Adaptive Time-to-Live Strategies for Query Result Caching in Web Search Engines

Sadiye Alici[1], Ismail Sengor Altingovde[2], Rifat Ozcan[1], B. Barla Cambazoglu[3], and Özgür Ulusoy[1]

[1] Computer Engineering Department, Bilkent University, Ankara, Turkey
{sadiye,rozcan,oulusoy}@cs.bilkent.edu.tr
[2] L3S Research Center, Hanover, Germany
altingovde@l3s.de
[3] Yahoo! Research, Barcelona, Spain
barla@yahoo-inc.com

Abstract. An important research problem that has recently started to receive attention is the freshness issue in search engine result caches. In the current techniques in literature, the cached search result pages are associated with a fixed time-to-live (TTL) value in order to bound the staleness of search results presented to the users, potentially as part of a more complex cache refresh or invalidation mechanism. In this paper, we propose techniques where the TTL values are set in an adaptive manner, on a per-query basis. Our results show that the proposed techniques reduce the fraction of stale results served by the cache and also decrease the fraction of redundant query evaluations on the search engine backend compared to a strategy using a fixed TTL value for all queries.

Keywords: Search engines, result cache, freshness, time-to-live.

1 Introduction

Search engines cache the results of frequently and/or recently issued queries to cope with large user query traffic volumes and to meet demanding query response time constraints [2]. Due to the strong power-law distribution in query streams, using a result cache significantly decreases the query volume hitting the backend index servers, leading to considerable resource savings. In practice, a typical entry in a result cache contains a query string, some data (e.g., URLs, snippets) about the top k search results of the query, and potentially some auxiliary data (e.g., timestamps). In the context of search systems using caches with limited capacity, the research issues involve cache admission [3], eviction [9], and prefetching [12], all of which are rather well explored in the literature. The availability of cheap on-disk storage devices, however, allows large-scale web search engines to cache practically all previous query results [7]. This renders the previously mentioned research problems relatively less interesting while creating a new challenge: maintaining the freshness of cached query results without incurring too much redundant computation overhead on the backend search system.

The main reason behind the result cache freshness problem in search engines is the fact that the documents in the index of the search engine are continuously modified due to new document additions, deletions, and content updates [4]. Hence, certain entries in the result cache face the risk of becoming stale as the time passes, i.e., some cached results no longer reflect the up-to-date top k results that would be returned by the backend search system. Serving stale search results may have a negative impact on the user satisfaction. On the other hand, bypassing the cache and evaluating the queries on the backend, thus guaranteeing the freshness of query results, increases the computational costs. Consequently, the research problem is to come up with techniques that can accurately identify stale cache entries without incurring much redundant query processing overhead.

In literature, there are two lines of techniques for maintaining the freshness of a result cache. In the first line of techniques, cache entries are invalidated, i.e., they are treated as stale, based on some feedback obtained from the indexer [1,4,5]. The indexer can generate this feedback based on some update statistics associated with terms and documents. Although such techniques are effective in improving the freshness, their implementation is non-trivial and they incur additional computational costs. In the second line of techniques, cache entries are proactively refreshed depending on features such as query frequency and result age [7]. Those techniques are easy to implement, but relatively less effective.

All solutions proposed so far are coupled with a time-to-live (TTL) mechanism, where each result entry is associated with a TTL value that indicates the time point after which the result entry will be considered stale. In practice, the TTL value has to be carefully set. While a too large value may cause serving stale results to the users, using a too small value may diminish the efficiency gains of having a result cache. To the best of our knowledge, the potential benefits of assigning adaptive TTL values tailored according to the features of queries and their results are left unexplored. Given the large space of user intents and web results, however, it is reasonable to assume that the results of different queries will remain fresh for different time periods. For instance, while the result of the query "stem cell research" can remain fresh for a couple of days, the results of a news-related query (e.g., "Libya war") may change within a couple of hours or even minutes. Moreover, the freshness of queries may show temporal variation. For example, while the results of the query "champions league" usually change on a daily basis, the changes may be every minute during the matches.

In this paper, we investigate adaptive TTL strategies that aim to assign a separate TTL value to each query. We evaluate a number of parameterized functions and also build a machine learning model to assign the TTL values to queries. Our machine learning model exploits a number of query- and result-specific features that help predicting the best TTL values. The experimental results show that the adaptive TTL strategies outperform the fixed TTL strategy in terms of both result freshness and redundant query processing cost metrics.

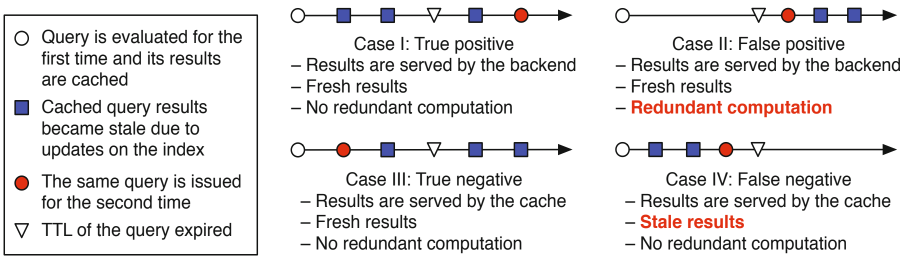

Query is evaluated for the first time and its results are cached
Cached query results became stale due to updates on the index
The same query is issued for the second time
TTL of the query expired
Case I: True positive
– Results are served by the backend
– Fresh results
– No redundant computation
Case II: False positive
– Results are served by the backend
– Fresh results
– Redundant computation
Case III: True negative
– Results are served by the cache
– Fresh results
– No redundant computation
Case IV: False negative
– Results are served by the cache
– Stale results
– No redundant computation

Fig. 1. Potential cases when performing a result cache lookup for a query

2 Fixed TTL Strategy

The standard technique in previous works is to associate each cache entry with a fixed TTL value [1,4,5,7]. These works assume a caching system such that the results of a query q are cached when the query is first seen at time $t(q)$ and its expiration time is set to $t(q)+T$, where T is a TTL value that is identical for all queries. If the query is re-issued before its expiration time $t(q)+T$, the results are readily served by the cache. Otherwise, the cached results are considered to be stale and the query is re-evaluated at the backend. In general, depending on whether the query results are really stale or not and whether the query evaluation is redundant or not, there are four possible outcomes of this decision (Fig. 1):

– Case I: The query is received after its TTL value has expired and while the cached results are stale. Since the TTL value is expired, the results are served by the backend. The backend computation is not redundant because the newly computed results are more fresh than those in the cache.
– Case II: The query is received after its TTL value has expired and while the cached results are not stale. Since the TTL value is expired, the results are served by the backend. The backend computation is redundant because the newly computed results are identical to those in the cache.
– Case III: The query is received before its TTL value has expired and while the cached results are not stale. The fresh results in the cache are served to the user without creating any query processing overhead on the backend.
– Case IV: The query is received before its TTL value has expired and while the cached results are stale. The stale results in the cache are served to the user, potentially degrading the user satisfaction.

3 Adaptive TTL Strategies

The TTL mechanism leads to an interesting trade-off between the stale query traffic ratio and the redundant query evaluation cost. In general, large TTL values lead to more stale results while small TTL values lead to higher processing

Table 1. Increment functions and parameters (T' is the current TTL, c is a parameter)

Type	$F(.)$	Parameter values
Linear	$c \times T'$	1/10, 1/5, 1/2
Polynomial	$(T')^c$	−3, −2, −1, −1/2, 1/2
Exponential	$c^{T'}$	1/2, 1, 2

overhead. The techniques we will discuss in this section aim to select the TTL value, adaptively, on a per query basis, trying to increase the freshness of served results and reduce the query processing overhead. We investigate three alternative strategies: average TTL, incremental TTL, and machine-learned TTL.

3.1 Average TTL

This is a simple approach that observes the past update frequency $f(q)$ for the top k result set of a query over a time interval of I units of time and sets the TTL value of q to $I/f(q)$. For instance, if the top k result set of a query is updated 3 times within a time interval of 30 days, the TTL of the query is set to 10. Since this strategy simply computes an average without taking into account the distribution of result updates in time, it fails to capture bursty update periods.

3.2 Incremental TTL

A simple yet effective way of assigning adaptive TTL values to queries is to adjust the TTL value based on the current TTL value. That is, each time a result entry having an expired TTL value is requested, we compare the newly computed results with those in the cache to decide whether the query results are stale or not. In this work, we adopt a conservative approach and declare the results of a query stale unless all URLs in the cached results and the newly computed results as well as their ranks are the same.

Given the dynamic and rapidly growing nature of the Web, it is reasonable to assume that the query results will be frequently updated. Moreover, in many cases, bursty updates are likely due to spontaneous events. Taking these observations into account, we assign the TTL value of a query as follows:

i) If the cached results are decided to be stale, we immediately set the new TTL value of the query to the lowest possible TTL value $T_{\min}$. The rationale behind decreasing the TTL value is to avoid missing a bursty update period.
ii) If the cached results are decided to be fresh, we use this as an evidence to increase the TTL value of the query according to the output of some increment function $F(.)$.[1] The function $F(.)$ takes as input the current TTL value T' of the query and outputs an increment value $F(T')$ to be added to the current TTL value, i.e., the new TTL value T is computed by the

[1] In this study, we evaluate a wide range of increment functions (see Table 1).

Table 2. Features used by the machine learning model

Feature	Description
`QueryTermCount`	Number of terms in the query
`QueryFreqeuency`	Number of times query appears in the log
`ResultCount`	Number of query results
`ReplacedDocsInResult`	Number of new query results
`RerankedDocsInResult`	Number of query results whose ranks have changed
`Web2ResultCount`	Number of query results from Web 2.0 sites
`NewsResultCount`	Number of query results from news sites

formula $T = T' + F(T')$. The TTL value of a query can be increased until a maximum allowed TTL value $T_{\max}$ is reached. This threshold serves as an upper bound to avoid assigning very high TTL values to queries.

3.3 Machine-Learned TTL

When assigning a new TTL value to a query, the strategy of the previous section makes use of only the current TTL value of the query. To make the same decision, we also build a machine learning model using several features extracted from a training query log. Some of our features aim to capture basic query characteristics such as query length and frequency. Some other features try to capture the tendency of the query results to change. For example, we compare the newly computed top k results of a query with the cached results and determine the number of new documents in the current results or the number of documents whose ranks have changed. The expectation is that the amount of change in the results will provide clues about the freshness period of the query results. We also use features reflecting the type of query results (e.g., news or social media pages). We envision that the number of results from Web 2.0 or news domains may also serve as a feature since queries whose results contain such pages are more likely to become stale. The features we use in the model are given in Table 2.

To train the machine learning model, we observe the updates on the result sets of some training queries over a time interval $[t_1, \ldots, t_K]$, where t_i represents a discrete time point. At each update of the result set of a query q, we create a training instance with the features given in Table 2. The target TTL value we learn is the time interval during which the query results remain fresh starting from the current update time. As an example, suppose that the results of a query are updated twice, at time points t_i and t_j during the training period. For this query, we compute a feature vector at time t_i and set the target TTL value as $t_j - t_i$. For the update at time t_j, we again compute features, but this time we set the target TTL value to $t_K - t_j$, i.e., we assume an implicit update at the end of the training period. The feature vectors for test instances are created in a similar fashion, every time a cache entry with an expired TTL value is requested.

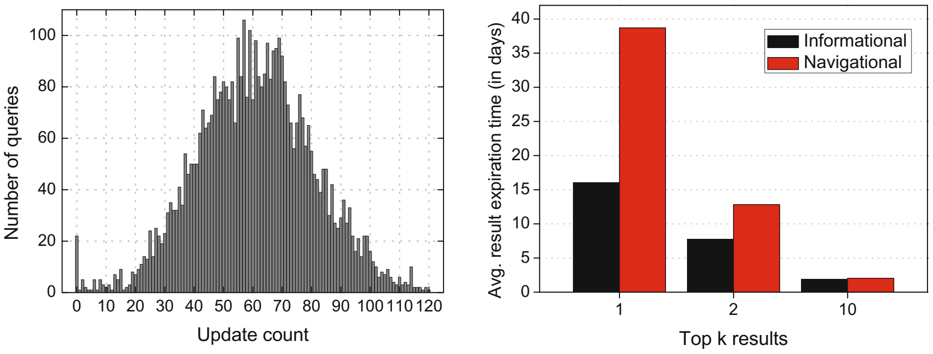


Fig. 2. Distribution of result updates during the evaluation period of 120 days (left) and the average freshness period (in days) for the top k results of informational and navigational queries (right)

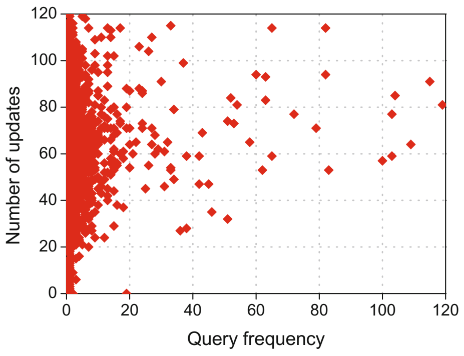
4 Characterization of Result Updates

To characterize the behavior of result updates, we use a randomly selected subset of 4,500 queries from the AOL query log [15]. For all queries, we obtain the top 10 results from the Yahoo! public search API for 120 days, from November 2010 to April 2011. As we mentioned before, we assume a strict definition for a result update: if any of the results or their order in the top k set changes with respect to the results of the previous day, we consider the results updated. In Section 5, we also investigate a case where this assumption is relaxed.

Our query log exhibits properties similar to earlier findings in terms of query length and frequency. The majority of queries have length between 1 and 4. The average query length is 2.9. Query frequencies follow a power-law distribution as shown in previous studies. To save space, we do not provide the related plots.

Figure 2 (left) shows the distribution of result updates. In our data, we observe that the query results become stale around 2.5 days, on average. This finding conforms with the finding in [11]. That study argues that the high rate of changes in query results might be due to the instability of indexing and/or ranking mechanisms used by the search engine. Nevertheless, it is not possible to know whether a change in the result set of a query is an artifact of the search system or due to updates in the underlying data used for ranking (e.g., IDF values of terms, web graph). Therefore, for our purposes, whenever the top k result set of a query differs from the result set of the previous day, we consider this as a proper update of the results that should be detected by an ideal invalidation mechanism (assuming that the same query is issued in both days).

A useful feature that can be determined from the query and its result set is the query intent (e.g., informational, navigational, or transactional [6]). In particular, while assigning adaptive TTL values, it seems quite promising to exploit navigational queries, which aim to find a certain page on the Web, as



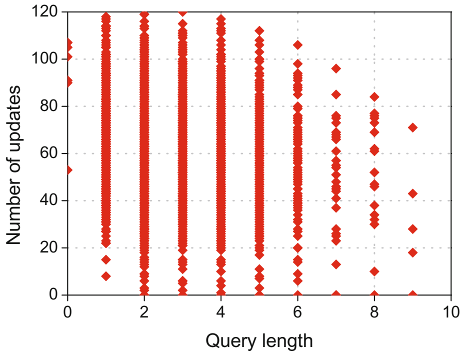


Fig. 3. Number of result updates versus query frequency (left) and the average number of result updates versus query length (right) during the evaluation period of 120 days

the URLs of frequently accessed web sites are more likely to be stable over time. To this end, we obtain a sample of 400 navigational and 400 informational queries from our query set. These queries are first filtered using the automated techniques in [14] and are further verified in a post-processing step by human judges. Figure 2 (right) shows that when we apply our conservative definition of result update, it turns out that the top 10 results of a navigational query change as frequently as the results of an informational query. This finding is different from what is reported in [11], where the stability is defined based on the result relevance. On the other hand, we also find that the top 1 or top 2 results of navigational queries have a freshness period that is almost twice longer than those of informational queries. This is important since earlier works report that answers of the navigational queries are usually returned at the highest ranks [13]. Therefore, the technique discussed in Section 3.3 can be applied using the update history of only the top-ranked results for navigational queries, which can be effectively identified by the search engines. We consider this as an obvious extension of our methods and do not make use of query intent in this paper.

Next, we investigate the relationship of certain features extracted from query results with the update frequency of results. According to Fig. 3 (left), the results of queries with higher frequencies are also updated more frequently, but the relative difference with respect to less frequent queries is rather low. A similar observation can also be made in Fig. 3 (right), according to which there seems to be no correlation between the update frequency and query length (especially for queries up to seven terms). This might be due to the observation that both head (shorter) and tail (longer) queries involve popular terms [16], whose posting lists may change more often. Subsequently, the query results can also change frequently, regardless of the query frequency or length.

An analysis similar to ours is provided in [11], which investigates the stability of query results with respect to the query frequency and length. Interestingly, the findings of that work (see Fig. 8 in [11]) are contradictory to ours in that they find less frequent and longer queries to be more unstable. However, their measure of instability is based on the variance of the human-judged quality of results (measured by NDCG) computed among the top-5 results over the entire evaluation period. In contrast, herein, we consider whether the top 10 results of a query change between any two consecutive days disregarding the relevance of results, which is unknown when assigning a TTL value to a query.

5 Experiments

5.1 Setup

We follow the simulation setup described in [1,4]. We assume that all 4,500 queries in our query set are submitted to the search engine every day throughout 120 days. We further assume an infinitely large cache [7] so that the performance does not depend on the cache size or eviction algorithms. On the first day of the 120-day simulation period, all query results are stored in the cache. For the following days, we either execute the query and replace the result in the cache (with the ground truth result of that particular day as obtained from the Yahoo! API) or return the cached results to the user. At the end of each day, we compute two metrics: the stale traffic (ST) ratio and the false positive (FP) ratio [4]. The ST ratio is the fraction of stale query results served by the system and the FP ratio is the fraction of redundant query evaluations. For a given day, we compare the returned result to the ground truth to see whether a stale result is returned or a redundant evaluation is done, and update the respective metrics. The reported values are over 120 days and all queries. Unless stated otherwise, query results that differ from the ground truth in terms of URLs or their ranks are considered to be stale, i.e., we impose a strict result equivalence requirement. For the machine-learned TTL strategy, we build a linear regression model using Weka [10].[2] To determine the number of results from the news and social Web 2.0 sites, we manually compile lists from various Web resources (6,349 and 477 web sites, respectively). For the incremental TTL strategy, after some tuning, we set $T_{\min}$ to 1 (i.e., no caching) and $T_{\max}$ to 5.

5.2 Results

In Fig. 4, we plot the ST ratio versus the FP ratio for fixed TTL values between 1 and 5 days, where a TTL value of 1 means that each query is executed everyday, i.e., no caching. The plot also includes the performance of the incremental TTL strategy described in Section 3.2. The results show that the incremental TTL strategy achieves lower ST and FP ratios. For instance, for the FP ratios between 0.2 and 0.3, the incremental TTL strategy achieves almost half of the ST ratio that could be obtained using a fixed TTL strategy. Note that, in an additional experiment, we randomly set the TTL value to one or two, each time the TTL of a query expires. We found that this strategy yields a point that is on the line that connects the TTL values one and two in Fig. 4. Therefore, the performance gains of the incremental TTL strategy shown in Fig. 4 are realistic.

[2] The generated regression model assigns a non-zero weight to all features in Table 2.

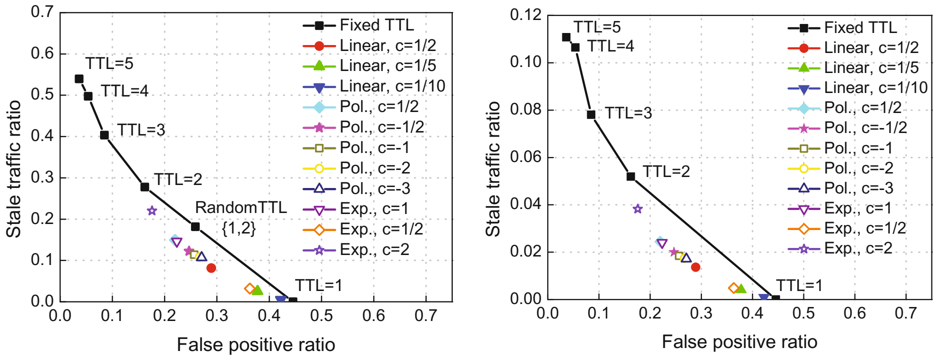


Fig. 4. Stale traffic ratio versus false positive ratio for incremental TTL strategies with strict result equivalence check (left) and Jaccard-based equivalence check (right)

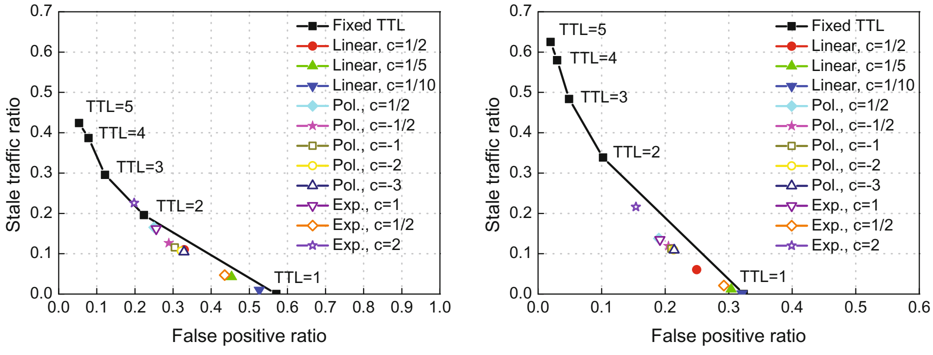


Fig. 5. Stale traffic ratio versus false positive ratio for queries whose results are infrequently (left) or frequently (right) updated

Figure 4 (left) also shows that the actual ST ratios are rather high, in comparison to previous works [1,4]. This is due to the high update frequency of web search results as shown in the previous section. On the other hand, it is possible to obtain lower ST ratios by relaxing our strict equivalence requirement while comparing cached results with the ground truth. In Fig. 4 (right), we report the ST ratio based on the Jaccard similarity metric, i.e., if the cached results are not exactly the same as the ground truth, we compute the staleness of the served result by obtaining the Jaccard similarity of these two result sets and subtracting this latter value from one. The resulting staleness value is then used to compute the ST ratio. Figure 4 (right) shows that our adaptive strategies perform better also in this case while all absolute ST ratios drop. This latter observation implies that most of the changes in the query results may be minor, e.g., two documents may swap places in the top 10 result list. In Figs. 5 (left) and 5 (right), we show the performance for queries that are updated less often or more often than the average update frequency, respectively. The increment functions are useful in both cases, but the gains are larger in the latter scenario.

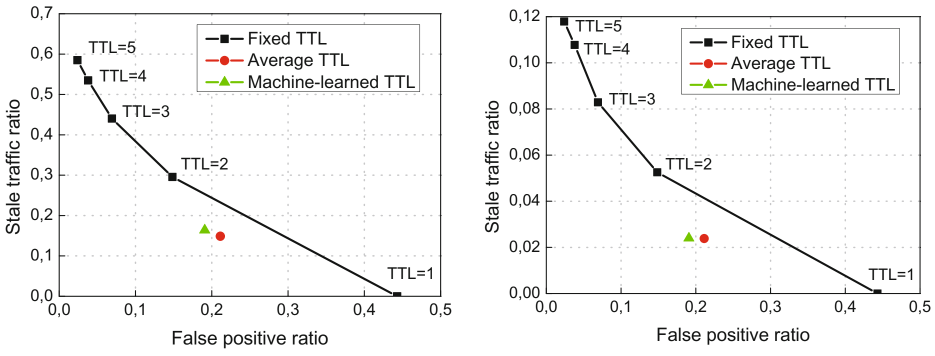


Fig. 6. Stale traffic ratio versus false positive ratio for the average TTL and machine-learned TTL strategies with strict (left) and Jaccard-based (right) equivalence check

In Fig. 6 (left), we report the performance of the adaptive TTL strategies based on simple averaging and machine learning (ML). The average TTL values are computed over the first 60 days and evaluated over the second 60 days of the evaluation period, for all queries. For the ML-based TTL strategy, we used five-fold cross validation. In each fold, 80% of the queries are used as the training set and only the first 60 days of these queries are used for training. The remaining 20% of queries are evaluated for the second 60 days. Therefore, we guarantee that queries in the training and test sets are content-wise and temporally distinct. Figure 6 (right) reports similar results, using the Jaccard similarity of the returned results and ground truth while computing the ST ratio. Although the results in Fig. 6 indicate that both the average TTL and ML-based TTL strategies can improve the performance of the fixed TTL strategy, the gains, however, are very close to the gains attained by the incremental TTL strategy. Hence, in practice, the incremental TTL strategy may be preferable since it does not require a training phase.

6 Related Work

The main reason that leads to updates in query results is the changes in the web content. A large-scale study of web change is conducted by Fetterly et al. [8], where the authors monitor the weekly change of around 150 M web pages for a period of 11 weeks. It is observed that almost 3% of all web documents change weekly and the pages from the .com domain change more frequently than the pages in the .edu or .gov domains.

Previous works that aim to maintain the freshness of a result cache adopt the practice of associating fixed TTL values with query results. In [7], TTL is used as the primary mechanism to prevent stale results. The TTL scheme is further augmented with an intelligent refresh strategy which uses the idle cycles of the backend to precompute the expired query results. More recently, other solutions that assume an incremental index update policy are proposed in [4]. In this case, each time the index is modified, the documents that are updated are also compared to the cached queries and the results that can potentially change are marked as invalid. Again, the query results are associated with TTL values to prevent having too stale results in the cache. Finally, in [1], a timestamp-based invalidation framework is proposed on top of an incrementally updated index.

In the literature, the idea of using an adaptive TTL is explored in the context of maintaining the coherency of virtual data warehouses that consume data from different web pages [17]. In that work, static, semi-static, dynamic, and adaptive TTL mechanisms are proposed. The static approach uses a fixed TTL value as in our baseline. The semi-static TTL approach sets the TTL value based on a previously observed maximum update rate. The dynamic TTL approach relies on the most recent changes. The adaptive TTL approach is a combination of the semi-static and dynamic approaches using min() and max() functions. It is shown that the adaptive TTL approach outperforms the other approaches for this problem domain. In a recent work [5], the cache invalidation predictor module proposed in [4] is augmented with a TTL-like approach, called virtual clock. This approach invalidates query results based on some events such as the number of times the result is served from the cache and the number of times new documents match the query but does not alter the top-k result set. Even though this approach is different from the fixed TTL approach, it is not adaptive because it sets a fixed virtual clock value independent of the query.

Kim and Carvalho [11] analyze the updates in web search results from an instability perspective. They monitor the results of 1,000 queries in three major search engines and observe that the top-10 result sets change for almost 90% of queries in ten days. They analyze the reasons for these changes and evaluate the degree of instability using different measures, such as the overlap between results and the change in the NDCG measure.

7 Conclusion

We proposed three types of adaptive TTL strategies to improve the performance over a fixed TTL approach in terms of stale traffic and false positive ratios. The experiments showed that the adaptive approaches achieve better results than the baseline. The future work directions involve developing a formal model of result updates over time and investigating more accurate adaptive TTL strategies based on this model. Another promising direction is to combine the adaptive TTL strategies with the relatively sophisticated approaches [1,4]. Yet another direction is using the clicks on results as relevance judgments and deciding on the staleness only when the clicked URLs change in a query result set.

Acknowledgments. This work is partially supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under grant no. 110E135, EU FP7 projects LivingKnowledge (contract no. 231126) and COAST (contract no. 248036).

References

1. Alici, S., Altingovde, I.S., Ozcan, R., Cambazoglu, B.B., Ulusoy, O.: Timestamp-based result cache invalidation for web search engines. In: Proc. 34th Int'l ACM SIGIR Conf. Research and Development in Information Retrieval, pp. 973–982 (2011)
2. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The impact of caching on search engines. In: Proc. 30th Int'l ACM SIGIR Conf. Research and Development in Information Retrieval, pp. 183–190 (2007)
3. Baeza-Yate, R., Junqueira, F.P., Plachouras, V., Witschel, H.F.: Admission Policies for Caches of Search Engine Results. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 74–85. Springer, Heidelberg (2007)
4. Blanco, R., Bortnikov, E., Junqueira, F., Lempel, R., Telloli, L., Zaragoza, H.: Caching search engine results over incremental indices. In: Proc. 33rd Int'l ACM SIGIR Conf. Research and Development in Information Retrieval, pp. 82–89 (2010)
5. Bortnikov, E., Lempel, R., Vornovitsky, K.: Caching for Realtime Search. In: Clough, P., Foley, C., Gurrin, C., Jones, G.J.F., Kraaij, W., Lee, H., Mudoch, V. (eds.) ECIR 2011. LNCS, vol. 6611, pp. 104–116. Springer, Heidelberg (2011)
6. Broder, A.: A taxonomy of web search. SIGIR Forum 36(2), 3–10 (2002)
7. Cambazoglu, B.B., Junqueira, F.P., Plachouras, V., Banachowski, S., Cui, B., Lim, S., Bridge, B.: A refreshing perspective of search engine caching. In: Proc. 19th Int'l Conf. World Wide Web, pp. 181–190 (2010)
8. Fetterly, D., Manasse, M., Najork, M., Wiener, J.: A large-scale study of the evolution of web pages. In: Proc. 12th Int'l Conf. World Wide Web, pp. 669–678 (2003)
9. Gan, Q., Suel, T.: Improved techniques for result caching in web search engines. In: Proc. 18th Int'l Conf. World Wide Web, pp. 431–440 (2009)
10. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations 11(1), 10–18 (2009)
11. Kim, J., Carvalho, V.R.: An Analysis of Time-Instability in Web Search Results. In: Clough, P., Foley, C., Gurrin, C., Jones, G.J.F., Kraaij, W., Lee, H., Mudoch, V. (eds.) ECIR 2011. LNCS, vol. 6611, pp. 466–478. Springer, Heidelberg (2011)
12. Lempel, R., Moran, S.: Predictive caching and prefetching of query results in search engines. In: Proc. 12th Int'l Conf. World Wide Web, pp. 19–28 (2003)
13. Liu, Y., Zhang, M., Ru, L., Ma, S.: Automatic Query Type Identification Based on Click Through Information. In: Ng, H.T., Leong, M.-K., Kan, M.-Y., Ji, D. (eds.) AIRS 2006. LNCS, vol. 4182, pp. 593–600. Springer, Heidelberg (2006)
14. Ozcan, R., Altingovde, I.S., Ulusoy, O.: Exploiting navigational queries for result presentation and caching in web search engines. J. Am. Soc. Inf. Sci. Technol. 62(4), 714–726 (2011)
15. Pass, G., Chowdhury, A., Torgeson, C.: A picture of search. In: Proc. 1st Int'l Conf. Scalable Information Systems (2006)
16. Skobeltsyn, G., Junqueira, F., Plachouras, V., Baeza-Yates, R.: ResIn: a combination of results caching and index pruning for high-performance web search engines. In: Proc. 31st Int'l ACM SIGIR Conf. Research and Development in Information Retrieval, pp. 131–138 (2008)
17. Srinivasan, R., Liang, C., Ramamritham, K.: Maintaining temporal coherency of virtual data warehouses. In: Proc. IEEE Real-Time Systems Symposium, pp. 60–70 (1998)