workloads.

Unlike earlier works that have a focus on limited-capacity caches [5, 15], more recent works assume result caches with infinite capacities [11]. The main reason behind this infinite cache assumption is the cheap availability of storage devices that are large enough to store the results of all previous user queries issued to the search engine. Under the infinite cache assumption all future queries can be served by the cache except for compulsory misses which have to be served by the search backend. However, an infinite result cache suffers from the staleness problem [11]. The dynamic nature of web document collections requires frequent index updates, which may render some cache entries stale [1, 8]. In some cases, the search results served by the cache may not be fresh enough in terms of their content and this may degrade the user satisfaction.

In practice, an effective solution to the freshness problem is to associate every result entry in the cache with a time-to-live (TTL) value [2, 11]. In this technique, a cache entry is considered stale once its TTL expires. The hits on the expired entries are treated as misses and are processed by the backend, leading to fresh results. This way, the TTL approach sets an upper bound on the staleness of any search result served by the cache. Unfortunately, it sacrifices some of the efficiency gains achieved by means of result caching. Since the cache hits on the expired cache entries are treated as cache misses, the query traffic volume hitting the backend search system significantly increases with respect to a scenario where no TTL value is associated with the cache entries. In general, the increased backend query volume leads to an increase in the query response latencies and more backend resources are needed to handle the query traffic. Moreover, there is a higher risk that the spikes in the query volume will lead to an overloaded backend, in which case certain queries may have to be processed in the degradation mode, i.e., search results are only partially computed for these queries and the user experience is hampered.

The above-mentioned negative consequences of the TTL approach can be alleviated by combining it with a prefetch-

ing[1] strategy that has the goal of updating results of cache entries that are expired or about to be expired before a user requests them [11]. An ideal prefetching strategy would have all queries be readily served by the cache. In practice, there is no perfect knowledge of queries that will be issued in the future. Hence, prefetching strategies can only be heuristics.

The observations made before form the main motivation of this paper. In particular, we aim to devise strategies to identify cache entries that are likely to be requested when they are expired. We proactively fetch the associated search results using the idle compute cycles of the backend. The main challenge associated with prefetching is to predict when an expired cache entry will be requested. The predicted times can be used to select queries whose results are worth prefetching and to prioritize them to obtain the highest performance benefit. Although related ideas on cache freshness [11] and batch query processing [12] appear in the literature (see Section 7), our work is novel in terms of the following contributions.

- We quantify the available opportunity for prefetching (i.e., the amount of backend capacity that can be used for prefetching) and the potential benefit (i.e., the amount of requests for expired cache entries), using a query workload obtained from Yahoo! web search. To best of our knowledge, these were not reported before.
- We propose offline and online strategies to select and prioritize queries that will potentially benefit from prefetching. The offline strategy relies on the observation that the queries tend to repeat on a daily basis and applies query log mining to identify queries whose results are to be prefetched. The online strategy relies on a machine learning model that predicts the next occurrence time of issued queries. Prefetching operations are then prioritized based on these expected times.
- We conduct simulations to observe some important performance metrics, including the hit rate, query response time, freshness, and query degradation rate.

The rest of the paper is organized as follows. In Section 2, we discuss our system model and motivate the result prefetching problem through observations made over a real-life web search query log. In Sections 3 and 4, we present the proposed offline and online prefetching strategies, respectively. We provide the details of our data and experimental setup in Section 5. The experimental results are presented in Section 6. We provide the related work in Section 7. The paper is concluded in Section 8.

2. PRELIMINARIES

System architecture. In this work, we assume the search engine architecture shown in Fig. 1. User queries are issued to the search engine's main frontend, which contains an infinite result cache where the entries are associated with a TTL value. Queries whose results are found in the cache and not yet expired are readily served by the result cache. Otherwise, they are issued to the frontend of a selected backend search cluster, which is composed of many nodes that host a large index build on the document collection (only one cluster is displayed in the figure). After a query is processed at the backend cluster, the computed results are cached together with the time of the computation so that the expiration time can be determined. The query prefetching module interacts with both the search engine frontend and the search cluster frontend. This module is responsible for selecting a set of queries that are expired or about to be expired (from the query logs or the result cache) and issuing them to the search cluster's frontend, which issues them to the backend. The results computed by the backend are then cached like regular user queries.

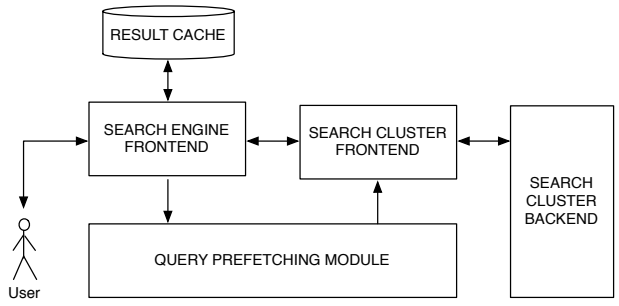

Figure 1: The sketch of a search engine architecture with a query prefetching module.

Query prefetching problem. Our focus in this work is on the query prefetching module. The idea behind this module is to avoid, as much as possible, potential cache misses by proactively computing, i.e., prefetching, the results of queries that are about to expire before they are requested by the users. At first glance, the problem of prefetching looks trivial as it seems easy to identify queries that are expired or about to be expired. The problem, however, is quite challenging because not all prefetching operations are useful and prefetching the results of a query consumes backend resources. In particular, if the results of a query are prefetched, but the prefetched results are not requested before they are expired, prefetching leads to waste of resources.

The most important benefit of prefetching is the increase in the cache hit rate [11]. This immediately corresponds to reduced average query response times as more queries can be served by the cache. In addition, the freshness can also be improved if unexpired cache entries are prefetched.[2] Finally, the fraction of queries whose results are computed in the degraded mode can be reduced if prefetching can decrease the amount of processing at peak query traffic times. In summary, the benefits expected from prefetching are reduced query response time, improved result freshness, and reduced query result degradation.

The query results need to be prefetched, as much as possible, when the user query traffic volume is low so that the user queries are not negatively affected from the query processing overhead incurred due to prefetching. Hence, the feasibility of prefetching depends on the availability of the low traffic hours. This raises the question whether the low traffic periods are long enough to prefetch sufficiently many query results. Moreover, the fraction of queries that can benefit from prefetching needs to be quantified. Finally, the potential risk for query result degradation needs to be identified.

[1]The term prefetching is used in [13, 16] differently to imply requesting the successive results pages for a query.

[2]It is interesting to note that prefetching only expired cache entries results in increased staleness. In fact, cache staleness not necessarily impacts on results freshness as some expired results might not be retrieved in the future.

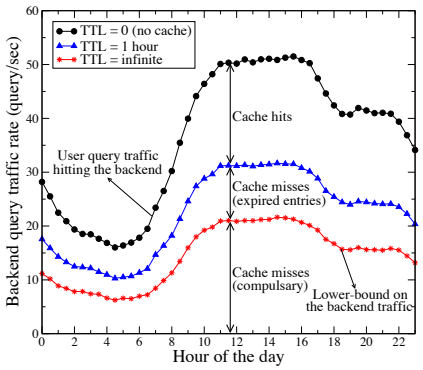

Figure 2: The user query traffic hitting the backend under different TTL values.

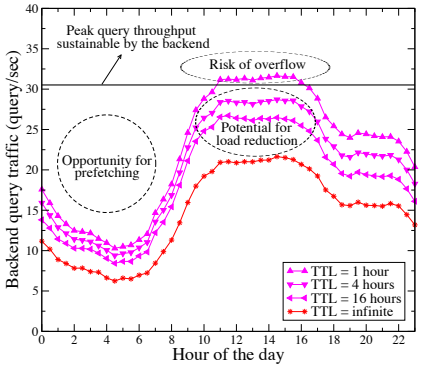

Figure 3: The variation in the user query traffic distribution within a day and its implications.

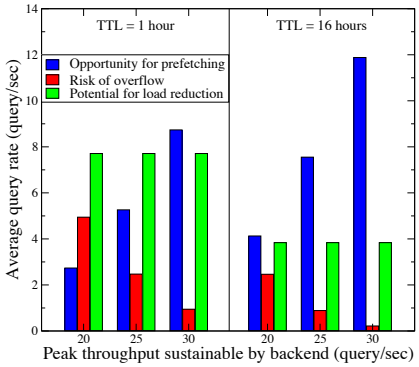

Figure 4: The opportunity, risk, and potential benefit (quantified as query per second).

We will look into these issues in what follows by analyzing a sample taken from the query traffic received by Yahoo! web search during a particular day.

Motivating observations. The upmost curve in Fig. 2 shows the user query traffic received by the search engine, i.e., the traffic that would hit the backend if there was no result cache. The bottom curve indicates the backend traffic in case of an infinite result cache with an infinite TTL, i.e., only the compulsory cache misses hit the backend. The curve in between shows the backend traffic volume when the TTL is set to one hour. We observe that, during the peak hours, a significant amount of cache misses (about one-third) are due to expired cache entries.

According to Fig. 3, there is a good amount of opportunity for prefetching (especially during the night) when compared to the amount of requests on expired cache entries. We observe that the traffic volume reaches its peak from 11AM to 4PM, resulting in a risk for query result degradation in scenarios where the backend traffic rate is comparable to the peak sustainable throughput (PST) of the backend (e.g., when the TTL is one hour). In general, the backend query load decreases with increasing TTL values, but the potential for reduction in the query load due to prefetching also decreases since there are fewer expired queries.

In Fig. 4, we try to quantify the existing opportunity for prefetching, the risk of overflow, and the potential for load reduction, assuming two different TTL values and three different PST values. The reported values are in terms of queries per second, averaged over the entire day. As an example, with a TTL of one hour and a PST of 20 query/sec, the overflow rate is about five queries per second.

Prefetching strategies. We evaluate two alternative types of prefetching strategies. The first set of techniques are offline and rely on query log mining. In these strategies, the prefetched queries are selected from the previous day's query logs. The second set of prefetching strategies are online. The idea here is to predict the next occurrence times of cached queries and try to prefetch expired or about-to-expire cache entries before they are requested while being in an expired state. Both types of strategies have two phases: the selection phase, where the queries whose results are to be prefetched are determined, and the ordering phase, where the selected queries are sorted in decreasing order of their importance and are prefetched in that order.

3. OFFLINE STRATEGIES

There is a high correlation between the repetition of some queries and the time of the day, i.e., some queries are more likely to be issued around the same time of the day. According to Fig. 2 in [10], a large fraction of queries are submitted about 24 hours after their latest submission. Hence, a reasonable strategy is to prefetch certain queries in the previous day's query logs. This strategy is based on query log mining and is completely offline. Hence, the query selection process does not incur any overhead on the runtime system.

3.1 Offline Query Selection

Let T denote the TTL of the result cache. Let τ denote the current time point and τ' denote the time point exactly 24 hours before τ. Assume that the time interval $[\tau', \tau)$ is split into N time intervals, each interval having a length of $s = (\tau - \tau')/N$ time units. Let $\langle \delta_1, \delta_2, \ldots, \delta_N \rangle$ be a sequence of N time intervals, where $\delta_i = [\tau' + (i-1) \times s, \tau' + i \times s)$. Each time interval δ_i is associated with a query bucket b_i that stores the queries issued to the search engine within the respective time interval. The queries to be prefetched during a particular future time interval $[\tau + (i-1) \times s, \tau + i \times s)$ are limited to those in the set of buckets $\langle b_i, b_{i+1}, \ldots, b_{i+\lfloor T/s \rfloor} \rangle$, i.e., the queries issued within the past time interval $[\tau' + (i-1) \times s, \tau' + (i-1) \times s + T)$. Hence, the prefetched queries are selected as

$$\mathcal{Q}_i = \bigcup_{j=i}^{i+\lfloor T/s \rfloor} b_j. \quad (1)$$

At the end of a time interval δ_i, we switch to the next query set $\mathcal{Q}_{i+1}$ and start prefetching results of queries in that set.

3.2 Offline Query Ordering

Since it may not be possible to prefetch all queries associated with a time interval, the queries need to be ordered based on their importance, i.e., the benefit that can be achieved by prefetching a query. While ordering the queries, the past frequency of a query plays an important role. Herein, we discuss three simple strategies to determine the importance of a query.[3] The strategies differ in the way the past query frequency is weighted. In what follows, $w(q)$

[3]It is possible to come up with variants of these strategies.

represents the weight of a query q and $\mathcal{S}$ is the sample of queries considered in the selection phase mentioned above.

Unit-weighted frequency. This strategy assumes that a query's importance is equal to its frequency in $\mathcal{S}$, i.e.,

$$w(q) = \sum_{q_j \in \mathcal{S}} 1, \tag{2}$$

where q_j denotes an occurrence of q in the query sample $\mathcal{S}$. This strategy simply prefers prefetching frequent queries as an attempt towards increasing the likelihood of prefetching a query that will repeat again.

Workload-weighted frequency. This strategy assigns a heavier weight to those queries appearing during high-traffic periods, i.e., those that are issued when the backend query traffic volume is higher. The motivation here is to prefetch more queries from busy traffic hours so that the number of queries hitting the search cluster backend is reduced in those hours. We compute the weight of a query as

$$w(q) = \sum_{q_j \in \mathcal{S}} v(q_j), \tag{3}$$

where $v(q_j)$ represents the backend query traffic volume in queries per second when q_j is observed. This strategy aims at reducing the backend query volume at busy hours and, indirectly, it also aims at reducing the number of degraded query results.

Time-weighted frequency. This strategy give a higher priority to queries that occurred closer in time to τ'.

$$w(q) = \sum_{q_j \in \mathcal{S}} (\tau' + T - t(q_j)), \tag{4}$$

where $t(q)$ denotes the time at which q_j is issued. The assumption here is that the likelihood of prefetching a repeating query will increase.

4. ONLINE STRATEGIES

Online strategies are designed to identify queries that are already expired or about to expire and are going to be issued to the backend due to a TTL miss. Those queries are processed beforehand, possibly when the load at the backend is low. The key feature of this strategy is the prediction of the next time point (e_q) at which query q will be issued while its cached results are in an expired state.

In the rest of the section, we use the following notation. We denote by l_q the latest request time and by u_q the latest computation time of the results of query q. As in the previous section, τ denotes the current time and T denotes the TTL. We denote by s the selection period (i.e., we update the prefetching queue at every s time units) and by f_q the number of occurrences of q up to τ.

To estimate e_q, each time q is submitted, we predict the time to its next appearance, n_q. Under the assumption that q appears every n_q seconds, e_q can be calculated as $l_q + k \times n_q$, where k is the smallest natural number greater than 0 that satisfies $u_q + T < l_q + k \times n_q$, i.e., $k = \lfloor \frac{u_q + T - l_q}{n_q} \rfloor + 1$.

The value for n_q is computed through a machine learning model using the features given in Table 1. Temporal features are based on when a query is submitted. Query string features are based on the syntactic content of the query. Result page features capture the characteristics of the results returned by the search engine. Frequency features are based on counters associated with terms of the queries. We use gradient boosted decision trees to train our model [14].

Table 1: The features used by the learning model

Feature type	Feature	Description
Temporal	`hourOfDay`	Hour of submission
	`timeGap`	Time since last occurence
Query string	`termCount`	No. of terms in query
	`avgTermLength`	Avg. term length
Result page	`pageNo`	Requested result page no
	`resultCount`	No. of matching results
Frequency	`queryFreq`	No. of occ. of query
	`sumQueryTF`	Sum
	`minQueryTF`	Minimum
	`avgQueryTF`	Average
	`maxQueryTF`	Max. term query log freq.
	`sumDocTF`	Sum
	`minDocTF`	Minimum
	`avgDocTF`	Average
	`maxDocTF`	Max. term document freq.

4.1 Online Selection

The key point in the online strategy is to prefetch queries satisfying $\tau < e_q \leq \tau + T$ when there is processing capacity. As a naïve method of query selection, one could scan the list of queries in the cache and pick those with e_q values that satisfy the constraint. In practice, this is not feasible as the cost of scanning the cache for every submitted query is prohibitive. Hence, we resort to use a bucket data structure where each bucket stores queries with e_q values within a time-period $[t, t+s)$. Insertions and deletions in the bucket list can be realized in $O(1)$-time.

As illustrated in Fig. 5, the prefetching module maintains a list of buckets. Each time a query is requested, we remove it from the current bucket, modify e_q and place it back in a new bucket. In particular, we have two situations. If the request is a hit, we modify e_q and potentially move the query into another bucket. Otherwise, if it is a miss, we delete it from its current bucket and issue it to the backend. Once it has been processed, we update e_q and eventually insert q in the bucket list. Finally, once in every s seconds we select queries having $e_q \in (\tau, \tau + T]$ from the corresponding buckets and place these in the prefetching queue.

4.2 Online Ordering

To prioritize the queries in the prefetching queue, we evaluate the following three methods:

Based on age-frequency. A query is assigned a weight

$$w(q) = (\tau - u_q) \times f_q.$$

In other words, older and more frequent queries get a higher priority. This strategy tends to favor frequent queries that were refreshed long time ago. The goal of this method is to optimize the freshness of the results.

Based on age-frequency and degradation. We extend the weight function to include degradation.

$$w(q) = (\tau - u_q) \times f_q \times (1 + d_q).$$

This strategy favors older and degraded queries. The goal of this method is to optimize the freshness of the results and reduce the degradation.

Based on processing cost and expected load. Each query is prioritized according to its expected processing cost

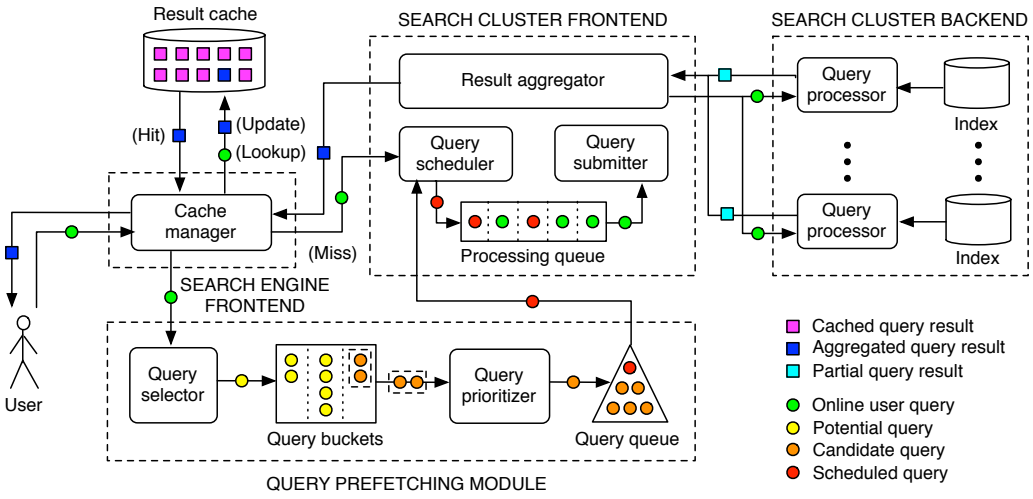

Figure 5: The proposed search engine architecture with query prefetching capability.

times the expected difference in the backend load.

$$w(q) = \rho(q) \times L(e_q),$$

where $\rho(q)$ is the estimated processing cost and $L(e_q)$ is the backend load that is observed 24 hours prior to e_q. This strategy aims at reducing the query response time and increasing the throughput.

5. DATA AND SETUP

Prefetching strategies. We evaluate a number of competing strategies. The first type of strategies are baseline strategies that do not employ any prefetching. These are `no-ttl`, which assumes that there is an infinite result cache with no TTL for entries so that any repetition of a query is a hit, and `no-prefetch`, which assumes that there is an infinite result cache with a TTL, but the prefetching logic is not activated. The second type of strategies are the offline strategies discussed in Section 3: `off-freq`, `off-vol`, and `off-time`. The third type of strategies are the online strategies discussed in Section 4: `on-age`, `on-agedg`, and `on-load`. The fourth type are the oracle versions of online strategies in which the exact next occurrence times of queries are assumed to be known. Following the convention in naming the online strategies, we refer to the oracle strategies as `oracle-age`, `oracle-agedg`, and `oracle-load`. The last implementation is an interpretation of the age-temperature refreshing strategy [11], which we refer to as `age-temp`.

The original `age-temp` strategy maintains a two-dimensional bucket structure, where one of the dimensions corresponds to the result age and the other dimension corresponds to the query frequency (temperature). In our implementation of this strategy, each age bucket is defined by the selection interval s and the temperature is computed as $\log_2(f_q)$, where f_q is the number of times a query is seen since the beginning of the first day. Each time a query is requested, we increase its frequency and potentially move it to a bucket corresponding to a higher temperature. Each time a query is processed, we move it to the bucket with the same temperature but the age dimension corresponding to τ. Furthermore, in our experiments, the temperature of a query cannot decrease. As an alternative, we have evaluated the use of a sliding time-window to dynamically maintain the number of occurrences. However, we have observed that with a window of reasonable size, e.g., 24 or 48 hours, the hit rate tends to be lower. Therefore, in our experiments, we count occurrences starting from the beginning of the first day. Further, we prioritize buckets by the corresponding age $\times$ temperature value and restrict selection to the first $30,000$ queries satisfying a minimum age requirement, which prevents too fresh queries from being repeatedly refreshed. We denote the minimum result age as R and use it as the limiting factor for queries selected also by the other methods. Finally, among the selected queries, we give a higher priority to expired queries instead of unexpired ones.

In the `offline` strategies, the only cache state information that is made available to the prefetching module is whether an entry is older than R or not. Other information relies only on the query log from the past day and includes nothing about the current cache state. In the `online` strategies, the decisions are made based on the current cache state and the prediction model mentioned before. As predicting, based on a limited query log, the arrival times of queries that request expired entries is a challenging problem, we allow an entry to be prefetched up to two times without being hit in between.

We use the `oracle` strategies to show the effect of accurate prediction of the next request for an expired entry. The oracle strategies have no information about future hits, neither the number of occurrences nor the time at which they will occur. Nevertheless, they demonstrate that accurate prediction of the next requests for expired entries is enough to achieve good performance. Note that, in our results, we report only the results for the `oracle-agedg` strategy since it always outperforms the other two alternatives.

Simulation setup. We sample queries from five consecutive days of Yahoo! web search query logs. The queries in the first three days are used to warmup the result cache. No prefetching is done in these days. At the end of the third day, we start allowing queries into our data structures. The age-temperature baseline starts with queries collected from the first three days. The offline strategies start with queries collected during the third day. The online strategies use the first three days to create a model. As no predictions are

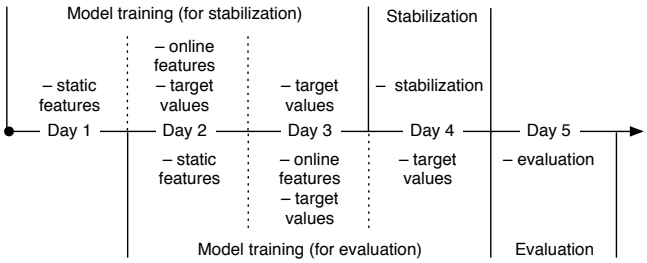

Figure 6: Temporal splitting of the query log with respect to the tasks.

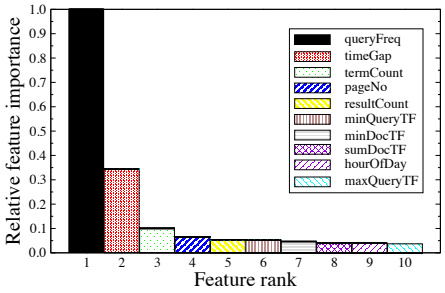

Figure 7: Feature importance.

made before the beginning of the fourth day, we use average inter-arrival times to initiate the bucket structure at the end of day three. We assume that the prefetching logic is activated at the beginning of the fourth day. Finally, for all of the techniques, we use the fourth day to stabilize the performance and the fifth day for the performance evaluation.

For the prediction model used by the online strategies, the first day of the training period is used to compute the static features (e.g., query and term frequencies).[4] The instances for which the next arrival times are tried to be predicted are extracted from the second day. The target values (i.e., the next arrival times that are tried to be predicted) are obtained from the second and third days. This setup is illustrated in Fig. 6. The importance values for the features used by the model are displayed in Fig. 7. As expected, the query frequency and the time gap between the two consecutive occurrences of a query turn out to be the most important features. For the fourth and fifth days, we train a learning model using the preceding three days.

System parameters and query processing logic. The experiments are conducted using a discrete event simulator that takes as input the user queries along with their original submission timestamps. We simulate a system with K compute nodes and P query processors, which allows up to P queries to be concurrently processed with a speedup proportional to K. We assume that the entire inverted index is maintained in the memory. In all experiments, we set the TTL (T) to 16 hours, which is chosen by an analysis of the opportunity, risk, and potential benefits of prefetching shown in Fig. 4. We fix P to 8 and vary K to create three different scenarios corresponding to poor ($K = 850$), medium ($K = 1020$), and high performance ($K = 1224$) systems. With 850 nodes, both `no-prefetch` and `no-ttl` experience a degradation of efficiency at peak performance. With 1020 nodes (20% above the first scenario), `no-ttl` reaches the peak sustainable throughput only marginally. With 1224 nodes (20% above the second scenario), both `no-prefetch` and `no-ttl` perform well.

If more than P queries are concurrently issued to the backend, the most recent queries are put into a processing queue. To prevent the queue from growing indefinitely, we monitor the processor usage P_l over the last minute. When $P_l > 0.95 \times P$, we degrade the currently processed query by P/Q_τ, where Q_τ is the number of queries being processed at the backend at that time.

[4]We use a document collection containing several billion documents to compute the features that rely on the document frequencies of terms.

Since prefetching happens while user queries are scheduled, we monitor the average number of user queries being processed over the last minute Q_l and maintain a budget of $B = \lfloor f \times P - Q_l \rfloor$ processors to be used for prefetching. By default, we use $f = 0.875$ to keep the processor utilization around 87.5 percent. To prevent the system from prefetching recently processed queries, we restrict the minimum age of queries being selected with a parameter R. In our experiments we use three different values, TTL/2, TTL/4 and TTL/8. Finally, the prefetching queue is updated/reset once in every $s = 10$ minutes.

Evaluation metrics. We evaluate the performance using five different metrics. The metrics are cache hit rate, query response time, average hit age, degradation rate, and backend query traffic rate. The hit rate reflects how good the prefetching algorithm is at having query results available at request. The change in response time reflects the real improvement from prefetching versus its overhead. The upper bound for both metrics is limited by the number of compulsory misses and corresponds to the hit rate and latency of `no-ttl`. The average hit age reflects how good the prefetching method is at keeping cached results fresh. The degradation rate reflects the amount of degraded results being served to the user and is defined as the total degradation divided by the number of queries being answered by the frontend. The total degradation is the sum of the relative degradations of the results returned by the frontend. Finally, the average backend query traffic measures the average number of queries issued to the backend. The lower bound on this number is again defined by `no-ttl`.

6. EXPERIMENTS

In this section, we present the results of our experiments. Our first observation is that in all the cases that we analyze (except for one), the best possible results are obtained, as expected, by the `oracle-agedg` strategy (the last line in the tables). Obviously, not being feasible in practice, the oracle strategy is presented only as an upper bound to the results that can be attained by the other techniques. Indeed, as the prediction accuracy gets closer to that of the oracle, we should get better and better results.

Note that, in the sole case of the average hit age results reported in Table 6, the `oracle-agedg` strategy leads to suboptimal results. This behavior can be explained by observing that the age decreases as we increase the number of times an entry is subject to prefetching. The oracle, instead, minimizes the amount of prefetching and thus some entries (especially those requested further in the future) tend to have higher ages.

Table 2: Cache hit rate

	$K=850$			$K=1020$			$K=1224$		
Strategy	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$
`no-prefetch`	0.493	–	–	0.494	–	–	0.494	–	–
`no-ttl`	0.600	–	–	0.600	–	–	0.600	–	–
`age-temp`	0.517	0.517	0.516	0.533	0.533	0.533	0.542	0.542	0.543
`off-freq`	0.512	0.512	0.510	0.515	0.515	0.514	0.517	0.518	0.517
`off-vol`	0.511	0.511	0.509	0.514	0.514	0.513	0.516	0.517	0.517
`off-time`	0.511	0.511	0.509	0.515	0.515	0.515	0.519	0.521	0.521
`on-age`	0.534	0.535	0.533	0.546	0.547	0.547	0.549	0.550	0.553
`on-agedg`	0.534	0.535	0.533	0.546	0.547	0.547	0.549	0.550	0.553
`on-load`	0.493	0.493	0.494	0.494	0.494	0.495	0.494	0.494	0.497
`oracle-agedg`	0.597	0.596	0.592	0.599	0.599	0.600	0.600	0.600	0.600

Table 2 shows the cache hit rate of different strategies for various parameters. The `no-prefetch` and `no-ttl` strategies are the two extreme cases. The `no-prefetch` strategy does not involve prefetching and relies only on the TTL to cope with staleness problem. The `no-ttl` strategy correspond to an infinite cache scenario, where only the compulsory misses incur a cost. In between these two extremes, by varying the K and R parameters, we observe that, in all cases, the offline strategies perform worse than the `age-temp` baseline. Our online strategies, however, outperform the baseline. In particular, we compare the hit rate performance of a strategy A versus that of B by computing the relative hit rate improvement with respect to `oracle-agedg` as $I=\frac{\mathsf{HR}_A-\mathsf{HR}_B}{\mathsf{HR}_\mathrm{O}-\mathsf{HR}_B}$, where HR_A, HR_B, and HR_O are the hit rates of the A, B, and `oracle-agedg` strategies, respectively. The improvement towards the oracle of our `on-age` versus the baseline `age-temp` ranges from 12% of `on-age` with $K=1224$ and $R=T/8$ to 22.78% of `on-age` with $K=850$ and $R=T/4$ with an average of 19.23%. Fig. 8 shows the hit rate trend of four different strategies: `no-prefetch`, `no-ttl`, `on-agedg`, and `oracle-agedg`. According to the figure, our method consistently outperforms the `no-prefetch` strategy.

In Table 3, we report the response time of our system by varying the simulation parameters. Query response time and hit rate are related and results confirm this behavior. As in the case of hit rate, we report the improvement with respect to the oracle that are similar to those of hit rate figures. The improvement towards the oracle of our `on-age` versus the baseline `age-temp` ranges from 12.4% of `on-age` with $K=1224$ and $R=T/8$ to 27.5% of `on-age` with $K=850$ and $R=T/8$ with an average of 21.1%. The hourly trend for the average response time is shown in Fig. 9, where the effect of peak load reduction is evident. From 11AM to 6PM, the peak load response time is greatly reduced. Therefore, in this case, the user experience should be greatly improved.

Another important aspect to consider when evaluating prefetching strategies is the load we put on the backend (Table 4). If we do not adopt any prefetching strategy, the load on the backend would certainly be affected only by the hit rate. Indeed, the higher the hit rate the less the number of queries hitting the backend. This is confirmed by the numbers in Table 4 as the `no-prefetch` strategy is the one attaining, in all cases, the minimum amount of workload at the backend. We remark that our goal is to optimize the exploitation of the infrastructure, i.e., to increase the overall load by keeping the amount of degraded queries as low as possible. Numbers in Table 4, thus, are more explanatory if read in conjunction with the results presented in Table 5. Disregarding the case $K=1224$, in which the computational capacity is too high to be overloaded even by a great number of prefetching operations (with the exception of `on-load` which tries to prefetch expensive queries first), in all cases, `on-age` and `on-agedg` attain the greatest reduction in terms of degraded queries, despite the average backend query traffic is greater than the baseline. Figs. 10 and 11 confirm the results reported in the tables and show even more remarkably the effect of prefetching on the backend query traffic, which results to be roughly flattened and far apart from the `no-cache` case. The degradation (Fig. 11) is also greatly reduced during the peak hours. In this case, we report degradation of query results for the case $K=850$ instead of $K=1020$ as in the other cases. This is because, for $K=1020$, the degradation is negligible.

We also report the average hit age measured in number of minutes passed since the last update. It is worth being pointed out that measuring the average hit ratio does not say much about the quality or staleness of results in the cache. In fact, if the majority of entries are updated but requested after a period of time very close to the TTL, the ideal strategy would not update them. Indeed, this would make the average hit age increase when those queries are actually requested only slightly before their expiration. We report these results in Table 6 and Fig. 12.

Finally, we measure the accuracy of prefetching methods. We define the accuracy as the fraction of correct prefetching actions. That is, the average between the fraction of prefetching operations never followed by a hit with respect to the total number of prefetching operations and the fraction of compulsory misses relative to the number of misses. As illustrated in Table 7, while performing less prefetching, the offline techniques have in general higher accuracy. In addition, it is worth to note that there is a high potential for improvement. If we compare prefetching accuracy with that of `oracle-agedg`, we can observe that we are still far from being close to the maximum. On the other hand, the worst accuracy is achieved by `on-load`, which illustrates that prioritizing frequent queries maximizes the probability of useful prefetching. We note that the `age-temp` baseline performs poorly with respect to our strategies.

In what follows, we enumerate what we retain to be the take away messages from this work. First, the baseline relies only on the knowledge of the past frequency and the current age. This approach cares about keeping popular entries fresh and does not care if they will be used again in the future. Second, the offline strategies rely only on the information about the queries submitted in the previous day. Even though they are worse than the baseline, they perform surprisingly well, given that they use only one day of history. Third, the online strategies rely on predicted future query expirations. This prediction task is very difficult to

Table 3: Average query response time (in ms)

	$K=850$			$K=1020$			$K=1224$		
Strategy	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$
no-pefetch	309.1	–	–	157.5	–	–	112.5	–	–
no-ttl	183.6	–	–	122.0	–	–	95.5	–	–
age-temp	278.1	277.4	277.4	153.8	153.7	153.9	119.0	119.0	119.0
off-freq	280.5	279.9	280.1	157.2	156.2	155.3	120.3	117.8	114.8
off-vol	280.1	280.6	282.3	157.6	156.9	155.9	120.6	118.6	116.3
off-time	280.9	283.0	285.5	157.8	157.1	156.9	120.6	118.8	117.2
on-age	257.8	255.5	257.5	148.1	147.9	147.9	116.6	116.5	116.4
on-agedg	253.8	254.8	257.3	147.9	147.7	147.7	116.6	116.5	116.4
on-load	316.2	311.2	312.4	173.3	173.2	171.2	133.7	133.3	132.5
oracle-agedg	189.7	189.8	191.2	129.5	129.0	128.9	99.7	99.4	99.0

Table 4: Average backend query traffic rate (query/sec)

	$K=850$			$K=1020$			$K=1224$		
Strategy	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$
no-prefetch	18.34	–	–	18.34	–	–	18.34	–	–
no-ttl	14.50	–	–	14.50	–	–	14.50	–	–
age-temp	23.48	23.44	23.41	27.28	27.09	26.93	32.75	32.44	32.07
off-freq	23.01	21.00	19.97	25.22	22.62	20.74	29.03	24.67	21.65
off-vol	22.98	20.97	19.95	25.19	22.59	20.83	29.00	24.97	22.23
off-time	23.04	21.18	20.08	25.15	22.73	21.28	29.23	25.50	22.92
on-age	25.11	24.93	24.33	30.34	30.03	29.15	36.99	36.43	35.26
on-agedg	25.07	24.86	24.28	30.35	30.03	29.15	36.99	36.43	35.26
on-load	18.95	19.03	19.24	19.53	19.60	19.88	20.78	20.86	21.17
oracle-agedg	20.92	20.49	19.85	22.47	21.63	20.74	23.99	23.19	21.62

Table 5: Average degradation rate ($\times 10^{-4}$)

	$K=850$			$K=1020$			$K=1224$		
Strategy	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$
no-prefetch	166.9	–	–	1.398	–	–	0.000	–	–
no-ttl	7.2	–	–	0.003	–	–	0.000	–	–
age-temp	76.6	79.0	95.0	0.485	0.412	0.418	0.000	0.000	0.000
off-freq	87.9	88.2	89.8	0.645	0.731	0.849	0.000	0.000	0.000
off-vol	88.8	87.8	87.9	0.632	0.606	0.645	0.000	0.000	0.000
off-time	94.0	91.1	93.4	0.807	0.661	0.689	0.000	0.000	0.000
on-age	50.3	48.8	51.3	0.319	0.223	0.118	0.000	0.000	0.000
on-agedg	50.6	49.5	49.6	0.201	0.169	0.284	0.000	0.000	0.000
on-load	233.2	241.5	175.4	173.3	173.2	171.3	133.7	133.3	132.5
oracle-agedg	5.7	5.7	6.4	0.000	0.000	0.000	0.000	0.000	0.000

Table 6: Average hit age (in minutes)

	$K=850$			$K=1020$			$K=1224$		
Strategy	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$
no-prefetch	357	–	–	356	–	–	356	–	–
no-ttl	4737	–	–	4735	–	–	4734	–	–
age-temp	342	356	375	170	209	289	137	170	246
off-freq	222	255	322	139	172	239	111	147	227
off-vol	222	254	327	138	172	329	111	148	226
off-time	226	259	329	144	179	242	118	155	232
on-age	217	255	333	135	163	235	114	148	222
on-agedg	216	254	333	135	163	235	114	148	222
on-load	356	356	353	356	356	355	352	352	360
oracle-agedg	272	291	336	258	280	324	265	287	332

Table 7: Prefetching accuracy

	$K=850$			$K=1020$			$K=1224$		
Strategy	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$	$R=T/8$	$R=T/4$	$R=T/2$
age-temp	0.529	0.525	0.514	0.550	0.543	0.533	0.543	0.535	0.523
off-freq	0.543	0.593	0.629	0.551	0.586	0.610	0.534	0.563	0.592
off-vol	0.537	0.588	0.622	0.546	0.580	0.602	0.531	0.556	0.573
off-time	0.534	0.573	0.601	0.542	0.572	0.586	0.533	0.554	0.561
on-age	0.599	0.592	0.594	0.597	0.587	0.588	0.570	0.560	0.557
on-agedg	0.599	0.593	0.595	0.597	0.587	0.588	0.570	0.560	0.557
on-load	0.412	0.441	0.528	0.417	0.451	0.516	0.419	0.475	0.495
oracle-agedg	0.842	0.846	0.855	0.798	0.799	0.804	0.723	0.720	0.726

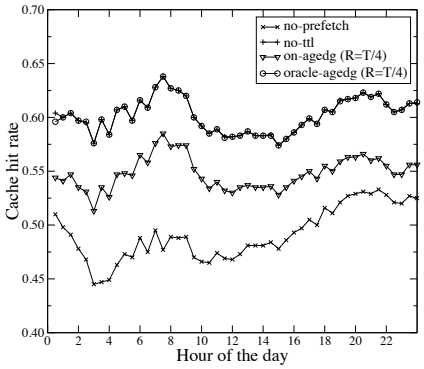

Figure 8: Result cache hit rate ($K = 1020$).

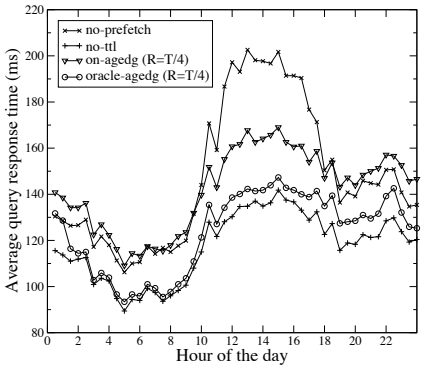

Figure 9: Average query response time ($K = 1020$).

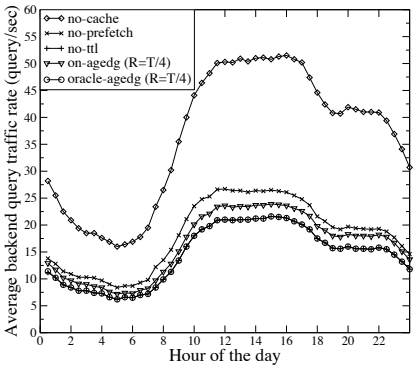

Figure 10: Backend query traffic rate (only user queries, $K = 1020$).

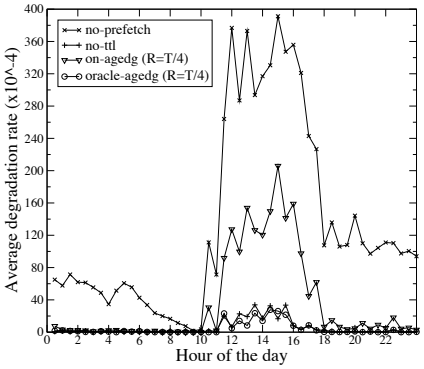

Figure 11: Average degradation rate ($K = 850$).

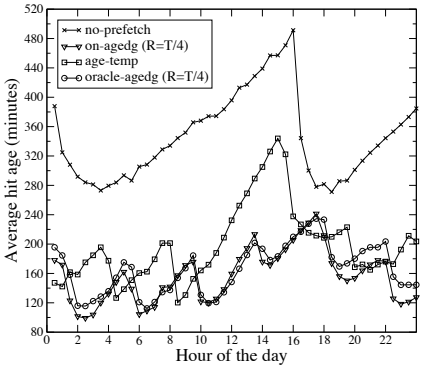

Figure 12: Average hit age ($K = 1020$).

carry out, as experiments show, yet a mildly good prediction policy gives very good results in terms of search backend exploitation, low query degradation, and hit-ratio. Fourth, even if oracle has a perfect knowledge of all the future misses, it does not have all the information needed to order the queries in the best possible way to reduce the number of misses, for instance. However, we use the oracle strategies as an upper bound to the effectiveness of prefetching with respect to the cache content. Finally, in Figs. 8–12, we observe that, during the peak hours, the benefits of prefetching are greatly amplified thus confirming our initial hypothesis, i.e., prefetching has a great potential for improving search engine utilization and peak-load performance.

7. PREVIOUS WORK

Herein, we survey the previous work on result caching, freshness, and batch query processing. Interested readers may refer to [10] for a broader survey on search efficiency.

Result caching. So far, a large body of research focused on increasing the result cache hit rates [4, 20] or to reduce the query processing cost of backend search systems [15, 21]. Depending on how the cache entries are selected, static [6], dynamic [22], or hybrid [13] caching strategies are followed. The earlier works assumed limited-capacity caches, where the main research issues are admission [5], prefetching [16], and eviction [20], whereas the recent works mostly adopted an infinite cache assumption [11]. A number of proposals are made to combine other layers of caching with result caching, resulting in two-level [3, 22], three-level [17, 18], or even five-level [19] caching architectures.

Cache freshness. The most recent works deal with result caching in the context of maintaining the freshness of the results served by the cache [1, 8, 9, 11]. In this line of research, the cache entries are associated with TTL values that are fixed for all queries. In [2], each query is associated with a different TTL value depending on its freshness requirements. In several works, the TTL approach is coupled either with more sophisticated techniques such as invalidation of potentially stale cache entries [1, 8, 9], where the goal is to predict the stale cache entries by exploiting certain information obtained during index updates. The search results whose freshness is potentially affected by the updates are then marked as invalid so that they are served by the backend system in a successive request, rather than by the cache. The approach followed in [11], on the other hand, relies on proactive refreshing of cached search results. In this approach, the indexing system does not provide any feedback on staleness. Instead, some presumably stale search results are selected and refreshed based on their likelihood of being requested in future and also depending on the availability of free processing cycles at the backend search system. Our work proposes strategies that are alternative to that in [11] and also focuses on different performance metrics.

Batch query processing. In [12], some efficiency optimizations are proposed for batch query processing in web search engines. Those optimizations (e.g., query reordering) are orthogonal to ours and, in case of a partially disk-resident index, they may be coupled with our prefetching techniques. Since we assume an in-memory index, however, we did not consider those optimizations in our work.

8. CONCLUSIONS AND FURTHER WORK

We investigated the impact of prefetching on search engines. We showed that an oracle strategy that has a perfect information of future query occurrences can achieve the best attainable results. We made an attempt to close the gap with the oracle by testing two alternative set of prefetching techniques: offline and online prefetching. We showed that the key aspect in prefetching is the prediction methodology. Furthermore, we showed that our accuracy in predicting future query occurrences is only about a half of the best that can be done. We plan to extend this work in several directions. First, we are going to design more effective techniques for predicting the expiration of a query. Second, we are going to evaluate the economic impact of prefetching on the search operations. Finally, we would like to design speculative prediction techniques that will be able to prefetch queries that were not even submitted. This would reduce the number of compulsory misses that form an upper bound on the cache performance.

9. ACKNOWLEDGMENTS

Simon Jonassen was an intern at Yahoo! Research and supported by the iAd Centre, Research Council of Norway and NTNU.

10. 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 *Proc. 34th Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 973–982, 2011.

[2] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy. Adaptive time-to-live strategies for query result caching in web search engines. In *Proc. 34th European Conf. Information Retrieval*, pages 401–412, 2012.

[3] I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy. Second chance: a hybrid approach for dynamic result caching in search engines. In *Proc. 33rd European Conf. Information Retrieval*, pages 510–516, 2011.

[4] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. 30th Annual Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 183–190, 2007.

[5] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. Witschel. Admission policies for caches of search engine results. In N. Ziviani and R. Baeza-Yates, editors, *String Processing and Information Retrieval*, volume 4726 of *Lecture Notes in Computer Science*, pages 74–85. Springer Berlin / Heidelberg, 2007.

[6] R. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, editors, *String Processing and Information Retrieval*, volume 2857 of *Lecture Notes in Computer Science*, pages 56–65. Springer Berlin / Heidelberg, 2003.

[7] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: the Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[8] R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza. Caching search engine results over incremental indices. In *Proc. 33rd Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 82–89, 2010.

[9] E. Bortnikov, R. Lempel, and K. Vornovitsky. Caching for realtime search. In *Proc. 33rd European Conf. Information Retrieval*, pages 104–116, 2011.

[10] B. B. Cambazoglu and R. Baeza-Yates. Scalability challenges in web search engines. In M. Melucci, R. Baeza-Yates, and W. B. Croft, editors, *Advanced Topics in Information Retrieval*, volume 33 of *The Information Retrieval Series*, pages 27–50. Springer Berlin Heidelberg, 2011.

[11] 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 *Proc. 19th Int'l Conf. World Wide Web*, pages 181–190, 2010.

[12] S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel. Batch query processing for web search engines. In *Proc. 4th ACM Int'l Conf. Web Search and Data Mining*, pages 137–146, 2011.

[13] 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. Information Systems*, 24(1):51–78, 2006.

[14] J. H. Friedman. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38:367–378, February 2002.

[15] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proc. 18th Int'l Conf. World Wide Web*, pages 431–440, 2009.

[16] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. 12th Int'l Conf. World Wide Web*, pages 19–28, 2003.

[17] H. Li, W.-C. Lee, A. Sivasubramaniam, and C. L. Giles. A hybrid cache and prefetch mechanism for scientific literature search engines. In *Proc. 7th Int'l Conf. Web Engineering*, pages 121–136, 2007.

[18] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. 14th Int'l Conf. World Wide Web*, pages 257–266, 2005.

[19] M. Marin, V. Gil-Costa, and C. Gomez-Pantoja. New caching techniques for web search engines. In *Proc. 19th ACM Int'l Symp. High Performance Distributed Computing*, pages 215–226, 2010.

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

[21] R. Ozcan, I. S. Altingövde, and O. Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Trans. Web*, 5(2):9:1–9:25, 2011.

[22] 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 *Proc. 24th Annual Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 51–58, 2001.