Information Processing and Management 53 (2017) 834–850

Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/infoproman

A machine learning approach for result caching in web search engines

Tayfun Kucukyilmaz[a,*], B. Barla Cambazoglu[b,1], Cevdet Aykanat[c], Ricardo Baeza-Yates[d,1]

[a] TED University, Computer Engineering Department, Ankara, Turkey
[b] Independent Researcher, Barcelona, Spain
[c] Bilkent University, Computer Engineering Department, Ankara, Turkey
[d] NTENT, Carlsbad, CA, USA

ARTICLE INFO

Article history:
Received 5 September 2016
Revised 4 January 2017
Accepted 9 February 2017
Available online 11 March 2017

Keywords:
Query result caching
Machine learning
Feature-based caching
Static caching
Static-dynamic caching

ABSTRACT

A commonly used technique for improving search engine performance is result caching. In result caching, precomputed results (e.g., URLs and snippets of best matching pages) of certain queries are stored in a fast-access storage. The future occurrences of a query whose results are already stored in the cache can be directly served by the result cache, eliminating the need to process the query using costly computing resources. Although other performance metrics are possible, the main performance metric for evaluating the success of a result cache is hit rate. In this work, we present a machine learning approach to improve the hit rate of a result cache by facilitating a large number of features extracted from search engine query logs. We then apply the proposed machine learning approach to static, dynamic, and static-dynamic caching. Compared to the previous methods in the literature, the proposed approach improves the hit rate of the result cache up to 0.66%, which corresponds to 9.60% of the potential room for improvement.

© 2017 Elsevier Ltd. All rights reserved.

1. Introduction

Scalability and efficiency are two crucial aspects of performance in search engines (Cambazoglu & Baeza-Yates, 2015). A commonly used technique for improving search engine performance is result caching (Baeza-Yates et al., 2007a). In result caching, precomputed results (e.g., URLs and snippets of best matching pages) of certain queries are stored in a fast-access storage. The future occurrences of a query whose results are already cached can be directly served by the result cache, eliminating the need to process the query using costly computing resources. Result caching has two immediate benefits to a search engine. First, by reducing the computational load on the server side, it enables higher query processing throughput. Second, it reduces the average query response time perceived by the users.

In the result caching problem, the goal is to maintain a set of previously computed query results in a limited-capacity cache such that some performance metric is optimized over the time as new queries are received. Although other performance metrics are possible (Altingovde, Ozcan, & Ulusoy, 2009; Gan & Suel, 2009), the main performance metric for

* Corresponding author.
E-mail addresses: tayfun.kucukyilmaz@tedu.edu.tr (T. Kucukyilmaz), barla@berkantbarlacambazoglu.com (B.B. Cambazoglu), aykanat@cs.bilkent.edu.tr (C. Aykanat), ricardo.baeza@upf.edu (R. Baeza-Yates).
[1] The work was performed while the author was affiliated with Yahoo Labs, Barcelona, Spain.

http://dx.doi.org/10.1016/j.ipm.2017.02.006
0306-4573/© 2017 Elsevier Ltd. All rights reserved.

evaluating the success of a result cache is hit rate, i.e., the fraction of queries that are answered by the cache. In practice, increasing the hit rate requires careful selection of queries whose results are to be cached.

In the literature, there are two types of result caches: static and dynamic. In static caching, the cache is filled in an offline manner with the results of queries selected from the past query logs. The underlying assumption in static caching is the steady behavior in the query stream, i.e., queries which were popular in the past remain popular in the future. Therefore, popular queries are preferred over the rest when making the caching decisions. In dynamic caching, the caching decisions are online and the goal is to identify queries that are least likely to result in a cache hit and evict such queries to make room for other, more recent queries. The underlying assumption in dynamic caching is the bursty behavior in the query stream, i.e., queries which are submitted more recently have higher probability of reoccurring in the future. Fagni, Perego, Silvestri, and Orlando (2006) show that a hybrid caching approach combining these two types of caches performs better than using them in isolation.

Past research has tried to improve the performance of the result caching by relying on a single or limited number of features. The primary objective of this work is to devise a unifying caching framework which exploits an extensive set of features. To this end, we propose a machine learning approach that combines a large variety of features extracted from search engine query logs and evaluate the impact of this framework on the hit rate of a search engine result cache. We devise different learning models for static and dynamic result caching. For the former type of caches, we focus on the offline cache allocation problem. For the latter type of caches, we focus on the online eviction problem. As a secondary objective, we aim to identify the potential room for improvement in result cache hit rate. To this end, we propose and evaluate various oracle algorithms to understand the potential room for improvement in hit rate. Although machine learning is used in different caching tasks (e.g., time-to-leave prediction (Alici, Altingovde, Ozcan, Cambazoglu, & Ulusoy, 2012), refreshing (Jonassen & Bratsberg, 2012), and admission (Ozcan, Altingovde, Cambazoglu, Junqueira, & Ulusoy, 2012)), our approach is the first that employs machine learning in result caching and eviction.

Our experiments are conducted using a large, real-life query log obtained from Yahoo Web Search. We evaluate our models within the state-of-the-art static-dynamic caching framework proposed in (Fagni et al., 2006). Compared to this state-of-the-art framework, the proposed approach improves the hit rate by 0.47%, which corresponds to 7.8% of the possible improvement. Although the improvement in the hit rate is quite modest, it presents a large potential financial benefit for commercial search engines.

The rest of this paper is organized as follows. In Section 2, the previous work on result caching is surveyed. In Section 3, we present the features used in our machine learning models. Section 4 presents the proposed techniques. The details of our query log and experimental setup are explained in Section 5. In Section 6, we present the results of our experiments. We present an extended discussion on the result caching problem and the results of our experiments in Section 7. Finally, we conclude the paper in Section 8.

2. Related work

The query result caching problem is investigated by Markatos (2001) for the first time in the literature. The author evaluates four eviction policies for dynamic result caching, revealing that policies that take into account both frequency and recency perform better than those that rely only on recency. The author also proposes a static caching policy.

Fagni et al. (2006) describe a new caching architecture referred to as static-dynamic caching. In this architecture, the result cache is split into a static and a dynamic part. The static part is filled with the most frequent queries using a query log while the dynamic part uses an LRU-based eviction policy. The proposed architecture is shown to outperform only static or only dynamic caches.

Result caching is often considered together with other types of caches. Saraiva et al. (2001) propose a two-level cache architecture consisting of a dynamic result cache and a dynamic posting list cache, reporting performance improvements over using only a single type of cache. Baeza-Yates, Saint-Jean, and de Moura (2003) consider a similar two-level architecture in a static caching setting. Altingovde, Ozcan, Cambazoglu, and Ulusoy (2011) coupled a result cache with a document id cache, which stores only document ids without snippets, further reducing the query traffic going to the backend search system. Long and Suel (2005) introduce, on top of result and posting list caching, a third level of cache where precomputed intersections of posting lists are stored. Li, Lee, Sivasubramaniam, and Giles (2007) propose a hybrid architecture involving result, posting list, and document caches. Tolosa, Becchetti, Feuerstein, and Marchetti-Spaccalema (2014) present a technique that couples posting lists and term intersection lists in a single static cache. Zhou, Li, Dong, Xu, and Xiao (2015) also propose a three-level cache hierarchy consisting of a result cache, posting list cache, and a term intersection cache, generated using the top-N frequent item sets (terms) mined from query logs. Five-level cache architectures that are even more sophisticated are proposed by Marin, Gil-Costa, and Gomez-Pantoja (2010) and Ozcan et al. (2012).

Result caching is considered in the context of various problems without being the main focus. For example, Cambazoglu, Altingovde, Ozcan, and Ulusoy (2012) propose last resort techniques for computing the results of previously unseen queries using already cached query results, thus potentially improving the performance and availability of the search engine. Skobeltsyn, Junqueira, Plachouras, and Baeza-Yates (2008) investigate the impact of result caching on dynamic index pruning. Puppin, Silvestri, Perego, and Baeza-Yates (2010) propose incremental caching policies that consider the backend workload of a distributed search architecture based on collection selection. Frances, Bai, Cambazoglu, and Baeza-Yates (2014) combined result caching with replication and query forwarding in a multi-site web search setting.

The freshness of large result caches has recently attracted a lot of research attention. The proposed techniques can be categorized as refreshing and invalidation techniques. In refreshing, the main goal is to predict queries whose results are likely to be stale and recompute the results of such queries using the idle cycles of the backend of the search engine (Cambazoglu et al., 2010; Jacobs & Longo, 2015; Jonassen, Cambazoglu, & Silvestri, 2012). In cache invalidation, the backend system informs the result cache about the updates on the index and stale queries are identified and expired using this information (Alici, Altingovde, Ozcan, Cambazoglu, & Ulusoy, 2011; Bai & Junqueira, 2012; Blanco et al., 2010; Bortnikov, Lempel, & Vornovitsky, 2011; Prokhorenkova, Ustinovskiy, Samosvat, Lefortier, & Serdyukov, 2014; Sazoglu, Ulusoy, Altingovde, Ozcan, & Cambazoglu, 2015). In our work, we focus on limited-capacity caches and do not consider the freshness problem.

Several works propose feature-based approaches for improving the performance of the result cache. Baeza-Yates, Junqueira, Plachouras, and Witschel (2007b) devise an admission policy for dynamic result caching. In this policy, the query length feature is used to decide whether the results of a query which is observed for the first time should be cached or not. Gan and Suel (2009) present a feature-based eviction policy for dynamic result caches. They exploit various features to classify every incoming query into query buckets, each of which is maintained as an LRU cache. These buckets are prioritized with respect to their relative hit rates and queries are evicted starting from the bucket with the lowest hit rate. Ozcan, Altingovde, and Ulusoy (2008) propose using the standard deviation of query frequency in static result caching. In their approach, the cache is filled with queries whose frequency remains stable over time instead of the most frequent queries. Lempel and Moran (2004) use probabilistic features for prefetching successive result pages. Ma et al. (2014) proposed a prefetching-aware admission policy for result caches. In their approach, two features that are based on query prefetching history are used to decide which queries to cache and how to manage their admissions. Prokhorenkova et al. (2014) devise a staleness degree feature based on query history to determine cache degradation and to refresh search engine result pages accordingly. The techniques proposed in our work are also feature-based. However, in our work, we evaluate a large number of features under a machine learning framework, combining the predictive power of individual features.

Machine learning has been used in the literature for various result caching tasks. In a recent study, Jonassen et al. (2012) devise a machine learning framework for predicting stale queries in a result cache. Queries that are predicted to be stale are scheduled to optimize certain metrics such as query degradation and response latency. Alici et al. (2012) develop a machine learning model to learn the update patterns in query results. Based on the learned model, the time-to-leave values of queries are set in an adaptive manner on a per-query basis instead of setting a fixed time-to-leave value for all queries. Ozcan, Altingovde, Cambazoglu, and Ulusoy (2013) perform singleton query prediction using machine learning. The predictions are then used to decide which queries should be admitted to a dynamic result cache. To the best of our knowledge, machine learning has not been used before for result caching or eviction in dynamic result caching, both of which forming the focus of our work.

3. Features

The machine learning models that we build in our work rely on a large number of features, which we will briefly explain in this section. For clarity of the presentation, we classify these features into six categories: query, session, index, term frequency, and query frequency. Table 1 provides a list of the extracted features.

Query features. These features are derived from the query. Therefore they do not change values with recurrent submissions of the query. The QUERY_LENGTH and TERM_COUNT features denote the number of characters and terms in the query string, respectively. PROTOCOL_PRESENT is a binary feature, which takes the value of 1 if the query string contains the substring “HTTP” or “FTP”, or the value of 0, otherwise. DOMAIN_PRESENT is another binary feature that takes the value of 1 if the query contains any of the top-level domain names, or the value of 0, otherwise. MISSPELLED is also a binary feature and denotes whether the query is misspelled or not. AVG_TERM_LENGTH is the average number of characters in a query term. PAGE_NUMBER refers to the result page number requested in the query. QUERY_TIME denotes the hour of the day the query was submitted.

Session features. This set involves six features that are extracted from the past query sessions. USER_LOGGED_IN is the likelihood of the query being issued while the user is logged in. CTR is the clickthrough rate of the query, i.e., the average number of clicked results for the query. Similarly CTR_TOP_ONE refers to the clickthrough rate for the top result of the query. HIT_COUNT is the number of results matching the query. DAYTIME_COUNT refers to the likelihood that the query will be submitted during daytime, i.e., between 7:00 and 19:00. The TIME_COMPATIBILITY feature refers to whether the current submission time of the query is compatible with its submission times observed in the past. In particular, we classify a query into three groups: daytime query, nighttime query, or anytime query. If a query is submitted during daytime more than 80% of the time in the observed portion of the query log, then it is considered as a daytime query.[2] The same strategy is adopted using nighttime query submissions for classifying a query as a nighttime query. If a query is not classified as daytime or nighttime, then it becomes an anytime query. If a query is classified as an anytime query or if the current query is submitted at daytime/nighttime and it was classified within the compatible time group (i.e., daytime/nighttime query), the TIME_COMPATIBILITY feature takes the value of 1, or the value of 0, otherwise.

[2] The 80% cutoff was empirically found after some observations.

Table 1
The features used by the machine learning models .

Type	Feature	Description
Query	QUERY_LENGTH	Number of characters in the query string
	TERM_COUNT	Number of terms in the query string
	PROTOCOL_PRESENT	Presence of a protocol string in the query string
	DOMAIN_PRESENT	Presence of a domain name in the query string
	MISSPELLED	Presence of misspelling
	AVG_TERM_LENGTH	Average number of characters in query terms
	PAGE_NUMBER	Requested result page number
	QUERY_TIME	Hour of the day the query was submitted
Session	USER_LOGGED_IN	Whether the user is logged in or not
	CTR	Clickthrough rate
	CTR_TOP_ONE	Clickthrough rate for the top result
	HIT_COUNT	Number of matching results
	DAYTIME_COUNT	Daytime query frequency
	TIME_COMPATIBLILTY	Daytime/nighttime compatibility
Index	MIN_POSTING_COUNT	Number of postings for the rarest term
	MAX_POSTING_COUNT	Number of postings for the most common term
	AVG_POSTING_COUNT	Average posting list size of query terms
Term freq.	MIN_TERM_FREQ_MINUTE	Min. query term freq. in the last one minute
	MAX_TERM_FREQ_MINUTE	Max. query term freq. in the last one minute
	AVG_TERM_FREQ_MINUTE	Avg. query term freq. in the last one minute
	MIN_TERM_FREQ_HOUR	Min. query term freq. in the last one hour
	MAX_TERM_FREQ_HOUR	Max. query term freq. in the last one hour
	AVG_TERM_FREQ_HOUR	Avg. query term freq. in the last one hour
	MIN_TERM_FREQ_DAY	Min. query term freq. in the last one day
	MAX_TERM_FREQ_DAY	Max. query term freq. in the last one day
	AVG_TERM_FREQ_DAY	Avg. query term freq. in the last one day
Query freq.	QUERY_FREQ	Query frequency
	QUERY_FREQ_MINUTE	Query frequency in the last one minute
	QUERY_FREQ_HOUR	Query frequency in the last one hour
	QUERY_FREQ_DAY	Query frequency in the last one day

Index features. These features are obtained by aggregating the posting list lengths of the query terms in various ways. MIN_POSTING_COUNT, MAX_POSTING_COUNT, and AVG_POSTING_COUNT are obtained by computing the minimum, maximum, and average posting list sizes, respectively.

Term frequency features. These features are related to the observed frequency of query terms in the past. As in the case of index features, we use the minimum, maximum, and average functions to aggregate the frequencies of individual query terms. Similar to the session features, these features are maintained in an online fashion. In particular, we compute the aggregate frequencies using a time window spanning the last one minute, one hour, and one day before the submission time of the current query.

Query frequency features. We use the same method described above to compute the frequency of the query in the last one minute, one hour, and one day. We also compute the frequency over all past query occurrences.

4. Techniques

In this section, we present several caching strategies for the query result caching problem with the objective of maximizing the hit rate. To this end, we first consider two extreme result cache organizations: fully static and fully dynamic result caching. Then, we provide techniques for the state-of-the-art static-dynamic caching approach.

For each of the mentioned cache organizations, we present the evaluated techniques under three headings: baseline, oracle, and proposed techniques. The baseline techniques are selected from those that are previously proposed in the result caching literature. The oracle techniques provide an upper bound for the cache hit rates that can be achieved by the proposed strategies under different assumptions.

4.1. Static result caching

In static result caching, the cache is filled with queries in an offline fashion, prior to deployment of the cache. The basic strategy is to fill the cache with queries according to a query quality metric that is expected to maximize the hit rate. Herein, we consider six different static result caching techniques.

4.1.1. Baseline techniques

Offline LRU (Off–LRU): Off–LRU is the underlying technique for recency. In Off–LRU, most recent queries are assumed to have high probability of being resubmitted. Thus, such queries are used to fill the static cache.

Most frequently used (MFU): MFU is the underlying technique for frequency (Markatos, 2001). MFU is based on the assumption that the most frequent queries in the past are more likely to be submitted again in the future than other queries. Thus, the static cache is filled with the most frequent queries in the query log.

Query deviation sorted (QDEV): This strategy is based on the work presented in Ozcan et al. (2008). In their work, the authors emphasize that frequency-based strategies have the disadvantage of over-valuing the bursty behavior in the query traffic and propose a query stability metric. To this end, in order to calculate the stability of a query, the query log is divided into fixed-length time frames. The query occurrences within each time frame is considered as a unit and the variation in query occurrences between time frames is calculated for each query using the standard deviation of query frequencies. Queries having the lowest variation are considered as the best candidates for admission into the static cache.

4.1.2. Oracle techniques

Theoretical oracle (TO): In order to present a tighter upper bound for the static caching problem, we rely on a clairvoyant caching technique: if the future frequencies of every query were known, an optimal solution to the static caching problem could be achieved. That is, given a limited-capacity cache, it would be possible to select the best set of queries to keep in a static cache for achieving the maximum possible hit rate. To this end, in the TO technique, we select the most frequent queries in the test set for filling the static cache. Although TO is an optimal static caching technique, a practical implementation of TO is impossible since it requires having perfect knowledge of future query occurrences.

Practical oracle (PO): In this technique, in order to provide a more refined upper bound, we again select the most frequent queries in the test set when filling the static cache. However this time, we restrict ourselves to the test queries that also occur in the training set.

4.1.3. Proposed techniques

Machine learned static caching (MLSC): In this strategy, we propose a machine learning approach to the static result caching problem. For our classifier, we define each query occurrence as an instance and the next arrival time (IAT-Next) of each instance as the class label. The IAT-Next of a query is defined as the number of queries between two occurrences of a query in the query stream.

Our machine learning strategy for selecting the content of the static cache is as follows: We first fit a regression model using the IAT-Next of the query occurrences in the training set. Then, we predict a next occurrence time (IAT-Next) for each query that appears in the training set using this regression model. In our approach, the queries with smaller IAT-Next values are expected to appear earlier than the queries with larger IAT-Next values. Consequently, the queries with smaller IAT-Next values would be encountered more frequently than the queries with larger IAT-Next values. We then use the predicted IAT-Next values for finding the estimated test-phase frequencies of each training query in the dataset and use this value as the admission metric for the static cache.

Note that, MLSC and Off-LRU use very similar approaches to fill the static cache. While the Off-LRU technique uses the training set frequencies, MLSC uses the predicted frequencies of past queries. The difference between our machine learned caching technique and Off-LRU is mainly in the assumption that queries carry characteristic markers which can be extracted and used for training a machine learning model, and that we may use this model to improve the hit rate. In this respect, the performance of MLSC can also be considered as a validation of the benefits of using feature-based caching methods.

4.2. Dynamic result caching

In this section, we consider the dynamic result caching problem and examine several dynamic caching techniques. In general, dynamic result caching can be considered as complementary to static result caching. In static result caching, it is not possible to change the queries that are in the cache after deployment, while in dynamic result caching, the queries can be evicted from the cache in order to free up space for admitting more recent and prospective queries.

4.2.1. Baseline techniques

Least recently used (LRU): This technique captures the query recency. The cache maintains queries that are issued more recently while least recently issued queries are evicted from the cache.

4.2.2. Oracle techniques

Belady's algorithm (BELADY): If one knows the future occurrences of queries, the best possible eviction strategy for a limited-size dynamic cache is to evict the queries that would be referenced latest in the future. This optimal caching strategy is referred to as the clairvoyant algorithm or Belady's algorithm (Belady, 1966).

4.2.3. Proposed techniques

Machine learned dynamic caching (MLDC): In this technique, we model the dynamic result caching problem with a machine learning approach. In our approach, we try to predict the next arrival time (IAT-Next) of each query and make the admission and eviction decisions based on the assumption that the queries that are expected to occur latest in the future would be the best candidates for eviction. We then define the IAT-Next prediction problem as a regression problem. To this

end, we define each query occurrence as an instance and IAT-Next of an instance as the class label, where the IAT-Next of a query is defined as the number of query occurrences between the two occurrences of a query.

As our machine learned caching strategy, we use a two-classifier approach for predicting the IAT-Next value of queries. First, we train a singleton classifier in order to predict the singleton queries, where queries that appear only once in a query stream are defined as singleton. Then, we train a second classifier for non-singleton queries in the training set for finding a regression model for the IAT-Next values. The rationale behind using a two-classifier approach is that, the singleton queries appear only once in the dataset and do not have any associated inter-arrival times. Since these queries are uninformative with regard to IAT-Next, if they were used in training, the resulting IAT-Next regression model would be biased and inaccurate.

The singleton classifier maps each test instance to the interval [0, 1] by fitting a regression model, where the class label 0 in the training set denotes a singleton query and the class label 1 denotes a non-singleton query. The predictions made based on this classifier will provide a probability estimate for each test query being a singleton or not.

The second classifier processes only non-singleton queries in the training set for fitting a second regression model, where the class labels for this classifier are the IAT-Next values for each query occurrence. For the test set, the same labels are the objectives to be predicted. In our approach, these predictions would be used as the quality metric of a query during eviction; it would be more beneficial to evict queries that are expected to arrive later than the others. Note that, in our approach, the problem of predicting the singleton queries is a subset of the latter problem (i.e., next arrival time prediction) in regard to the information to be predicted since both classifiers compute an estimation for the future expectancy of test queries. However, since the second classifier is trained only for the non-singleton queries, singleton prediction is also the superset of the latter problem in terms of the instance size.

In order to combine the results of these two classifiers, we used two different approaches. In the first approach, we used the singleton classifier as an admission policy and the IAT-Next classifier as an eviction policy. According to this admission policy, the queries that are predicted as singletons are not admitted to the cache, and according to the eviction policy, the result cache is ordered using the IAT-Next predictions, which are generated by the IAT-Next regression model. In the second approach, the predictions of the singleton classifier are used as the support values for the IAT-Next regression model. The latter approach constructs an eviction policy using a linear combination of the two machine learning models. Our experimental results show that the latter approach performs consistently better than the former approach. For this reason, in the forthcoming discussions, our machine-learned dynamic caching technique will be limited to the second approach.

During our experiments, we have made an important observation regarding the performance of the machine learned dynamic result caching. When MLDC is used as the dynamic caching technique, the misclassified infrequent queries (i.e., queries that are falsely predicted as popular queries) pollute the cache and degrade the performance of the result cache due to the fact that such queries would continue to rank better than some more suitable queries, preventing those queries from ever being admitted to the result cache.

In order to prevent the cache pollution due to misclassification of infrequent queries, we propose a segmentation method for MLDC that honors the LRU policy more actively. According to our segmentation method, each time when a fixed number of queries is processed, a fresh cache partition, which we call a segment, is started. That is, all newly processed queries will only be written to the currently active segment while the old segments become stale and kept only until all of their content either got resubmitted (the resubmitted queries will be moved to the active segment) or evicted (the evicted queries will be removed from the cache). Starting a new segment can be considered as starting a new dynamic cache during execution. Additionally, whenever a query needs to be evicted from the result cache, the eviction decisions are given only on the queries that are in one of the inactive segments. This way, the segmentation provides the machine learned dynamic caching a method for evicting misclassified queries, preventing a permanent pollution of the cache. Note that, in this strategy, all segments are still governed by the same policy, while by honoring the LRU, it is possible to evict old and polluting queries from the cache without any interference with the query ordering.

4.3. Static-dynamic result caching

In the current caching literature, static-dynamic caching (Fagni et al., 2006) (SDC) is considered as the state-of-the-art result caching technique. In this subsection, we apply our machine learning approach to SDC. To this end, we will evaluate several techniques that work in conjunction with SDC.

4.3.1. Baseline techniques

Static-dynamic cache (SDC): According to SDC (Fagni et al., 2006), the result cache is divided into two segments: a static segment and a dynamic segment. In essence, SDC uses the static segment to answer frequently submitted queries while the LRU-based dynamic segment is used to answer unexpected, bursty queries. The ratio of the static and dynamic segments play an important role in the performance of SDC. In the literature, the best ratio for this division is found through experimentation and may vary depending on the query submission characteristics of different datasets. Herein, we also relied on the query logs to find a good partitioning between static and dynamic segments.

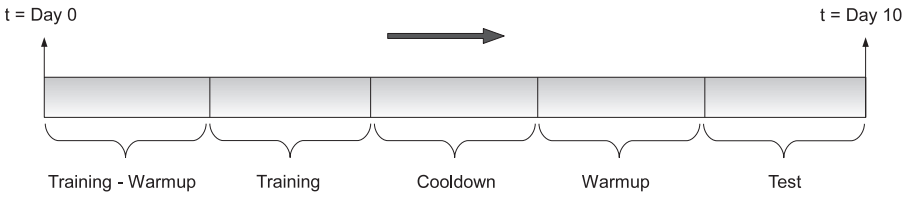


Fig. 1. The temporal organization of different phases in our experimental setup.

4.3.2. Oracle techniques

SDC-dynamic oracle (SDC-BELADY): This policy is based on the SDC strategy, where Belady's Algorithm is used as the eviction policy in the dynamic segment. The static segment is created using the most frequent queries in the training set. The performance of this policy provides insight concerning the room for improvement in the dynamic segment of SDC.

SDC-static oracle (SDC-PO): In this strategy, similar to the strategy presented in Section 4.1, we set the content of the static segment using the most frequent queries in the test set provided that those queries also appear in the training set. The dynamic segment is governed by a LRU-based eviction policy. The performance of this policy provides an insight concerning the room for improvement in the static segment of SDC.

4.3.3. Proposed techniques

Static machine learned cache + LRU (SML+LRU): In this policy, we apply our proposed machine learning approach to the static segment of SDC. The machine learning approach for the static segment is the same method presented in Section 4.1, while the dynamic segment uses an LRU-based eviction policy. During our evaluations we also experimented with the orthogonal caching technique: Off-LRU + MLDC. However, the hit rates produced by this technique is very similar to SML + LRU. In order to provide a clearer presentation, we omit the results of this technique in the upcoming discussions.

Machine learned static-dynamic cache (MLSDC): In this policy, we apply our proposed machine learning approach to both static and dynamic segments of SDC. The training method for each segment are the same approaches presented in Sections 4.1 and 4.2, respectively.

5. Data and setup

We conduct extensive experiments on a realistic dataset to evaluate the caching techniques presented in Section 4. In this section, we first introduce the query log and discuss the experimental setup. Then, we present the machine learning setup used in our experiments.

5.1. Query log

The dataset is constructed using queries issued to Yahoo Web Search during a 10-day period. We apply several preprocessing operations before experimentation. Punctuation marks in the queries are cleared and query strings are normalized by converting all characters to lowercase. All query terms are rearranged in alphabetical order in order to eliminate dissimilarity as a result of term positions. Finally, spelling correction is applied to all queries.

5.2. Experimental setup

In order to test the effectiveness of the presented caching techniques, we divide the dataset into five parts, which we dub phases: training-warmup, training, cooldown, warmup, and test. Fig. 1 illustrates these phases.

The first part of the query log is used in the training-warmup phase. Queries in the training-warmup phase are used for warming up the cache prior to the training phase. The purpose of this phase is two-fold: First, the training-warmup phase allows the following training phase to start with an already filled cache, leading to a more realistic caching scenario. Second, as noted earlier in Table 1, several features used in this work are defined over time frames which may have a span up to one day. The training-warmup phase is used to exclude queries with incomplete information.

The queries in the training phase are used for generating two machine learning models. Each query in the training phase is labeled with two values: a binary value denoting whether the query is a singleton or not, in order to fit a singleton regression model, and a numeric value denoting the future inter-arrival times (IAT-Next) of each query in order to fit an IAT-Next regression model. Using these models, the proposed machine-learned caching techniques will make admission and eviction decisions.

Although the machine-learned caching models, in a realistic setting, are designed to deal with an infinite query stream, the models are generated using a finite-size training data. This phenomenon leads to an anomaly during the training. The last occurrence of each query in the training set is marked with an infinite IAT-Next value due to the fact that the training set is finite. In practice, a subset of such queries in the training set are likely to reappear after some time in the future. In order to alleviate this issue, we employ a cooldown phase, which provides the machine learning models some future

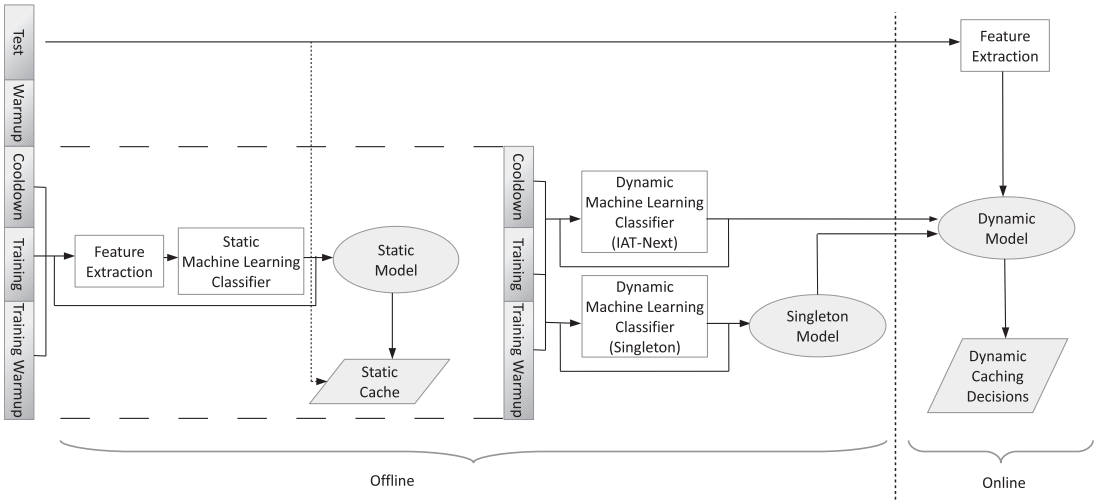


Fig. 2. The deployment strategy for the machine learned result caching framework.

knowledge about the query stream. This way, the generated machine learning models can capture the characteristics of the query stream better.

The last two parts of the query log are used as the warmup and test phases. The queries in the warmup phase are used to refill the cache prior to testing in order to prevent a cold-start problem. The queries in the test phase are used to evaluate the performance of different caching techniques.

Fig. 2 illustrates the deployment strategy for the proposed machine learned result caching framework. It also summarizes how different parts of the dataset are utilized. As indicated in Fig. 2, the framework consists of an offline and an online component. The offline component extracts various query specific features from a given query log and trains two machine learning models. A static result caching model which predicts the IAT-Next values for the query log and utilizes those predictions to select the most "frequently expected" queries in the static cache. The queries that are to be placed in the static cache are determined from the predicted IAT-Next values of the query log. The training warmup, training, and cooldown parts of the dataset are used to emulate the query log in our dataset while training the machine learning models.

The same classifier model above, also depicted as the dynamic IAT-Next classifier, is then used to produce a dynamic result caching model. The produced model makes IAT-Next predictions for any incoming query in the query stream. We also train a dynamic singleton classifier to improve the effectiveness of the dynamic model. This classifier models whether a query is singleton or not. Combination of these two machine learning models are then used as an admission/eviction policy for the dynamic result cache. The test dataset is used to simulate the query stream. For each incoming query, first, the static cache is checked for the availability of the query. If the incoming query produces a static cache miss, features are extracted and an admission decision is given using the predicted IAT-Next value of the query.

We perform two tests in order to evaluate the statistical significance of the generated machine learning models. First, for both singleton and IAT-Next classifier models, we perform a sampled two-tailed z-score test of the possible error rates in our predictions. The proposed singleton classifier model predicts whether a query in the dataset is a singleton or not with a 11.243 z-score, corresponding to a p-value of almost 0. The proposed IAT-Next classifier model, which is used for both admitting queries into the static cache and predicting the IAT-Next values for any incoming query has a -1.817 z-score, corresponding to a p-value of 0.859.

Second, to investigate whether our predictions produce a distribution similar to that of the query log, we perform a Kullback–Leibler (KL) divergence test (Kullback, 1997; Kullback & Leibler, 1951). According to this test, the singleton classifier model has a an entropy of 0.049 relative the original data distribution. This shows that the proposed model has an extremely similar data distribution to the query log. For the dynamic IAT-Next classifier model, we first create 1000 equal-depth buckets and place the predicted IAT-Next values of the query stream into the corresponding buckets. For the query log, we also discretize each query's inter-arrival time using the same strategy. Using the bucket contents, the KL divergence test shows that the predictions of the proposed caching technique has a relative entropy of 0.128 against the distribution of the query log. This confirms that the predicted inter-arrival time distribution is close to that of the original query stream.

5.3. Machine learning setup

We experimented with several machine learning tools, such as Weka (Hall et al., 2009), Orange (Curk et al., 2005), Liblinear (Fan, Chang, Hsieh, Wang, & Lin, 2008), and GBDT (Ye, Chow, Chen, & Zheng, 2009) to train our machine learning models. We analyzed the results of several machine learning algorithms such as multi layer perceptron, pace regression,

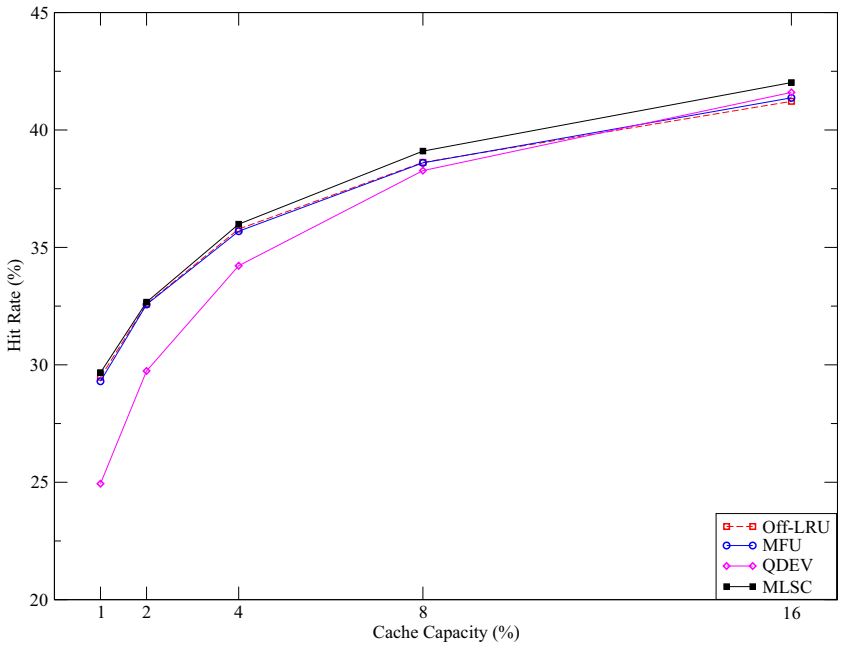


Fig. 3. Hit rates attained by different static caching techniques as the cache capacity varies.

support vector machines, k-nearest neighbors, logistic regression, and gradient boosted decision trees in order to find the most suitable learning algorithm.

Our experiments indicated that the learning algorithms provided by both Weka and Orange are not suitable for evaluating large scale data due to their poor running time performance. In terms of accuracy, GBDT performed consistently better than Liblinear in all experiments. Thus, we have decided to use the stochastic gradient boosted decision trees algorithm (Friedman, 2002) provided by GBDT for training the proposed machine-learned caching models.

GBDT is one of the most commonly used learning algorithms in machine learning today. The results produced by GBDT are simple and easy to interpret. Also, the models created by decision-tree-based methods are non-parametric and non-linear. For our experiments, we have used a version of GBDT described in (Ye et al., 2009). After evaluating the effectiveness of GBDT with different number of decision trees and different number of nodes in each decision tree, taking into account both running time and regression accuracy, we set the maximum number of decision trees as 40 and the number of nodes in each tree as 20 in our experiments.

6. Results

In this section, we present a comprehensive experimental evaluation and comparison of the discussed caching techniques. The proposed experiments are categorized within three groups: static, dynamic, and static-dynamic caching. We perform our experiments with varying cache capacities. We select the cache capacity as a function of distinct queries in the test set. During our evaluations, we have used 1%, 2%, 4%, 8%, and 16% of the number of test queries as the cache capacity. As the evaluation metric, we use the cache hit rate (the percentage of queries served by the cache). Additionally, we conducted several experiments to evaluate the effects of the proposed machine-learned result caching technique on the query response time of a search engine.

6.1. Static caching

Fig. 3 summarizes the hit rates of different static caching techniques with various cache capacities. According to our experiments, Off-LRU and MFU perform almost equally. QDEV performs the worst for small cache capacities. This is due to the fact that some infrequent queries have significantly better stability values than some popular queries. However, when the cache capacity is large enough, QDEV performs better than both Off-LRU and MFU since the cache is large enough to accommodate both unpopular-yet-stable and popular queries at the same time.

The experiment also shows that the proposed MLSC technique performs consistently better than the remaining techniques. For small cache capacities, the improvement attained by MLSC is relatively small. However, as the cache capacity grows larger, MLSC increases the hit rate considerably (up to 0.66% when the cache capacity is 16%).

Table 2 shows the most important 10 features in MLSC (provided by the GBDT toolkit). Two of the top three features are variants of query frequency, validating the importance of frequency in static caching. Additionally, word count, page number,

Table 2
The most important 10 features for the machine-learned static caching (MLSC) technique.

Rank	Feature	Imp.
1	QUERY_FREQ	100.00
2	TIME_COMPATIBLE	35.38
3	QUREY_FREQ_DAY	34.80
4	WORD_COUNT	17.81
5	MIN_POSTING_COUNT	17.15
6	PAGE_NUMBER	11.94
7	QUERY_FREQ_HOUR	11.92
8	QUERY_LENGTH	11.24
9	CTR	9.66
10	USER_LOGGED	8.84

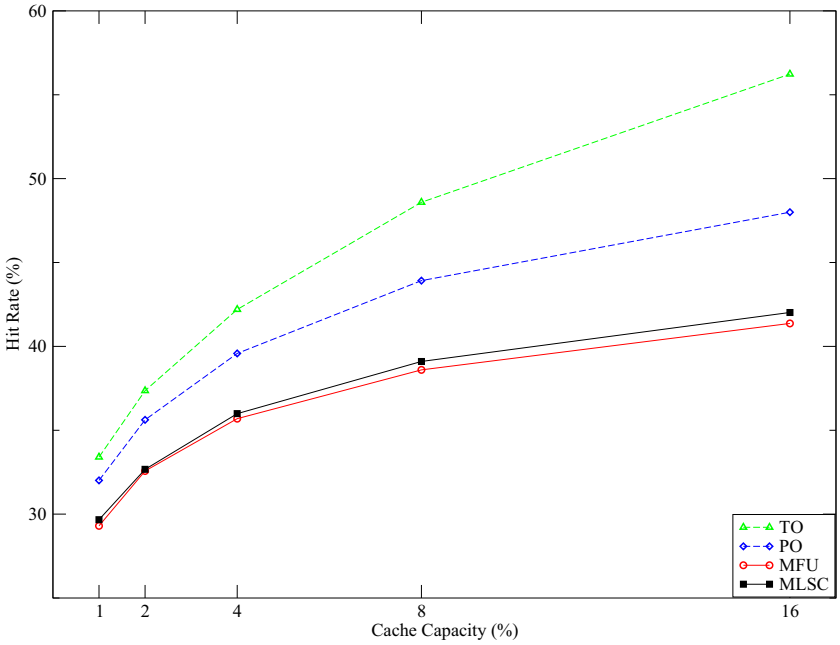


Fig. 4. Comparison of the proposed machine-learned static caching technique (MLSC) with two oracle techniques (PO and TO).

query length, and inverted list sizes of query terms are identified among the characteristics that may be used to improve the performance of a static result cache.

Fig. 4 compares the proposed MLSC technique with two oracle static caching techniques (TO and PO) and the baseline MFU technique. This new experiment indicates that, although MLSC was previously shown to increase the hit rate up to 0.66% with respect to its closest competitor, this improvement constitutes 9.6% of the potential room for improvement (when the hit rate of PO is taken as an upper bound). Moreover, the comparison of the two oracle techniques, TO and PO, demonstrates that TO improves the hit rate by 8%, on average, with respect to PO, due to the admission of frequent queries that only appear in the test set to the static cache. Obviously, TO is not a practical oracle due to exploitation of future information about the query stream.

6.2. Dynamic caching

As pointed out earlier in Section 4.2.3, the machine-learned dynamic caching technique (MLDC) suffers from false positives. i.e., infrequent queries admitted to the cache. Since a purely machine-learned cache has no means to evict such queries, they reside in the cache indefinitely, shrinking the effective cache size. To alleviate this problem, we use the cache segmentation method presented in Section 4.2.3. Before conducting our experiments on dynamic caching, we first perform a set of initial experiments using different cache capacities and segment sizes. Our aim here is two-fold: to evaluate the effect of varying segment sizes on MLDC and to discover whether there is a fixed best segment size for different cache capacities.

Fig. 5 presents the performance of MLDC for different cache capacities and segment sizes. For the cache capacity of 1%, increasing the segment size up to 30,000 queries also increases the hit rate. However, when the segment size is increased further, the hit rate starts to decline monotonically, down to 2.7% when the segment size is equal to the test set size.

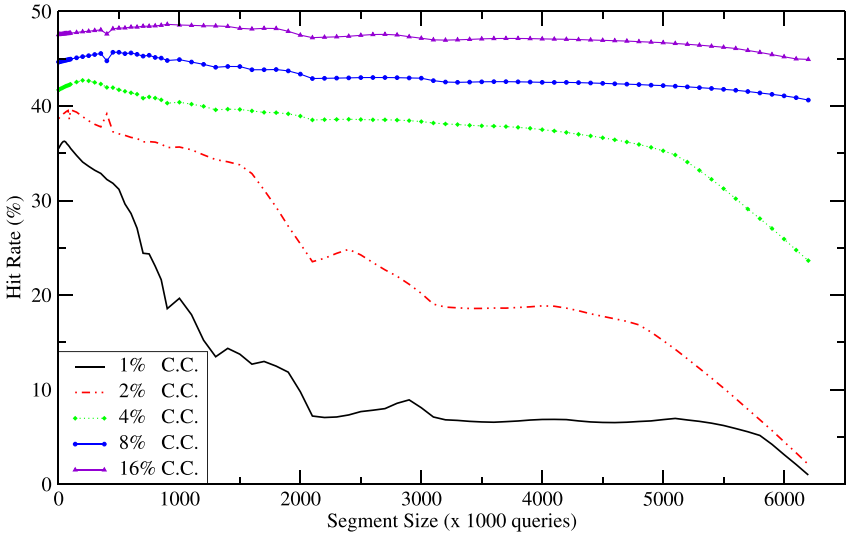


Fig. 5. Effect of segment size on hit rate in machine-learned dynamic caching.

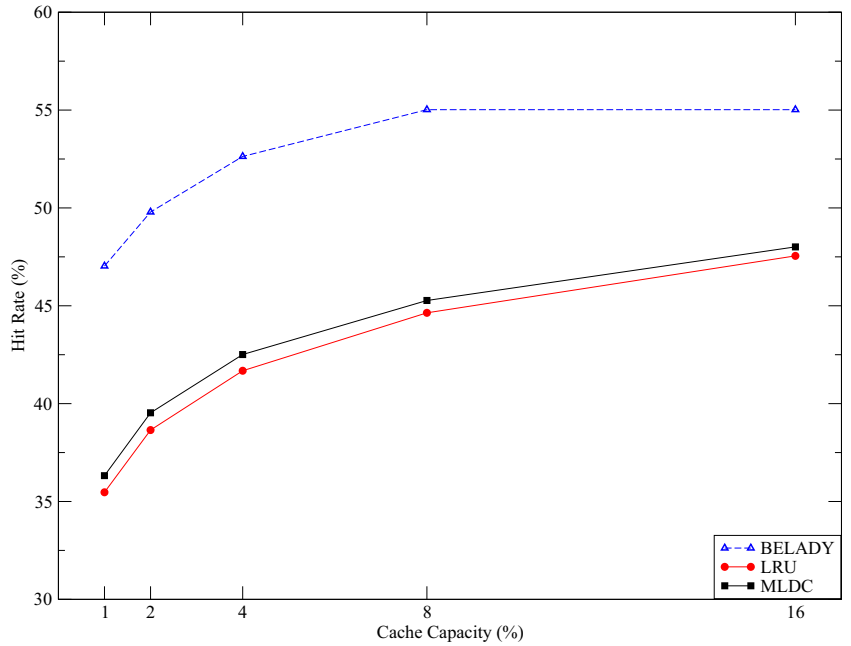


Fig. 6. Comparison of the proposed machine-learned dynamic caching technique (MLDC) with a baseline (LRU) and an oracle (BELADY) technique.

Similar results are observed for 2% and 4% cache capacities. Setting the segment size too small leads to degeneration of the machine-learned caching technique, ultimately to an LRU-like technique, eliminating the benefits of the machine-learned model. Setting the segment size too large leads to cache pollution due to misclassification of infrequent queries and inability to remove them adequately from the dynamic segment. The effects of the cache pollution are more visible when the cache capacity is small. For larger cache capacities, even when very large segment sizes are used, the performance degradation because of cache pollution is almost negligible due to the fact that despite cache pollution due to misclassified queries, larger caches can still accommodate the most popular queries within the dynamic cache.

In Fig. 6, the hit rate attained by MLDC is compared with the baseline (LRU) and oracle (BELADY) dynamic caching techniques for different cache capacities. During this experiment, we have applied the aforementioned segmentation method to MLDC. To this end, we first established a best segment size by training a machine learning model for MLDC using the training data and testing it over the same training data. Then, we have used the best segment sizes for each cache capacity for MLDC. The results show that the proposed MLDC technique performs consistently better than the baseline LRU policy

Table 3
The most important 10 features for the machine-learned dynamic caching (MLDC) technique.

Rank	Singleton prediction		Next arrival time prediction	
	Feature	Imp.	Feature	Imp.
1	QUERY_FREQ	100.00	TIME_COMPATIBLE	100.00
2	TIME_COMPATIBLE	41.12	CTR_TOP_ONE	49.14
3	WORD_COUNT	17.26	MIN_TERM_FREQ_HOUR	43.21
4	USER_LOGGED	14.86	WORD_COUNT	42.20
5	PAGE_NUMBER	10.88	QUERY_FREQ	41.16
6	CTR	10.21	USER_LOGGED	33.19
7	QUERY_LEN	9.10	HIT_COUNT	30.69
8	CTR_TOP_ONE	9.02	CTR	27.57
9	MIN_TERM_FREQ_DAY	8.52	MIN_TERM_FREQ_DAY	26.93
10	HIT_COUNT	8.29	SPELL_CORR	22.82

for all cache capacities. For the cache capacity of 16%, MLDC improves the hit rate by 0.65% with respect to the baseline. This constitutes 7.4% of the potential room for improvement with respect to the oracle.

Table 3 shows the most important 10 features and their relative importance values for the singleton prediction model and the IAT-Next regression model for MLDC. In the table, Columns 1–3 denote the most important features for singleton prediction, and Columns 4–6 denote the most important features for IAT-Next regression. We note that query frequency is selected as the most important feature in singleton prediction, while other features, such as TIME_COMP (daytime/nighttime compatibility of a query) and CTR_TOP_ONE (clickthrough rate for the top result) are selected as the most important features for IAT-Next regression.

6.3. Static-dynamic caching

We note that BELADY is an optimal cache eviction policy not only for the dynamic setting, but also for the static-dynamic setting. Also, two more clairvoyant techniques, SDC-BELADY and SDC-PO, are evaluated here to provide more insight about the possible room for improvement in the static-dynamic caching problem, for the dynamic and static cache segments of SDC, respectively.

Fig. 7 shows the hit rates of different static-dynamic caching techniques for varying cache capacities. A striking observation is that, although BELADY is an optimal caching strategy for the static-dynamic setting, both SDC-BELADY and SDC-PO oracles outperform it at large cache capacities. This is due to the fact that, being a dynamic caching technique, BELADY is prone to non-compulsory misses (misses that occur when a query is encountered for the first time). The static segments of the latter two algorithms do not cause any non-compulsory misses since the static segments are precomputed and deployed

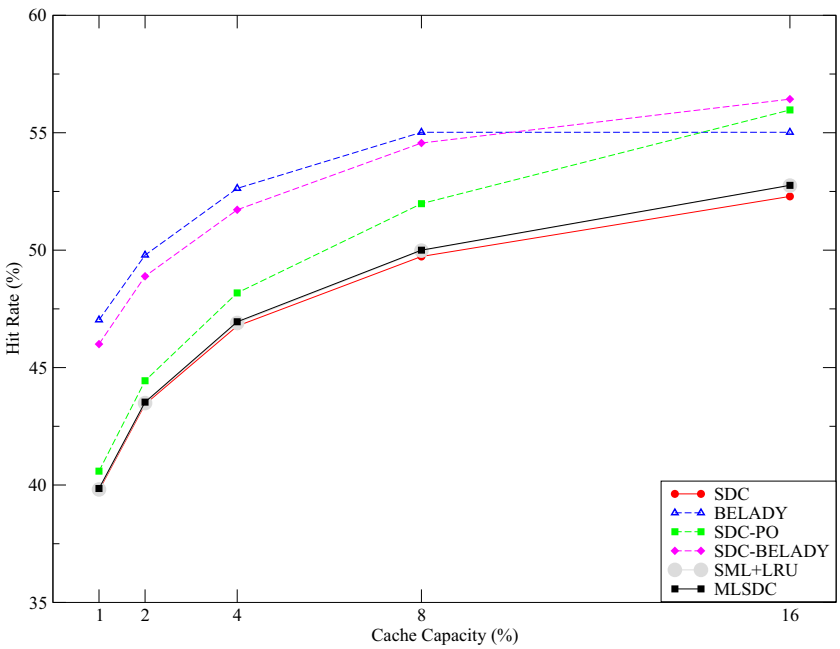


Fig. 7. Hit rates attained by different static-dynamic caching techniques as the cache capacity varies.

Table 4
The most important 10 features for the machine-learned static-dynamic caching (SML+LRU and MLSDC) techniques.

1% Cache capacity				
Rank	Singleton prediction		Next arrival time prediction	
	Feature	Imp.	Feature	Imp.
1	QUERY_FREQ	100.00	QUERY_FREQ_DAY	100.00
2	QUERY_FREQ_DAY	62.11	WORD_COUNT	31.70
3	QUERY_FREQ_HOUR	24.88	QUERY_FREQ	22.66
4	WORD_COUNT	23.08	MIN_TERM_FREQ_DAY	21.42
5	MIN_POSTING_COUNT	19.83	MIN_POSTING_COUNT	20.14
6	PAGE_NUMBER	16.14	QUERY_FREQ_HOUR	18.84
7	AVG_POSTING_COUNT	8.61	SPELL_CORR	15.82
8	QUERY_FREQ_DAY	6.32	AVG_POSTING_COUNT	13.33
9	SPELL_CORR	4.97	AVG_TERM_FREQ_DAY	11.05
10	MIN_TERM_FREQ_MIN	2.79	PAGE_NUMBER	9.43
16% Cache capacity				
Rank	Singleton prediction		Next arrival time prediction	
	Feature	Imp.	Feature	Imp.
1	QUERY_FREQ	100.00	QUERY_FREQ_DAY	100.00
2	QUERY_FREQ_DAY	37.11	WORD_COUNT	29.63
3	TIME_COMPATIBLE	35.72	QUERY_FREQ	21.96
4	WORD_COUNT	19.41	HIT_COUNT	20.64
5	MIN_POSTING_COUNT	17.03	MIN_POSTING_COUNT	19.68
6	QUERY_FREQ_HOUR	14.93	TIME_COMPATIBLE	19.62
7	PAGE_NUMBER	14.42	QUERY_FREQ_HOUR	18.24
8	QUERY_LENGTH	9.30	SPELL_CORR	15.57
9	DAYTIME_COUNT	7.64	AVG_POSTING_COUNT	13.13
10	AVG_POSTING_COUNT	5.75	DAYTIME_COUNT	11.08

in an off-line manner. In a practical cache deployment scenario, the miss rate due to the non-compulsory misses would be almost negligible.

The comparison of the two machine learning techniques with SDC shows that both techniques consistently outperform SDC for all cache capacities. For the cache capacity of 16%, MLSDC improves the hit rate with respect to SDC by 0.47%, which constitutes 7.8% of the possible room for improvement against the best oracle. When the proposed machine-learned caching techniques are compared with each other, they are seen to perform almost equally. The addition of a machine learning strategy to the dynamic segment does not seem to improve the performance of SDC and, although MLSDC performs consistently better than SML+LRU, the improvements are quite small. Consequently, one can argue that in a static-dynamic caching setting, the benefits of the machine-learned dynamic segment is almost completely shadowed by the improvements obtained through the machine-learned static segment.

Table 4 shows the most important 10 features for MLSDC for two different cache capacities. In the table, Columns 1–3 show the most important features and their relative importance values for singleton prediction, and Columns 4–6 show the most important features and their relative importance values for IAT-Next regression. The first 10 rows show the importance values for a cache capacity of 1% and the last 10 rows show the importance values for a cache capacity of 16%. According to the table, the most important features are similar for both singleton prediction and IAT-Next regression. Although the most important features for IAT-Next regression still contain some features that hint on mine able properties of queries, when compared to the results in Table 3, features related to query frequency play a more important role on the dynamic model when a static segment is introduced. According to these results, the addition of the static segment seems to affect the IAT-Next regression adversely, making it resemble a singleton prediction model.

6.4. Impact on query response time

One of the main aspects of employing caching in search engines is to improve the query response time. In a typical distributed search engine setting, several operations need to be performed in succession in order to respond to a user query. Upon receiving an incoming query, a central broker performs a lookup (C_{lookup}) on its result cache. In the case of cache hit, i.e., the query result can be found in the result cache, the result is fetched from the cache and returned immediately. In the case of a cache miss, i.e., the query result cannot be found in the result cache, the central broker sends the query to the search nodes. Each search node fetches the corresponding posting lists for each term in the query (C_{pl}), processes them (i.e., decompression and calculating the similarity between the query and documents) and rank them (C_{rank}). Then, the search node retrieves the best ranked documents from the disk (C_{doc}) and generates snippets for these documents (C_{snip}) and sends

Table 5
Simulation parameters.

Cost	Notation	Value
Block size	S_{block}	512 bytes
Document size	S_{doc}	512 bytes
Posting size	S_p	8 bytes
Document count	n	10
Disk seek	D_{seek}	8.5 msec
Rotational latency	D_{rot}	4.17 msec
Disk block read	D_{block}	4.883 nsec
Ranking cost per posting	$C_{scoring}$	200 nsec
Cache lookup cost	C_{lookup}	40 nsec
IAT prediction cost	C_{pred}	166 nsec
Feature extraction cost	C_{feat}	800 nsec
Snippet generation cost Per byte	C_{snip}	10 nsec
Posting list access cost	C_{pl}	$\sum_{t_i \in q}(D_{seek} + D_{rot} + D_{read} * \left\lceil \frac{\lvert I_i \rvert * S_p}{S_{block}} \right\rceil)$
Ranking cost	C_{rank}	$C_{scoring} * \sum_{t_i \in q} \lvert I_i \rvert$
Document retrieval cost	C_{doc}	$D_{seek} + D_{rot} + D_{read} * \left\lceil \frac{n * S_{Doc}}{S_{block}} \right\rceil$
Snippet generation cost	C_{snip}	$n * S_{doc} * C_{snip}$

the generated result page to the user. Each of these operations have an associated processing cost and are denoted with the above abbreviations in this work.

In order to evaluate the effects of deploying a machine-learned result cache on a search engine, we have set up a simulation framework and conducted several experiments by following the experimental framework presented in Altingovde et al. (2009); Ozcan et al. (2012). In this work, we restrict our experiments to a single search node. We assume that the node acts both as a broker and a search node and has a local document collection. For simulating the document collection, we have randomly selected 20 million web pages from the ClueWeb09 Category-B dataset (Gabrilovich, Ringgaard, & Subramanya, 2013). The terms in the collection are case-folded and punctuation marks are removed. The resulting posting list contains roughly 103 million distinct terms.

For the query log, we used the publicly available AOL query log (Pass, Chawdhury, & Torgesson, 2006) which contains queries submitted to the AOL search engine during a 12 week period. The click-data and anonymized queries are removed from the query log, and the terms in the queries are case-folded. The resulting dataset contains roughly 26 million queries.

As pointed out in Ozcan et al. (2012), it is well known that any large-scale search engine node also incorporates a sizeable posting list cache along with a result cache for reducing disk access times. In order to provide more realistic results, we assume that postings with largest size-frequency ratios are stored in an in-memory posting list cache, requiring no disk access. For this purpose, we divide the query log into two equal-sized parts. Using the first half of the query log, we have extracted the query term frequencies and assume that 0%, 25%, and 50% of the query terms are stored in the posting list cache of the search node. The remaining half of the query log is then used to run our simulations. Finally, since the network and communication costs would constitute only a fraction of the total query processing time (Altingovde et al., 2009), we omit the networking costs in our simulations.

During our experiments, we compare the query processing performance of a search node facilitating two different result caching techniques: the state-of-the-art Static-Dynamic Caching (SDC) and the proposed machine-learned result caching technique MLSDC. For SDC, an incoming query may induce two different costs on the search node depending on whether result cache lookup is a hit or a miss. A result cache hit would only cost lookup time (C_{lookup}) regardless of the query result being in the static segment or dynamic segment. In the case of a miss, the total cost of the query to the search node is $(C_{lookup} + C_{pl} + C_{rank} + C_{doc} + C_{snip})$.

In the case of the proposed result caching technique (MLSDC), the associated costs of a hit on the static and dynamic cache segments are different. First, for each of the incoming queries, a lookup is performed on the static segment, which incurs the lookup cost (C_{lookup}). If the query is a static segment miss, then a set of query-specific features need to be extracted (C_{feat}) and an IAT-Next prediction needs to be performed using the decision tree regression models (C_{pred}) along with the lookup cost. If the incoming query is a cache miss, the cost of the query to the search node is $(C_{lookup} + C_{feat} + C_{pred} + C_{pl} + C_{rank} + C_{doc} + C_{snip})$. Table 5 presents the simulation parameters for each of the cost components in our experiments. Also, a detailed summary of different result cache responses and their associated costs for the evaluated caching technique is presented in Table 6.

Table 7 summarizes the results of our simulations. In the table, column 1 represents the result cache capacity, while column 2 shows the static posting list cache capacity for the corresponding experiment. Columns 3 and 4 summarize the average response time for a query in a search node accompanied with the state-of-the-art SDC or the proposed MLSDC result caching techniques for varying posting list and result cache capacities. The last column of the table shows the relative improvement of MLSDC over SDC. The simulation results lead to two conclusions. First, our experiments clearly show that

Table 6
The associated result cache hit and miss costs for SDC and MLSDC.

Lookup result	SDC	MLSDC
Static hit	C_{lookup}	C_{lookup}
Dynamic hit	C_{lookup}	$C_{feat} + C_{pred} + C_{lookup}$
Miss	$C_{lookup} + C_{pl} + C_{rank} + C_{doc} + C_{snip}$	$C_{feat} + C_{pred} + C_{lookup} + C_{pl} + C_{rank} + C_{doc} + C_{snip}$

Table 7
Experiment results for the response time of SDC and MLSDC .

Result cache Capacity % queries	Posting list Cache capacity % index size	Average query Response time (SDC) milliseconds	Average query Response time (MLSDC) milliseconds	Relative Imp. %
1%	0%	100.205	99.116	1.1
	25%	16.299	16.006	1.8
	50%	10.490	10.299	1.8
2%	0%	97.095	95.527	1.6
	25%	15.815	15.417	2.5
	50%	10.162	9.889	2.7
4%	0%	93.490	91.990	1.6
	25%	15.286	14.878	2.7
	50%	9.804	9.538	2.7
8%	0%	88.859	86.984	2.1
	25%	14.693	14.199	3.3
	50%	9.405	9.077	3.5
16%	0%	82.443	80.194	2.7
	25%	13.979	13.474	3.6
	50%	8.929	8.590	3.8

despite the newly incurred feature extraction and prediction costs, deployment of a machine learned result cache consistently improves the query response time of an SDC-based result cache. The proposed algorithm also benefits better from the existence of a posting list cache, achieving higher rates of improvement. Second, our experiments verify that the dominating factor in query processing is the posting list access and ranking costs, and the newly incurred feature extraction and prediction costs are negligible with respect to these costs.

7. Discussion

In the literature, data sets having heavy-tailed data distributions can be treated in three parts (Brynjolfsson, Hu, & Smith, 2003) according to the frequency distribution of data items: head, torso, and tail. The queries in the search result caching problem also exhibit a heavy tailed distribution. The head queries are the most frequent queries. They form a large fraction of the overall search engine query traffic relative to their small number. The queries in the torso appear less frequently, while those in the tail appear quite rarely, possibly only a couple of times during the lifetime of a search engine query log.

The distribution of queries plays a significant role on the performance of a static-dynamic cache, where the cache is partitioned into static and dynamic segments in order to exploit the heavy tailed data distribution of queries. By admitting few queries that reside in the head of the query log in an offline manner, the static segment of the cache can directly answer a large portion of queries. This segment usually handles queries that are known to be frequent for a long time. The dynamic segment of the cache usually helps to answer relatively infrequent queries in the torso. Thus, it is mostly responsible for responding to unexpected, bursty queries. In practice, although distinguishing head queries is a relatively trivial problem, distinguishing tail and torso queries is difficult. The accurate admission of tail queries to the dynamic segment, for all practical purposes, requires either an impractically large cache capacity or clairvoyance.

The result of our experiments on the performance of the static-dynamic caching approach supports the observations mentioned above. In a static-dynamic cache setting, in the case of a cache capacity of 1%, all queries with a frequency higher than 136 are stored in the static cache. However, in the case of a cache capacity of 16%, the queries stored in the static cache have frequencies as low as 4. That is, when the cache capacity is large enough, the remaining dynamic caching problem reduces to distinguishing singleton queries from very infrequent queries. Our experiments show that the combination of machine-learned static caching and machine-learned dynamic caching presented does not yield a cumulative improvement on the hit rates. This is mainly because the improvement attained by the machine-learned static caching technique overlaps and shadows the improvement obtained by the machine-learned dynamic caching technique.

8. Conclusion

In this paper, we presented machine learning models for result caching in web search engines. These models were trained using a large number of features extracted from the query string, query logs, search index, and search session, and were exploited to make caching decisions, aiming to increase the hit rate of the cache. We evaluated different learning models for both static and dynamic result caching. In the case of static caching, the learning model is used to select queries that will be admitted to the cache. In the case of dynamic caching, the learning model predicted the likelihood of observing a query in the near future, thus facilitating admission and eviction.

The performance of the proposed learning models were evaluated using a real-life, large-scale query sample obtained from Yahoo Web Search, assuming three difference cache organizations: static, dynamic, and static-dynamic caching. The conducted experiments demonstrated that the proposed learning approach can attain improvements in hit rate compared to state-of-the-art baselines (i.e., SDC). Although the attained improvements may not appear very large in magnitude (e.g., around 1% increase in hit rate), they may still have important implications for reducing the financial costs associated with query processing in commercial search engines.

Our work also provided upper bounds regarding the hit rates that any result caching approach can attain in the best case. To obtain these bounds, we conducted experiments with oracle caching approaches that make perfect caching decisions. Within the possible room for improvement, the proposed learning-based caching approaches could increase the hit rate of baseline caching techniques by up to 9.6%.

References

Alici, S., Altingovde, I. S., Ozcan, R., Cambazoglu, B. B., & Ulusoy, O. (2011). Timestamp-based result cache invalidation for web search engines. In *Proceedings of the 34th international ACM SIGIR conference on research and development in information retrieval* (pp. 973–982). SIGIR '11. ACM, New York, NY, USA.

Alici, S., Altingovde, I. S., Ozcan, R., Cambazoglu, B. B., & Ulusoy, O. (2012). Adaptive time-to-live strategies for query result caching in web search engines. In *Proceedings of the 34th european conference on advances in information retrieval. ECIR'12* (pp. 401–412). Springer-Verlag, Berlin, Heidelberg.

Altingovde, I., Ozcan, R., & Ulusoy, O. (2009). A cost-aware strategy for query result caching in web search engines. In M. Boughanem, C. Berrut, J. Mothe, & C. Soule-Dupuy (Eds.), *Advances in information retrieval*. In *Lecture Notes in Computer Science: vol. 5478* (pp. 628–636). Springer Berlin, Heidelberg.

Altingovde, I. S., Ozcan, R., Cambazoglu, B. B., & Ulusoy, O. (2011). Second chance: A hybrid approach for dynamic result caching in search engines. In *Proceedings of the 33rd european conference on advances in information retrieval* (pp. 510–516). ECIR'11. Springer-Verlag, Berlin, Heidelberg.

Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., & Silvestri, F. (2007a). The impact of caching on search engines. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval* (pp. 183–190). ACM, New York, NY, USA.

Baeza-Yates, R., Junqueira, F., Plachouras, V., & Witschel, H. (2007b). Admission policies for caches of search engine results. In N. Ziviani, & R. Baeza-Yates (Eds.), *String processing and information retrieval*. In *Lecture Notes in Computer Science: vol. 4726* (pp. 74–85). Springer Berlin, Heidelberg.

Baeza-Yates, R., Saint-Jean, F., & de Moura, E. S. (2003). A three level search engine index based in query log distribution. In M. A. Nascimento, & A. L. Oliveira (Eds.), *String processing and information retrieval*. In *Lecture Notes in Computer Science: vol. 2857* (pp. 56–65). Springer Berlin, Heidelberg.

Bai, X., & Junqueira, F. P. (2012). Online result cache invalidation for real-time web search. In *Proceedings of the 35th international ACM SIGIR conference on research and development in information retrieval* (pp. 641–650).

Belady, L. A. (1966). A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal, 5*, 78–101.

Blanco, R., Bortnikov, E., Junqueira, F., Lempel, R., Telloli, L., & Zaragoza, H. (2010). Caching search engine results over incremental indices. In *Proceedings of the 33rd international ACM SIGIR conference on research and development in information retrieval* (pp. 82–89). ACM, New York, NY, USA.

Bortnikov, E., Lempel, R., & Vornovitsky, K. (2011). Caching for realtime search. In *Proceedings of the 33rd european conference on advances in information retrieval* (pp. 104–116). ECIR'11. Springer-Verlag, Berlin, Heidelberg.

Brynjolfsson, E., Hu, Y. J., & Smith, M. D. (2003). Consumer surplus in the digital economy: Estimating the value of increased product variety at online booksellers. *Management Science, 49*(11), 1580–1596.

Cambazoglu, B. B., Altingovde, I. S., Ozcan, R., & Ulusoy, O. (2012). Cache-based query processing for search engines. *ACM Transactions on the Web, 6*(4), 1–24.

Cambazoglu, B. B., & Baeza-Yates, R. A. (2015). *Scalability challenges in web search engines. synthesis lectures on information concepts, retrieval, and services*. Morgan & Claypool Publishers.

Cambazoglu, B. B., Junqueira, F. P., Plachouras, V., Banachowski, S., Cui, B., Lim, S., & Bridge, B. (2010). A refreshing perspective of search engine caching. In *Proceedings of the 19th international conference on world wide web* (pp. 181–190).

Curk, T., Demsar, J., Xu, Q., Leban, G., Petrovic, U., Bratko, I., … Zupan, B. (2005). Microarray data mining with visual programming. *Bioinformatics, 21*(3), 396–398.

Fagni, T., Perego, R., Silvestri, F., & Orlando, S. (2006). Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems, 24*(1), 51–78.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., & Lin, C.-J. (2008). Liblinear: A library for large linear classification. *Journal of Machine Learning Research, 9*, 1871–1874.

Frances, G., Bai, X., Cambazoglu, B. B., & Baeza-Yates, R. (2014). Improving the efficiency of multi-site web search engines. In *Proceedings of the 7th ACM international conference on web search and data mining* (pp. 3–12). WSDM'14. ACM, New York, New York, USA.

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statatical Data Analysis, 38*(4), 367–378.

Gabrilovich, E., Ringgaard, M., & Subramanya, A. (2013). FACC1: Freebase annotation of ClueWeb corpora version 1. (Release Date 2013-06-26, Format Version 1, Correction level 0), http://lemurproject.org/clueweb09/.

Gan, Q., & Suel, T. (2009). Improved techniques for result caching in web search engines. In *Proceedings of the 18th international conference on world wide web* (pp. 431–440).

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The weka data mining software: An update. *SIGKDD Exploration Newsletters, 11*(1), 10–18.

Jacobs, T., & Longo, S. (2015). A study of caching strategies for web service discovery. In *IEEE international conference on web services* (pp. 464–471). ICWS. New York, New York, USA.

Jonassen, S., & Bratsberg, S. E. (2012). Improving the performance of pipelined query processing with skipping. In X. Wang, I. Cruz, A. Delis, & G. Huang (Eds.), *Web information systems engineering. lecture notes in computer science* (pp. 1–15). Springer Berlin Heidelberg.

Jonassen, S., Cambazoglu, B. B., & Silvestri, F. (2012). Prefetching query results and its impact on search engines. In *Proceedings of the 35th international ACM SIGIR conference on research and development in information retrieval* (pp. 631–640).

Kullback, S. (1997). *Information theory and statistics (dover books on mathematics)*. Dover Publications Inc.

Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics, 22*, 79–86.

Lempel, R., & Moran, S. (2004). Optimizing result prefetching in web search engines with segmented indices. *ACM Transactions on Internet Technologies, 4*(1), 31–59.

Li, H., Lee, W.-C., Sivasubramaniam, A., & Giles, C. L. (2007). A hybrid cache and prefetch mechanism for scientific literature search engines. In *Proceedings of the 7th international conference on web engineering* (pp. 121–136). ICWE'07. Springer-Verlag, Berlin, Heidelberg.

Long, X., & Suel, T. (2005). Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international conference on world wide web* (pp. 257–266). ACM, New York, NY, USA.

Ma, H., Liu, W., Wei, B., Shi, L., Bao, X., Wang, L., & Wang, B. (2014). PAAP: Prefetch-aware admission policies for query results cache in web search engines. In *Proceedings of the 37th international ACM SIGIR conference on research and development in information retrieval, SIGIR'14. gold coast, queensland, australia* (pp. 983–986).

Marin, M., Gil-Costa, V., & Gomez-Pantoja, C. (2010). New caching techniques for web search engines. In *Proceedings of the 19th ACM international symposium on high performance distributed computing* (pp. 215–226). HPDC'10. ACM, New York, NY, USA.

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

Ozcan, R., Altingovde, I. S., Cambazoglu, B. B., Junqueira, F. P., & Ulusoy, O. (2012). A five-level static cache architecture for web search engines. *Information Processing & Management, 48*(5), 828–840.

Ozcan, R., Altingovde, I. S., Cambazoglu, B. B., & Ulusoy, O. (2013). Second chance: A hybrid approach for dynamic result caching and prefetching in search engines. *ACM Transactions of the Web, 8*(1), 3:1–3:22.

Ozcan, R., Altingovde, I. S., & Ulusoy, O. (2008). Static query result caching revisited. In *Proceedings of the 17th international conference on world wide web* (pp. 1169–1170). ACM, New York, NY, USA.

Pass, G., Chawdhury, A., & Torgesson, C. (2006). A picture of search. *The first international conference on scalable information systems, hong hong, 20006.*

Prokhorenkova, L. O., Ustinovskiy, Y., Samosvat, E., Lefortier, D., & Serdyukov, P. M. (2014). Adaptive caching of fresh web search results. In *Advances in information retrieval: 37th european conference on IR research, ECIR 2015, vienna, austria, march 29 - april 2, 2015* (pp. 110–122).

Puppin, D., Silvestri, F., Perego, R., & Baeza-Yates, R. (2010). Tuning the capacity of search engines: Load-driven routing and incremental caching to reduce and balance the load. *ACM Transactions on Information Systems, 28*(2), 5:1–5:36.

Saraiva, P. C., Silva de Moura, E., Ziviani, N., Meira, W., Fonseca, R., & Riberio-Neto, B. (2001). Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th annual international ACM SIGIR conference on research and development in information retrieval* (pp. 51–58). SIGIR '01. ACM, New York, NY, USA.

Sazoglu, F. B., Ulusoy, O., Altingovde, I. S., Ozcan, R., & Cambazoglu, B. B. (2015). Propagating expiration decisions in a search engine result cache. In *Proceedings of the 24th international conference on world wide web* (pp. 107–108). WWW'15. ACM, New York, NY, USA.

Skobeltsyn, G., Junqueira, F., Plachouras, V., & Baeza-Yates, R. (2008). Resin: A combination of results caching and index pruning for high-performance web search engines. In *Proceedings of the 31st annual international ACM SIGIR conference on research and development in information retrieval* (pp. 131–138).

Tolosa, G., Becchetti, L., Feuerstein, E., & Marchetti-Spaccalema, A. (2014). Performance improvements for search systems using an integrated cache of lists+intersections. In *String processing and information retrieval lecture notes in computer science, edition: vol. 8799* (pp. 227–235). Springer International Publishing.

Ye, J., Chow, J.-H., Chen, J., & Zheng, Z. (2009). Stochastic gradient boosted distributed decision trees. In *Proceedings of the 18th ACM conference on information and knowledge management* (pp. 2061–2064). CIKM '09. ACM, New York, NY, USA.

Zhou, W., Li, R., Dong, X., Xu, Z., & Xiao, W. (2015). An intersection cache based on frequent itemset mining in large scale search engines. In *3rd IEEE workshop on hot topics in web systems and technologies, hotweb. november 12, - 13.* (pp. 19–24).