EFFICIENT RESULT CACHING MECHANISMS IN SEARCH ENGINES

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Fethi Burak Sazoğlu

September, 2014

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Özgür Ulusoy (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. İsmail Sengör Altıngövde (Co-advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Uğur Güdükbay

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ahmet Coşar

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

EFFICIENT RESULT CACHING MECHANISMS IN SEARCH ENGINES

Fethi Burak Sazoğlu
M.S. in Computer Engineering
Supervisors: Prof. Dr. Özgür Ulusoy and Asst. Prof. Dr. İsmail Sengör Altıngövde
September, 2014

The performance of a search engine depends on its components such as crawler, indexer and processor. The query latency, accuracy and recency of the results play crucial role in determining the performance. High performance can be provided with powerful hardware in the data center, but keeping the operational costs restrained is mandatory for search engines for commercial durability. This thesis focuses on techniques to boost the performance of search engines by means of reducing both the number of queries issued to the backend and the cost to process a query stream. This can be accomplished by taking advantage of the temporal locality of the queries. Caching the result for a recently issued query removes the need to reprocess this query when it is issued again by the same or different user. Therefore, deploying query result cache decreases the load on the resources of the search engine which increases the processing power. The main objective of this thesis is to improve search engine performance by enhancing productivity of result cache. This is done by endeavoring to maximize the cache hit rate and minimizing the processing cost by using the per query statistics such as frequency, timestamp and cost. While providing high hit rates and low processing costs improves performance, the freshness of the queries in the cache has to be considered as well for user satisfaction. Therefore, a variety of techniques are examined in this thesis to bound the staleness of cache results without blasting the backend with refresh queries. The offered techniques are demonstrated to be efficient by using real query log data from a commercial search engine.

Keywords: Query result Caching, web search engines, financial cost, time-to-live.

ÖZET

ARAMA MOTORLARI İÇİN VERİMLİ ÖNBELLEKLEME MEKANIZMALARI

Fethi Burak Sazoğlu
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticileri: Prof. Dr. Özgur Ulusoy ve Yrd. Doç. Dr. İsmail Sengör Altıngövde
Eylül, 2014

Arama motorlarının performansı indeksleyici, arka uç işlemcileri ve belge toplama botları gibi parçalarının performansına bağlıdır. Sorguların gecikme süresi, sorgu sonuçlarının doğruluğu ve güncellikleri performansı belirlemede önemli rol oynar. Arama motorlarında performans, güçlü donanımlarla sağlanabilir, fakat arama motorlarının ticari devamlılığı açısından operasyonel giderlerin kontrol altında tutulması gerekir. Bu nedenle, bu tez arama motorlarının performansını arka uca giden sorgu sayısını ve bir sorgu akımının sonuçlarının hesaplama maliyetini azaltarak iyileştiren tekniklere odaklanır. Bu, sorgulardaki zamansal lokalite özelliğinden yararlanılarak sağlanabilir. Yakın zamanda verilen sorguların sonuçları önbelleklenerek, bu sorguların aynı veya farklı kullanıcılar tarafından tekrarlanması durumunda oluşacak tekrar hesaplama maliyeti ortadan kaldırılabilir. Dolayısıyla, sorgu sonuç önbelleği eldeki kaynaklardaki yükü azaltarak hesaplama güçlerini artırır. Bu tez temel olarak sonuç önbelleğinin üretkenliğini geliştirerek arama motorunun performansını yükseltmeyi amaçlar. Frekans, sorgu zamanı ve sorgu maliyeti gibi sorgu istatistikleri kullanılarak önbellek isabet oranını artırarak ve toplam maliyeti düşürerek bu amaca ulaşılabilir. Arama motorlarının verimliliğini artırırken önbellekteki sonuçların taze tutulması kullanıcı memnuniyeti açısından önemlidir; bundan dolayı arama motorları tarafından gözardı edilemez. Sonuçların tazeliğini sınırlandırmak için çeşitli teknikler önerilerek, önbelleğin performansını düşürmeden bu çalışmada verimli çözümler bulunmaya çalışılmıştır.

Anahtar sözcükler: Sorgu sonuçlarının önbelleklenmesi, web arama motorları, finansal maliyet, sorgu yaşam süresi.

Acknowledgement

I would like to express my deepest gratitude to my advisor Prof. Dr. Özgür Ulusoy for his invaluable guidance, support and considerateness during this research. I also would like to thank my co-advisor Asst. Prof. Dr. İsmail Sengör Altıngövde for his guidance, encouragement and effort during the preparation of this thesis.

I would like to thank my committee members Prof. Dr. Uğur Güdükbay and Prof. Dr. Ahmet Coşar for their feedbacks and comments on the thesis.

I would like to thank Asst. Prof. Dr. Rıfat Özcan for his support and guidance during this research and Dr. Berkant Barla Cambazoğlu for his advice and feedback during my intership with him. I would like to acknowledge TÜBİTAK for their financial support within National Scholarship Programme for M.Sc. Students.

Moreover, I would like to thank my friends Ahmet Çınar and Mehmet Güvercin for their friendship and support. Finally, I would like to express my gratitude to my parents and my brothers Zahid and Erdem for their constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the number of websites increases, finding content across web relies more upon commercial search engines, which has the task to collect information about websites and present a portion of them to its users concerning their search terms. The increasing number of websites presents challenges to the search engines such as efficient and effective allocation of their resources for query processing while ensuring user satisfaction. Users are content as long as they reach fast and accurate search results. Since commercial search engines relies on user data to constantly improve their search algorithms, number of users a search engine has is essential. However, maintaining a web search engine includes tasks such as crawling, parsing, indexing of the web pages, partitioning them into clusters and populating the caches which have a vital role in the success of the query search results and eventually the search engine.

Search engine caches are one of the crucial components as they mitigate the burden on search engine backend by serving the readily available results in the cache without calculating the result in the backend from scratch. Therefore, search engines endeavor to maximize the search queries served from the cache. However, this comes with the overhead of keeping the cache fresh as the web pages are not static, updated frequently. In order for search engines to provide up to date results, they are required to reflect these changes in their indices to the result caches. While too much update of the entries in the caches may render

them redundant, very little update may degrade user satisfaction by serving them stale results. Thus, it is crucial to optimize this process with respect to hardware and cache algorithms. With the cheap production costs of necessary hardware, cache size is considered insignificant and assumed to be infinite. Recent research has focused more on cache algorithms that maximizes served fresh results from cache and minimizing unnecessary processing of search results at the backend rather than giving them from the cache.

Caching policies take care of cache admission and eviction operations for finite caches. Prefetching of query results and cache invalidation mechanisms are deployed by search engines for keeping the query results fresh. Cache invalidation algorithms determine possible cache entries which are stale and invalidate them so that when a hit occurs in an invalidated entry, the result is served from the backend and the corresponding cache entry is updated accordingly. The basic cache invalidation algorithm is time-to-live (TTL). It is a query-agnostic algorithm which sets an upper bound on the staleness of query results. The first part of our thesis is devoted to examine this strategy by combining them with different TTL mechanisms.

Power consumption of the commercial search engines constitutes another vital component of the companies. These search engines deploy massive data centers in order to handle searches coming from all over the world with minimum latency. Search engine result caching can also lower this power consumption by eliminating the need to reprocess the search queries and providing previously available results. In the second part of our thesis, we take this notion one step further and consider not only the power consumption but also the electricity prices when conducting caching operations. This work utilizes hourly available electricity prices to alter caching policies accordingly and lower the electricity bills consequently.

The rest of this thesis is organized as follows. In Chapter 2 we provide the related work about search engine caching and give information about the work done so far, and the motivations for these works. In Chapter 3, we investigate a number mechanisms for setting TTL values of the result caches. In Chapter 4, similarity based TTL approach is analyzed. In Chapter 5, we introduce a cost

metric for result caching and compare the effects of different caching mechanisms on the cost of processing a query log. Chapter 6 combines cost and freshness factors and interprets the tradeoff between these metrics. Chapter 7 concludes the thesis.

Chapter 2

Related Work

Caching of query result is not a new concept for search engines. They deploy caches for Web pages in proxies which are referred when the cached page is requested by the same user or different users. Same logic is applied to the search engine result caches which are mostly employed at the backend of the search engines. The main objective of a result cache is to take advantage of the temporal locality of queries. In this technique, the results of previously processed user queries are stored in a cache. The results for the subsequent occurrences of a query are served by this cache, eliminating the need to process the query and generate its results using the computational resources in the backend search system. This technique helps reducing the query processing workload incurred on the search engine while reducing the response time for queries whose results are cached.

Markatos [1] was the first to demonstrate the temporal locality of queries and suggest that caching query results can lead to improvements in the performance of search engines by eliminating the need to reprocess recently submitted queries and decreasing the time to return query results to users. This implies lower loads on the search engine backend and increased throughput of user queries. Markatos examines traces of EXCITE search engine and finds out that 20-30% of the queries consists of re-submitted queries. In this work, Markatos compares static and dynamic caching methods. In static cache, the cache is filled with most popular queries and the cached queries are fixed, in the sense that when

a new query is issued to the search engine the content does not change. If the cache contains the result for a query it is returned to the user, but in case of absence, the query is redirected to the backend and cache content is not altered. However, in dynamic caching issued queries affect the content of the cache. Query result is returned by the cache when the cache contains the query (hit), whereas absence of the query in the cache (miss) requires replacement of a victim query with this query in case the cache is full. Markatos experiments with replacement algorithms such as LRU, FBR, LRU/2 and SLRU. His experiments reveal that static cache outperforms dynamic cache for small cache sizes and dynamic cache gets better as the cache size increases.

The research on search engine result caching focused on what to cache and how to cache in order to improve the performance of the cache by boosting cache hit rates, which is the rate of items readily available in the cache when a request for this item is made. Fagni et al. alter how the items are cached by proposing a hybrid caching technique called Static and Dynamic Cache (SDC) [2]. The static part of the cache serves the results for most frequent queries, while the dynamic part steps in if the static part causes a cache miss. In case the dynamic part does not contain this query, the query is processed at the search backend and query admission and eviction policies are applied at the dynamic cache. Fagni et al. investigate temporal and spatial locality too. While the findings for temporal locality is parallel with [1], the query logs examined showed limited spatial locality, that is only a small portion of the users request two or more pages of query results. The experiments with different static cache size ratio shows that for some values, SDC outperforms only static and only dynamic caches. It achieves this by handling long-term popular queries with the static part, while serving short-term popular queries from its dynamic part.

There are other works in this area focusing on what to cache instead of how to cache, as in [1, 2]. This works consider caching the inverted lists alongside with the query results. Saraiva et al. [3] are the first to propose caching inverted lists in their two-level rank preserving caching architecture. The result cache resides between the client and query processor as in previous works. The query is first looked up in this cache and if the query is not in the result cache then

query processor computes the result for this query. The second level cache resides between the query processor and inverted lists database and holds the inverted lists for popular queries. Since the posting lists for popular terms is long, this work uses index pruning to restrict the space allocated to popular terms. The motivation to cache inverted lists is to utilize term locality which is asserted to be greater than query locality. The experiments compare the two-level cache with only query result cache and only inverted lists cache. The two-level cache attains query throughput 52% higher than the only invert lists case and 36% higher than the only query result case.

Baeza-Yates et al. [4] offer a three-level memory organization by capitalizing on real empirical data which is the query logs of a commercial search engine. The three-level memory organization consists of a cache of precomputed answers and part of an inverted index in the main memory and the remaining inverted lists in the secondary memory. They find an optimal split in the main memory for query results and part of the inverted lists by mathematically modeling the size of the inverted file and time to process a query using the size of the vocabulary and documents.

As an extension to [4], Long et al. propose another three-level cache in [5] which has the same result and list caching levels as well as another level for cache of posting list intersection. This cache level stores pairwise intersections of the index in the secondary memory (20% or 40 % of disk space of inverted index). This method searches for the available posting list intersection if the query result cache does not have the results. The last level which is the cache of popular inverted lists is applied if a miss occurs in intersection cache. They achieve 25% decrease in the CPU cost, which implies 33% increase in query throughput in their experimental setting that uses query logs from a search engine.

The works mentioned above assumes finite cache sizes, however today in practice, commercial web search engines deploy result caches that are large enough to store practically all query results computed in the past by the search engine [6]. Having a very large result cache renders basic caching techniques unnecessary (e.g., admission of queries [7], eviction of old cache entries [8], or prefetching of

successive result pages [9]). In case of very large result caches, the main problem is to preserve the freshness of cached query results. This is because commercial web search engine indexes are frequently updated as more recent snapshots of the Web are crawled and new pages are discovered. Eventually, the cached results of a query may differ from the actual results that can be obtained by evaluating the query on the current version of the index. Queries whose cached results are not consistent with those that would be provided by backend search system are referred to as stale queries. Identifying such queries and improving the overall freshness of a result cache is crucial because presenting stale query results to the users may have a negative effect on the user satisfaction for certain types of queries [10]. So far, two different lines of techniques addressed the freshness issue mentioned above: refreshing [6, 11] and invalidation [12, 13, 14, 15].

In the first set of techniques, cached query results that are predicted to be stale are refreshed by evaluating the associated queries at the search backend. The main motivation behind these techniques is to use the idle cycles of the backend search system to recompute the results of a selected set of supposedly stale queries. In general, the techniques based on refreshing are easy to implement as they do not require any interaction between the result cache and the backend search system when deciding which queries to refresh. On the other hand, identification of stale queries is a rather difficult task and this leads to an increase in the volume of queries whose results are redundantly recomputed at the backend with no positive impact on the freshness of the cache.

In the second set of techniques, the result cache is informed by the indexing system about the recent updates on the index. This information is then exploited at the cache side to identify cached query results that are potentially stale. More specifically, upon an update on the index, an invalidation module located in the backend system transfers these changes to the result caching module. This module then decides for every cached query result if the received changes on the index may render the results of the query stale, in which case the query is marked as invalid, i.e., considered to be not cached.

In all techniques mentioned above, the staleness decision for a query is given

via heuristics that do not yield perfect accuracy. Some stale queries may not be identified on time although their results have changed. Consequently, certain queries may remain in the cache for a long period with stale results. As a remedy to this problem, all of the above-mentioned techniques rely on a complementary mechanism known as time-to-live (TTL). In this mechanism, the validity of selected cache entries are expired based on a fixed criterion with the aim of setting an upper-bound on the possible staleness of a cache entry. In fact, on its own, this simple mechanism can provide freshness to a certain degree in the absence of more sophisticated refreshing or invalidation techniques.

The invalidation techniques use query statistics such as query cache age (the time passed since the query is cached), query cache frequency (number of hits since the query is cached) and number of clicks on the query. While this techniques use the statistics of a query to decide on its expiration, other queries or their posting lists can help evaluating this query too. [5] includes an additional level of caching in their three level caching architecture as mentioned above. In this intersection cache, the common documents of posting lists for term pairs are cached. Thus, exploiting the similarity of query terms across different queries by using this cache such that a query q_1 with terms t_1, t_2 and t_3 can use the intersection of posting lists of terms t_1 and t_2 which is cached when another query q_2 with terms t_1 and t_2 is issued.

To the best of our knowledge, utilizing query result similarity to change the TTL values of queries has not been subject to another study in this field. In [10], the TTL values are adaptively altered using the change of the query result when its TTL expires as the feedback. The TTLs are altered with predetermined functions, as a result, better stale ratio and false positive ratio values were obtained compared to fixed TTL setup. Other set of studies for cache invalidation includes [13, 14, 15]. In [15], instead of time based TTL values, frequency TTL is employed in which queries are invalidated with respect to the number of occurrences in the cache (cache hits). However, similarity TTL study focuses on altering time based TTL values using query result similarity.

In the literature, query result caching performances are evaluated with different metrics. Hit rate (or miss rate), which is a widely used metric for result caches [16], measures the proportion of query requests that are served (or missed) from the cache. This metric does not differentiate between queries as each miss has the same cost. Later, it was shown through cost-aware caching policies [8, 17] that queries have varying processing costs and the cache performance should be measured by taking these costs into account for cache misses. Real cost of a query can be calculated by integrating disk time for retrieval of posting lists, CPU time to uncompress the lists and CPU time to calculate the document scores. In [8], the cost of a query is simulated by considering the shortest posting list associated with the query terms. In a similar study [17], the cost is computed as the sum of the measured CPU time and simulated disk access time (under different posting list caching scenarios) and various static, dynamic, and hybrid cost-aware caching policies are proposed. In a more recent work [18], the performance of a hybrid dynamic result cache is also evaluated by the query cost metric.

Commercial web search engines rely on a large number of search clusters, each containing hundreds of nodes. Hence, they consume significant amounts of energy when processing user queries and the electricity bills for the large data centers form an important part of the operational costs of the search engine companies[19]. In a recent work [20], energy-price-driven query forwarding techniques are proposed to reduce the electricity bills. The main idea in that work is to exploit the spatio-temporal variation in electricity prices and forward queries to data centers that consume the cheapest electricity under certain performance constraints. This work considers that the data centers have varying processing capacities and query workloads. Being inspired by that work, a financial cost metric for evaluating the performance of query result caches is proposed. This new metric measures the total electricity cost incurred to the search engine company due to cache misses and assumes there is no interaction between different data centers. Since the electricity prices and the query traffic of the search engine both show high volatility within a day, it is important to analyze the overall financial cost of query result caching techniques in terms of real electric price. The most similar metric to our financial cost metric is the power consumption

metric used in [21]. In that work, a cache hierarchy consisting of result and list caches is evaluated in terms of the power consumption. Our financial cost metric considers not only power consumption but also the hourly electricity price rates and presents a more realistic financial cost evaluation.

Chapter 3

Strategies for Setting TTL Values for Search Engine Result Caching

3.1 Introduction

Query result caching is a commonly used technique in web search engines [16]. In this technique, the results of previously processed user queries are stored in a cache. The results for the subsequent occurrences of a query are served by this cache, eliminating the need to process the query and generate its results using the computational resources in the backend search system. This technique helps reducing the query processing workload incurred on the search engine while reducing the response time for queries whose results are cached. However, as the cache sizes increase with the cheap hardware used, the focus has been shifted to keeping the cache entries fresh.

As the index of the search engine is altered with the updates conducted on document set by addition, deletion and update operations on documents, it becomes crucial to keep the index and the result cache in sync. In other words, as index updates results of some queries, the results may change rendering the results in the cache useless. The user satisfaction is heavily depended on result freshness, especially for informational queries whose results prone to change

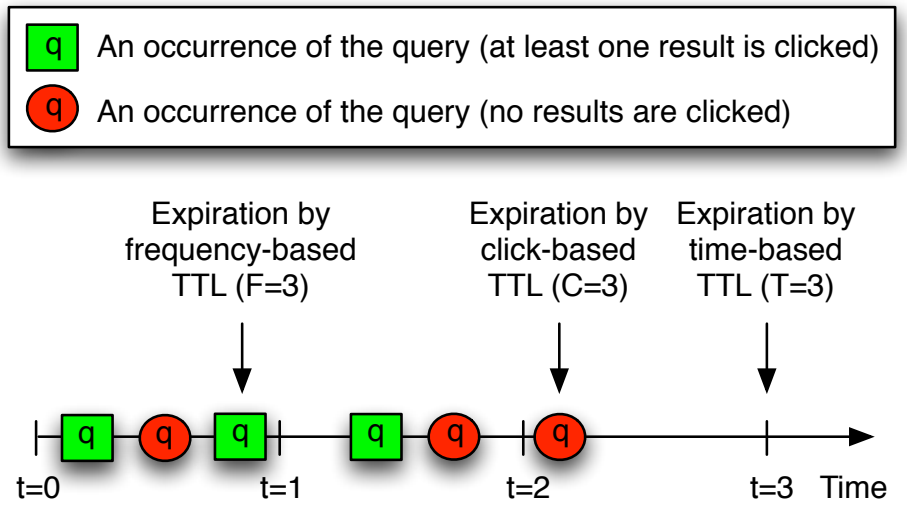

Figure 3.1: The points at which the results of a query are expired in different TTL approaches (the query results are assumed to be cached at time $t=0$).

constantly.

The focus of the work in this chapter[1] is on mechanisms for setting the TTL values of entries in result caches. We consider three alternative approaches, time-based TTL [6], frequency-based TTL [15], and click-based TTL. We evaluate the performance of these alternatives in terms of attained cache freshness and redundant query workload incurred to the backend. Moreover, we propose hybrid approaches that combine the above-mentioned basic approaches. Our results indicate that the best performance can be achieved when time-based TTL is combined with frequency-based TTL.

In this chapter, we first present the competing TTL approaches and the proposed approaches for combining them. We then provide the details of our experimental setup and the experimental results.

[1](Fethi Burak Sazoğlu, B. Barla Cambazoğlu, Rıfat Özcan, İsmail Sengör Altıngövde, and Özgür Ulusoy. 2013. Strategies for setting time-to-live values in result caches. In Proceedings of the 22nd ACM international conference on Conference on information & knowledge management (CIKM '13). ACM, New York, NY, USA, 1881-1884. DOI=10.1145/2505515.2507886 http://doi.acm.org/10.1145/2505515.2507886. Reprinted by permission with licence number 3458780725689.)

3.2 TTL Approaches

3.2.1 Basic Approaches

In this section, we present three different strategies for expiring the results of a cached query: Time-based TTL [10, 14, 6], frequency-based TTL [15], and click-based TTL. The functioning of these three strategies are illustrated in Figure 3.1. Throughout the section, we assume that the presented strategies are not accompanied by more sophisticated refreshing or invalidation mechanisms.

Time-based TTL. Time-based TTL is commonly used in result caches in search engines [6] as well as other types of caching systems. In this approach, every cached query result is associated with a fixed lifetime T. Given a query whose results are computed and cached at time t, the cached results are expired at time point $t+T$. Hence, the expiration point for the query results are known at the time of caching. The results of the query are considered to be invalid beyond time point $t + T$ (if the query results are not refreshed or already invalidated before that time by some other mechanism) and any request for the results leads to a cache miss. The time-based TTL strategy is especially useful for bounding the staleness of the results associated with infrequent (tail) queries. In general, larger T values increases the fraction of stale results served by the cache while smaller values lead to a larger fraction of queries whose results are redundantly computed. In Fig 3.1, the results of query q are expired $T = 3$ time units after they are cached.

Frequency-based TTL. A recently employed alternative is the frequency-based TTL (or virtual TTL) approach [15]. In this approach, unlike the time-based TTL approach where the expiration point (i.e., $t+T$) is fixed, the expiration point for the results of a query is determined depending on the recent occurrences of the query. In particular, the results of a query are assumed to be expired if the query was issued to the search engine F times since its results were cached. The frequency-based TTL approach is effective in bounding the staleness of very frequent (head) queries. In Fig 3.1, the results of query q are expired after the

query is issued to the search engine $F=3$ times.

Click-based TTL. To best of our knowledge, the click-based TTL strategy is not proposed before. This approach is somewhat similar to the frequency-based TTL approach in that it relies on the recent occurrence pattern of the query. In this approach, however, the expiration is determined only by occurrences in which no search results are clicked by the user. In particular, the results of a query are expired after C occurrences with no clicks (such occurrences do not have to be consecutive). The rationale here is to use the absence of clicks on search results as an indication of the staleness. In a sense, every occurrence of the query with no clicks on search results increases the confidence on that the query results are not fresh. In Fig 3.1, the results of query q are expired when the query results do not receive any click for $C=3$ times.

3.2.2 Hybrid Approaches

In this section, we describe two hybrid approaches that set the TTL based on a combination of the two or more of the basic TTL approaches presented in the previous section. We evaluate two logical operators in the combination: conjunction and disjunction. We experimented with other operators (e.g., multiplication), but the results were not better. Hence, we prefer to omit them herein.

Conjunction. In case of conjunction, all TTL approaches used in the combination should agree that the cached results should be expired. In a sense, this hybrid approach seeks for consensus to make an expiration decision. For instance, when the frequency- and time-based approaches are combined in the example given in Figure 3.1, the cached results will be expired at time point $t=3$, once both the query frequency reaches three and the age of the cache entry reaches three time units.

Disjunction. The disjunction approach is more aggressive with respect to the conjunction approach in that the results are expired as soon as one of the combined approaches raises a flag. Using the same example before, the cached

results are expired right before time point $t = 1$ because the frequency of the query reached three.

3.3 Setup

Data. We use a subset of a query log including the queries submitted to Spanish front-end of a commercial search engine. This constitutes to a set of 2,044,531 queries in timestamp order. We use the first half of the queries as the training set (i.e., to warm-up the cache) and remaining half as the test set. In our experiments, in addition to using this entire query stream, we also provide a more detailed performance analysis for the head and tail queries. To this end, we sort all unique queries in our query set by their submission frequencies, and label those in top-1% and bottom-90% as head and tail queries, respectively; and then construct the corresponding query streams that only include these identified queries. As before, we also make a 50/50 split of these streams for training and testing.

Simulation setup. We assume an infinitely large cache so that we can evaluate the proposed strategies independently from the other parameters such as the cache size and eviction strategies, as in [10]. We assume that for a query-timestamp pair (q, t), the top-k ($k <= 10$) URLs stored in the query log serve as the ground truth result R_t^* (i.e., the fresh answer for q at time t is R_t^*). During the simulations, when a query is first encountered, say at time t, its result R_t^* is cached. In a subsequent submission of the same query at time t', if the TTL assigned to this result has not yet expired, we assume this cached result is served; otherwise we refresh the result by taking the result $R_{t'}^*$ from the query log. The result R served from cache at a time point t is said to be stale if it differs from the result in the query log, R_t^*. As in [12, 14] we consider any two results as different if they don't have exactly the same URLs in the same order.

Evaluation Metrics. We evaluate the basic and hybrid TTL approaches in terms of the stale traffic (ST) ratio versus the false positive (FP) ratio (as in [12, 14]). Stale traffic ratio is the percentage of the queries for which the result

served from the cache turns out to be stale. False positive ratio is the percentage of redundant query executions, i.e., the fraction of the queries for which the refreshed result is found to be the same as the previous result that was already cached. As we aim to minimize both of these metrics, in the following results we report the performance for the parameter combinations that yield the minimum total value of the ST and FP ratios.

3.4 Results

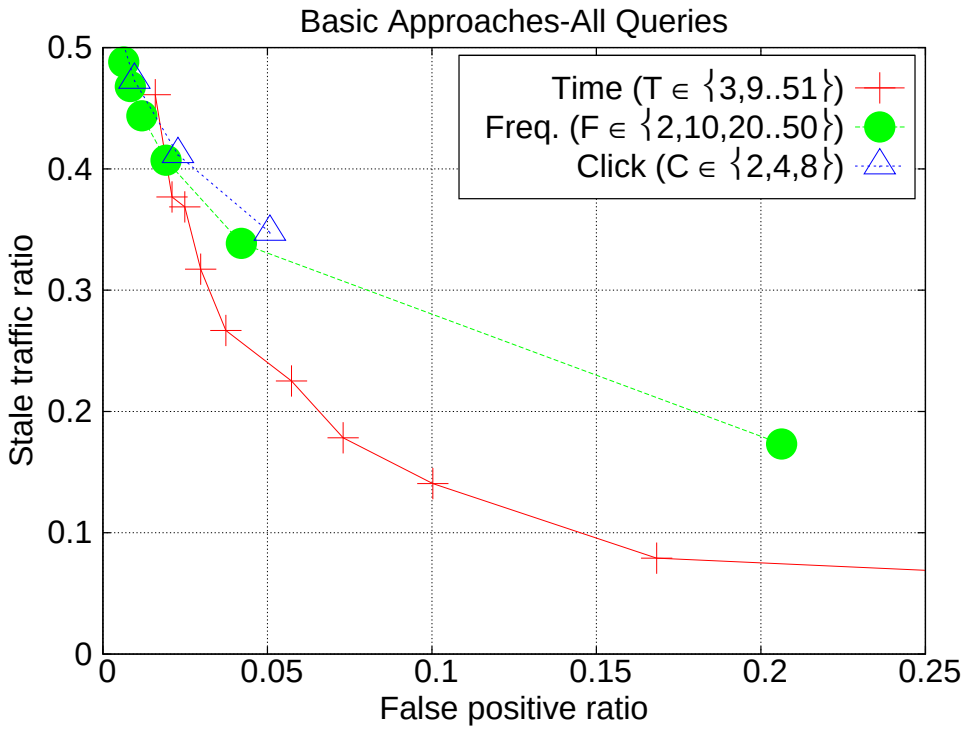

Figure 3.2: Stale traffic and false positive ratios for basic approaches over all queries

Figure 3.2 shows the simulation results for the basic approaches over the entire query stream (referred to as *all* queries hereafter). In this case, time-based TTL is superior to frequency- and click-based approaches, as both of the latter yield higher ST ratios than time-based TTL for the FP ratios larger than 2%, as a consequence non of frequency- and click-based TTL approach is better than

time-based TTL on their own.

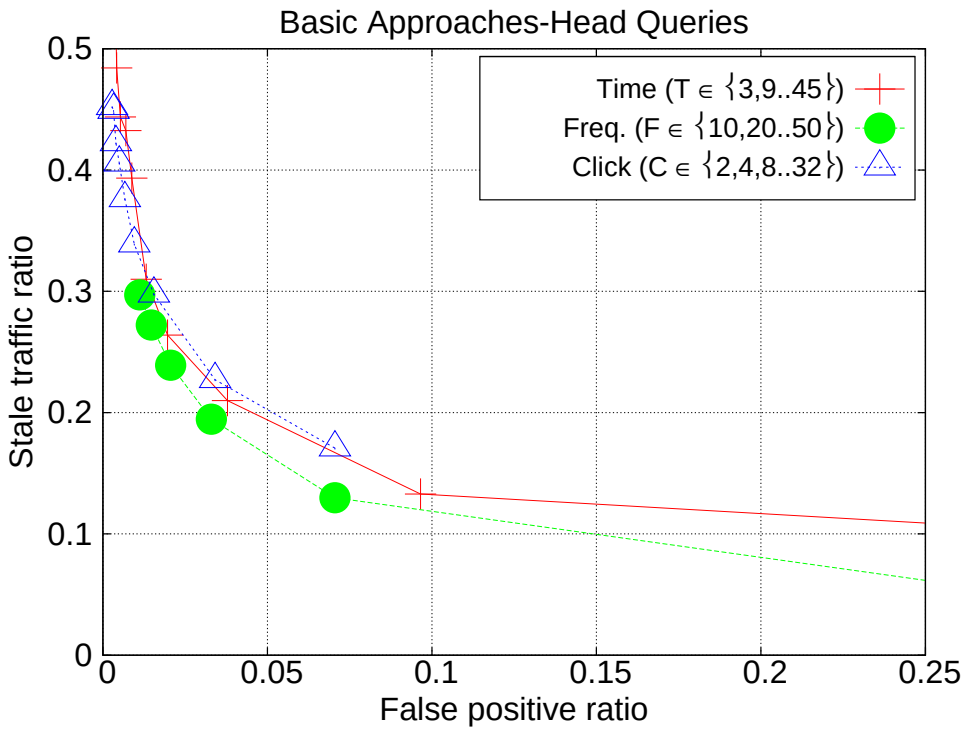

Figure 3.3: Stale traffic and false positive ratios for basic approaches over head queries

In contrast, for the head queries, we see that time- and click-based TTLs are comparable and the frequency-based TTL outperforms both of the latter (Figure 3.3). This is an intuitive finding; since the head queries are extremely popular, setting a fixed time interval as the TTL cannot capture the sudden updates on the underlying index, which yields lots of stale results for head queries. Frequency-based TTL applies an upper bound on the number of stale results that can be served from the cache (indeed, this is the underlying motivation for proposing the frequency-based TTL approach in [15]). This expiration mechanism allows expired queries to have different results from the cache, which results in less unnecessary query processing (fp ratio).

In our third experiment, we investigate the performance for the tail queries (Figure 3.4). We find that both of the frequency- and click-based approaches fail to improve the performance for the tail queries, and that is why they are inferior

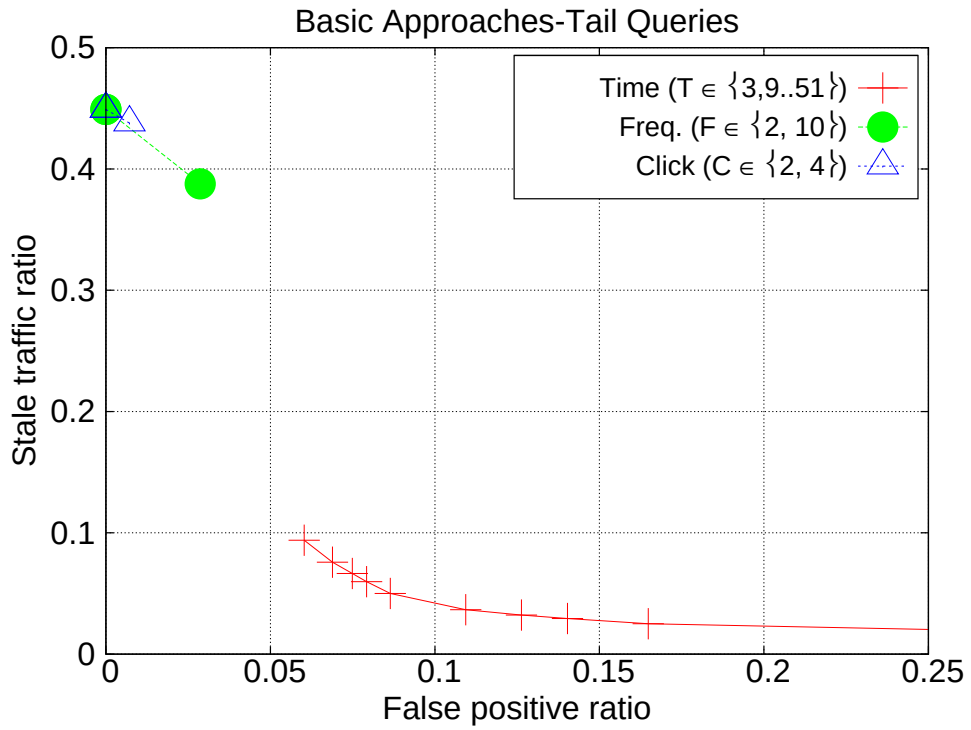

Figure 3.4: Stale traffic and false positive ratios for basic approaches over tail queries

to time-based TTL in the overall case, i.e., for the all queries (see Figure 3.2). This latter finding is caused by the fact that the submission frequency of tail queries is very low, hence the next time this tail query is submitted to the search engine its result will change. This situation leads to very long time periods until when an expiration decision can be made by the frequency- and click-based TTL strategies, even for the smallest values of F and C parameters (see the corresponding points for F and C are equal to 2 in Figure 3.4). And during this long time period, the underlying index and query results are likely to be updated, which yield very high ST ratios at the end.

Next we explore the performance for the hybrid approaches. While doing so we take the best basic strategy from Figures 3.2 to 3.4 as the baseline (i.e., time-based TTL for the all and tail query streams, and frequency-based TTL for the head queries). We create the conjunction and disjunction of the pairs of strategies, i.e., (time-based, frequency-based) and (time-based, click-based),

as well as all three of them. We discard the pair (frequency-based, click-based) for the readability of the plots, as it is found to be inferior to all other hybrid approaches anyway.

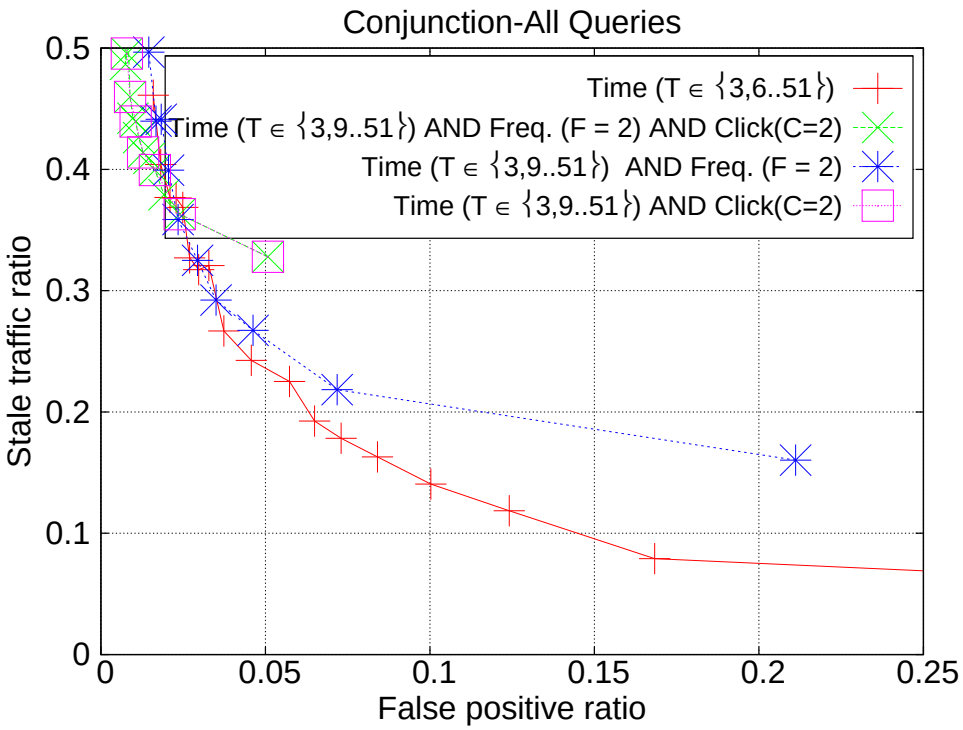

Figure 3.5: Stale traffic and false positive ratios for conjunction-based hybrid approaches over all queries.

In Figures 3.5 to 3.7, we present the results for the conjunction of the expiration decisions from the basic TTL approaches for the all, head and tail queries, respectively. It turns out that none of the hybrid approaches can outperform the baseline basic approach for any of these query sets. In other words, seeking a consensus among these approaches seems to delay the expiration decision and likely to cause more stale results.

For the hybrid approaches based on the disjunction of individual expiration decisions, the picture is different. Figure 3.8 shows that these hybrid methods can considerably outperform the baseline time-based TTL approach for the entire query stream. Figures 3.9 and 3.10 explain why this happens. In Figure 3.9, we see that all hybrid versions are also superior to the baseline also for the head

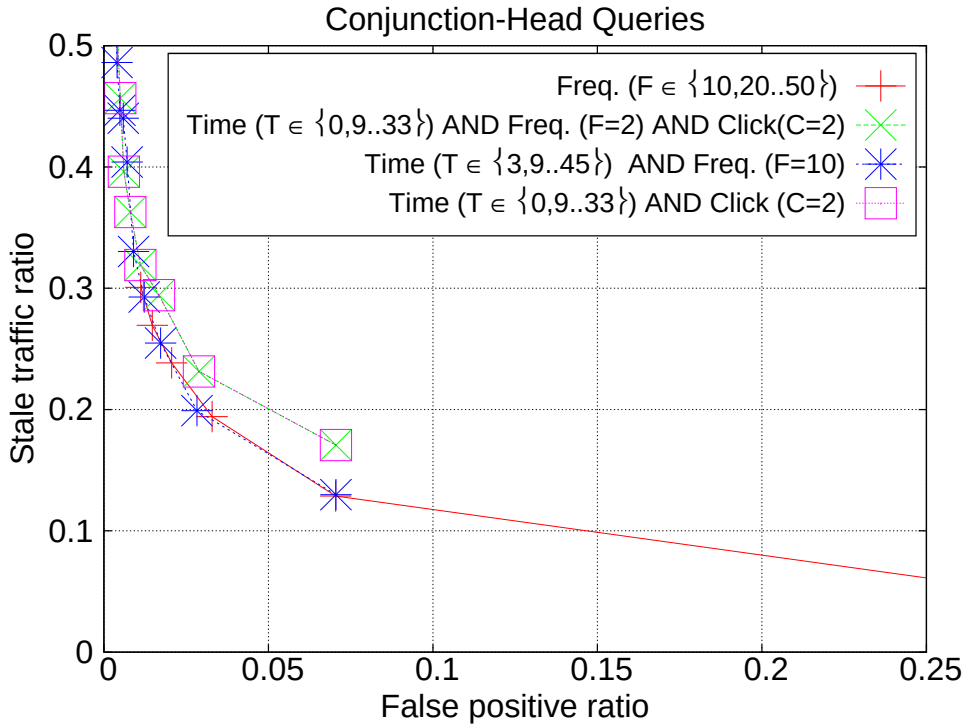

Figure 3.6: Stale traffic and false positive ratios for conjunction-based hybrid approaches over head queries.

queries, the best one (i.e., with the lowest ST ratios) being the strategy (time-based TTL OR frequency-based TTL). While doing so, these hybrid strategies do not degrade the performance for the tail queries (see Figure 3.10) and thus the improvements for the head queries are also reflected to the entire query stream. In other words, using a disjunction of decisions from the time- and frequency-based TTL strategies, we combine the best of two worlds: we improve the performance for the head queries without any adverse effects on the tail queries, and thus we end up with a better overall performance.

3.5 Conclusions

We evaluated the performance of three basic time-to-live (TTL) approaches for result caching: time-based TTL, frequency-based TTL, and click-based TTL.

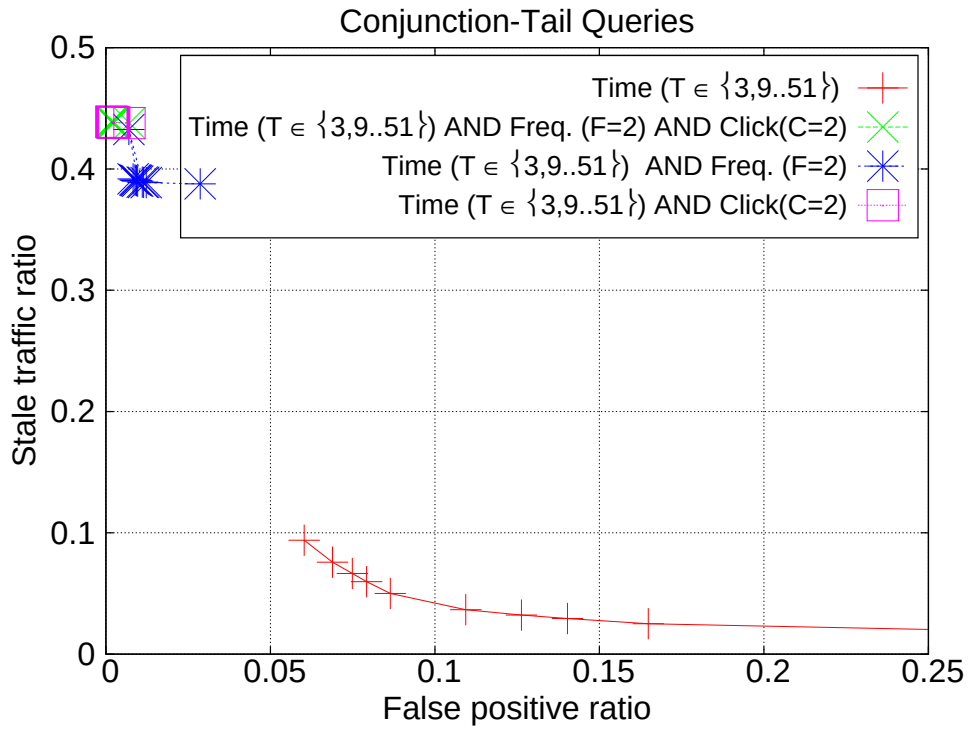

Figure 3.7: Stale traffic and false positive ratios for conjunction-based hybrid approaches over tail queries.

We further proposed hybrid TTL techniques that combine the basic approaches. We measured the attained stale query traffic ratio and redundant computation overhead via simulations on a real-life query log obtained from a commercial web search engine. Our experimental results indicate that the best performance is achieved when time-based TTL is combined with frequency-based TTL using a disjunction of the expiration decisions from these two approaches. We also found that combining click-based TTL with the latter two strategies do not bring further improvement in practice.

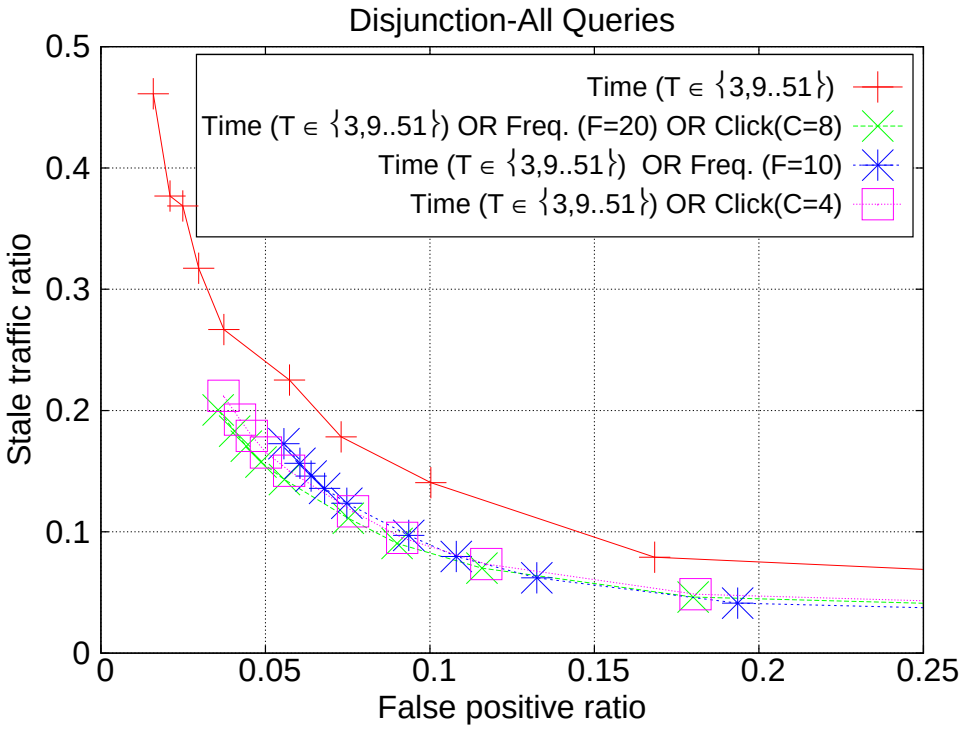

Figure 3.8: Stale traffic and false positive ratios for disjunction-based hybrid approaches over all queries.

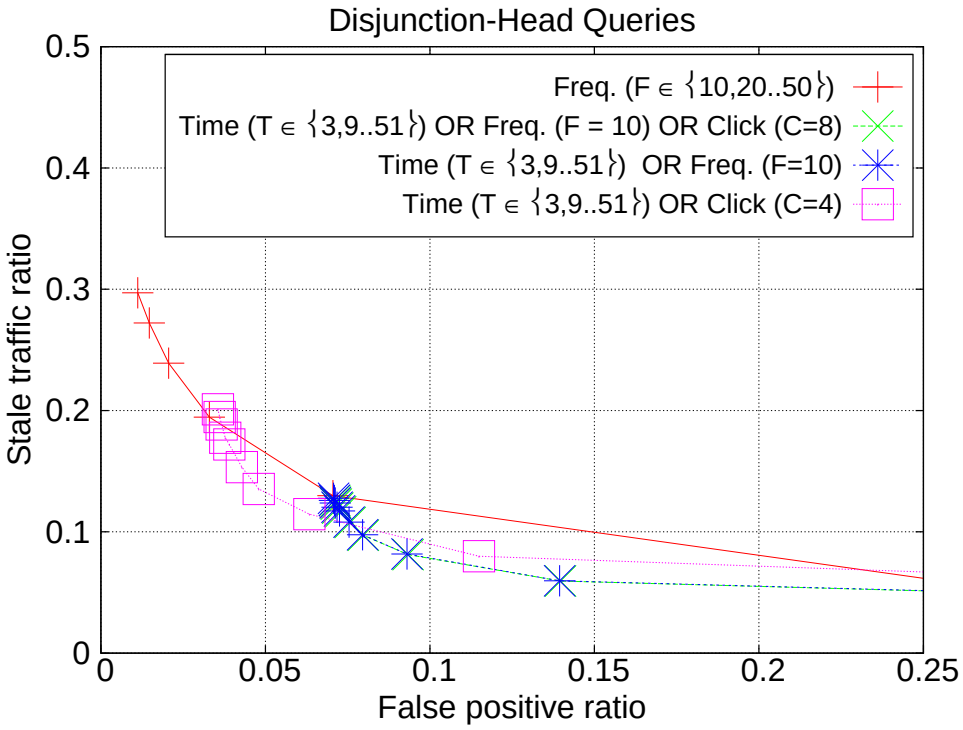

Figure 3.9: Stale traffic and false positive ratios for disjunction-based hybrid approaches over head queries.

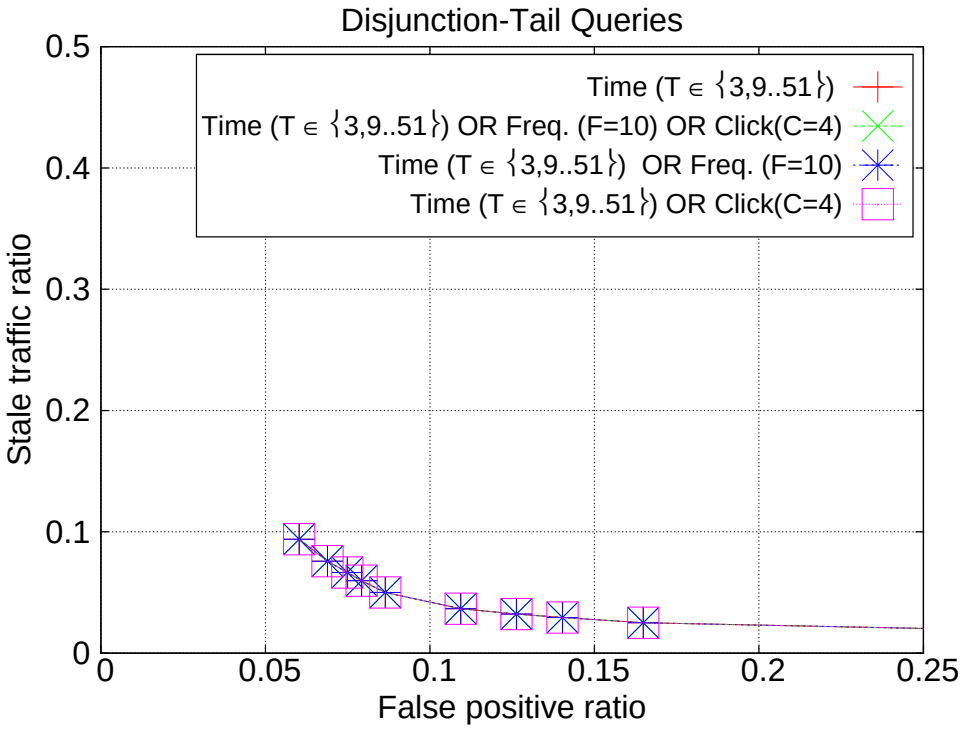

Figure 3.10: Stale traffic and false positive ratios for disjunction-based hybrid approaches over tail queries.

Chapter 4

Similarity based TTL Approach to Search Engine Result Caching

4.1 Introduction

Commercial search engines make use of different cache invalidation approaches for the freshness issue in result caches. In the previous chapter, basic approaches as well as their different combinations have been covered. The techniques discussed there utilize some query statistics such as query frequency (virtual TTL), number of clicks per result set (click TTL) which are obtained from the query log and constant time intervals that bound the staleness of query results (time-based TTL). The motivation for similarity-based TTL arises from the same source which is the search engine query log, however the statistics to exploit are slightly different than those used by the previous approaches. The similarity-based approach takes advantage of the result similarity of cached queries. It allows an expired query to signal this information to the similar queries in the cache.

4.2 Motivation

Query results are calculated by processing the query terms in the clusters of a search engine. Search engines use document and term partitioning to increase their processing power. The results are calculated at each node and merged to constitute the final result set. The posting lists of the documents in these clusters are used to give term-document similarity score to the queries which are sorted to prepare the result set. The posting lists contain the documents for terms and their frequencies term-wise.

There are two possible methods to determine similarity of queries, which are query text similarity and query result similarity. Query text similarity may be accounted for the similarity between the query results. However, query text similarity is not a reliable method to be used in assigning TTL values, as it cannot safely reflect the index updates. Therefore, query result similarity is employed in this chapter which can reliably propagate index changes to similar queries. The similarity between the results of two queries gives clues about how much their posting lists overlap. When the document set is modified by addition, deletion or update of a document, the underlying index is also updated which changes the query results. Queries affected from this change should decrement its TTL, because the results provided from cache would be rendered as stale. When a query is expired and its results change, which means the underlying index has been changed, it can alter the TTL values of its similar queries regarding query results. Since similar queries (in terms of query results) share common postings, consequently common documents; this change of query result probably occurs in the similar queries as well. Similarity based TTL method conveys this query result change information to its similar queries in the proportion of their similarity. If the similarity between two queries is low, then the propagated score due to this similarity is also small. The following sections have detailed description of the algorithms used for similarity based expiration of queries.

The experiments in this chapter uses time-based TTL [6] as the base methodology. The performance of similarity caching is evaluated in terms of cache freshness and redundant query workload incurred to the backend. In this chapter, the

algorithms are differentiated by their expire mechanism and the similarity score they use. All the queries keep a score for the queries. This score is updated when a similar query expires, and when it exceeds a certain threshold, either the TTL of the query is updated or the query is directly expired.

The rest of this chapter is organized as follows. In Section 4.3, the competing similarity TTL approaches are presented. The details of the experimental setup are presented in Section 4.4. Section 4.5 provides the experimental results. We conclude the chapter in Section 4.6.

4.3 Similarity-based TTL Algorithms

There are four algorithms presented in this chapter. According to the expiration mechanism, the algorithms are named by the effect of similarity score on expiration, which are `direct_expire` and `indirect_expire`. If the TTL is updated using the score and expired by only the TTL method, the algorithm will be referred as `indirect_expire` algorithm. Otherwise, if the query is expired by TTL or similarity score the algorithm will be called `direct_expire`, meaning that the query is expired if query age exceeds the TTL or the score exceeds the threshold value given as parameter.

For each of these methods two different types of scores are calculated. The first type is simply the similarity of two queries,

$$Value(q_1) = similarity(q_1, q_2) \tag{4.1}$$

This similarity type will be referred as `basic_score` in the algorithms. The other score type incorporates the cache ages of the queries into the score. Consider two queries q_1 and q_2 which are cached at times t_1 and t_2, respectively, and have query result similarity of s. Assume that q_1 expires at time t and it will update the similarity scores of its similar queries and q_2 is one of these similar queries. Assume that at the expiration time cache age of q_1 is $t - t_1$, and cache age of q_2

is $t - t_2$. In the second type of score, it is assumed that as the cache age of q_1 increases its effect to the score of q_2 diminishes. Similarly, the effect of the cache age of q_2 is assumed to have positive impact on its score. This score is calculated as follows:

$$Value(q_2) = similarity(q_1, q_2) \times \frac{1}{cache\_age(q_1)} \times cache\_age(q_2) \tag{4.2}$$

The query result similarity for queries q_1 and q_2 with respective results R_1 and R_2 is calculated using Jaccard similarity, which is

$$Similarity(q_1, q_2) = \frac{R_1 \cap R_2}{R_1 \cup R_2} \tag{4.3}$$

The cache age of the query q_1 is the time spent between query caching time (t_1) and the current time (t), which is

$$CacheAge(q_1) = t - t_1 \tag{4.4}$$

This score will be mentioned as `age_score` in the following algorithms.

The Algorithms. Query log is processed in the order of the timestamps. The cache is assumed to have infinite size. When a query is received, if it is not in the cache, it is stored in the cache alongside with its timestamp and top 10 most similar queries in the cache. If the cache contains the query, it is expired according to the rules explained above and if its result is changed, scores of its similar queries are incremented using the formulas above. The first half of the log is used for training and the second half is used for testing in which *fp*, *tp* , *fn* and *tn* statistics are calculated. Note that, the scores for similar queries are not decremented when the result of a query stays the same. The score is set to 0, when the query expires. The algorithms are named by combining expiration mechanism with the query score, as `BasicScore_DirectExpiration`, `BasicScore_IndirectExpiration`, `AgeScore_DirectExpiration` and `AgeScore_IndirectExpiration`.

Input: q: query, C: Cache, T_S: similarity threshold,
t_q: submission time of q, c_q: caching time of query q, s_q: score of query q.

$R_q \leftarrow \emptyset$ ▷ initialize the result set of q;

if $q \notin C$ **then** `/* Not Cached */`
  evaluate q over the backend and obtain R_q;
  insert R_q into C ;
else if $q \in C$ **then** `/* Cached */`
  get R_q from C;
  if $t_q - c_q \geq TTL$ *or* $s_q \geq T_S$ **then** `/* Query Expires */`
    increment scores of similar queries (using 4.2 and 4.3);
    update statistics of q in C ;
  else if $t_q - c_q < TTL$ *and* $s_q < T_S$ **then** `/* Query Not Expires */`
    update statistics of q in C
return R_q;

Algorithm 1: Similarity Algorithm `DirectExpiration`.

Table 4.1 categorizes the algorithms according to their expiration mechanisms and score calculation approach.

4.4 Experimental Setup

Data. In this chapter the same data from the previous chapter, which is the query log of the queries submitted to Spanish front-end of a commercial search engine, is used. The query log is split into two equal portions for training and testing purposes. The detailed information about the data can be found in the Appendix.

Simulation setup. Similar to the previous chapter, an infinitely large cache is assumed and for a query-timestamp pair (q, t), the top-k ($k <= 10$) URLs stored in the query log serve as the ground truth result R_t^* (i.e., the fresh answer for q at time t is R_t^*). The same simulation setup from the previous chapter is employed for experiments.

Evaluation Metrics. The evaluation is done in terms of the stale traffic (ST) ratio versus the false positive (FP) ratio (as in [12, 14]).

Input: q: query, C: Cache, T_S: similarity threshold,
t_q: submission time of q, c_q: caching time of query q, s_q: score of query q,
q_1: similar query for query q, s_{q1}: score of query q_1.

$R_q \leftarrow \emptyset$ ▷ initialize the result set of q;

if $q \notin C$ **then** `/* Not Cached */`
  evaluate q over the backend and obtain R_q;
  insert R_q into C ;
else if $q \in C$ **then** `/* Cached */`
  get R_q from C;
  if $t_q - c_q \geq TTL$ *or* $s_q \geq T_S$ **then** `/* Query Expires */`
    increment scores of similar queries (using 4.2 and 4.3), decrement TTLs for queries q_1 where $s_{q1} \geq T_S$;
    update statistics of q in C ;
  else if $t_q - c_q < TTL$ *and* $s_q < T_S$ **then** `/* Query Not Expires */`
    update statistics of q in C
return R_q;

Algorithm 2: Similarity Algorithm `IndirectExpiration`.

	Expiration by TTL or score exceeds threshold	Expiration by Only TTL
Score is query similarity	`BasicScore_DirectExp.`	`BasicScore_IndirectExp.`
Score is query similarity with query age	`AgeScore_DirectExp.`	`AgeScore_IndirectExp.`

Table 4.1: Similarity Algorithms.

4.5 Experimental Results

Inclusion of similarity TTL does not yield significant difference when compared with time TTL. The implications of this result are twofold. The first one is the query log data used in the experiments. The similarity of the queries may not be sufficient for the similarity feedback mechanism in the algorithms. The other implication would be the underlying time TTL utilized in the similarity TTL algorithms. The time TTL may eliminate the need for feedback from similar queries by invalidating (expiring) queries before similar queries give feedback to expiring queries. Comparing the results with the time TTL values shows that stale traffic ratio values diminish. Therefore, similarity TTL can be used to decrease proportion of stale results in cases where the TTL values stay the same.

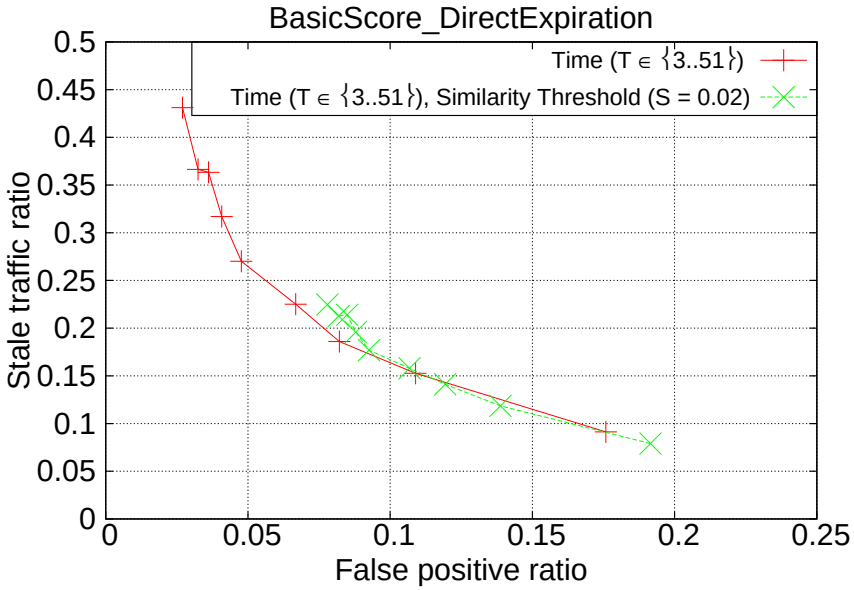

Figure 4.1: Stale traffic and false positive ratios for `BasicScore_DirectExpiration` Algorithm.

Table 4.2 shows the comparison of 4 similarity TTL algorithms in more detail. For each TTL value, the performance results are presented in terms of the gain in *st* and *fp* sums as percentage. Assume that st_s and fp_s are stale traffic and false positive ratios for similarity TTL, and st_f and fp_f are stale traffic and false positive ratios for time TTL. Then, the gain as percentage is calculated as;

$$Gain = \frac{(st_s + fp_s) - (st_f + fp_f)}{(st_f + fp_f)} \times 100 \tag{4.5}$$

This formula assumes that the impact of stale traffic and false positive ratios are the same for the performance. However, from commercial search engine's point of view lowering stale traffic ratios is more important than lowering false positive ratios as serving stale results is worse than redundant result computation at the backend in terms of user satisfaction. As Figures 4.1, 4.2, 4.3, and 4.4 indicate, the similarity TTL is more effective on decreasing stale traffic ratios than decreasing false positive ratios.

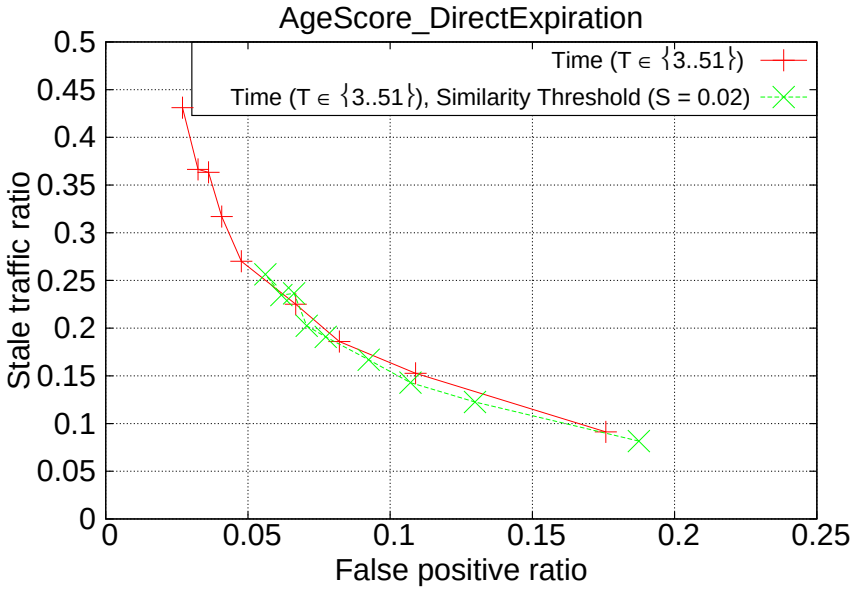

Figure 4.2: Stale traffic and false positive ratios for `AgeScore_DirectExpiration` Algorithm.

4.6 Conclusion

The result cache is a crucial component of search engines. The contribution of every correctly invalidated cache entry is twofold from search engines point of view. It can first deliver the query result without any processing directly from the cache that reduces the response size. Secondly, it discards the backend when processing such queries which decreases both the load on the backend and the power consumption of the data centres. Similarity TTL can be applied to any type of TTL such as time-based TTL and frequency-based TTL. The main idea is to incorporate the query result similarity to the existing result expiration methods. In case of an index update, this information can be propagated to the other similar queries proportional to the similarity.

The algorithms evaluated in this chapter could not decrease the false positive and stale traffic ratios when the time TTL curve is considered. However, if the experimental results are compared TTL-wise, the stale traffic ratios decrease dramatically. Therefore, for the same TTL values, similarity scores expire the

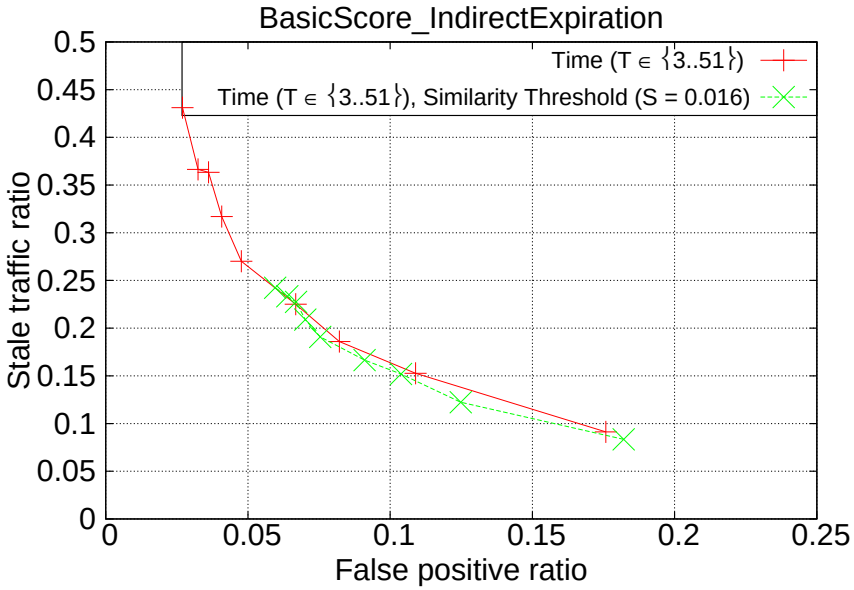

Figure 4.3: Stale traffic and false positive ratios for `BasicScore_IndirectExpiration` Algorithm.

queries correctly to prevent the search engine return more stale results. Thus, this method can be employed as a supplementary to other TTL techniques to decrease false positive and stale traffic ratios without changing TTL values.

Figure 4.4: Stale traffic and false positive ratios for `AgeScore_IndirectExpiration` Algorithm.

Alg. / TTL	Basic_Dir.Exp.	Age_Dir.Exp.	Basic_In.Exp.	Age_In.Exp.
0	0	0	0	0
3	-1.28615	0.565008	-0.734622	0.660112
6	-1.42111	2.25798	0.163838	2.58879
9	1.6461	5.53375	3.51272	5.51923
12	3.74574	6.33509	5.17047	6.8085
15	2.81806	4.63897	6.74739	6.37709
18	4.4994	7.09973	7.76227	8.92459
21	9.27627	11.7569	11.1364	10.3542
24	9.08804	12.2731	11.6274	12.0738
27	15.194	16.2393	15.6015	14.4704
30	26.134	22.1339	24.5124	24.1398
33	20.7351	21.9942	23.6592	24.2981
36	15.4061	15.2683	17.6225	16.1344
39	25.1787	26.2944	24.2339	28.2288
42	25.4186	27.8357	29.0419	27.9313
45	26.1094	25.4527	25.6711	25.8273
48	27.8268	27.6122	28.7488	28.8271
51	33.924	34.0936	31.7792	33.2333

Table 4.2: Similarity Algorithms Result Comparison as Gain in Percentage.

Chapter 5

A Financial Cost Metric for Result Caching

5.1 Introduction

Commercial web search engines cache query results for efficient query processing. The main purpose of result caching is to exploit temporal locality of search queries. Search engine result caching exploits the idea that a query submitted to the search engine will be resubmitted by the same or a different user in close proximity.

The main contributions of the work in this chapter[1] are the following. First, a financial cost metric is offered for query result caches. Second, the state-of-the-art static, dynamic, and hybrid caching techniques in the literature are evaluated using this new metric. Finally, a financial-cost-aware version of the well-known LRU strategy is proposed and shown to be superior to the original LRU strategy

[1](Fethi Burak Sazoğlu, B. Barla Cambazoğlu, Rıfat Özcan, İsmail Sengör Altıngövde, and Özgür Ulusoy. 2013. A financial cost metric for result caching. In Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval (SIGIR '13). ACM, New York, NY, USA, 873-876. DOI=10.1145/2484028.2484182 http://doi.acm.org/10.1145/2484028.2484182. Reprinted by permission with licence number 3458780850476.)

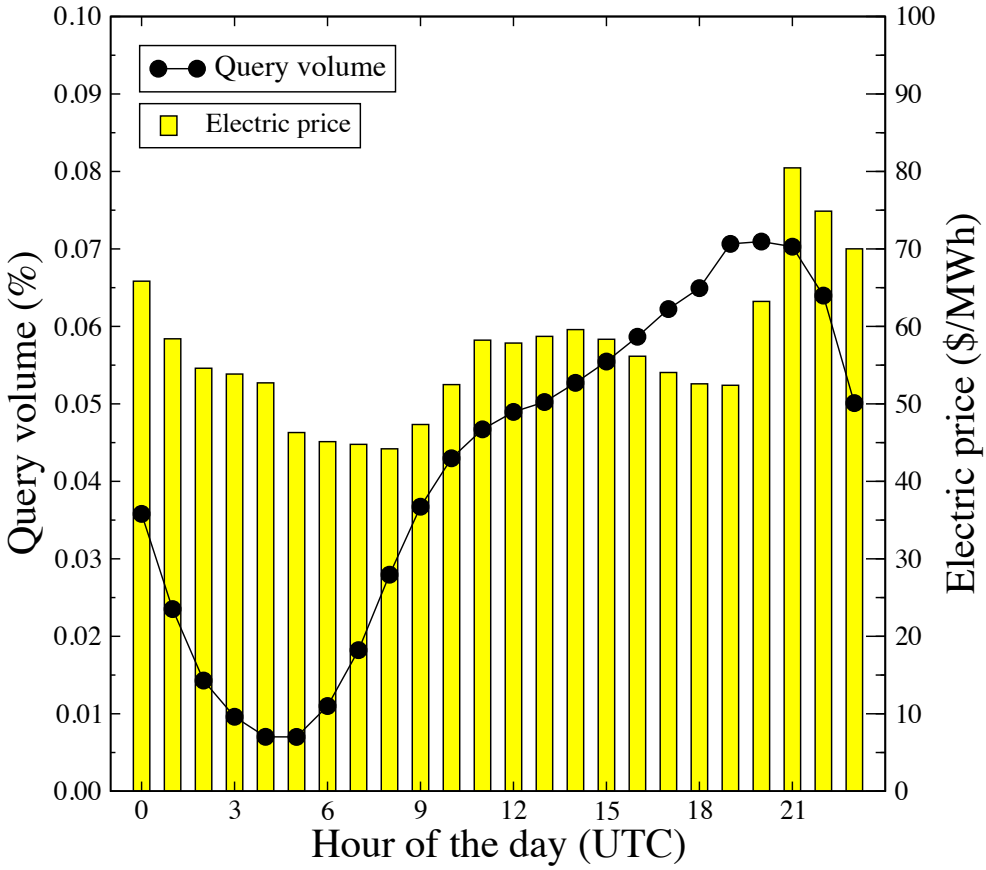

Figure 5.1: Hourly query traffic volume distribution and hourly variation in electricity prices.

under this metric.

5.2 Financial Cost Metric

Cost-aware caching strategies [8, 17] consider the time overhead of processing cache misses. The financial cost metric goes one step forward and computes the cost of electricity consumed when processing cache misses. The underlying motivation here is that the electricity prices show temporal variation and hence the financial cost of processing a query varies in time. In practice, the electricity price changes mainly based on supply-demand rates and certain seasonal effects [22]. The query traffic received by a search engine also rises and falls irregularly depending on time. As an example, Figure 5.1 shows the hourly electricity prices

taken from an electricity provider located in New York and the distribution of the query traffic received by Yahoo! web search. The electricity prices and query volume are normalized by the their respective mean value.

Our financial cost metric simply computes the processing time of the query as given in [17] weighted by the electricity price at the time of processing the query. Since only hourly electricity price is available, momentary electricity price cannot be obtained. Instead electricity price in that hour is taken.

Cache hits are assumed to incur no financial cost in terms of query processing. Proposed financial cost metric is defined as follows

$$C_q = T_q \times P[t], \tag{5.1}$$

where q is a query submitted at time t, $P[t]$ is the electricity price at time t, and T_q is the time needed to process q. $P[t]$ is taken as the price in that hour.

5.3 Result Caching Techniques

We evaluate the most well-known policies in terms of our financial cost metric. A new financial cost aware caching policy, which provides improvement over LRU algorithm by exploiting hourly variance of electricity price, is designed. In the remaining sections of this chapter, the frequency of a query q is denoted as F_q. Each caching policy is briefly described below.

Most Frequent (`MostFreq`): This policy basically fills the static cache with the results of the most frequent queries in the query log. Thus, the value of a cache item is simply determined as follows:

$$Value(q) = F_q. \tag{5.2}$$

Frequency and Cost (`FC`): This policy [17] combines the frequency and cost of queries in a static caching setting. The value of a cached query is determined

by the product of its (boosted) frequency and cost:

$$Value(q) = C_q \times (F_q)^K, \tag{5.3}$$

where $K > 1$.

Least Recently Used (`LRU`): This well-known dynamic caching policy chooses the least recently requested query as the victim for eviction.

Least Frequently Used (`LFU`): In this policy, each cache item has a frequency value that shows how many times the cache item is requested. The cache item with the lowest frequency value is replaced when the cache is full.

Least Frequently and Costly Used (`LFCU`): This policy [17] is the dynamic version of the `FC` static caching policy. When the cache is full, the item with the lowest value (determined by its frequency and cost in Equation (5.3)) is evicted.

Greedy Dual Size (`GDS`): This policy [23] computes a so-called *H_value* metric for each cached query q as

$$H\_value(q) = \frac{C_q}{S_q} + L. \tag{5.4}$$

This value combines the cost and size of the item with an aging factor L. This aging factor creates an effect similar to the recency component in the LRU policy. The cache item with the lowest *H_value* is chosen as the victim for eviction. In this work size is taken as 1.

Greedy Dual Size Frequency (`GDSF`): This policy [24] is a slightly modified version of the `GDS` policy. It further considers the frequency of the cache item when computing the *H_value*. We also boost the frequency component with a power coefficient K as in [17].

$$H\_value(q) = (F_q)^K \times \frac{C_q}{S_q} + L. \tag{5.5}$$

Static Dynamic Cache (`SDC`): `SDC` [2] is a hybrid caching policy that reserves a portion of the cache space for static caching and the remaining space for

dynamic caching. Static cache is populated with the most frequent queries over a period of time. Popular queries are served from this component of SDC while its dynamic component is included to capture the changes in query stream in short time intervals. Dynamic part is vital for serving queries that stays popular for short periods such as news queries. In SDC it is important to fine tune the proportion of sizes of static and dynamic parts to maximize the hit rates.

It is shown that this strategy outperforms both purely static and purely dynamic caches. Static cache alone cannot handle sudden changes in the query stream while pure dynamic cache has to constantly evict and readmit popular queries to make room for queries possibly with small frequencies depending on its size due. The policy of dynamic cache that keeps the most recent queries in the cache drops the hit rate comparing to its hybrid version.

Two-Part LRU Cache (`2P_LRU`): A two-part LRU cache (similar to [18]) is proposed to optimize the result cache performance in terms of the financial cost metric. The main motivation of this strategy is to take advantage of financial cost variation to make the cache cost-aware. This strategy combines the segmented LRU idea (evaluated in [1]) with an admission control mechanism based on a financial cost threshold. Algorithm 3 presents the pseudocode for the `2P_LRU` policy, which reserves a certain portion of the cache space for queries submitted in the high financial cost period (i.e., expensive cache (E)) and the rest for those submitted in the low cost period (i.e., cheap cache (C)).

We decide on the expensive and cheap time periods based on a financial cost threshold (T_P). If the current financial cost ($T\_q \times P[t]$) is less (greater) than this threshold, we say that the query is submitted in the cheap (expensive) time period, respectively. The partitioning of `2P_LRU` cache is to keep the expensive queries in the cache as long as possible as the space permits, while cheap portion of the cache are reserved for queries that can be reevaluated due to its cheap processing cost. The `2P_LRU` policy realizes this as follows: If the result of a requested query q is not found in the expensive and cheap caches, it is evaluated at the backend and then inserted into the cheap cache (incurs a financial cost of $T\_q \times P[t]$) regardless of the financial cost of the query, i.e., cheap or expensive

Input: q: query, E: Expensive cache, C: Cheap cache,
T_P: financial cost threshold, t_q: submission time of q,
T_q: time cost of q, $P[t]$: price at time t, C_q: financial cost of q

$R_q \leftarrow \emptyset$ ▷ initialize the result set of q;

if $q \notin E$ *and* $q \notin C$ **then** /* $C_q = T_q \times P[t_q]$ */
  evaluate q over the backend and obtain R_q;
  insert R_q into C (if full, evict the LRU item);
else if $q \in E$ **then** /* $C_q = 0$ */
  get R_q from E;
  update statistics of q in E;
else if $q \in C$ *and* $T_q \times P[t_q] < T_P$ **then** /* $C_q = 0$ */
  get R_q from C;
  update statistics of q in C;
else if $q \in C$ *and* $T_q \times P[t_q] \geq T_P$ **then** /* $C_q = 0$ */
  evict R_q from C and insert into E (if full, evict LRU item);

return R_q;

Algorithm 3: Two-part LRU caching algorithm.

periods. The intuition behind this choice is that, as there is a high chance for a query to be singleton (i.e., submitted only once) due to the power law distribution of the query stream, we do not want to waste the expensive cache capacity without having enough evidence that the query will be re-submitted. If the query result is found in the cheap cache, then we check the current financial cost of the query and determine the time period. If we are in the cheap period, we simply serve from the cheap cache and update the query access statistics (i.e., housekeeping for LRU). Otherwise, we evict the query result from the cheap cache and insert it into the expensive cache (i.e., after seeing that the query is not a singleton). If the expensive cache is full, the least recent query is evicted from expensive cache and inserted to the most recent portion of cheap cache. This gives a second chance to the evicted expensive cache item to reenter to the expensive cache. Finally, if the query result is found in the expensive cache, it is served from the cache and, again, the statistics are updated.

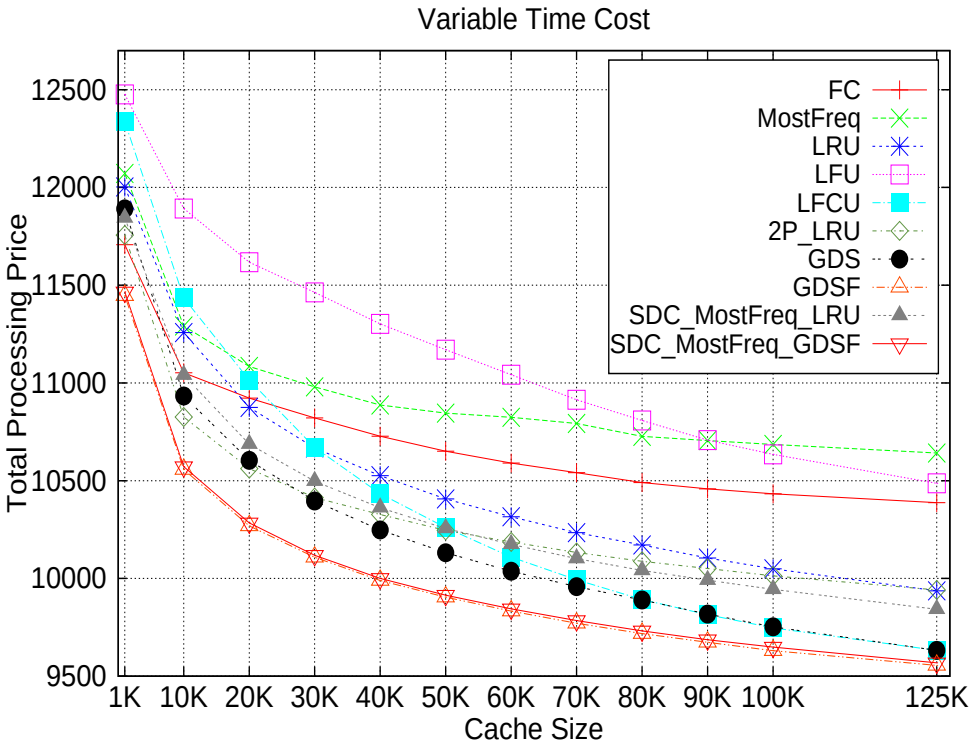

Figure 5.2: Financial cost evaluation of caching policies assuming variable query processing time costs.

5.4 Experiments

We evaluate the performance via a simulation using a subset of the AOL query log [25], which contains around 20 million queries. We use 2.2 million pages crawled from the Open Directory Project Web directory (`http://www.dmoz.org`) as our document collection. These pages are indexed using the Zettair search engine (`http://www.seg.rmit.edu.au/zettair/`) and a 1.1M query subset from the AOL query log is processed. We remove queries with no results. After this process, we end up with a stream of 809,795 queries over a period of six weeks. We reserve 446,952 queries (253,961 unique) submitted in the first three weeks as the training set and use this set to fill the static caches and/or to warm up the dynamic caches. The remaining 362,843 (209,636 unique) queries form the test set, and the hit rate and financial cost metrics are computed over this set.

Our experiments consider two different cases. In the first case, query processing times of queries (T_q) are assumed to be variable. These times are measured

as CPU times using the Zettair search engine. We refer to this case as "nonuniform (variable) time costs". In the second set of experiments, we consider only the price rate at the hour of the query submission ($P[t_q]$) and set the processing cost of queries to 1 (i.e., $T_q = 1$ for all queries). We refer to this case as "uniform (fixed) time costs". This latter scenario is motivated by the fact that search engines limit the time spent processing a query [20], i.e., we assume that the processing times of queries are nearly the same and close to this limit.

In our static cache simulation, when we compute the cost ($T_q \times P[t]$) for a query, we use the average electricity price observed in the training set when the query is processed at different times. For the dynamic caching setup, we consider the last time the query is issued and use the electricity price at that time point. We set various parameters as follows: For the `FC`, `LFCU`, and `GDSF` policies, K is set to 2.5 [17]. When dividing the cache space in `SDC`, %20 is reserved for the static portion and the rest is for the dynamic portion (tuned empirically). We also experimentally tune the `2P_LRU` policy and allocate %60 of the cache space for the cheap cache, and the remaining %40 for the expensive cache. We set the financial cost threshold (T_P) parameter to 0.02 and 0.9 in the variable and fixed time cost scenarios, respectively.

Figs. 5.2 and 5.3 present the performance of caching policies for the variable time cost scenario, in terms of the financial cost and hit rate metrics, respectively. For a typical large cache (90K or 100K), the policies can be ordered according to their performance as follows:

- Financial cost: `LFU` $\cong$ `MostFreq` $>$ `FC` $>$ `LRU` $>$ `2P_LRU` $>$ `SDC_MostFreq_LRU` $>$ `GDS` $\cong$ `LFCU` $>$ `GDSF` $\cong$ `SDC_MostFreq_GDSF`.
- Hit rate: `GDS` $<$ `LFCU` $<$ `FC` $<$ `GDSF` $\cong$ `LFU` $<$ `MostFreq` $<$ `2P_LRU` $\cong$ `SDC_MostFreq_GDSF` $<$ `LRU` $<$ `SDC_MostFreq_LRU`.

It is interesting to note that even though `LRU` and `SDC_MostFreq_LRU` policies are the best-performing policies, according to the hit rate metric, they are outperformed by the cost-based policies (`LFCU`, `GDS`, `SDC_MostFreq_GDSF`, `GDSF`) in terms of the financial cost metric. The proposed `2P_LRU` policy incurs lower financial costs than the `LRU` policy. The reductions in financial cost reach up to

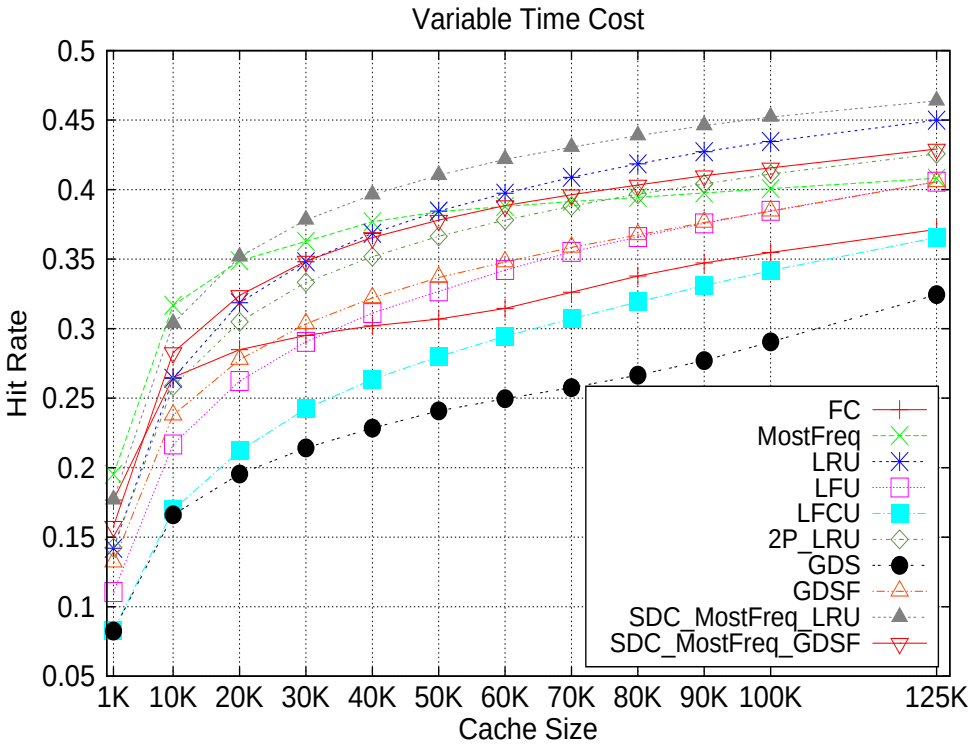

Figure 5.3: Hit rates of caching policies assuming variable query processing time costs.

3.8% and 1% for small and large cache sizes, respectively. The best policies, based on the financial cost metric, namely `GDSF` and `SDC_MostFreq_GDSF`, outperform `LRU` by around 4%.

In the second set of experiments, uniform time costs for queries are assumed. In this case, cost is determined only by the current price rate. Figures 5.4 and 5.5 present the plots for the evaluation of caching strategies based on the financial cost and hit rate metrics, respectively.

Figures 5.4 and 5.5 show the performance for the fixed time cost scenario. For large caches, ordering of policies in decreasing financial cost and increasing hit rate are as follows:

- Financial cost: `LFU` $>$ `MostFreq` $\cong$ `FC` $\cong$ `LFCU` $>$ `GDS` $\cong$ `LRU` $>$ `2P_LRU` $\cong$ `SDC_MostFreq_LRU` $>$ `GDSF` $\cong$ `SDC_MostFreq_GDSF`.
- Hit rate: `LFU` $<$ `MostFreq` $\cong$ `FC` $\cong$ `LFCU` $<$ `GDS` $\cong$ `LRU` $<$ `2P_LRU` $\cong$ `SDC_MostFreq_LRU` $<$ `GDSF` $\cong$ `SDC_MostFreq_GDSF`.

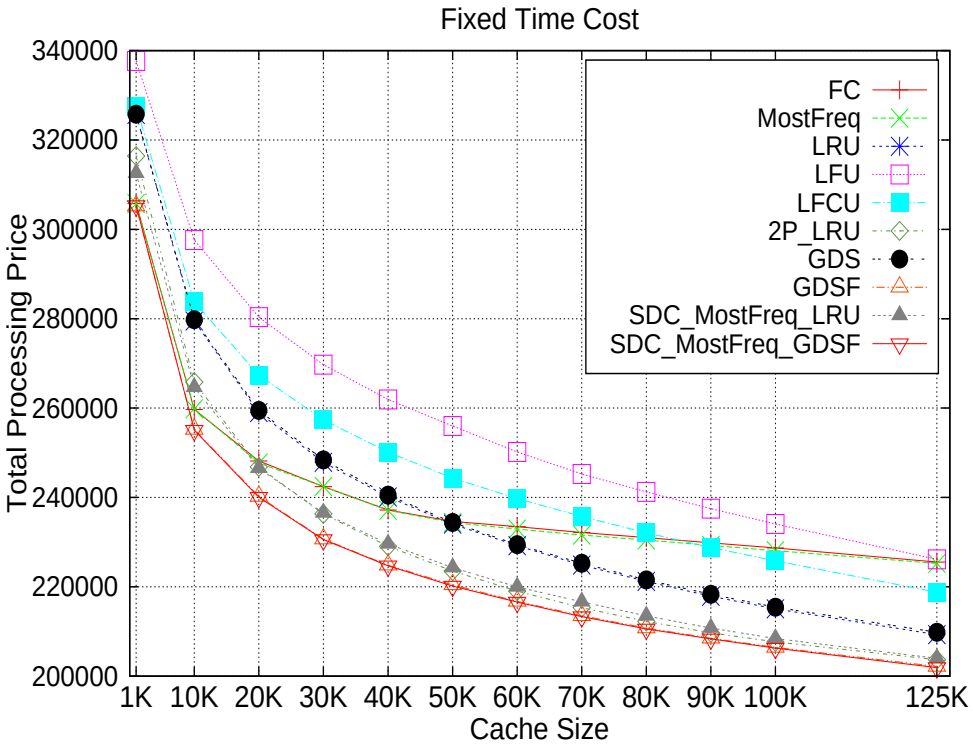

Figure 5.4: Financial cost evaluation of caching policies assuming fixed query processing time costs.

We note that, in this case, the two orderings are the same. This is because, when only the electricity price is considered as the cost, the variation between the costs of different queries is not high. Let $P(q)$ be the probability that query q leads to a cache hit. In this case, the objective function is to minimize the sum of all $(1-P(q))\times C_q$ values, i.e., the cost of all cache misses. If the variation between the C_q values of different queries is not high, then the objective becomes similar to minimizing the $(1 - P(q))$ function, i.e., increasing the hit rate. Therefore, the hit rate and financial cost metrics are highly correlated in this case. In this experiment setting, the proposed `2P_LRU` strategy is only outperformed by hybrid caches and the GDSF strategy.

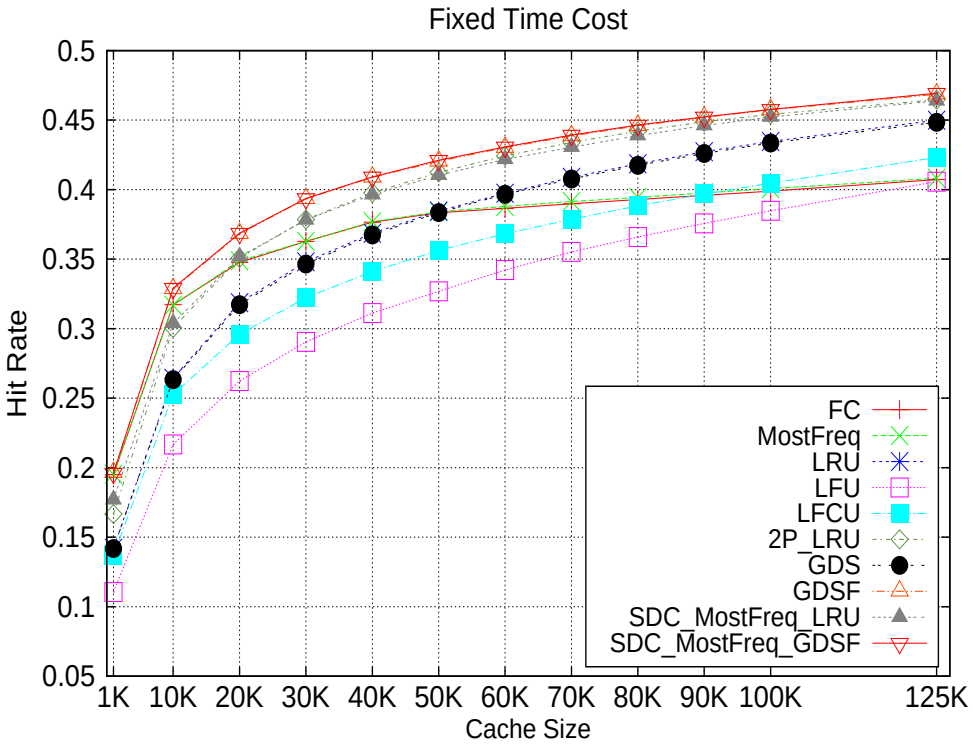

Figure 5.5: Hit rates of caching policies assuming fixed query processing time costs.

5.5 Conclusion

We proposed the financial cost as a new evaluation metric for query result caches in web search engines. We evaluated the well-known static, dynamic, and hybrid result caching strategies using this metric.In general, we observed that the improvement of cost based strategies in terms of the financial cost metric over cost-unaware strategies is more emphasized when there is a sufficient variation between the costs of queries. The proposed financial-cost-aware LRU cache, `2P_LRU`, outperforms the original LRU strategy. For fixed time costs, `2P_LRU` achieves performance results that are very close to the best performing policies, `GDSF` and `SDC_MostFreq_GDSF`. However, in practice, `2P_LRU` could be a better option since only LRU housekeeping mechanisms are performed in 2P_LRU strategy while GDSF strategy requires a priority queue structure that results in higher computational complexity.

Chapter 6

A Financial Cost Approach to Query Freshness in Query Result Caches

6.1 Introduction

The chapters 3 and 4 examined query result freshness in the cache while chapter 5 analyzed a financial cost metric to minimize the financial cost of processing a query stream. Keeping the query results in the cache fresh comes with the financial cost to refresh them regularly. Therefore, the work in this chapter is conducted to delve into the tradeoff between query freshness and query processing cost.

In the previous chapter, it is mentioned that commercial search engines strive to lower their electricity bills which constitutes one of the biggest economical challenge for their commercial entity. On one hand, reducing processing costs by obtaining the results from the cache seems to provide an improvement for their financial operations. On the other hand, this diminishes user satisfaction as the results given by the cache becomes stale when the underlying index is updated due to the changes in the document set. Thus, it is imperative for search engines

to pay regard to this tradeoff while administering result caches.

The evaluation metrics used in previous chapters become useless, as they do not reflect cost and freshness at the same time. For this reason a different metric namely ST and FP cost ratio is employed in this chapter. ST ratio demonstrates the ratio of the queries supplied from the cache whose results are stale, while FP cost ratio is the rate of the cost of the queries which are processed at the backend vainly (if the cache holds the same result as the backend) over total processing cost. FP cost ratio is the cost of redundant query processing of the expiration techniques. In other words, not expiring the queries for these cases will not lead to any performance degradation. In this chapter, two expiration techniques are discussed in terms of their effects to query result freshness and financial cost of processing.

6.2 Query Expiration Techniques

Query expiration techniques in this chapter can be described as lazy methods and eager methods. The lazy methods invalidate a query using some statistics (cache age, frequency, query cost etc.) about a query when the query is issued, and set the value of the parameters such as TTL with respect to those values and current state of the cache, which are explained in the previous chapters in more detail. The eager methods do not wait for the re-submission of a cached query to make expiration and refreshing decisions. They make use of the idle cycles at the query backend to selectively refresh queries. These methods consider user satisfaction more by keeping the query results fresh by exploiting idle cycles.

In their work [6], Cambazoglu et al. selectively refresh queries in the cache at every tic (1 or 2 seconds) to keep the results as fresh as possible and compare their method with the techniques such as TTL and flushing. They conduct experiments for different PST (peak sustainable throughput) values as well as different MRA (minimum refresh age, the minimum cache age for a query to be refreshed) values. Their proposed refreshing technique calculates a target number

of queries to process at each cycle to find the number of queries to refresh at this cycle by subtracting the queries issued at this cycle from the target number. This number has a lower and an upper bound to keep it in a sustained range. This cyclic refresh uses tick latency (the average latency for all queries issued at this cycle), to calculate the target number for the next tick. Since the data used in this chapter do not have processing times of the queries, the tick latencies cannot be calculated, thus constant PST values will be used to bound number of queries processed at each cycle.

The selection of queries to refresh at each cycle is decided by sorting them according to their cache age and frequency. The expired queries are given priority for refreshing and the remaining slot is allocated to queries with higher frequency and cache age. The implementation for this methodology is not given in full detail in the paper and electricity cost is out of scope of the paper. Therefore, the refreshing methodology in this chapter is a slight modification of the methodology mentioned in [6].

6.2.1 Lazy Techniques

Lazy techniques alter TTL values only at the time of the submission of the query to the search engine. Some of the techniques classify the hourly time slots as *expensive* and *cheap*. If the electricity price of a time slot is above the price threshold, that time slot is referred to as expensive time slot, otherwise it is called cheap time slot. Since the electricity prices are available for each hour, each time slot is an hour long.

Time TTL: A constant TTL value is used to expire queries. A query expires if the time it is in the cache exceeds this value.

Time TTL or Frequency TTL (Disjunction): A frequency TTL is augmented to the time TTL, which expires a query when the cache hit count for the query exceeds frequency TTL. In this technique, the query is expired if one of these TTLs expires as explained in chapter 3.

Energy Proactive: The expiration mechanism is similar to the Time TTL or Frequency TTL (Disjunction) technique. There is one more frequency TTL for the projected frequency, which is the estimated frequency of a query submitted in an expensive time slot until the closest cheap time slot. In other words, if a query q is issued in an expensive time slot, at this time the cache age of the query is A seconds, query frequency is F, and the next cheap slot is T_C seconds away, then, the projected frequency until the next cheap time slot is calculated as follows.

$$ProjectedFrequency(q) = \frac{F}{A} \times T_C \tag{6.1}$$

The query is expired if this value exceeds the second frequency TTL or the other TTLs(Time TTL, Frequency TTL) expire.

Energy Separate TTLs: Each hour is labeled as expensive or cheap hour according to a threshold value given as a parameter. Namely, if the price in that hour is greater(smaller) than the threshold, then this hour is labeled expensive(cheap). In this technique, there are two separate time TTL and frequency TTL values for expensive and cheap slots. The motivation is to do the processing in cheap time as much as we can, in order to reduce total processing cost. Therefore, setting high(low) time and frequency TTL values for expensive(cheap) time slots causes most of the query expirations to occur in cheap time.

Energy Skip: This technique labels the query issue time as expensive and cheap using a cost threshold. If the electricity cost in a time slot(an hour in this case) is lower (higher) than the threshold, it is labeled as cheap (expensive) time. In this technique, the queries are expired as in Time TTL or Frequency TTL (Disjunction) method, however if the expiration occurs in an expensive time slot, the refreshing of the query is skipped until it is reissued in a cheap time. That is, no query is expired in an expensive time slot.

Energy GDSF: This technique keeps a score for each query q calculated by using the GDSF formula used in previous chapters. Assume the cache age of the query is L, electricity cost at the time the query issued is C, and query frequency F (the frequency of the query since the last time it is cached), then the score is:

$$Score(q) = L + C \times F \tag{6.2}$$

The query is expired if this score exceeds a threshold given as parameter.

6.2.2 Eager Techniques

This methodology is more proactive in refreshing the queries. The previous lazy methods take action at the time of the submission of the queries to make refresh decisions by taking advantage of the query statistics. However, eager techniques do not wait for cache hit, instead selectively refresh certain number of queries depending on hardware and time constraints. This way, the empty cycles in search engine backend are utilized in order to provide fresh results for future requests.

The eager methods are derived from the refreshing notion mentioned in [6], in which queries are selectively refreshed in constant time periods (*ticks*). The main objective is to take advantage of idle cycles in search engine backend. The algorithm in [6] assumes a peak sustainable throughput (PST) for each tick in which all the queries are served within the query latency constraints. This work sets lower and upper bounds for PST and makes estimations within this range. At each tick, this technique calculates a target query throughput for the next cycle using query latency statistics from the current tick and after the queries issued in this tick are processed, the remaining budget that is calculated by subtracting the number of processed queries from the target number is used to issue refresh queries. The expired queries are given precedence, and the remaining slot is used to refresh most frequent queries that are issued less recently, considering that popular queries with high cache age should be refreshed first.

The approach followed in this work differs from [6] in including electricity price to refresh decisions. Since we do not have a production setting as in [6], we do not use a target throughput for each tick, as this requires monitoring the query latency values at each tick. Instead we use a fixed peak sustainable

throughput(PST) for each tick. First, the issued queries at each tick are processed and then the remaining slot is used for refresh queries. The number of refresh queries is calculated by subtracting the number of issued queries from the fixed PST value. The expired queries are processed first and then the refresh queries are issued. The refresh queries are selected associating a score calculated by using the GDSF formula given in Equation 6.2. The queries are sorted with respect to this score and queries with higher values are refreshed first to give precedence to the popular queries which are issued in hours with higher electricity prices and have higher cache age.

6.3 Experiments

Data. The data used in this chapter are the same as the data as used in Chapters 3 and 4, which are explained in detail in the Appendix. The first half of the data is used for warming the cache and the other half is used for calculating the results.

Setup. The data do not include the processing time for the queries, only electricity cost is taken into consideration when calculating the processing cost. Search engines set an upper bound on the processing times for the queries not to delay the return of the result set and await user for the results. Therefore, it is safe to assume that the variance among the processing time of the queries is not very high and electricity price is adequate for the financial cost of processing a query.

In chapter 3, the combination of time TTL and frequency TTL in disjunction mode was the best performing techniques in terms of stale traffic and false positive ratio and selected as the baseline for lazy techniques. The hourly electricity prices used in the experiments are the same prices from Chapter 5, which are normalized by the average value. The parameters for electricity price threshold starts with 0.7 and ends with 1.3, with regard to these normalized values.

For eager techniques, we present results with normalized electricity prices (uniform cost) as well as for the case where electricity cost is set to 1 (nonuniform

cost). In this case we observe the result of removing the impact of price on selecting which queries to refresh. The time TTL, frequency TTL and various threshold values for the parameters in all techniques are given empirically and the best outcomes are presented in the results, except for the time TTL values for eager case which are based on the values from [6]

Evaluation Metric. We use stale traffic ratio and false positive cost ratio to evaluate the results. Stale traffic ratio captures the freshness while false positive cost ratio captures the cost of redundant calculations. False positive cost ratio is calculated by dividing the cost of false positive cases to the total cost of processing the entire query stream.

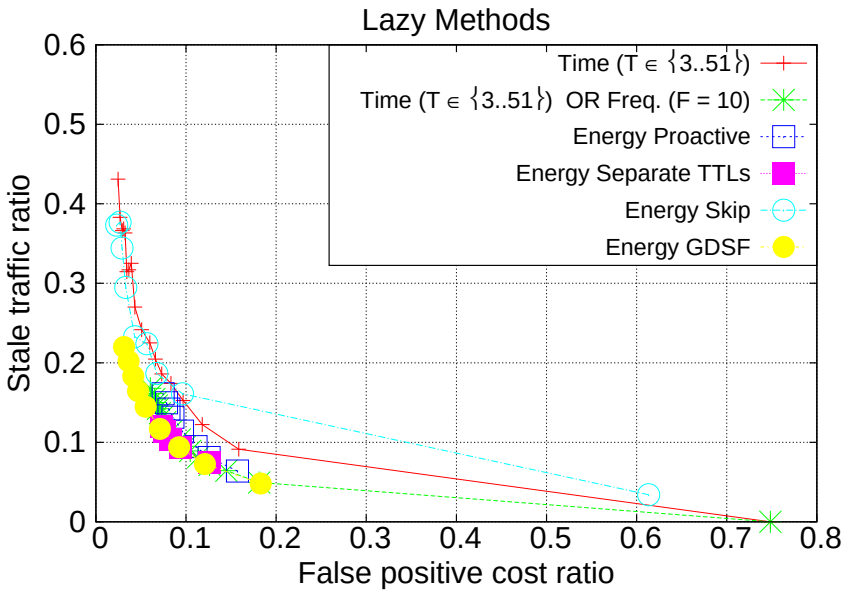

Figure 6.1: Stale traffic and false positive cost ratios for lazy techniques

6.4 Results

The results for lazy methods are given in Figure 6.1. The parameter values for some of the techniques are not included in Figure 6.1 and 6.2 due to insufficient space for legends in the plots and excess of parameters. The values for parameters

for lazy techniques are as follows. In the energy_proactive technique, the cost threshold value is 1.3, frequency TTL value is 20 and the second frequency TTL value 10. In energy separate TTLs technique, for the cheap time slots the time TTL value is 6 and the frequency TTL value is 21; while for the expensive time slot the parameter values for time TTL are 5, 15, 25, 35 and 45, and the frequency TTL is 60. For this case electricity cost threshold value is 0.9. In energy_skip technique, the frequency TTL value is 10 and the cost threshold value is 1.2. For the energy_GDSF technique the parameter values start from 6 and end at 54 with an increment of 6.

Energy Skip and time TTL techniques are the worst performing techniques. Energy Skip technique gives better results than time TTL technique for high TTL values. The false positive cost ratio was expected to be lower as energy skip case does not perform any processing in expensive time slots. However, not expiring queries in expensive time slots results in higher stale traffic ratios. In addition to these techniques, Energy Proactive technique performs worse than the baseline technique Time TTL or Frequency TTL (Disjunction). Energy GDSF method performs the best and energy separate TTLs method performs very close. In energy separate TTLs case the time TTL and frequency TTL values for expensive time slot are significantly higher when compared to the values for cheap time slot. This reduces processing cost by conducting the majority of the query expirations in cheap time slots. Considering computational overhead of energy GSDF method to reorder query scores at each score update, energy separate TTLs seems to be a better alternative for our baseline.

For the eager techniques, since the number of queries is relatively low, experimenting with various PST values did not give much different results. Therefore, results for different PST values are only presented for time TTL case. Figure 6.3 shows average cache age values for different TTL values. As expected, average age increases as the TTL value increases. For TTL values of 6, 12 and 18, average age values oscillate around 2.5, 5 and 7, respectively. Average cost values are demonstrated in Figure 6.4, and the values decrease as TTL increases and the results show correlation with expensive and cheap time slots.

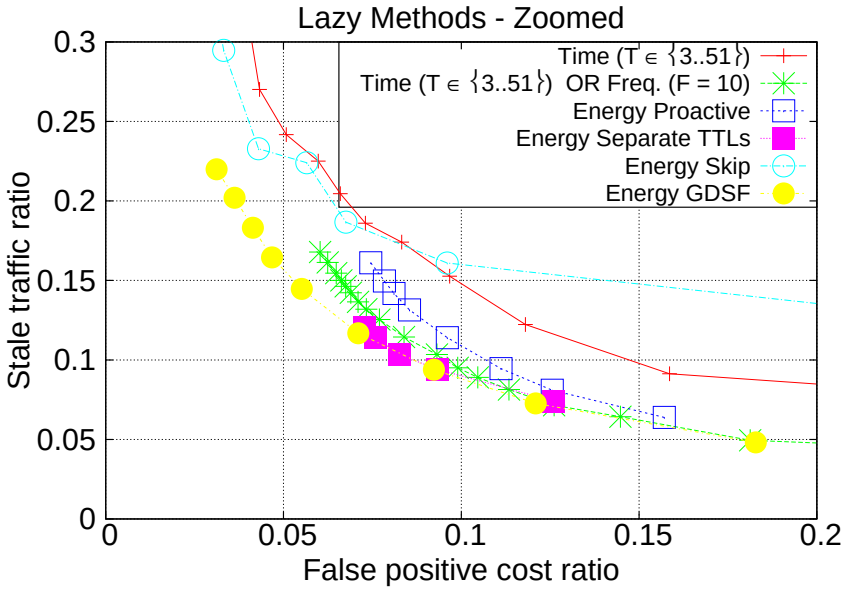

Figure 6.2: Stale traffic and false positive cost ratios for lazy techniques - zoomed

Figures 6.5 and 6.6 show the average age and processing cost for eager techniques. The average age values are very close in uniform and nonuniform cost cases. It is probably because the frequency part of the score formula dominates the cost. Therefore, when refresh queries are issued, similar set of queries are selected by both methods. Figure 6.6 shows that the average processing cost for variable cost case oscillates as in Figure 6.4 and converges to 1, but the values in Figure 6.4 are lower.

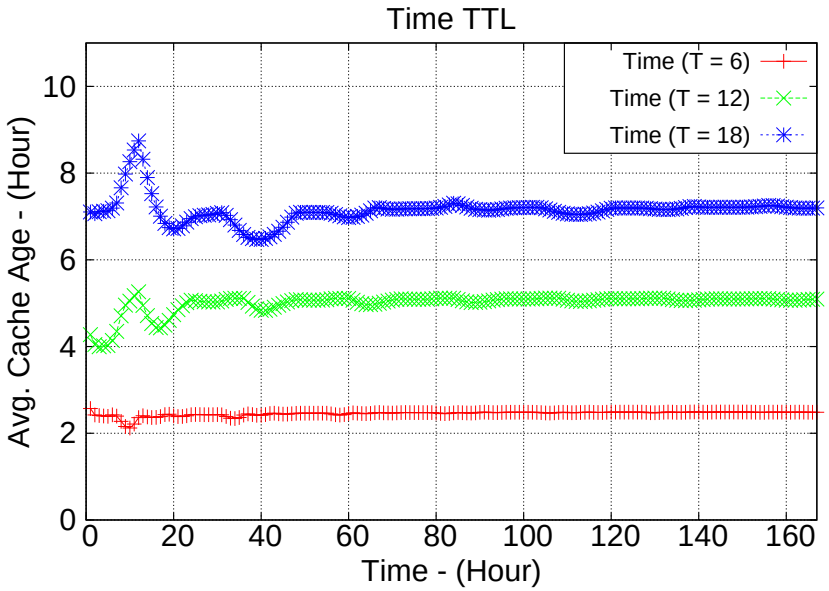

Figure 6.3: Average age over a week for Time TTL

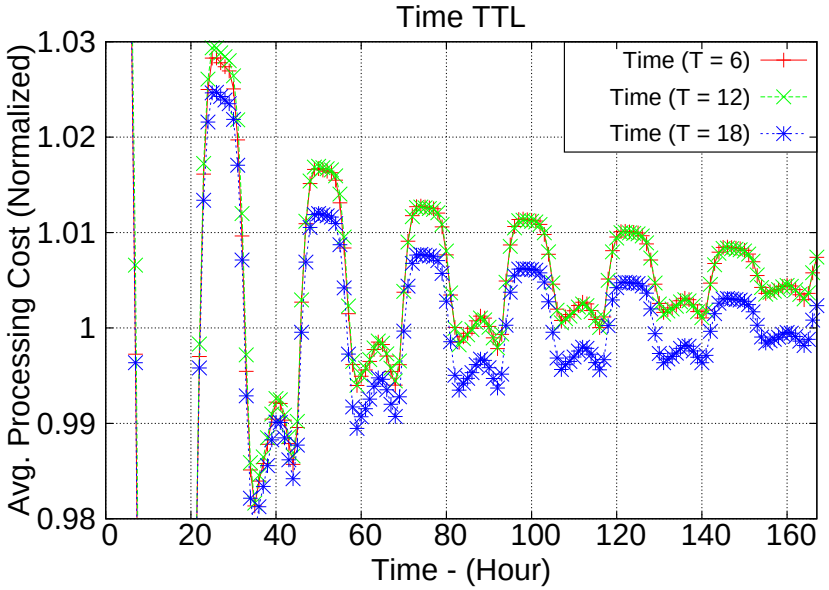

Figure 6.4: Average processing cost over a week for Time TTL

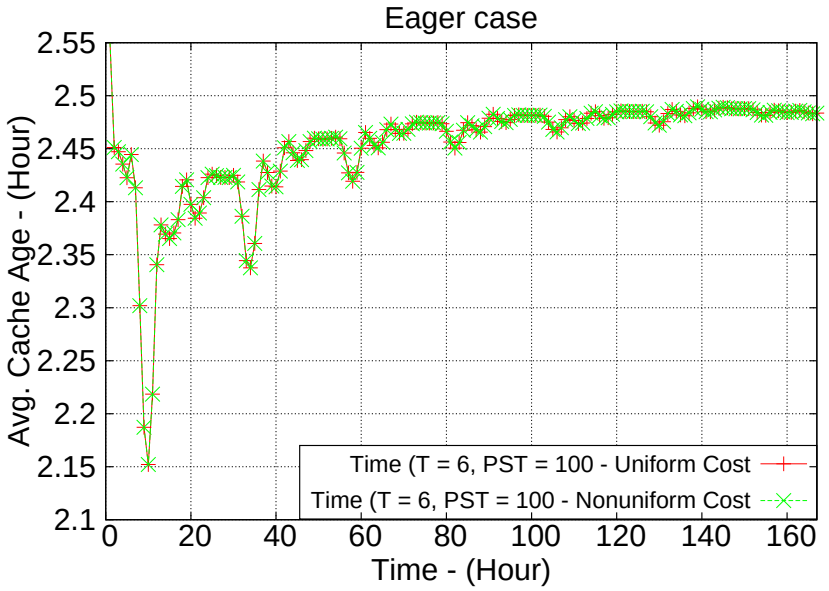

Figure 6.5: Average age over a week for the eager case - refreshing with uniform and nonuniform cost

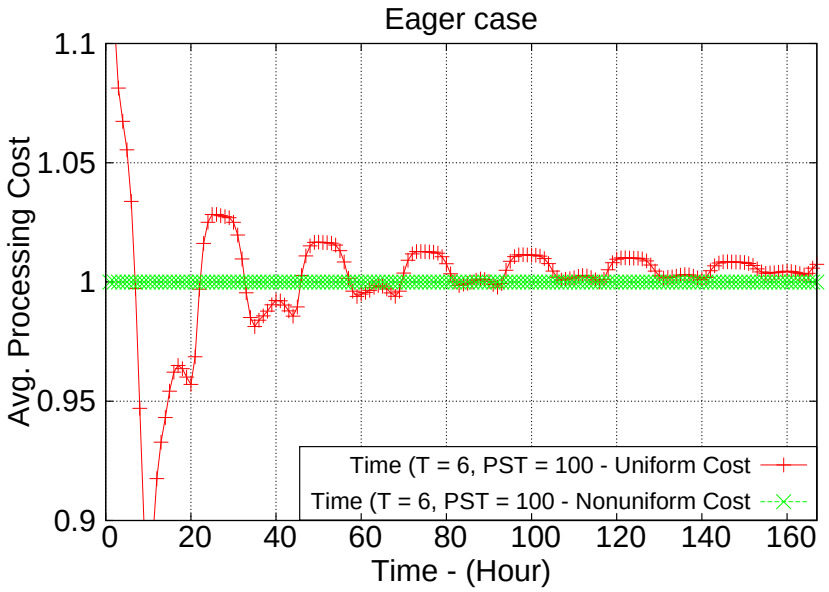

Figure 6.6: Average processing cost over a week for the eager case - refreshing with uniform and nonuniform cost

6.5 Conclusion

We examined the tradeoff between cost and freshness in result caching. Lazy and eager methodologies have been evaluated. The experiments demonstrated that for lazy techniques, even keeping separate TTL values for cheap and expensive time slots can improve cache performance. The best lazy technique was energy GDSF technique which combines cache age, frequency and cost in query scores and uses this to expire queries. These techniques performed better than the baseline technique which is time or frequency TTL (Disjunction) case explained in Chapter 3. However, the eager case did not show much improvement. The main reason for this result is the number of queries in the query log and the low variance of electricity costs. The limited number of queries results in limited number of refresh queries to be issued. Since the query age and frequency dominates the selection of refresh queries, the set of refresh queries become quite similar, which yields the results in this chapter.

Chapter 7

Conclusion

The overall performance of a search engine is impacted by metrics such as operational costs, financial costs and user satisfaction. The dependencies among this metrics have to be taken into consideration while developing the search engine. Improving user satisfaction is favourable for the search engine, but it has adverse effects on operational and financial costs. Thus, the search engines must be cautious in determining the correct measures for identifying its performance, and such a measure should not leave out any metric.

We both discussed the techniques to improve search engine performance and investigated the evaluation criteria to meet the requirements of each technique. We demonstrated that making use of the correct subset of query statistics in invalidating cache entries results in fast and accurate query results with minimal overload on storage and processing power. Although maximizing cache hit rate is perceived as decreased load on backend and increased query throughput, the user satisfaction component of the search engine performance criteria is negatively affected by this. Therefore, we examined various time-to-live mechanisms to maximize cache hit rate without returning results that contradicts with the current index. In addition to query specific feedback, other queries in the cache are taken into consideration to correctly invalidate a cache entry.

The latter part of the thesis investigated impacts of considering financial cost

for result caching. A new financial cost metric was proposed to contain the cost component. A memory friendly novel algorithm was compared with a state of the art algorithm that considers computation power and concluded that low processing costs can be obtained at the expense of some processing power which poses another contradiction among the metrics for overall performance of the search engine.

The last part of the thesis examined the tradeoff between processing cost and query freshness. Lazy and eager techniques were evaluated to find out how the processing cost can be minimized without affecting query freshness. The common feature of all the chapters is that they make use of the statistics in the query log. Even experiments including click data of the queries were conducted. From several query statistics in the query log it is crucial to combine them in the right way. We experimented with different combinations to find out the best one.

As a future work, we plan to further extend our analysis on the query log to find more feedback to invalidate cache entries, because invalidating cache entries without peeking at the index may not provide the most accurate feedback. Therefore, it becomes imperative to maximize the feedback from query log in order to selectively invalidate cache entries without too much processing overhead.

Bibliography

[1] E. P. Markatos, "On caching search engine query results," *Computer Communications*, vol. 24, no. 2, 2001.

[2] T. Fagni, R. Perego, F. Silvestri, and S. Orlando, "Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data," *ACM Transactions on Information Systems*, vol. 24, pp. 51–78, Jan. 2006.

[3] P. C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto, "Rank-preserving two-level caching for scalable search engines," in *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, (New York, NY, USA), pp. 51–58, ACM, 2001.

[4] R. Baeza-Yates and F. Saint-Jean, "A three level search engine index based in query log distribution," in *String Processing and Information Retrieval* (M. Nascimento, E. de Moura, and A. Oliveira, eds.), vol. 2857 of *Lecture Notes in Computer Science*, pp. 56–65, Springer Berlin Heidelberg, 2003.

[5] X. Long and T. Suel, "Three-level caching for efficient query processing in large web search engines," in *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, (New York, NY, USA), pp. 257–266, ACM, 2005.

[6] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge, "A refreshing perspective of search engine caching,"

in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, (New York, NY, USA), pp. 181–190, ACM, 2010.

[7] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. F. Witschel, "Admission policies for caches of search engine results," in *Proceedings of the 14th International Conference on String Processing and Information Retrieval*, SPIRE'07, (Berlin, Heidelberg), pp. 74–85, Springer-Verlag, 2007.

[8] Q. Gan and T. Suel, "Improved techniques for result caching in web search engines," in *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, (New York, NY, USA), pp. 431–440, ACM, 2009.

[9] R. Lempel and S. Moran, "Predictive caching and prefetching of query results in search engines," in *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, (New York, NY, USA), pp. 19–28, ACM, 2003.

[10] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy, "Adaptive time-to-live strategies for query result caching in web search engines," in *Proceedings of the 34th European Conference on Advances in Information Retrieval*, ECIR'12, (Berlin, Heidelberg), pp. 401–412, Springer-Verlag, 2012.

[11] S. Jonassen, B. B. Cambazoglu, and F. Silvestri, "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*, SIGIR '12, (New York, NY, USA), pp. 631–640, ACM, 2012.

[12] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy, "Timestamp-based result cache invalidation for web search engines," in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, (New York, NY, USA), pp. 973–982, ACM, 2011.

[13] X. Bai and F. P. Junqueira, "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*, SIGIR '12, (New York, NY, USA), pp. 641–650, ACM, 2012.

[14] R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza, "Caching search engine results over incremental indices," in *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, (New York, NY, USA), pp. 82–89, ACM, 2010.

[15] E. Bortnikov, R. Lempel, and K. Vornovitsky, "Caching for realtime search," in *Proceedings of the 33rd European Conference on Advances in Information Retrieval*, ECIR'11, (Berlin, Heidelberg), pp. 104–116, Springer-Verlag, 2011.

[16] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, (New York, NY, USA), pp. 183–190, ACM, 2007.

[17] R. Ozcan, I. S. Altingovde, and O. Ulusoy, "Cost-aware strategies for query result caching in web search engines," *ACM Transactions on the Web*, vol. 5, pp. 9:1–9:25, May 2011.

[18] I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy, "Second chance: a hybrid approach for dynamic result caching in search engines," in *Proceedings of the 33rd European Conference on Advances in Information Retrieval*, ECIR'11, (Berlin, Heidelberg), pp. 510–516, Springer-Verlag, 2011.

[19] A. Barroso Luiz and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2009.

[20] E. Kayaaslan, B. B. Cambazoglu, R. Blanco, F. P. Junqueira, and C. Aykanat, "Energy-price-driven query processing in multi-center web

search engines," in *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, SIGIR '11, (New York, NY, USA), pp. 983–992, ACM, 2011.

[21] M. Marin, V. Gil-Costa, and C. Gomez-Pantoja, "New caching techniques for web search engines," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), pp. 215–226, ACM, 2010.

[22] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs, "Cutting the electric bill for internet-scale systems," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, (New York, NY, USA), pp. 123–134, ACM, 2009.

[23] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 1997.

[24] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *SIGMETRICS Performance Evaluation Review*, vol. 27, pp. 3–11, Mar. 2000.

[25] G. Pass, A. Chowdhury, and C. Torgeson, "A picture of search," in *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, (New York, NY, USA), ACM, 2006.

Appendix A

Query Log Data

Strategies for setting TTL values for the methods described in Chapters 3, 4 and 6 use the same query log data. The subset of a query log of a commercial search engine is employed for the simulation of the search engine with the settings explained in the respective chapters. The subset is generated by applying some heuristics that are coherent with the nature of the experimental setting. The heuristics are as follows:

- Only the first page of the query results are included. The query results are compared to each other for stale ratio calculation purposes and to make this comparison correctly only the first pages are considered (most of the query result pages are 1).
- Page size for the results should be 10. Page sizes other than 10 make query result comparison more complicated. When comparing query results, different page sizes are counted as different results without further consideration.
- Number of matching results of a query should be at least half of the number of matching results returned to the last occurrence of the same query (by same query, it is suggested that the query terms are the same). When the total number of matching results drops dramatically, it usually implies that not all the clusters are able to contribute to the final result set, possibly

due to the limitations on the query response time. In such a situation, comparing top 10 query results of this case with a normal case does not produce healthy results for the simulations conducted. Thus, this heuristic is applied to eliminate this case.

- User location and data center are the same across the whole query log. Again, this is for the sake of the comparison of query results. Different data centers and user locations contribute to the customisation of query results, which change the result of the same query terms making the comparison of their results harder.

- For the query terms, the query normalized by the search engine is employed. This is safer, since user query terms may include some typo which obstruct distinguishing queries.

The queries submitted to Spanish front-end of the commercial search engine are taken for the experiments. This constitutes to a set of 2,044,531 queries in timestamp order, when the heuristics above are applied to the query log. The first half of the queries are used as the training set (i.e., to warm-up the cache) and remaining half is used as the test set. In the experiments, in addition to using this entire query stream, a more detailed performance analysis for the head and tail queries are also provided. To this end, all unique queries in the query set are sorted by their submission frequencies, and label those in top-1% and bottom-90% as head and tail queries, respectively; and the rest as torso. Then the query streams that only include these identified queries are constructed using the corresponding labels. As before, the stream is split as 50/50 for training and testing.