CACHING TECHNIQUES FOR LARGE SCALE WEB SEARCH ENGINES

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING
AND SCIENCE OF BİLKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Rıfat Özcan

September, 2011

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

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 dissertation for the degree of doctor of philosophy.

Prof. Dr. Fazlı Can

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

Assoc. 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 dissertation for the degree of doctor of philosophy.

Prof. Dr. Adnan Yazıcı

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

Prof. Dr. Enis Çetin

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of Graduate School of Engineering and Science

ABSTRACT

CACHING TECHNIQUES FOR LARGE SCALE WEB SEARCH ENGINES

Rıfat Özcan
Ph.D. in Computer Engineering
Supervisor: Prof. Dr. Özgür Ulusoy
September, 2011

Large scale search engines have to cope with increasing volume of web content and increasing number of query requests each day. Caching of query results is one of the crucial methods that can increase the throughput of the system. In this thesis, we propose a variety of methods to increase the efficiency of caching for search engines.

We first provide cost-aware policies for both static and dynamic query result caches. We show that queries have significantly varying costs and processing cost of a query is not proportional to its frequency (popularity). Based on this observation, we develop caching policies that take the query cost into consideration in addition to frequency, while deciding which items to cache. Second, we propose a query intent aware caching scheme such that navigational queries are identified and cached differently from other queries. Query results are cached and presented in terms of pages, which typically includes 10 results each. In navigational queries, the aim is to reach a particular web site which would be typically listed at the top ranks by the search engine, if found. We argue that caching and presenting the results of navigational queries in this 10-per-page manner is not cost effective and thus we propose alternative result presentation models and investigate the effect of these models on caching performance. Third, we propose a cluster based storage model for query results in a static cache. Queries with common result documents are clustered using single link clustering algorithm. We provide a compact storage model for those clusters by exploiting the overlap in query results. Finally, a five-level static cache that consists of all cacheable data items (query results, part of index, and document contents) in a search engine setting is presented. A greedy method is developed to determine which items to cache. This method prioritizes items for caching based on gains computed using items' past frequency, estimated costs, and storage overheads. This approach also considers the inter-dependency between items such that caching of an item may affect the gain of items that are not cached yet.

We experimentally evaluate all our methods using a real query log and document collections. We provide comparisons to corresponding baseline methods in the literature and we present improvements in terms of throughput, number of cache misses, and storage overhead of query results.

Keywords: Search engine, caching techniques, cost-aware caching, navigational queries.

ÖZET

BÜYÜK ÖLÇEKLİ ARAMA MOTORLARINDA ÖNBELLEKLEME TEKNİKLERİ

Rıfat Özcan
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Prof. Dr. Özgür Ulusoy
Eylül, 2011

Büyük ölçekli arama motorları artan ağ içeriği ve artan günlük sorgu sayısı ile mücadele etmek zorundadırlar. Sorgu cevaplarının önbelleklenmesi ise sistemin belli bir zamanda sorgu cevaplama sayısını arttırabileceği kritik yöntemlerden birisidir. Bu tezde, arama motorlarında önbelleklemenin verimliliğini arttırıcı çeşitli yöntemler önerilmektedir.

İlk olarak, statik ve dinamik sorgu cevabı önbellekleri için maliyet bazlı önbellekleme yöntemleri geliştirilmiştir. Sorguların ciddi oranda farklı maliyetlerinin olduğu ve bu maliyet ile frekans arasında doğru orantı olmadığı gözlemlenmiştir. Bu nedenle önbelleğe hangi öğelerin alınacağına karar verilirken frekansa ek olarak sorgu maliyetini de göz önüne alan yöntemler tasarlanmıştır. İkinci olarak, navigasyonel sorguların tanımlanarak diğer sorgulardan farklı olarak önbelleklendiği bir sorgu tipi bazlı önbellekleme yöntemi geliştirilmiştir. Sorgu cevapları genellikle her biri 10 cevap içeren sonuç sayfaları olarak sunulmakta ve önbellekte saklanmaktadır. Navigasyonel sorgularda amaç belirli bir ağ sayfasına ulaşmaktır ve bu sayfa arama motoru tarafından bulunursa yüksek sıralarda listelenmektedir. Bu sorgu tipi için cevapların bir sayfada 10 cevap olacak şekilde sunulup önbellekte saklanmasının maliyet etkinliği olmayan bir yöntem olduğu gösterilmiş ve alternatif cevap sunum modelleri önerilerek bunların önbellekleme üzerindeki etkisi araştırılmıştır. Üçüncü olarak, statik önbellekte sorgu cevapları için kümeleme bazlı bir saklama yöntemi önerilmiştir. Ortak cevap belgesi olan sorgular tek bağlantı yöntemi ile kümelenmiştir. Oluşan kümelerdeki sorgu cevaplarındaki örtüşmeden yararlanan kompakt bir saklama modeli sunulmuştur. Son olarak, bir arama motoru ortamında önbelleklenebilecek tüm veri öğelerini (sorgu cevapları, endeks parçası ve belge içeriği) içerisinde barındıran beş-seviyeli bir statik önbellek önerilmiştir. Öğelerin hangi sırayla önbelleğe alınacağını belirlemek için bir açgözlü algoritma geliştirilmiştir. Bu yöntem önbelleğe alınmada öğeleri geçmiş frekans değerleri, öngörülen maliyet ve önbellekte kaplayacağı alan açısından önceliklendirmektedir. Ayrıca öğeler arasındaki bağımlılık göz önüne alınarak bir öğenin önbelleğe alınmasından sonra henüz önbelleğe alınmamış bağımlı öğelerin kazanç değerleri değiştirilmektedir.

Önerilen bütün yöntemler gerçek sorgu kütüğü ve belge kolleksiyonu kullanan deneylerle test edilmiştir. Literatürde karşılık gelen referans yöntemler ile karşılaştırmalar sunulmuş ve üretilen iş, sorgu ıskalama sayısı ve sorgu cevaplarının kapladığı alan bazında gelişmeler elde edilmiştir.

Anahtar sözcükler: Arama motoru, önbellekleme teknikleri, maliyet-bazlı önbellekleme, navigasyonel sorgular.

Acknowledgement

I would like to express my deepest thanks and gratitude to my supervisor Prof. Dr. Özgür Ulusoy for his invaluable suggestions, support, guidance and patience during this research.

I would like to thank all committee members for spending their time and effort to read and comment on my thesis.

I would like to thank Dr. İsmail Sengör Altıngövde for his support and guidance during this research. Furthermore, I also thank to my colleagues Dr. Berkant Barla Cambazoğlu and Şadiye Alıcı.

I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting my PhD.

Finally, I would like to thank my family.

Contents

List of Figures

List of Tables

List of Publications

This dissertation is based on the following publications.

[Publication-I] R. Ozcan and I. S. Altingovde and O. Ulusoy, "*Cost-aware strategies for query result caching in web search engines,*" ACM Transactions on the Web, Vol. 5, No. 2, Article 9, 2011.

[Publication-II] R. Ozcan and I. S. Altingovde and O. Ulusoy, "*Exploiting navigational queries for result presentation and caching in web search engines,*" Journal of the American Society for Information Science and Technology, Vol. 62, No. 4, 714-726, 2011.

[Publication-III] R. Ozcan and I. S. Altingovde and B. B. Cambazoglu and F. P. Junqueira and O. Ulusoy, "*A five-level static cache architecture for web search engines,*" Information Processing & Management, In Press.

[Publication-IV] R. Ozcan and I. S. Altingovde and O. Ulusoy, "*Static query result caching revisited,*" In Proceedings of the 17th International Conference on World Wide Web, ACM, New York, NY, 1169-1170, 2008.

[Publication-V] R. Ozcan and I. S. Altingovde and O. Ulusoy, "*Space efficient caching of query results in search engines,*" In Proceedings of the 23rd Int. Symposium on Computer and Information Sciences, Istanbul, Turkey, 1-6, 2008.

[Publication-VI] R. Ozcan and I. S. Altingovde and O. Ulusoy, "*Utilization of navigational queries for result presentation and caching in search engines,*" In Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM), Napa Valley, California, USA, 1499-1500, 2008.

Chapter 1

Introduction

1.1 Motivation

The Internet has become the largest and most diverse source of knowledge (information) in the world. In the early days, Internet was covering static homepages with mostly textual information but today it is much more dynamic, reaching to tens of billion web pages[1], and it includes a wide range different web sites such as giant enterprise web sites, multimedia (image, audio, video) sharing web sites, blogs and social web sites. It is a challenge to find the web pages that contain relevant information from this huge amount of data. At this point, large scale search engines accept this challenge and try to answer millions of query requests each day. For instance, Google, one of the largest search engines for the time being, receives several hundred million queries per day[2]. Search engines have to be efficient in order to respond to each user within seconds and effective so that returned results are relevant to information needs of users.

Search engines crawl the web periodically in order to obtain the latest possible content of the web. Thousands of computers are needed to store and process such a collection. Next, an index structure is built on top of the crawled document

[1] http://www.worldwidewebsize.com/

[2] http://en.wikipedia.org/wiki/Google_Search

collection. It is shown that inverted index file is the state-of-the-art data structure for efficient retrieval [93]. Finally, queries are processed over the inverted index using a retrieval (ranking) function that computes scores for documents based on their relevance to the query and documents with highest scores are returned as the query result. Additionally, result pages containing title, url and snippet (i.e., two or three sentence summary of the document) information are prepared and displayed.

Query processing over a large inverted index file that is partitioned at thousands of computers is a complex operation. In the first stage of processing, a query must be forwarded to index servers containing the partial inverted index files since it is not possible to accommodate the full index file in one server. Each index server must access the disk to fetch the relevant parts of the index for the query terms. A retrieval (ranking) algorithm has to be executed in order to find the most relevant documents to the query. Finally, results from each index server are aggregated and the result page is prepared. This requires access to the contents of the documents.

A further challenge taken by large scale search engines is to perform the query processing task in a very short period of time, almost instantly. To this end, search engines employ a number of key mechanisms to cope with these efficiency and scalability challenges. Caching, as applied in different areas of computer science [29, 68, 75] successfully over many years, is one of these vital methods to cope with these demanding requirements. The fundamental principle in caching is to store items that will be requested in the near future. It is observed that some popular queries are asked by many users and query requests have temporal locality property, that means, a significant amount of queries submitted previously are submitted again in the near future. This shows an evidence for caching potential for query results [54]. Therefore, query results can be stored in a reserved cache space and queries can be answered from the cache without requiring any processing. In addition to query results, parts of the inverted index structure and contents of documents can also be cached. Caching in the context of large scale search engines has become an important and popular research problem in recent years. In this thesis, we propose efficient strategies to improve the efficiency of caching.

1.2 Contributions

In this thesis, we focus our attention on query result caching in large scale search engines. Our basic contribution is to develop cost-aware and query intent aware caching policies. We first propose cost-aware static and dynamic caching methods that take the cost of queries into account while deciding which queries to cache. Next, we deal with navigational queries (i.e., queries where the user is searching for a website) and propose result presentation models for this query intent. We analyze the impact of these models on the performance of query result caching. We also contribute by proposing an efficient storage mechanism for query results by clustering queries that have similar results. Finally, we provide a five-level static cache architecture that considers different types of items (query results, inverted index, document content, etc.) to cache at the same time using a greedy approach. In the following paragraphs, we provide the details of our contributions together with the organization of the thesis.

In Chapter 2, we provide the background information about large scale web search engines and caching. We present a typical architecture of a search engine and give details about query processing. Later, we describe different caching components incorporated in this architecture to increase the efficiency. The literature work on caching in the context of search engines is also given in this chapter.

In Chapter 3 (based on [64]), we propose cost-aware caching policies. We first show that query processing costs may significantly vary among different queries and the processing cost of a query is not proportional to its popularity. Based on this observation, we propose to explicitly incorporate the query costs into the caching policies. Our experiments using two large web crawl datasets and a real query log reveal that the proposed approach improves overall system performance in terms of the average query execution time.

In Chapter 4 (based on [65]), we propose result presentation models for navigational queries and analyze the effect of these models on caching. With navigational queries (e.g, “united airlines”) users try to reach a website and result browsing behavior is very different from the case for informational queries (e.g, “caching in web search engines”) where the user seeks for information about a topic. We propose metrics for evaluation of result presentation models. We then analyze the effects of result page models on the performance of caching. We also conduct a user study in order to investigate the user browsing behavior on different result page models.

In Chapter 5 (based on [61]), we propose a storage model for document identifiers of query results in a cache. Queries with common result documents are clustered. We propose a compact representation for these clusters.

In Chapter 6 (based on [60]), we propose a five-level static cache that consists of different items such as query results, inverted index and document contents. A greedy approach that prioritizes the items using a cost oriented method for caching is provided. The inter-dependency between items when making cache decisions is also considered in this approach.

Finally, we conclude the thesis and mention some future work directions about caching in search engines in Chapter 7.

Chapter 2

Background and Related Work

In this chapter, we provide background information for a large scale search engine and present related work in the literature about caching mechanisms applied in these systems. We first present a typical architecture of a search engine and give information about query processing in Section 2.1. Different cache types (result, posting list, document) employed in a large scale search engine together with literature work on these caches are provided in Section 2.2. Since the majority of work in this thesis focuses on the result caches, we further review result cache filling, evaluation, and freshness issues in more depth in Sections 2.3, 2.4, and 2.5, respectively.

2.1 Large Scale Web Search Engines

Large scale web search engines like Google, Yahoo!, and Bing answer millions of queries from all over the world each day. Each query is answered within one or two seconds and the search service provided by them is available at all times. This operation conditions search engines to take extra measures for efficiency, effectiveness, fault tolerance, and availability. Figure 2.1 shows a typical top-down architecture for a large scale web search engine consisting of geographically distributed search clusters [11]. Each *search cluster* contains a replica of web

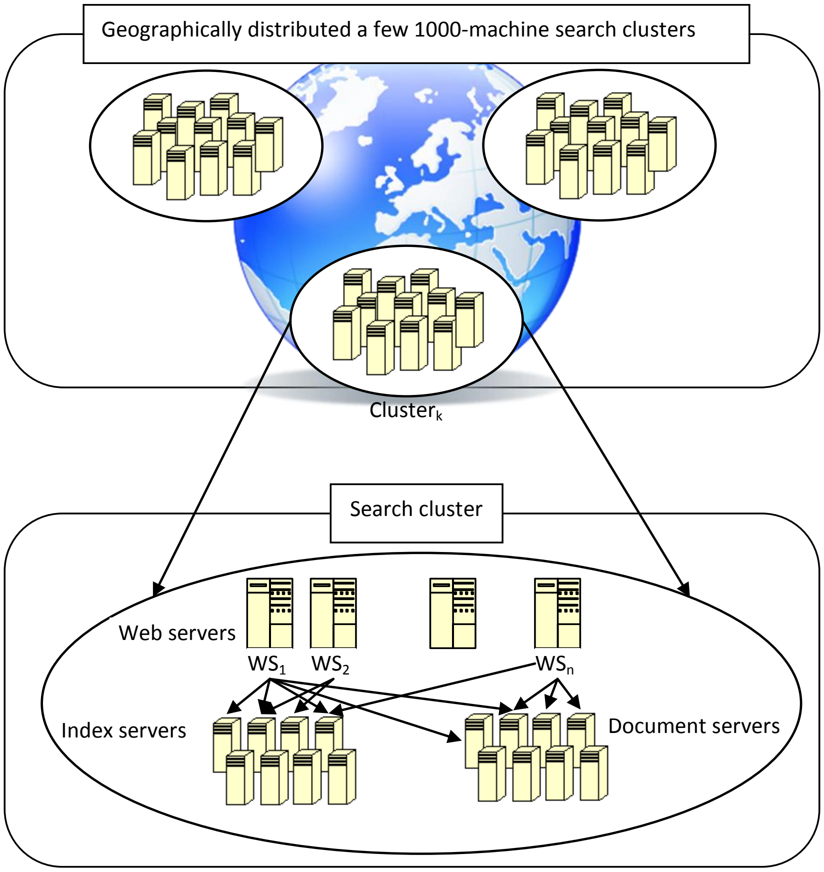


Figure 2.1: Large scale search engine consisting of geographically distributed search clusters.

crawl and operates a few thousands nodes (computers). Each user query submitted through the web page interface of the search engine is directed to the nearest search cluster through DNS (Domain Name Service) mechanism. Each search cluster contains nodes for web servers, index servers, document servers, and additionally, ad servers [11].

Web servers handle the query requests by communicating to the index servers and document servers. All documents in a web crawl are partitoned among document and index servers. As shown in Figure 2.2, each index server contains a portion of the full inverted index for the web crawl. Each document server store a subset of the crawled web pages. Web server (query broker) nodes send

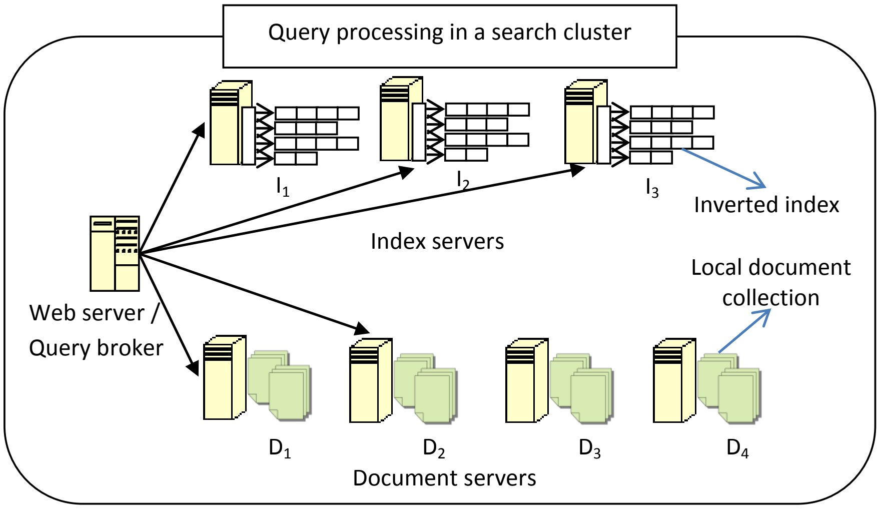

Query processing in a search cluster
I_1
I_2
I_3
Inverted index
Index servers
Local document collection
Web server / Query broker
D_1
D_2
D_3
D_4
Document servers

Figure 2.2: Query processing in a search cluster.

query requests to index servers. Each index server processes the query using the inverted index (and any extra available information), applies a ranking (scoring) algorithm to sort the documents based on their relevance to the query, and forms the top-k list. An example inverted index for the three documents is shown in Figure 2.3. Vocabulary data structure contains the list of terms contained in the document collection. It also contains pointers to the location of posting lists for each term. Index part consists of term posting lists. Each posting list contains the document ids (additionally, frequency and/or position of the term in the document) which belong to the document that contains this term. Inverted index construction requires several additional steps like tokenization, stopword removal and stemming (for details, please see [87, 93]).

2.2 Caching in Web Search Engines

Caching is a mechanism to store items that will be requested in the near future. It is a well-known technique applied in several domains, such as disk page caching in operating systems [75, 77], databases [29], and web page caching in proxies [68]. Its main goal is to exploit the temporal and spatial localities of the requests in

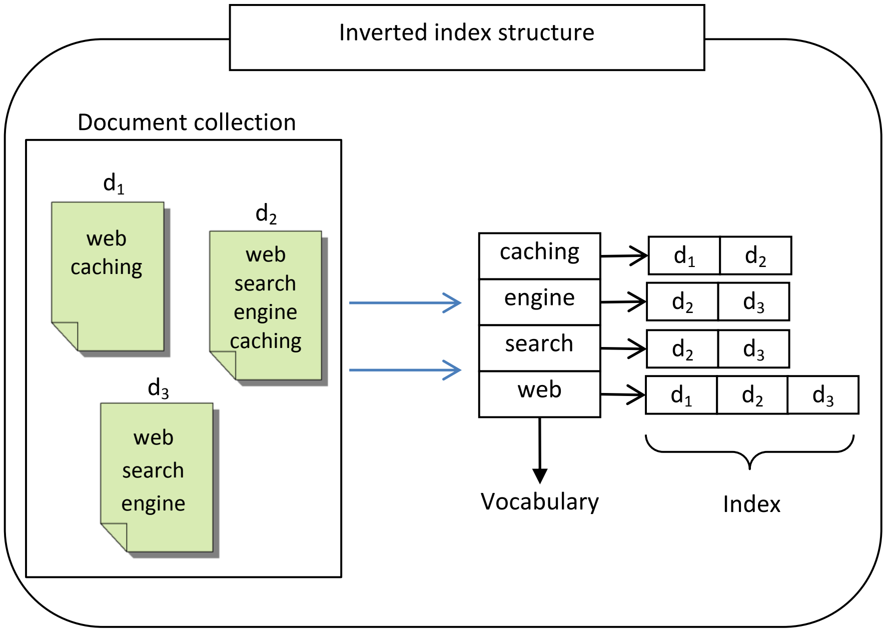


Figure 2.3: Inverted index data structure.

the task at hand. Operating systems attempt to fill the cache with frequently (and/or recently) used disk pages and expect them to be again requested in the near future.

Caching is one of the key techniques search engines use to cope with high query loads. Figure 2.4 shows different cache types exploited in a search cluster. Three different types of items can be cached: query results, posting lists, and document content. Typically, search engines cache query results (result cache) [54, 30, 47, 61] in the query broker machine or posting lists for query terms (list cache) in the index servers, or both [10, 51, 6, 7]. Query results can be cached in two different formats: *HTML result cache* and *docID (or Score) result cache*. HTML result cache stores the complete (ready to be sent to the user) result pages. A *result page* contains links and titles of usually 10 or more result documents, and snippets. *Snippet* is a small textual portion of the full result document related to the query. DocID (or Score) result cache stores only the result document ids of queries. Even though this representation is very compact for storing, an additional step of snippet generation is needed. Caches containing posting lists

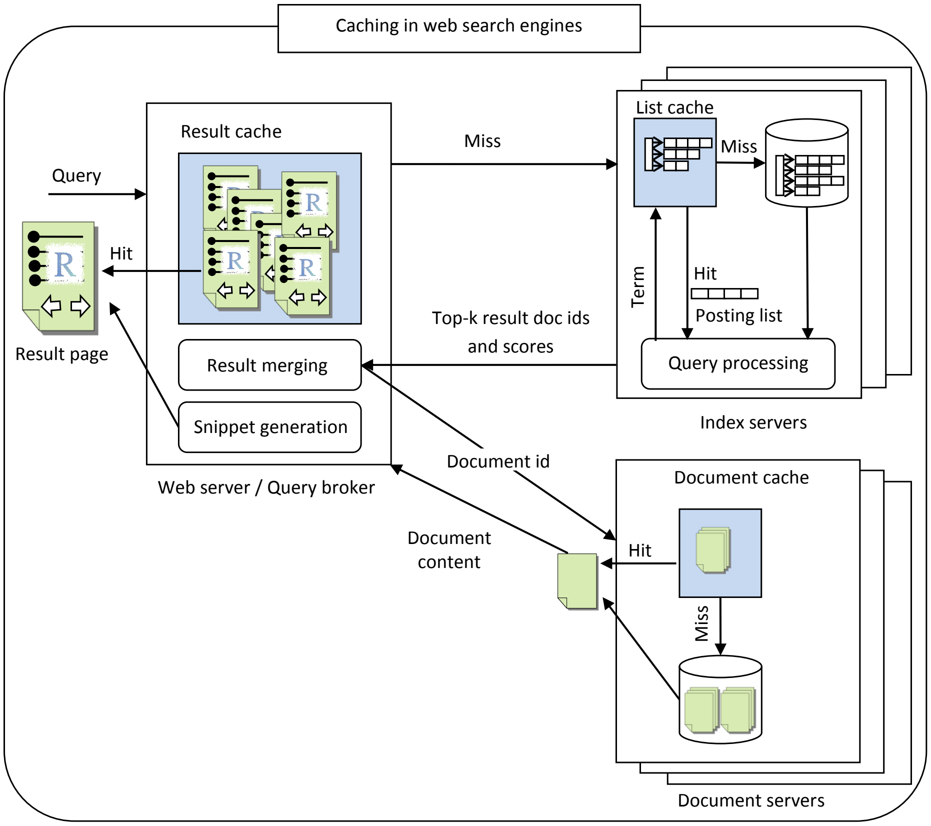


Figure 2.4: Caching in a large scale search engine architecture.

can be categorized into two: List and intersection caches. *List cache* stores the full (or a portion of) posting list of terms. *Intersection cache* [51] includes common postings of frequently occuring pairs of terms. Additionally, document servers can cache the frequently accessed document content (document cache).

Query processing with caching in search engine proceeds as follows: When a query is submitted to the query broker node, result cache must be checked first. If the result of the submitted query is already stored in the HTML result cache, then there is no need for further processing and it can be sent to the user. If the DocID cache is employed and the submitted query result is found in the cache, then query broker machine contacts to the document servers in order to get the content of result documents for snippet generation to produce the HTML result page. If the query result cannot be found in the result cache, then the broker contacts to all (or some) index servers for query processing. Each index server executes the query using its own (local) inverted index. This requires access to the posting lists for each of the query terms. It first checks the intersection cache to see if posting list for any pair (or triple) of query terms is cached. Furthermore, it looks for posting lists of the query terms in the list cache. Index server accesses the disk for the query terms that are not found in the list or intersection cache. In the final stage, a ranking (scoring) method computes the scores for documents containing the query terms using the posting lists. A wide range of ranking approaches such as BM25 and machine learning based methods [87] are proposed in the literature, but we do not describe them in detail since they are not in the scope of this thesis. Documents are sorted based on decreasing score and top-k (k is practically between 10 and 1000) result document ids are sent back to the broker with their corresponding scores. Broker aggregates (merges) all these results and produces a final top-k list. In the final stage, snippets must be constructed for documents in the final top-k in order to produce the HTML result page. Document servers first check their document cache. If it is not found in the cache, disk access is required. Snippet generation algorithm [83, 81] produces a textual summary of document considering the query terms. Finally, the broker node sends the result page to the user.

A search engine may employ a static or dynamic cache of different data items (query results, posting lists, and documents), or both [30]. In the static case, the cache is filled with entries as obtained from earlier logs of the search engine and its content remains intact until the next periodical update. In the dynamic case, the cache content changes dynamically with respect to the query traffic, as new entries may be inserted and existing entries may be evicted. There are many cache replacement policies adapted from the literature, such as Least Recently Used (LRU), Least Frequently Used (LFU), etc.

In Table 2.1, we classify the previous studies that essentially focus on caching techniques for large scale search engines. We group those works in terms of the cache component focused on and also whether the caching strategy employed is static or dynamic. An early proposal that discusses posting list caching in an information retrieval system is presented in [80]. The authors use LRU as the caching policy in their dynamic cache replacement algorithm. Markatos [54]

Table 2.1: Classification of earlier works on caching in web search engines. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

Cache Type	Static	Dynamic
Result	Markatos [54] Baeza-Yates and Saint-Jean [10] Fagni et al. [30] Ozcan et al. [62]	Markatos [54] Saraiva et al. [73] Lempel and Moran [47] Long and Suel [51] Fagni et al. [30] Gan and Suel [32] Puppin et al. [69] Cambazoglu et al. [21]
Score	–	Fagni et al. [30]
Intersection	–	Long and Suel [51]
List	Baeza-Yates and Saint-Jean [10] Baeza-Yates et al. [6]	Tomasic and Garcia-Molina [80] Saraiva et al. [73] Long and Suel [51]
Document	–	–

works on query result caching and analyzes the query log from the EXCITE search engine. He shows that query requests have a temporal locality property, that is, a significant amount of queries submitted previously are submitted again a short time later. This fact shows evidence for the caching potential of query results. That work also compares static vs. dynamic caching approaches for varying cache sizes. The analysis in [54] reveals that the static caching strategy performs better when the cache size is small, but dynamic caching becomes preferable when the cache is relatively larger. In a more recent work, we [62] propose an alternative query selection strategy, which is based on the stability of query frequencies over time intervals instead of using an overall frequency value. Lempel and Moran introduce a probabilistic caching algorithm for a result cache that predicts the number of result pages to be cached for a query [47].

To the best of our knowledge, Saraiva et al. [73] present the first work in the literature that mentions a two-level caching to combine caching of query results and inverted lists. This work also uses index pruning techniques for list caching because the full index contains lists that are too long for some popular or common words. Experiments measure the performance of caching the query results and inverted lists separately, as well as that of caching them together. The results show that two-level caching achieves a 52% higher throughput than caching only inverted lists and a 36% higher throughput than caching only query results. Baeza-Yates and Saint-Jean propose a three-level search index structure using query log distribution [10]. The first level consists of precomputed answers (cached query results) and the second level contains the posting lists of the most frequent query terms in the main memory. The remaining posting lists need to be accessed from the secondary memory, which constitutes the third level in the indexing structure. Since the main memory is shared by cached query results and posting lists, it is important to find the optimal space allocation to achieve the best performance. The authors provide a mathematical formulation of this optimization problem and propose an optimal solution. More recently, another three-level caching approach is proposed [51]. As in [10], the first and second levels contain the results and posting lists of the most frequent queries, respectively. The authors propose a third level, namely, an intersection cache, containing common postings of frequently occurring pairs of terms. The intersection cache resides in the secondary memory. Their experimental results show significant performance gains using this three-level caching strategy. Garcia [33] also proposes a cache architecture that stores multiple data item types based on their disk access costs.

The extensive study of Baeza-Yates et al. [6] covers many issues in static and dynamic result/list caching. The authors propose a simple heuristic that takes into account the storage size of lists when making caching decisions. They compare the two alternatives: caching query results vs. posting lists. Their analysis shows that caching posting lists achieves better hit rates [6]. The main reason for this result is that repetitions of terms in queries are more frequent than repetitions of queries. Puppin et al. propose a novel incremental cache architecture for collection selection architectures [69]. They cache query results computed by a subset of index servers selected by a collection selection strategy. If a cache hit occurs for a particular query, additional index servers are contacted to get more results to incrementally improve the cached result of the query.

Fagni et al. [30] propose a hybrid caching strategy, which involves both static and dynamic caching. The authors divide the available cache space into two parts: One part is reserved for static caching and the remaining part is used for dynamic caching. Their motivation for this approach is based on the fact that static caching exploits the query popularity that lasts for a long time interval and dynamic caching handles the query popularity that arises for a shorter interval (e.g., queries for breaking news, etc.). Experiments with three query logs show that their hybrid strategy, called Static-Dynamic Caching (SDC), achieves better performance than either purely static caching or purely dynamic caching. The idea of having a score cache (called the DocID cache in the work) is also mentioned in this study.

2.3 Result Cache Filling Strategies

Although there are many studies on query result caching for search engines, there is no survey that classifies or compares all these works. We classify query result cache filling strategies as

- frequency based
- recency based
- cost based

similar to the classification provided by Podlipnig and Böszörmenyi [68] for proxy caching.

2.3.1 Frequency based

The objective of frequency based methods is to cache the most frequently requested query result pages in the cache. Static result caches are filled based on a previous query log. Queries are sorted based on decreasing frequency and the cache is filled with most frequent query result pages [30, 54, 62]. Static cache content is periodically updated.

Dynamic result caches can also employ frequency based methods such as the widely-known LFU method [32]. This method keeps frequency of each item in the cache. It tries to keep the most frequently requested result pages in the cache by replacing least frequently requested items each time when the cache is full.

2.3.2 Recency based

Recency based methods consider the last time the item in the cache is requested by any user. The objective is to keep the most recently used items in the cache as much as possible. LRU is a popular recency based approach that works well in other domains as well. This approach chooses the least recently used items for replacement when the cache is full. Dynamic result caches can also employ this approach [6, 54].

2.3.3 Cost based

Cost based methods consider that items are associated with different costs to reproduce if not found in the cache. In the result cache domain, this means that the cost of a query result page is to process the query in the index servers and producing the snippets by accessing the document contents. The objective of cost-aware methods is to keep the items in the cache that will provide the highest cost gain.

In the literature, the idea of cost-aware caching is applied in some other areas where miss costs are not always uniform [40, 41, 49]. For instance, in the context of multiprocessor caches, there may be non-uniform miss costs that can be measured in terms of latency, penalty, power, or bandwidth consumption, etc. [40, 41].

In [40], the authors propose several extensions of LRU to make the cache replacement policy cost sensitive. The initial cost function assumes two static miss costs, i.e., a low cost (simply 1) and a high cost (experimented with varying values). This work also provides experiments with a more realistic cost function, which involves miss latency in the second-level cache of a multiprocessor. Web proxy caching is another area where a cost-aware caching strategy naturally fits. In this case, the cost of a miss would depend on the size of the page missed and its network cost, i.e., the number of hops to be traveled to download it. The work of Cao and Irani [22] introduces GreedyDual-Size (GDS) algorithm, which is a modified version of the Landlord algorithm proposed by Young [90]. The GDS strategy incorporates locality with cost and size concerns for web proxy caches. We are aware of only two works in the literature that explicitly incorporate the notion of costs into the caching policies of web search engines. Garcia [33] proposes a heterogeneous cache that can store all possible data structures (posting lists, accumulator sets, query results, etc.) to process a query, with each of these entry types associated with a cost function. However, the cost function in that work is solely based on disk access times and must be recomputed for each cache entry after every modification of the cache.

The second study [32] that is simultaneous to our cost-aware caching work (detailed in Chapter 3) also proposes cache eviction policies that take the costs of queries into account. In this work, the cost function essentially represents queries' disk access costs and it is estimated by computing the sum of the lengths of the posting lists for the query terms. The authors also experiment with another option, i.e., using the length of the shortest list to represent the query cost, and report no major difference in the trends. In Chapter 3, we also introduce the notion of cost for caching result pages, however instead of predicting the cost we use the actual CPU processing time obtained the first time the query is executed, i.e., when the query result page is first generated. Our motivation is due to the fact that actual query processing cost is affected by several mechanisms, such as dynamic pruning, list caching, etc. (see [6]); and it may be preferable to use the actual value when available. Furthermore, we use a more realistic simulation of the disk access cost, which takes into account the contents of the list cache under several scenarios.

Note that, there are other differences between our cost-aware caching work and that of Gan and Suel [32]. In Chapter 3, we provide several interesting findings regarding the variance of processing times among different queries, and the relationship between a query's processing time and its popularity. In light of our findings, we propose cost-aware strategies for static, dynamic, and static-dynamic caching cases [64]. Gan and Suel's work [32] also considers some cost-aware techniques for dynamic and hybrid caching. However, they focus on issues like query burstiness and propose cache eviction policies that exploit features other than the query cost.

2.4 Result Cache Evaluation Metrics

In the literature, performance of result caching strategies is evaluated using different measures. Here, we list and describe these metrics in detail below:

- *Hit Ratio* is the ratio of query requests served from the cache to all query requests.
- *Miss Ratio* is the ratio of query requests that required query processing (not served from the cache) to all query requests. Note that the summation of hit and miss ratios is 1.
- *Throughput* measures the number of query requests that can be answered by the search engine within a unit of time (e.g., 1 second).
- *Response Time* evaluates the time between query submission and return of the result page.

2.5 Result Cache Freshness

A more recent research direction that is orthogonal to the above works is investigating the cache freshness problem. In this case, the main concern is not the capacity related problems (as in the eviction policies) but the freshness of the query results that is stored in the cache. To this end, Cambazoglu et al. [21] propose a blind cache refresh strategy: they assign a fixed time-to-live (TTL) value to each query result in the cache and re-compute the expired queries without verifying whether their result have actually changed or not. They also introduce an eager approach that refreshes expired or about-to-expire queries during the idle times of the search cluster. In contrast, Blanco et al. [14, 15] attempt to invalidate only those cache items whose results have changed due to incremental index updates. They propose cache invalidation predictor (CIP) module that tries to provide coherency between index and result cache. This module invalidates cache entries that are affected by the newly crawled, updated, or deleted documents. Bortnikov et al. [16] extend the work on CIP module and evaluate its performance in real-life cache settings.

Timestamp-based cache invalidation techniques [1, 2] are proposed recently. Timestamps are assigned to queries based on generation time of the result pages in the cache. Similarly, posting list for each term and each document in the collection are also timestamped based on their update times. Query results are invalidated based on some rules that compare the query result timestamps with term or document timestamps. Their experiments show that timestamp-based approach achieves a performance close to that of CIP module but it is more cost efficient.

Chapter 3

Cost-Aware Query Result Caching

Web search engines need to cache query results for efficiency and scalability purposes. Static and dynamic caching techniques (as well as their combinations) are employed to effectively cache query results. In this chapter, we propose cost-aware strategies for static and dynamic caching setups. Our research is motivated by two key observations: i) query processing costs may significantly vary among different queries, and ii) the processing cost of a query is not proportional to its popularity (i.e., frequency in the previous logs). The first observation implies that cache misses have different, i.e., non-uniform, costs in this context. The latter observation implies that typical caching policies, solely based on query popularity, cannot always minimize the total cost. Therefore, we propose to explicitly incorporate the query costs into the caching policies. Simulation results using two large web crawl datasets and a real query log reveal that the proposed approach improves overall system performance in terms of the average query execution time.

The rest of this chapter is organized as follows. In Section 3.1, we provide the motivation for our work and list our contributions. We provide the experimental setup that includes the characteristics of our datasets, query logs and computing resources in Section 3.2. Section 3.3 is devoted to a cost analysis of query processing in a web search engine. The cost-aware static, dynamic, and hybrid caching strategies are discussed in Section 3.4, and evaluated in Section 3.5. We conclude the chapter in Section 3.6.

3.1 Introduction

In the context of web search engines, the literature involves several proposals concerning what and how to cache. However, especially for query result caching, the cost of a miss is usually disregarded, and all queries are assumed to have the same cost. In this chapter, we essentially concentrate on the caching of query results and propose cost-aware strategies that explicitly make use of the query costs while determining the cache contents. Our research is motivated by the following observations: First, queries submitted to a search engine have significantly varying costs in terms of several aspects (e.g., CPU processing time, disk access time, etc.). Thus, it is not realistic to assume that all cache misses would incur the same cost. Second, the frequency of the query is not an indicator of its cost. Thus, caching policies solely based on query popularity may not always lead to optimum performance, and a cost-aware strategy may provide further gains.

In this chapter, we start by investigating the validity of these observations for our experimental setup. To this end, it is crucial to model the query cost in a realistic and accurate manner. Here, we define query cost as the sum of the actual CPU execution time and the disk access cost, which is computed under a number of different scenarios. The former cost, CPU time, involves decompressing the posting lists, computing the query-document similarities and determining the top-N document identifiers in the final answer set. Obviously, CPU time is independent of the query submission order, i.e., can be measured in isolation per query. On the other hand, disk access cost involves fetching the posting lists for query terms from the disk, and depends on the current content of the posting list cache and the query order of queries. In this work, we compute the latter cost under three realistic scenarios, where either a quarter, half, or full index is assumed to be cached in memory. The latter option, storing the entire index in memory, is practiced by some industry-scale search engines (e.g., see [27]). For this case, we only consider CPU time to represent the query cost, as there is no disk access. The former option, caching a relatively smaller fraction of the index, is more viable for medium-scale systems or memory-scarce environments. In this study, we also consider disk access costs under such scenarios while computing the total query processing cost. The other cost factors, namely, network latency and snippet generation costs, are totally left out, assuming that these components would be less dependent on query characteristics and would not be a determining factor in the total cost. Next, we introduce cost-aware strategies for the static, dynamic, and hybrid caching of query results. For the static caching case, we combine query frequency information with query cost in different ways to generate alternative strategies. For the dynamic case, we again incorporate the cost notion into a typical frequency-based strategy in addition to adapting some cost-aware policies from other domains. In the hybrid caching environment, a number of cost-aware approaches developed for static and dynamic cases are coupled. All these strategies are evaluated in a realistic experimental framework that attempts to imitate the query processing of a search engine, and are shown to improve the total query processing time over a number of test query logs.

The contributions of our work in this chapter are summarized as follows:

1. We introduce a cost-aware strategy that takes the frequency and cost into account at the same time for the static caching. We also propose a cost-aware counterpart of the static caching method that we have discussed in another work [62]. The latter method takes into account the stability of query frequency in time, and can outperform typical frequency-based caching.

2. We introduce two cost-aware caching policies for dynamic query result caching. Furthermore, we adapt several cost-aware policies from other domains.

3. We also evaluate the performance of cost-aware methods in a hybrid caching environment (as proposed in [30]), in which a certain part of the cache is reserved for static caching and the remaining part is used for the dynamic cache.

4. We experimentally evaluate caching policies using two large web crawl datasets and real query logs. Our cost function, as discussed above, takes into account both actual CPU processing time that is measured in a realistic setup with list decompression and dynamic pruning, and disk access time computed under several list caching scenarios. Our findings reveal that the cost-aware strategies improve overall system performance in terms of the total query processing time.

3.2 Experimental Setup

Datasets. In this study, we use two datasets. For the first, we obtained the list of URLs categorized at the Open Directory Project (ODP) web directory (www.dmoz.org). Among these links, we successfully crawled around 2.2 million pages, which take 37 GBs of disk space in uncompressed HTML format. For the second dataset, we create a subset of the terabyte-order crawl collection provided by Stanford University's WebBase Project [84]. This subset includes approximately 4.3 million pages collected from US government web sites during the first quarter of 2007. These two datasets will be referred to as "ODP" and "Webbase" datasets, respectively. The datasets are indexed by the Zettair search engine [92] without stemming and stopword removal. We obtained compressed index files of 2.2 GB and 7 GB on disk (including the term offset information in the posting lists) for the first and second datasets, respectively.

Query Log. We create a subset of the AOL Query Log [66], which contains around 20 million queries of about 650K people for a period of three months. Our subset contains around 700K queries from the first six weeks of the log. Queries submitted in the first three weeks constitute the training set (used to fill the static cache and/or warm up the dynamic cache), whereas queries from the second three weeks are reserved as the test set. In this study, during both the training and testing stages, the requests for the next page of the results for a query are considered as a single query request, as in [9]. Another alternative would be to interpret each log entry as <query, result page number> pairs [30]. Accordingly, we presume that a fixed number of N results are cached per query. Since N would be set to a small number in all practical settings, we presume that the actual value of N would not significantly affect the findings in this study. Here, we set N as 30, as earlier works on log analysis reveal that search engine users mostly request only the first few result pages. For instance, Silverstein et al. report that in 95.7% of queries, users requested up to only three result pages [74].

Experimental Platform. All experiments are conducted using a computer that includes an Intel Core2 processor running at 2.13GHz with 2GB RAM. The operating system is Suse Linux.

3.3 An Analysis of the Query Processing Cost

3.3.1 The Setup for Cost Measurement

The underlying motivation for employing result caching in web search engines (at the server side) is to reduce the burden of query processing. In a typical broker-based distributed environment (e.g., see [20]), the cost of query processing would involve several aspects, as shown in Figure 3.1. The central broker, after consulting its result cache, sends the query to index nodes. Each index node should then fetch the corresponding posting lists to the main memory (if they are not already in the list cache) with the cost C_{DISK}. Next, the postings are processed and partial results are computed, with the cost C_{CPU}. More specifically, the CPU cost involves decompressing the posting lists (as they are usually stored in a compressed form), computing a similarity function between the query and the postings, and obtaining the top-N documents as the partial result. Then, each node sends its partial results to the central broker, with the cost C_{NET}, where they are merged. Finally, the central broker generates the snippets for the

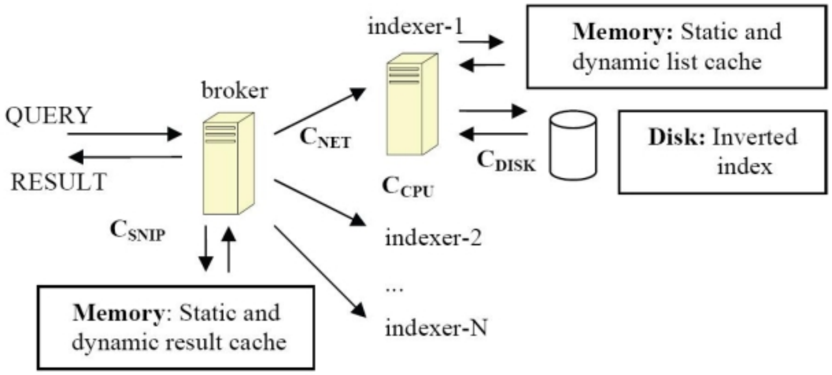


Figure 3.1: Query processing in a typical large scale search engine. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

query results, with the cost C_{SNIP}, and sends the output page to the user. Thus, the cost of query processing is the sum of all of these costs, i.e., $C_{DISK} + C_{CPU} + C_{NET} + C_{SNIP}$.

For the purposes of this work, we consider the sum of the CPU execution time and disk access time (i.e., $C_{DISK}+C_{CPU}$) as the major representative of the overall cost of a query. We justify leaving out the network communication and snippet generation costs as follows: Regarding the issue of network costs, a recent work states that for a distributed system interconnected via a LAN, the network cost would only be a fraction of the query processing cost (e.g., see Table 2 in [6]). The snippet generation cost is discarded because its efficiency is investigated in only a few previous studies (e.g., [83, 81]), and none of these discusses how the cost of snippet generation compares to other cost components. Furthermore, we envision that the two costs, namely, network communication and snippet generation, are less likely to vary significantly among different queries and neither would be a dominating factor in the query processing cost. This is because, regardless of the size of the posting lists for query terms, only a small and fixed number of results with the highest scores, (typically, top-10 document identifiers) would be transferred through the network. Similarly, only that number of documents would be accessed for snippet generation. Thus, in the rest of this chapter, we essentially use $C_{CPU} + C_{DISK}$ as the representative of the query processing cost in a search engine. In this study, all distinct queries are processed using the Zettair search engine in batch mode to obtain the isolated CPU execution time per query. That is, we measured the time to decompress the lists, compute similarities and obtain the identifiers of the top-N documents, where N is set to 30. Since Zettair is executed at a centralized architecture, there is no network cost. To be more accurate, we describe the setup as follows.

1. We use Zettair by its default mode, which employs an early pruning strategy that dynamically limits the number of accumulators used for a query. In particular, this dynamic pruning strategy adaptively decides to discard some of the existing accumulators or add new ones (up to a predefined target number of accumulators) as each query term is processed (see [48] for details). Following the practice in Lester et al. [48], we also set the target number of accumulators to approximately 1% of the number of documents in the collection, namely, 20K. Employing a dynamic pruning strategy is a crucial choice for the practicality of our proposal, since no real web search engine would make a full evaluation and the CPU execution time clearly depends on the partial evaluation strategy used in the system.

2. All query terms in the log are converted to lower case. The queries are modified to include an additional "AND" conjunct between each term, so that the search engine runs in the "conjunctive" mode. This is the default search mode of the major search engines [26, 57]. Stopwords are not eliminated from the queries. No stemming algorithm is applied. Finally, all phrase queries are discarded.

In contrast to C_{CPU}, it is hard to associate a query with an isolated disk access cost, because for a real-life system disk access time depends on the query stream and the posting lists that are buffered and/or cached in the memory at the processing time. Thus, instead of measuring the actual disk access time, we compute this value per query under three different scenarios, where 25%, 50%, or 100% of the index is assumed to be cached. It is now widely accepted that any large scale search engine involves a reasonably large list cache that accompanies

Table 3.1: Disk parameters for simulating C_{DISK}. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

Parameter Name	Value
Seek time	8.5 ms
Rotational delay	4.17 ms
Block read time	4.883 microseconds
Block size	512 bytes

the result cache. Indeed, it is known that some major search engines store all posting lists in the main memory, an approach totally eliminating the cost of disk access ([27, 76]). Therefore, our scenarios reflect realistic choices for the list cache setup.

For each of these scenarios, query term frequencies are obtained from the training logs and those terms with the highest ratio of term frequency to posting list length are stored in the cache, as proposed in [6]. That study also reports that a static list cache filled in this manner yields a better hit rate than dynamic approaches (see Figure 8 in [6]); so our framework only includes the static list cache. For each scenario, we first determine which terms of a given query cause a cache miss, and compute the disk access cost of each such term as the sum of seek time, rotational latency, and block transfer time, which is typical (e.g., see [70]). In this computation, we use the parameters of a moderate disk, as listed in Table 3.1.

To sum up, our framework realistically models the cost of processing a query in terms of the CPU and disk access times. The cost of a query computed by this model is independent of the query load on the system, as it only considers the isolated CPU and disk access times. More specifically, disk access time in this setup depends on the contents of the list cache (and, in some sense, previous queries), but not on the query load.

Before discussing cost-aware result caching strategies, we investigate answers to the following questions: i) Do the costs of different queries really vary considerably? ii) Is there a correlation between the frequency and the cost of the queries? We expect the answer to the first question to be positive; i.e., there should be a large variance among query execution times. On the other hand, we expect the answer of the second question to be negative; that is, frequency should not be a trustworthy indicator of processing time, so that cost-aware strategies can further improve solely frequency-based strategies. We explore the answers to these questions in the next section and show that our expectations are justified.

3.3.2 Experiments

After a few initial experiments with the datasets and query log described above, it turns out that a non-trivial number of queries yields no answer for our datasets. As an additional problem, some of the most frequent queries in the query log appear much less frequently in the dataset, which might bias our experiments. Since previous works in the literature emphasize that the dataset and query log should be compatible to ensure a reliable experimental framework [85], we first focus on resolving this issue. The ODP dataset contains pages from a general web directory that includes several different categories and the AOL log is a general domain search engine log. That is, the content of our dataset matches the query log to a certain extent, although the dataset is, of course, much more restricted than the real AOL collection. Thus, for this case, it would be adequate to simply discard all queries that have no answer in the ODP dataset. On the other hand, recall that the Webbase dataset includes pages crawled only from the .gov domain. Thus, there seems to be a higher possibility of mismatch between the Webbase dataset and AOL query log. To avoid any bias that might be caused by this situation, we decided to obtain a domain-restricted version of the query log for the Webbase collection. In what follows, we describe the query logs corresponding to the datasets as used in this study.

a) ODP query log: We keep all queries in the log that yield non-empty results on the ODP dataset.

Table 3.2: Characteristics of the query log variants. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

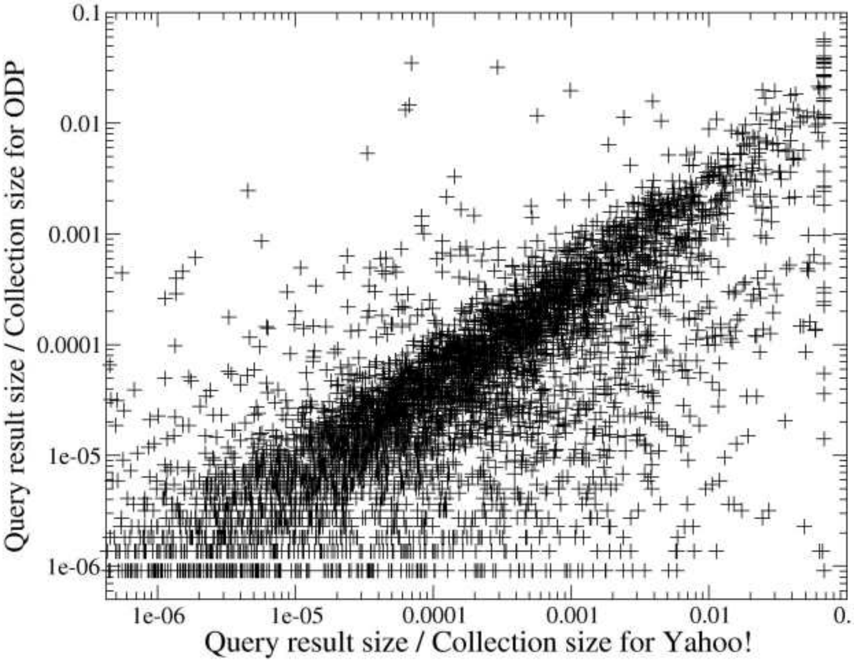
Query Log	Number of distinct queries		Number of all queries	
	Training	Test	Training	Test
ODP query log	253,961	209,636	446,952	362,843
Webbase query log	211,854	175,557	386,179	313,884
Webbase semantically aligned query log	28,794	24,066	45,705	37,506

b) Webbase query log: We keep all queries in the log that yield non-empty results on the Webbase dataset.

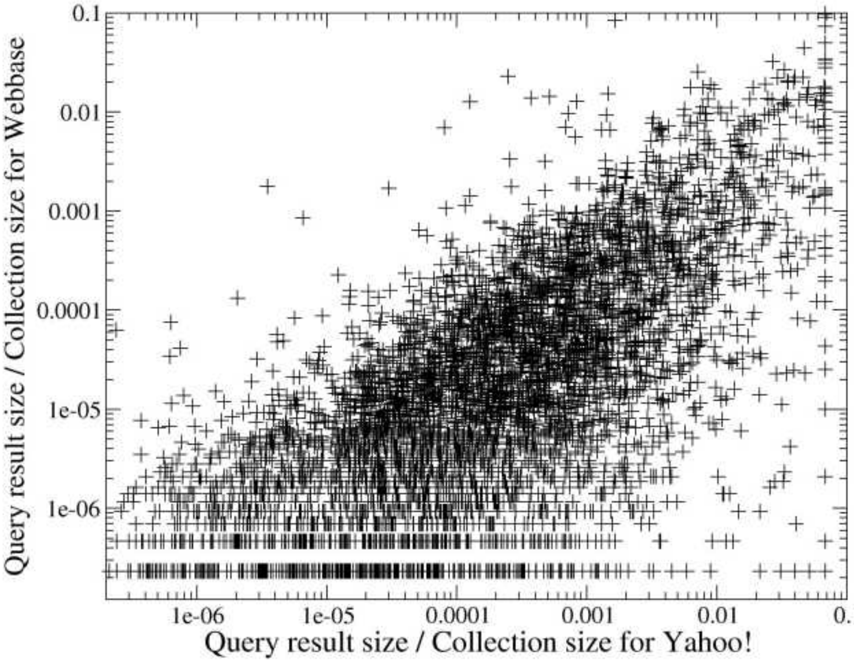
c) Webbase semantically aligned query log: Following a similar approach discussed in a recent work [82], we first submit all distinct queries in our original query log to Yahoo! search engine's "Web search" service [89] to get the top-10 results. Next, we only keep those queries that yield at least one result from the .gov domain.

In Table 3.2, we report the number of the remaining queries in each query log. To experimentally justify that these query logs and the corresponding datasets are compatible, we conduct an experiment as follows: We process randomly chosen 5K queries from each of the three query logs in conjunctive mode on the corresponding collection, and record the total number of results per query. Next, we submit the same queries to Yahoo! (using its web search API), again in conjunctive (default) processing mode. For each case, we also store the number of results per query as returned by the search engine API.

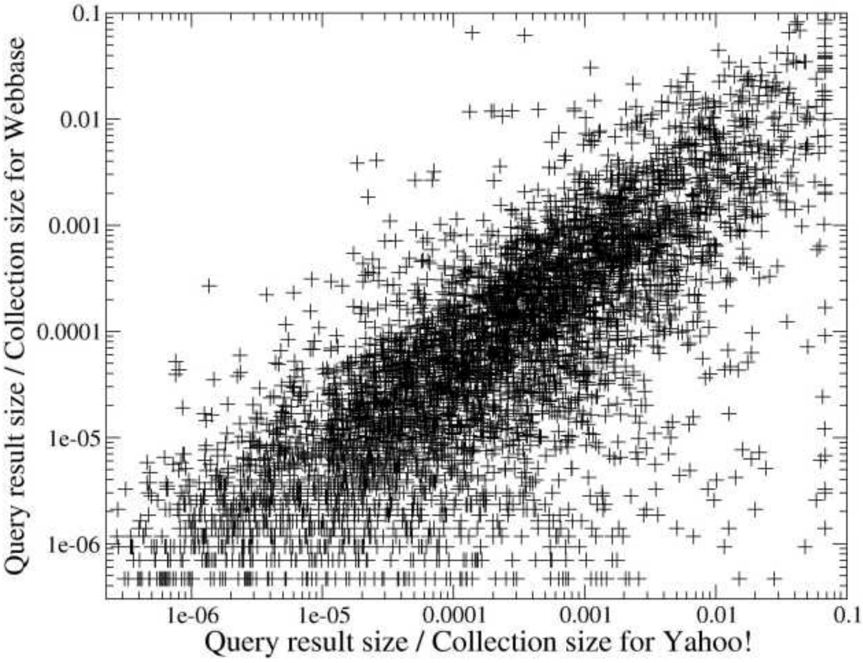
In Figure 3.2, we represent these 5K queries on a log-log scale plot, where the y-axis is the ratio of the number of results retrieved in our corresponding collection to the collection size, and the x-axis is the same ratio for Yahoo! collection. We assume that the underlying collection of Yahoo! includes around 31.5 billion pages, which is the reported number of results when searching for the term "a" on the Yahoo! website. The figure reveals that the ratio of the number of results


0.1
0.01
0.001
0.0001
1e-05
1e-06
Query result size / Collection size for ODP
1e-06
1e-05
0.0001
0.001
0.01
0.1
Query result size / Collection size for Yahoo!

(a)


0.1
0.01
0.001
0.0001
1e-05
1e-06
Query result size / Collection size for Webbase
1e-06
1e-05
0.0001
0.001
0.01
0.1
Query result size / Collection size for Yahoo!

(b)



(c)

Figure 3.2: Correlation of "query result size/collection size" on Yahoo! and a) ODP, b) Webbase, and c) Webbase semantically aligned for the conjunctive processing mode. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

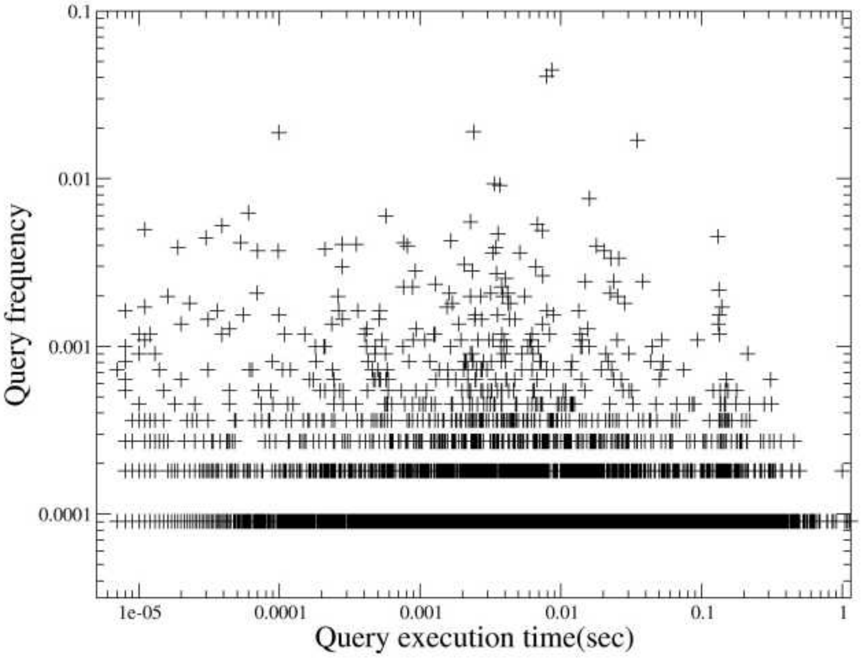
per query in each collection is positively correlated with the ratio in the Yahoo! search engine, i.e., yielding correlation coefficients of 0.80, 0.57, and 0.86 for the ODP, Webbase, and Webbase semantically aligned logs, respectively. Thus, we conclude that our collections and query sets are compatible, and experimental evaluations would provide meaningful results.

Next, for each of the query logs in Table 3.2, we obtain the CPU execution time of all distinct queries using the setup described in the previous section. The experiments are repeated four times and the results reveal that the computed CPU costs are stable and can be used as the basis of the following discussions. For instance, for ODP log, we find that the average standard deviation of query execution times is 2 ms. Considering that the average query processing time is 93 ms for this case, we believe that the standard deviation figure (possibly due to system-dependent fluctuations) is reasonable and justifies our claim of the stability of execution times.

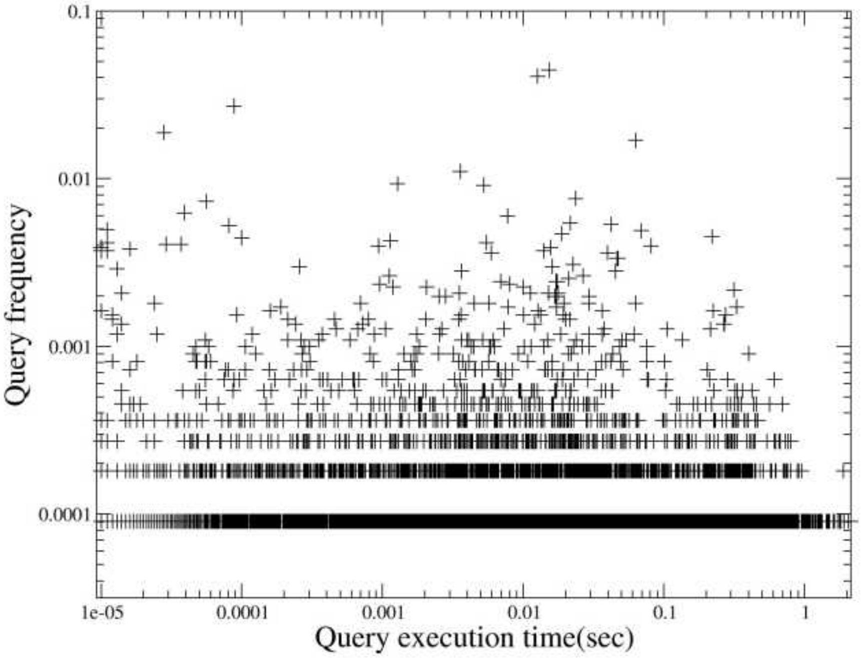
In Figure 3.3, we provide the normalized log-log scatter plots that relate the query's frequency in the log and the CPU execution time[1] for randomly selected 10K queries. These plots reveal the answers to the questions raised at the end of the previous section. First, we see that the query execution time covers a wide range, from a fraction of a millisecond to a few thousand milliseconds. Thus, it may be useful to devise cost-aware strategies in the caching mechanisms. Second, we cannot derive a high positive correlation between the query frequency and the processing times. That is, a very frequent query may be cheaper than a less-frequent query. This can be explained by the following arguments: In earlier works, it is stated that query terms and collection terms may not be highly correlated (e.g., a correlation between Chilean crawl data and the query log is found to be only 0.15 [10]), which means that a highly frequent query may return fewer documents, and be cheaper to process. Indeed, in a separate experiment reported below, we observe that this situation also holds for the AOL query log using both the ODP dataset and the Yahoo! search engine.

In Figure 3.4, we show the normalized log-log scatter plots that relate the query frequency in the log and result-set size, i.e., the ratio of number of query results to the size of ODP collection, for randomly selected 10K queries. As can be seen from the figure, the correlation is very low; i.e., the correlation coefficient is -0.01. In order to show that the same trend also holds true for a real search engine, we provide the same plot obtained for Yahoo! in Figure 3.5. Here, we obtain the number of results returned for randomly selected 5K queries using the Yahoo! Web search API. Figure 3.5 again demonstrates that queries with the same frequency might have a very different number of results, and that there is no positive correlation between query frequency and query result frequency; i.e., for this case, the correlation coefficient is only 0.03. Similar findings are also

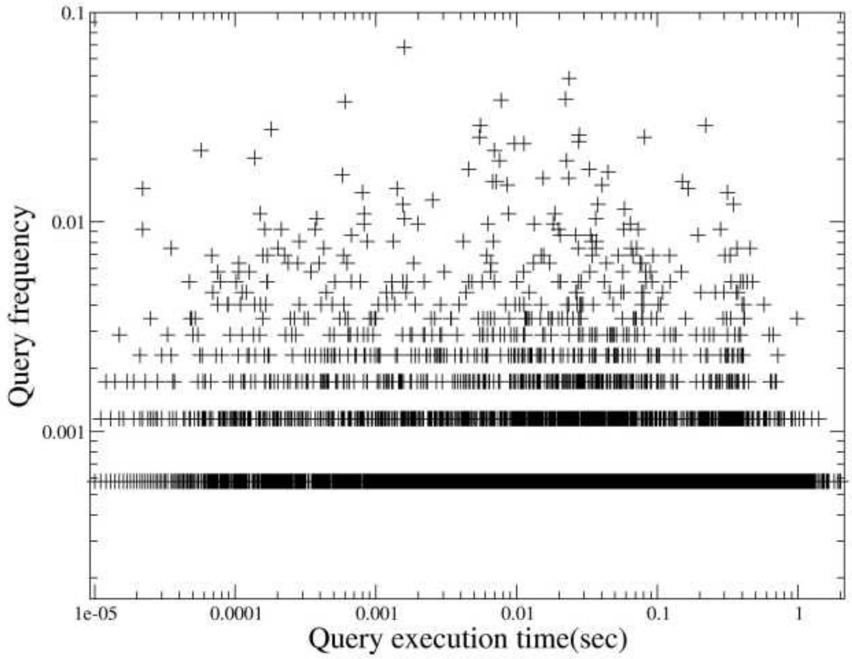
[1] Note that, this experiment considers the scenario where the entire index is assumed to be in the memory and thus C_{DISK} is discarded. The findings for other scenarios, i.e., those involving C_{DISK}, are similar and not reported here for brevity.


0.1
0.01
0.001
0.0001
Query frequency
1e-05
0.0001
0.001
0.01
0.1
1
Query execution time(sec)

(a)


0.1
0.01
0.001
0.0001
Query frequency
1e-05
0.0001
0.001
0.01
0.1
1
Query execution time(sec)

(b)



(c)

Figure 3.3: Normalized log-log scatter plot of the query CPU execution time and query frequency in the a) ODP, b) Webbase, and c) Webbase semantically aligned query log. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

observed for Webbase dataset and corresponding query logs.

Finally, note that, even for the cases where the above trend does not hold (i.e., the frequent queries return a large number of result documents), the processing time does not necessarily follow the proportion, due to the compression and early stopping (pruning) techniques applied during query processing (see Figure 10 in [6], for example). Our findings in this section are encouraging in the following ways: We observe that the query processing costs, and accordingly, the miss costs, are non-uniform and may vary considerably among different queries. Furthermore, this variation is not directly correlated to the query frequency, a feature already employed in current caching strategies. These call for a cost-aware caching strategy, which we discuss next.

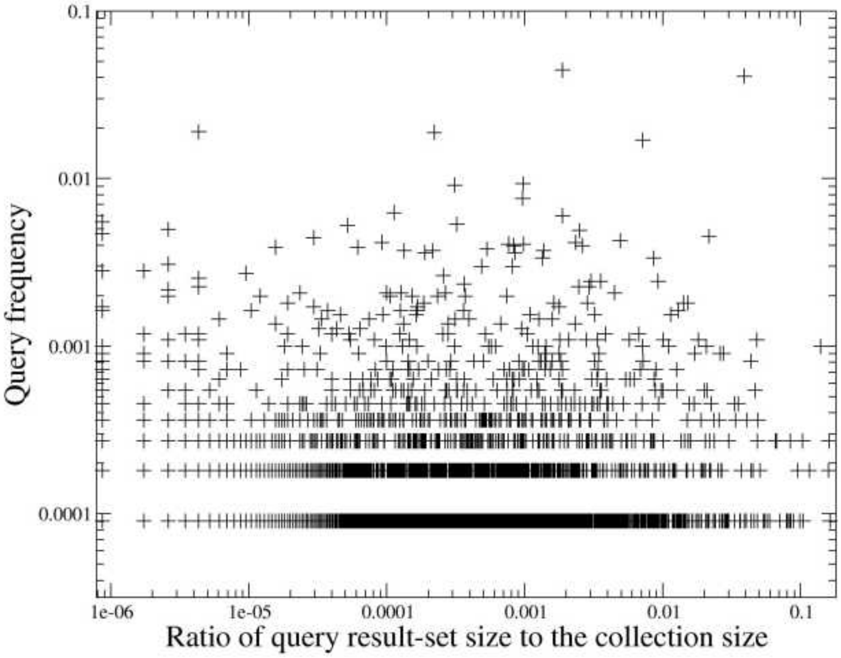


Figure 3.4: Normalized log-log scatter plot of the query result-set size and the query frequency in the ODP query log for 10K queries. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

3.4 Cost-Aware Static and Dynamic Caching

In this section we describe our cost-aware strategies for static, dynamic, and hybrid result caching.

3.4.1 Cost-Aware Caching Policies for a Static Result Cache

As discussed in the literature [6], filling a static cache with a predefined capacity can be reduced to the well-known knapsack problem, where query result pages are the items with certain sizes and values. In our case, we presume that cache capacity is expressed in terms of the number of queries; i.e., each query (and its results) is allocated a unit space. Then, the question is how to fill the knapsack

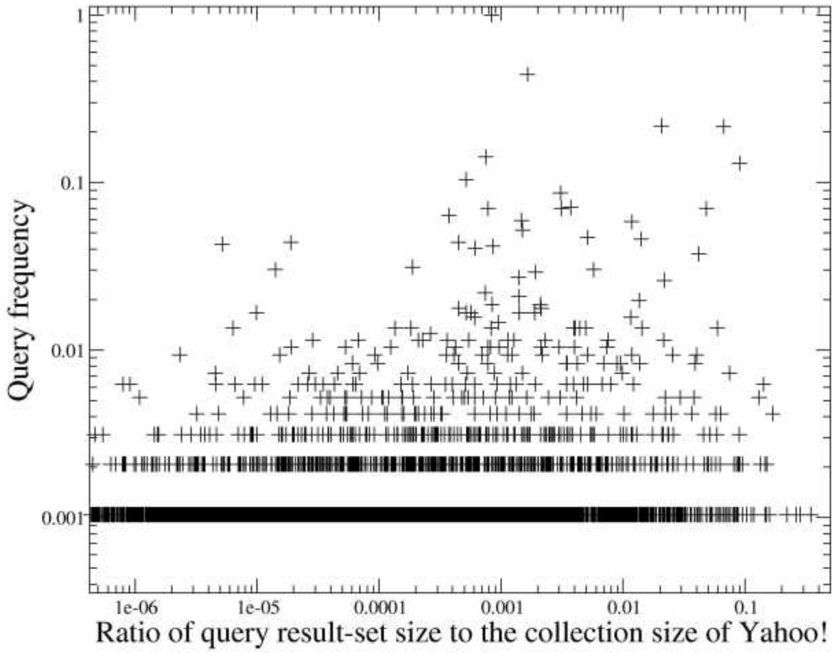


Figure 3.5: Normalized log-log scatter plot of the query result-set size in Yahoo! and the query frequency in the query log for 5K queries. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

with the items that are most valuable. Setting the value of a query to its frequency in previous query logs, i.e., filling the cache with the results of the most frequent past queries, is a typical approach employed in search engines. However, as we argue above, miss costs of queries are not uniform. Therefore, the improvement promises of such a strategy evaluated in terms of, for example, hit rate, may not translate to actual improvements that can be measured in terms of query processing time or throughput. To remedy this problem, we propose to directly embed miss costs into the static caching policies. In what follows, we specify a number of cost-aware static caching approaches in addition to the traditional frequency-based caching strategy, which serves as a baseline. In these discussions, for a given query q, its cost and frequency are denoted as Cq and Fq, respectively.

Most Frequent (MostFreq). This is the baseline method, which 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:

$$Value(q) = F_q \tag{3.1}$$

Frequency Then Cost (FreqThenCost). Previous studies show that query frequencies in a log follow a power-law distribution; i.e., there exist a few queries with high frequencies and many queries with very low frequencies [88]. This means that for a reasonably large cache size, the MostFreq strategy should select among a large number of queries with the same relatively low- frequency value, possibly breaking the ties at random. In the FreqThenCost strategy, we define the value of a query with the pair (Fq, Cq) so that while filling the cache we first choose the results of the queries with the highest frequencies, and from the queries with the same frequency values we choose those with the highest cost. We anticipate that this strategy would be more effective than MostFreq especially for caches with larger capacities, for which more queries with the same frequency value would be considered for caching. In Section 3.5, we provide experimental results that justify our expectation.

Stability Then Cost (StabThenCost). In a recent study, we introduce another feature, namely, query frequency stability (QFS), to determine the value of a query for caching [62]. This feature represents the change in a query's popularity during a time interval. The underlying intuition for this feature stems from the fact that in order to be cached, queries should be frequent and remain frequent over a certain time period. The QFS feature is defined as follows: Assume that query q has the total frequency of f in a training query log that spans a time period T. Consider that this time period is divided into n equal time intervals[2] and query q has the following values of frequency: $F = \{f_1, f_2, \ldots, f_n\}$; one for each T/n time units. Then, the stability of query q is defined by the following formula:

$$QFS_q = \sum_{i=1}^{n} \frac{|f_i - f_\mu|}{f_\mu} \tag{3.2}$$

[2] In this study, we assume one day as the time interval

where $f_\mu = \frac{f}{n}$ is the mean frequency of q during T.

In this previous study, it is shown that using the QFS feature for static caching yields better hit rates than solely using the frequency feature. Here, we combine this feature with the query cost and define the value of a query with the pair (QFS_q, C_q). That is, while filling the cache, queries are first selected based on their QFS value and then their associated cost values.

Frequency and Cost (FC_K). For a query q with cost C_q and frequency F_q, the expected value of the query q can be simply computed as the product of these two figures, i.e., $C_q \times F_q$. That is, we expect that the query would be as frequent in future as in past logs, and caching its result would provide the gain as expressed by this formula. During the simulations reported in Section 3.5, we observe that this expectation may not hold in a linearly proportional manner; i.e., queries that occur with some high frequency still tend to appear with a high frequency, whereas the queries with a relatively lower frequency may appear even more sparsely, or totally fade away, in future queries. A similar observation is also discussed in [32]. For this reason, we use a slightly modified version of the formula that is biased to emphasize higher frequency values and depreciate lower ones, as shown below.

$$Value(q) = C_q \times F_q^K \tag{3.3}$$

where $K > 1$.

3.4.2 Cost-Aware Caching Policies for a Dynamic Result Cache

Although static caching is an effective approach for exploiting long-term popular queries, it may miss short-term popular queries submitted to a web search engine during a short time interval. Dynamic caching handles this case by updating its content as the query stream changes. Different from static caching, a dynamic cache does not require a previous query log. It can start with an empty cache and it fills its entries as new queries are submitted to a web search engine. If the result page for a submitted query is not found in the cache, the query is executed and its result is stored in the cache. When the cache is full, a victim cache entry is chosen based on the underlying cache replacement policy. A notable number of cache replacement policies are proposed in the literature (e.g., see [68] for web caches). In the following, we first describe two well-known strategies, namely, least recently used and least frequently used, to serve as a baseline. Then, we introduce two cost-aware dynamic caching strategies in addition to adapting two other approaches from the literature to the result caching domain.

Least Recently Used (LRU). This well-known strategy chooses the least recently referenced/used cache item as the victim for eviction.

Least Frequently Used (LFU). In this policy, each cached entry has a frequency value that shows how many times this entry is requested. The cache item with the lowest frequency value is replaced when the cache is full. This strategy is called “in-cache LFU” in [68].

Least Costly Used (LCU). This is the most basic cost-aware replacement policy introduced in this study. Each cached item has an associated cost. This method chooses the least costly cached query result as the victim.

Least Frequently and Costly Used (LFCU_K). This policy is the dynamic version of the FC_K static cost-aware policy. We employ the same formula specified for the FC_K strategy in Section 3.4.1.

Greedy Dual Size (GDS). This method maintains a so-called H-value for each cached item [22]. For a given query q with an associated cost C_q and result page size S_q, the H-value is computed as follows:

$$H\_value(q) = \frac{C_q}{S_q} + L \tag{3.4}$$

In this formula, L is an aging factor that is initialized to zero at the beginning. This policy chooses the cache item with the smallest H-value. Then, the value of

L is set to the evicted item's H-value. When a query result is requested again, its H-value is recalculated since the value of L might have changed. The size component in the formula can be ignored as all result pages are assumed to use the same amount of space, as we discuss before.

Greedy Dual Size Frequency (GDSF_K). This method is a slightly modified version of the GDS replacement policy [3]. In this case, the frequency of cache items is also taken into account. The corresponding H-value formula is presented below.

$$H\_value(q) = F_q^K \times \frac{C_q}{S_q} + L \tag{3.5}$$

In this strategy, the frequency of the cache items are also kept and updated after each request. As discussed in Section 3.4.1, again we favor the higher frequencies by adding an exponent $K(> 1)$ to the frequency component. Note that with this extension, the formula also resembles the generalized form of GDSF as proposed in [23]. However, that work proposes to add weighting parameters for both frequency and size components while setting the cost component to 1. In our case, we have to keep the cost, C_q, and apply weighting only for the frequency values.

3.4.3 Cost-Aware Caching Policies for a Hybrid Result Cache

The hybrid result caching strategy proposed in [30] involves both static and dynamic caching and it outperforms its purely static and purely dynamic counterparts. We employ the hybrid caching framework in order to see the effect of cost-awareness in such a state-of-the-art query result caching environment. In this cache configuration, total cache size is divided into two parts, for static and dynamic caches. The fractions of the cache space reserved for each kind of cache is based on the underlying query log for the best performance. The static cache part is filled based on a training query log. Then, the dynamic cache part

Table 3.3: Hybrid cache configurations. (Ozcan, R., Altingovde, I.S., Ulusoy, O., “Cost-Aware Strategies for Query Result Caching in Web Search Engines,” ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

Hybrid cache configuration	Static cache strategy	Dynamic cache strategy
Non cost-aware	MostFreq	LRU
Only static cache is cost-aware	FC_K	LRU
Both static and dynamic caches are cost-aware	FC_K	LFCU_K
	FC_K	GDS
	FC_K	GDSF_K

is warmed up by submitting the remaining training queries into this part of the cache. Later, cache performance is evaluated using the disjoint test set. Table 3.3 shows the hybrid cache configurations experimented with in this work. We essentially consider three types of cache configurations, based on whether a cost-aware strategy is employed in each cache part. The baseline case does not involve the notion of cost at all; the static and dynamic caches employ traditional MostFreq and LRU strategies, respectively [30]. In the second case, only the static portion of the hybrid cache can be cost-aware; the static part employs the FC_K strategy and the dynamic part is still based on LRU. In the third configuration, both the static and dynamic portions might be cost-aware. For this case, three different combinations are experimented with. All three combinations use FC_K for the static cache part, but they use three different cost-aware dynamic caching policies, namely, LFCU_K, GDS, and GDSF_K.

3.5 Experiments

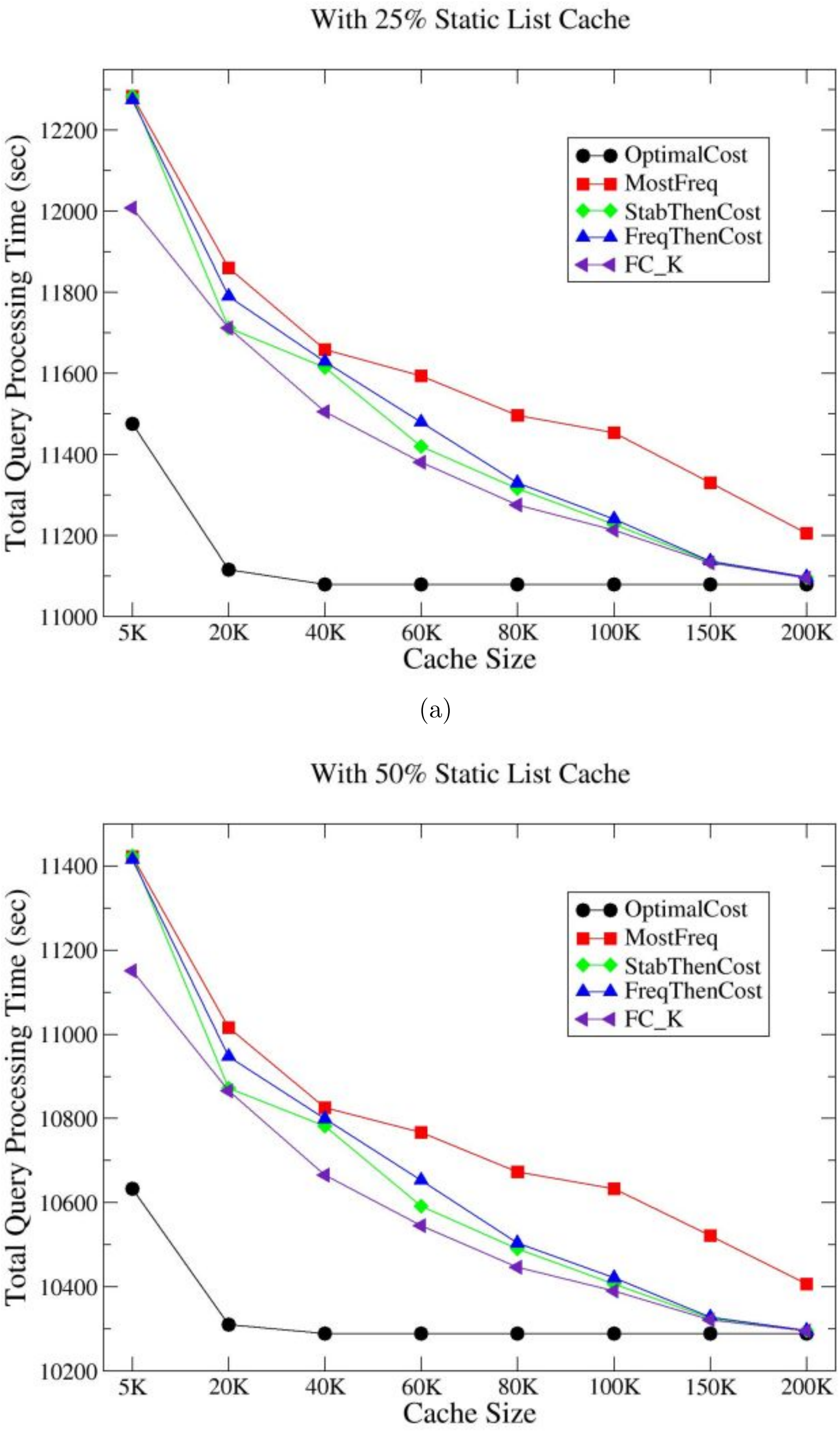
In this section, we provide a simulation-based evaluation of the aforementioned cost-aware policies for static, dynamic, and hybrid caching environments. As described in Section 3.2, the query log is split into training and test sets, and the former is used to fill the static caches and/or warm up the dynamic caches, whereas the latter is used to evaluate the performance. Cache size is specified in terms of the number of the queries. Remarkably, we do not measure the cache hit rate (due to non-uniform miss costs) but use the total query processing time for evaluation. For the cache hits, we assume that the processing time is negligible; i.e., the cost is 0. To simulate the cache misses, we use the query processing cost, $C_{CPU} + C_{DISK}$, which is also employed in the training stage. That is, for all distinct queries in the log, we store the actual CPU execution time (C_{CPU}) per query that is reported by the Zettair search engine. As mentioned before, CPU cost measurements are repeated four times and found to be stable; i.e., no fluctuations are observed. Furthermore, for each list cache scenario, namely, caching 25%, 50%, and 100% of the index, we compute the simulated disk access time, C_{DISK}, per query. Whenever a cache miss occurs, the cost of this query is retrieved as the sum of C_{CPU} and C_{DISK} for the given list cache scenario, and added to the total query processing time.

3.5.1 Simulation Results for Static Caching

In this section, we essentially compare the four strategies, namely MostFreq, FreqThenCost, StabThenCost, and FC_K, for static caching. In Figure 3.6(a), 3.6(b), and 3.6(c), we provide the total query execution times using these strategies for the ODP log when 25%, 50%, and 100% of the index is cached, respectively. For the sake of brevity, the corresponding experimental results for the Webbase and Webbase semantically aligned logs are given in Figure 3.7 only for the case where the entire index is cached.

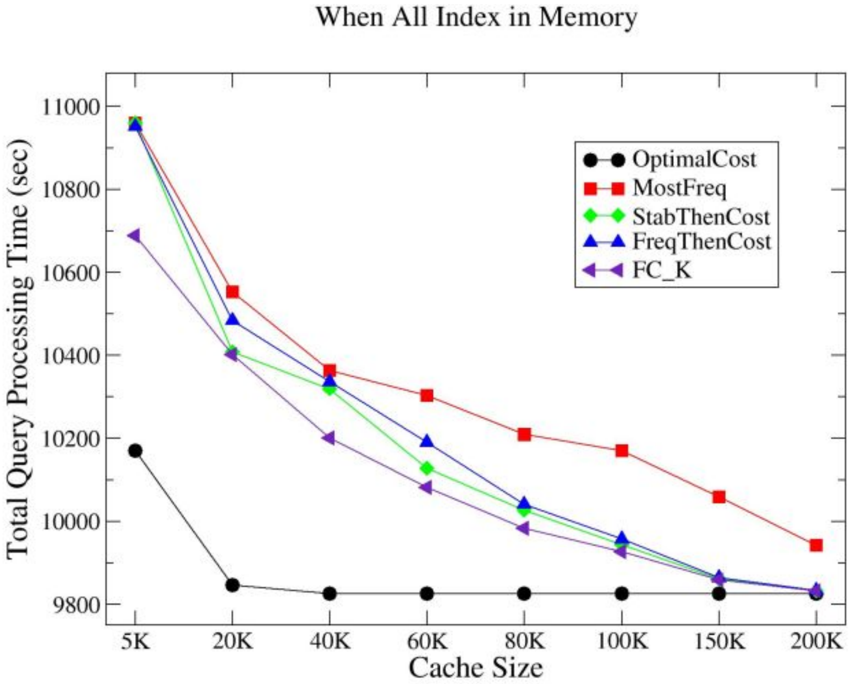
We also provide the potential gains for the optimal cost-aware static caching strategy (OptimalCost), where the test queries are assumed to be known beforehand. Since we know the future frequency of the training queries, we fill the cache with the results of the queries that would yield the highest gain, i.e., frequency times cost. Clearly, this is only reported to illustrate how far the proposed strategies are away from the optimal.

In all experiments, cost-aware strategies (FreqThenCost, StabThenCost and


With 25% Static List Cache
Total Query Processing Time (sec)
11000
11200
11400
11600
11800
12000
12200
5K
20K
40K
60K
80K
100K
150K
200K
Cache Size
OptimalCost
MostFreq
StabThenCost
FreqThenCost
FC_K
With 50% Static List Cache
Total Query Processing Time (sec)
10200
10400
10600
10800
11000
11200
11400
5K
20K
40K
60K
80K
100K
150K
200K
Cache Size
OptimalCost
MostFreq
StabThenCost
FreqThenCost
FC_K

(a)

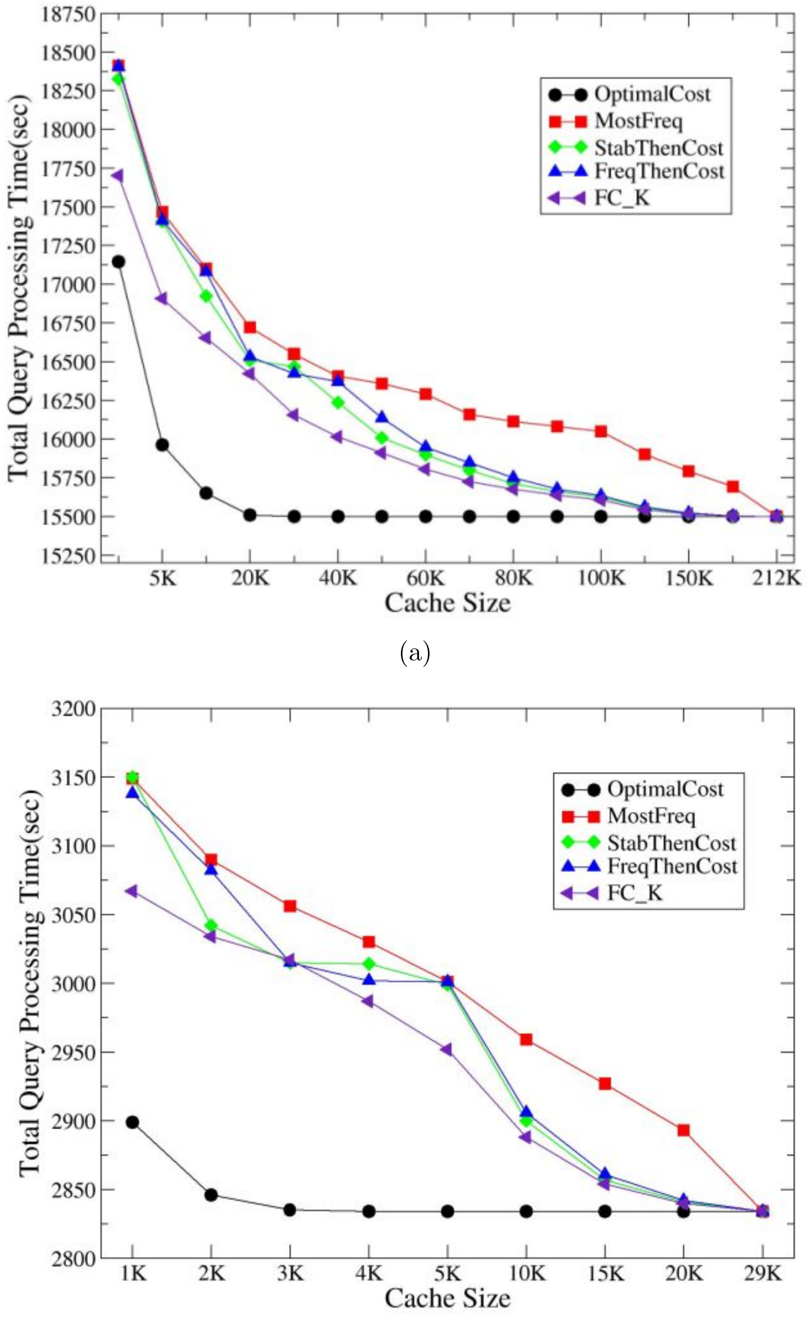
(b)



(c)

Figure 3.6: Total query processing times (in seconds) obtained using different static caching strategies for the ODP log when a) 25%, b) 50%, and c) 100% of the index is cached. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

FC_K) reduce the overall execution time with respect to the baseline, i.e., MostFreq. We observe that the improvement gets higher as higher percentages of the index are cached. This is important because large scale search engines tend to cache most of the index in memory. The best-performing strategy, FC_K (where K is set to 2.5 by experimentally tuning), yields up to a 3% reduction in total time for varying cache sizes. It is also remarkable that the gains for the optimal cache (denoted as OptimalCost) are much higher, which implies that it is possible to further improve the value function employed in the cost-aware strategies.



(a)

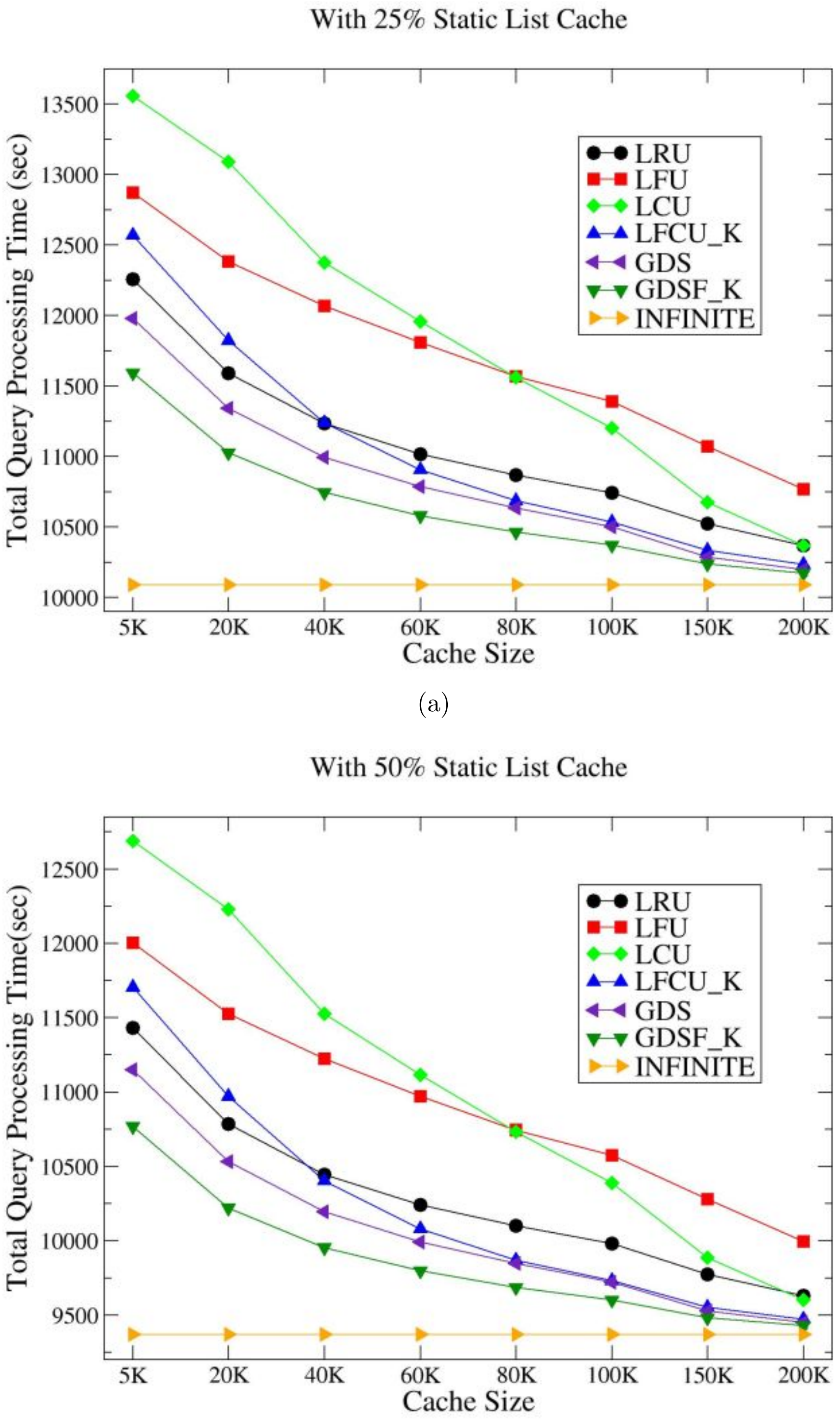
(b)

Figure 3.7: Total query processing times (in seconds) obtained using different static caching strategies for the a) Webbase and b) Webbase semantically aligned logs when 100% of the index is cached. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

3.5.2 Simulation Results for Dynamic Caching

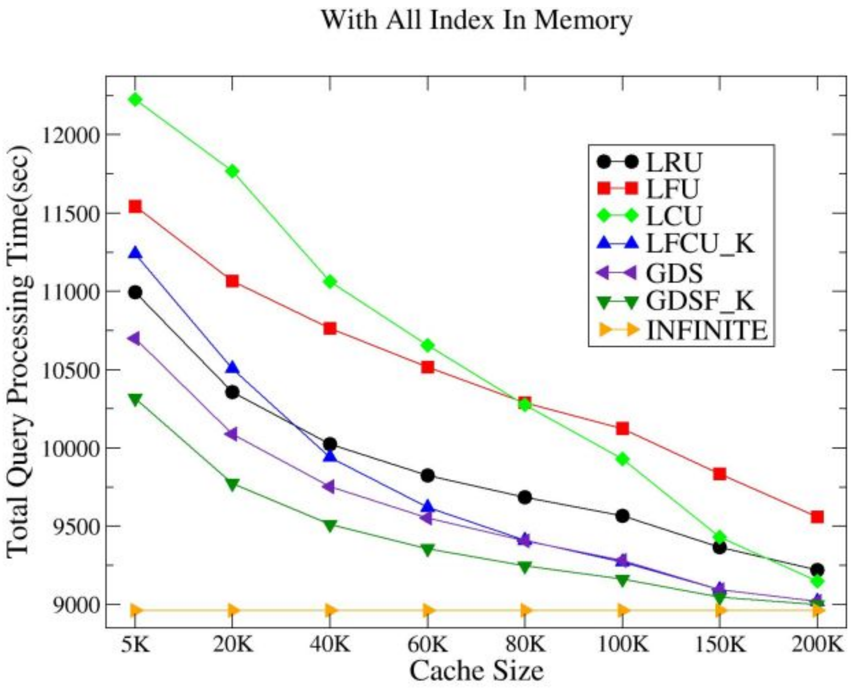
In this section, we experiment with the dynamic caching approaches mentioned in Section 3.4. Note that, LRU and LFU strategies do not use cost values in the replacement policies, while all the other strategies are cost-aware. As a lower bound, we also show the performance of the infinite-sized dynamic cache (INFINITE), for which no replacement policy is necessary. In Figure 3.8, we display the total query execution times obtained using different dynamic caching strategies for the ODP log when 25%, 50%, and 100% of the index is cached. The results of the same experiment for the Webbase and Webbase semantically aligned logs are given in Figure 3.9 only for the case where 100% of the index is cached. The other cases, namely caching 25% and 50% of the index, yield similar results and are not reported here.

In all experiments, the trends are very similar. As in the case of static caching, the improvements are more emphasized as the percentage of index that is cached in the memory increases. Therefore, in the following, we only discuss the case for the ODP log when 100% of the index is cached. First, we see that the cost-aware version of LFU, which is LFCU_K (with K=2, tuned by only using the training log), outperforms LFU in all reported cache sizes. The reductions in total query processing times reach up to 8.6%, 9.4%, and 7.4% for the ODP, Webbase, and Webbase semantically aligned logs, respectively. Although LRU is slightly better than LFCU_K for small cache sizes, the cost-aware strategy performs better for medium and large cache sizes. It is seen that the GDSF_K policy (again with the best-performing value of K tuned by only using the training log, namely, 2.5) is the best strategy among all policies for all cache sizes and all three logs. We achieved up to 6.2%, 7%, and 5.9% reductions in total query processing time compared to the LRU cache for the ODP, Webbase, and Webbase semantically aligned logs, respectively.


With 25% Static List Cache
Total Query Processing Time (sec)
13500
13000
12500
12000
11500
11000
10500
10000
5K
20K
40K
60K
80K
100K
150K
200K
Cache Size
● LRU
■ LFU
◆ LCU
▲ LFCU_K
◀ GDS
▼ GDSF_K
▶ INFINITE
With 50% Static List Cache
Total Query Processing Time(sec)
12500
12000
11500
11000
10500
10000
9500
5K
20K
40K
60K
80K
100K
150K
200K
Cache Size
● LRU
■ LFU
◆ LCU
▲ LFCU_K
◀ GDS
▼ GDSF_K
▶ INFINITE

(a)

(b)

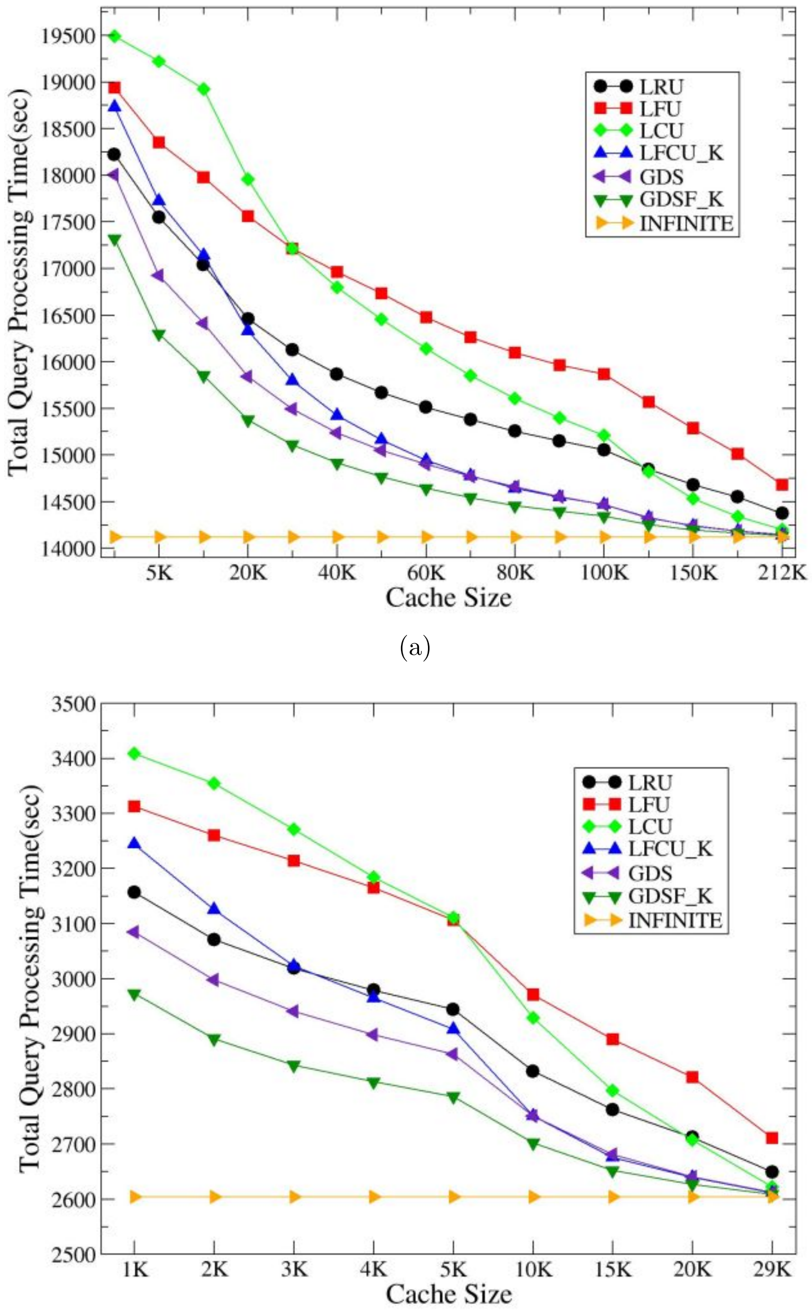


(c)

Figure 3.8: Total query processing times (in seconds) obtained using different dynamic caching strategies for the ODP log when a) 25%, b) 50%, and c) 100% of the index is cached. (Ozcan, R., Altingovde, I.S., Ulusoy, O., “Cost-Aware Strategies for Query Result Caching in Web Search Engines,” ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

3.5.3 Simulation Results for Hybrid Caching

In this section, we experimentally evaluate the hybrid caching approaches. In Figure 3.10, we provide the total query execution times using different hybrid caching strategies for the ODP log when 25%, 50%, and 100% of the index is cached. As before, we report the results for the Webbase and Webbase semantically aligned logs only for the latter scenario, in Figure 3.11(a) and 3.11(b), respectively. The fraction parameter for dividing the cache is tuned experimentally using the training query log. We observe that using a split parameter of 50% yields the best performance for the majority of the cases among the five different hybrid cache configurations given in Table 3.3. So, in all of the experiments reported below, we equally divide the cache space between static and dynamic



(a)

(b)

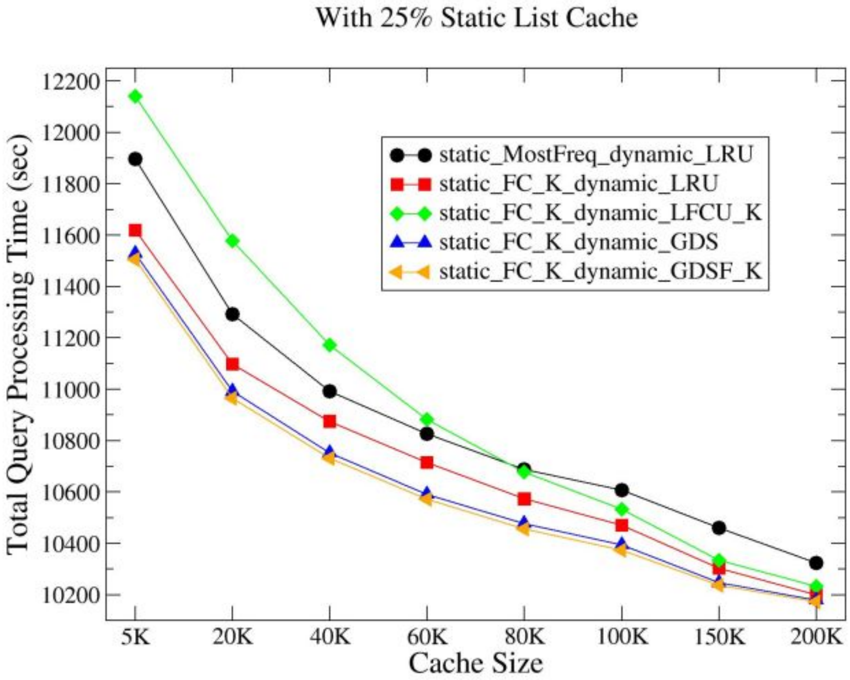
Figure 3.9: Total query processing times (in seconds) obtained using different dynamic caching strategies for the a) Webbase and b) Webbase semantically aligned logs when 100% of the index is cached. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

caches. We see that cost-aware policies also improve performance in the hybrid caching environment. For brevity, we discuss the reduction percentages for the ODP log for the scenario where 100% of the index is cached. The other cases exhibit similar trends with smaller gains.

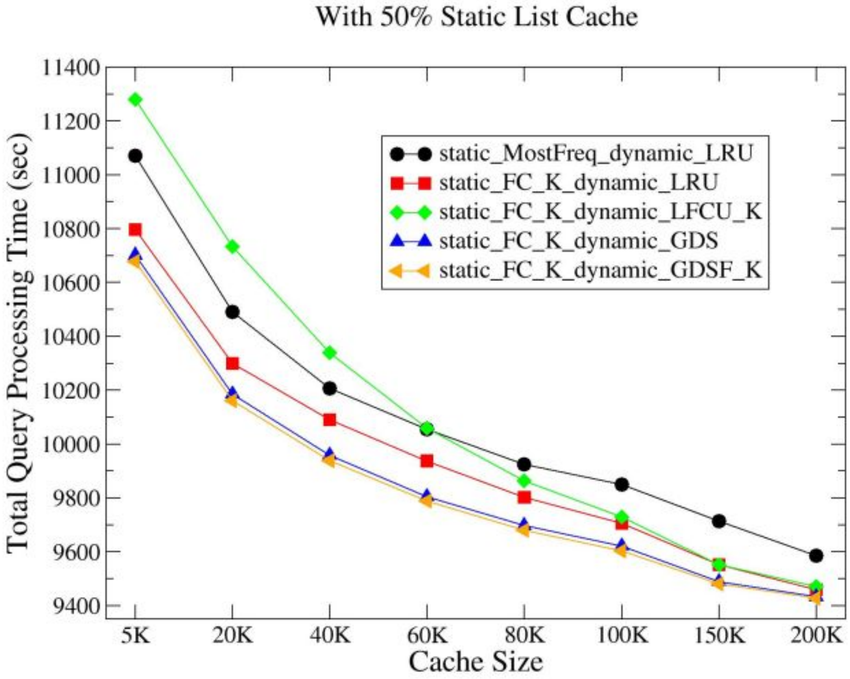
To start with, we observe that it is possible to obtain improvements even when only the static portion of the cache is cost-aware. In Figures 3.10 and 3.11, the static_FC_K_dynamic_LRU case outperforms the baseline, especially for the larger cache sizes. For this case, we achieve up to 2.6%, 3.5%, and 2.6% reductions in total query processing time for the ODP, Webbase, and Webbase semantically aligned logs, respectively. Furthermore, if the dynamic portion also employs a cost-aware cache eviction policy, reductions in total query processing time are more emphasized, especially for the GDSF_K policy (i.e., up to 3.8%, 4.9%, and 3.5% reductions for the ODP, Webbase, and Webbase semantically aligned logs, respectively).

In Figure 3.12, we compare the best performing strategies for static and dynamic caching (namely, FC_K, and GDSF_K) to the hybrid caching that combines both strategies. Our findings confirm previous observations that result caching significantly improves system performance. For instance, even the smallest static cache configuration (including only 5K queries) yields a 14% drop in total query processing time. We also show that hybrid caching is superior to purely static and dynamic caching for smaller cache sizes, whereas it provides comparable performance to dynamic caching for larger cache sizes.

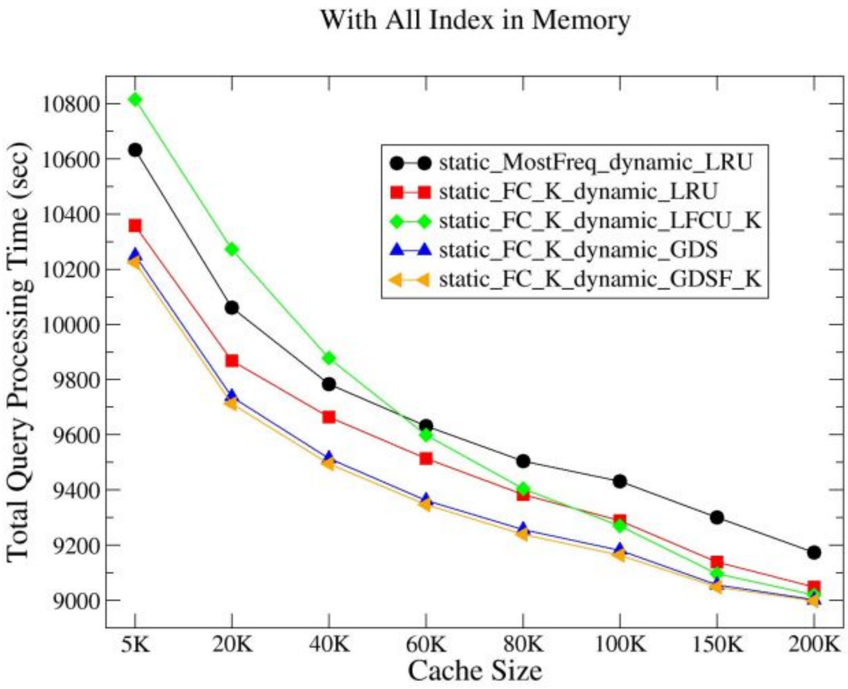
The performance of dynamic caching strategies may suffer from the use of concurrency control mechanisms in a parallel query processing environment. Fagni et al. [30] argue that such cache-access concurrency mechanisms can cause a relatively higher overhead for fully dynamic caching strategies when compared to hybrid strategies, which include a static (i.e., read-only) part. Note that, since the performance gap between the static and dynamic/hybrid strategies is rather large (e.g., see Figure 3.12), it is less probable that this access overhead would make any difference in relative performance of these strategies. For the purposes of this work, we anticipate that the cost of cache-access mechanisms would not


With 25% Static List Cache
Total Query Processing Time (sec)
12200
12000
11800
11600
11400
11200
11000
10800
10600
10400
10200
5K
20K
40K
60K
80K
100K
150K
200K
Cache Size
static_MostFreq_dynamic_LRU
static_FC_K_dynamic_LRU
static_FC_K_dynamic_LFCU_K
static_FC_K_dynamic_GDS
static_FC_K_dynamic_GDSF_K

(a)


With 50% Static List Cache
Total Query Processing Time (sec)
11400
11200
11000
10800
10600
10400
10200
10000
9800
9600
9400
5K
20K
40K
60K
80K
100K
150K
200K
Cache Size
static_MostFreq_dynamic_LRU
static_FC_K_dynamic_LRU
static_FC_K_dynamic_LFCU_K
static_FC_K_dynamic_GDS
static_FC_K_dynamic_GDSF_K

(b)



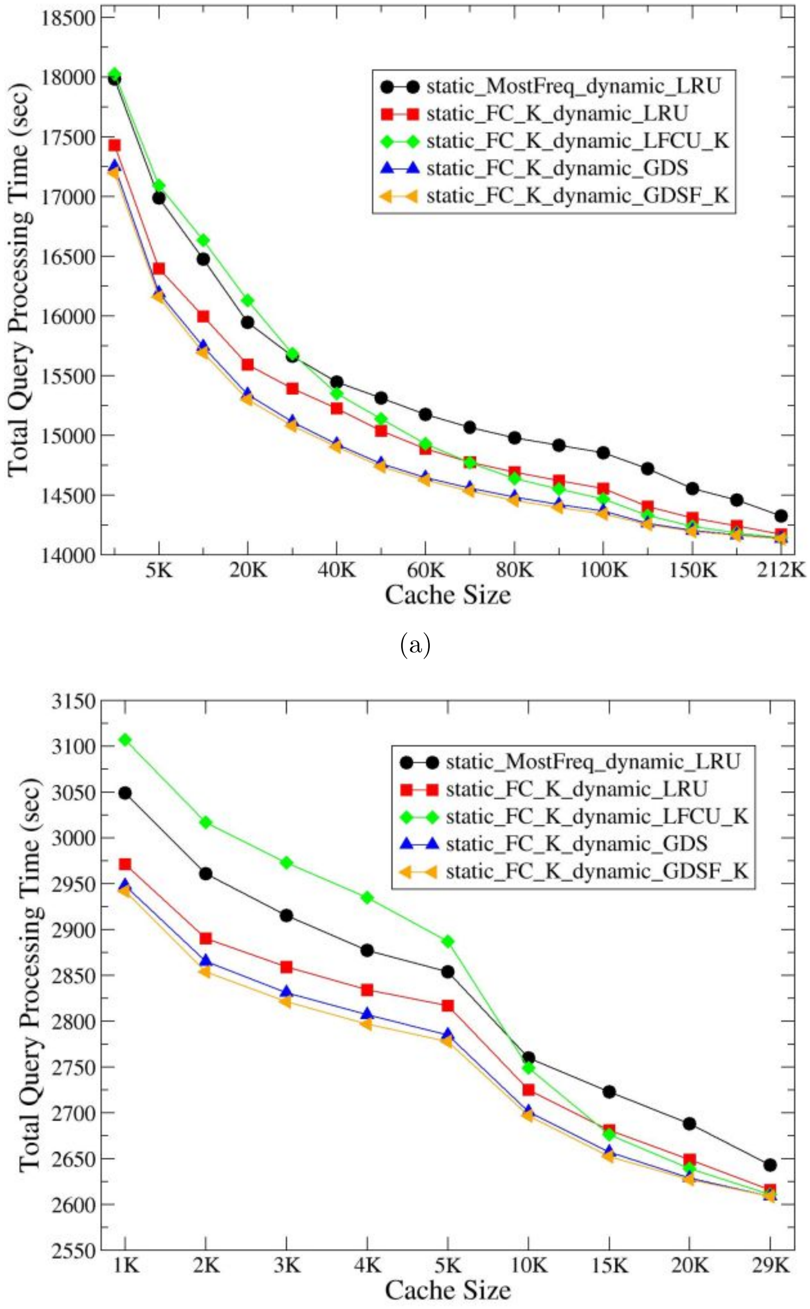
(c)

Figure 3.10: Total query processing times (in seconds) obtained using different hybrid caching strategies for the ODP log when a) 25%, b) 50%, and c) 100% of the index is cached. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

be that significant in comparison to other query processing cost components, and so we do not explicitly consider cache-access overhead in the experiments.

3.5.4 Additional Experiments

In this section, we provide two additional experiments for our cost-aware caching policies. Note that, in all simulation results reported previously, we measure CPU execution times when the queries are processed in the conjunctive mode (i.e., "AND" mode). In a set of additional experiments, we also measure the query execution times in the disjunctive (OR) mode. Figure 3.13 provides the simulation result for dynamic caching with the ODP log when the entire index is



(a)

(b)

Figure 3.11: Total query processing times (in seconds) obtained using different hybrid caching strategies for the a) Webbase and b) Webbase semantically aligned logs when 100% of the index is cached. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

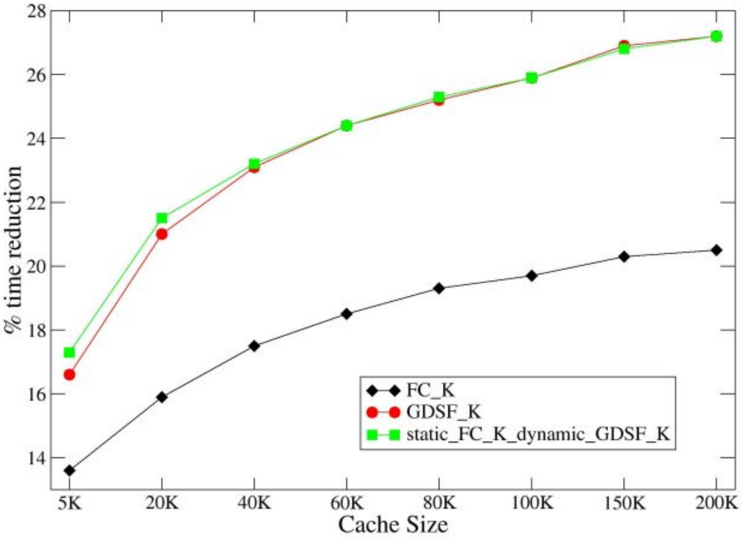


Figure 3.12: Percentages of time reduction due to caching using best static, dynamic, and hybrid approaches for the ODP log when 100% of the index is cached. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

assumed to be cached. The other cache types and query logs are not discussed to save space. Notably, the overall query processing times in Figure 3.13 are longer than those of the corresponding case with the conjunctive processing, as expected (please compare with the plot in Figure 3.8(c)). We also see that all trends are the same, except the LCU strategy performs slightly better in the conjunctive mode.

As a final experiment, we analyze the average response time of queries under different query loads for several caching methods. In this simulation, we assume that the time between each query submission follows an exponential distribution as shown in previous works (e.g., [19]). In particular, we vary the mean query inter-arrival time between 50 ms and 500 ms, corresponding to high and low workload scenarios, respectively. We also assume that the search system involves two processors. In Figure 3.14, we provide average response time figures for dynamic caching strategies (namely LFU, LRU, LFCU_K and GDSF_K), and only for the case where all index is stored in memory, for the sake of simplicity.

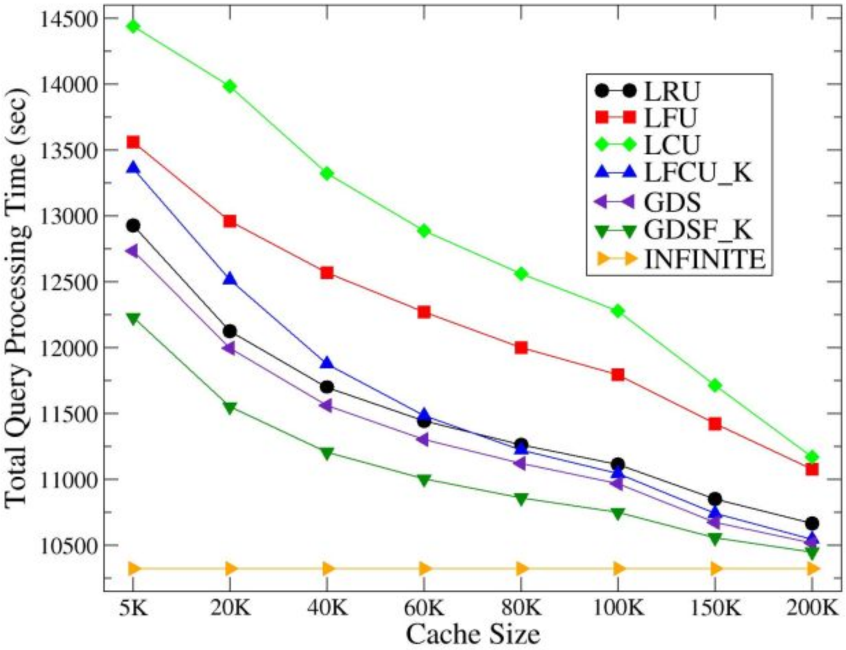


Figure 3.13: Total query processing times (in seconds) obtained using different dynamic caching strategies for the ODP log when queries are processed in the disjunctive processing mode. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

Our findings reveal that cost-aware strategies also improve the average response time under different load scenarios. Note that, our results presented here are not conclusive, and we leave an in-depth investigation of response time related issues as a future work.

3.6 Conclusion

In this chapter, we justify the necessity of cost-based caching strategies by demonstrating that query costs are not uniform and may considerably vary among the queries submitted to a search engine. We propose cost-aware caching strategies for the query result caching in web search engines, and evaluate it for static,

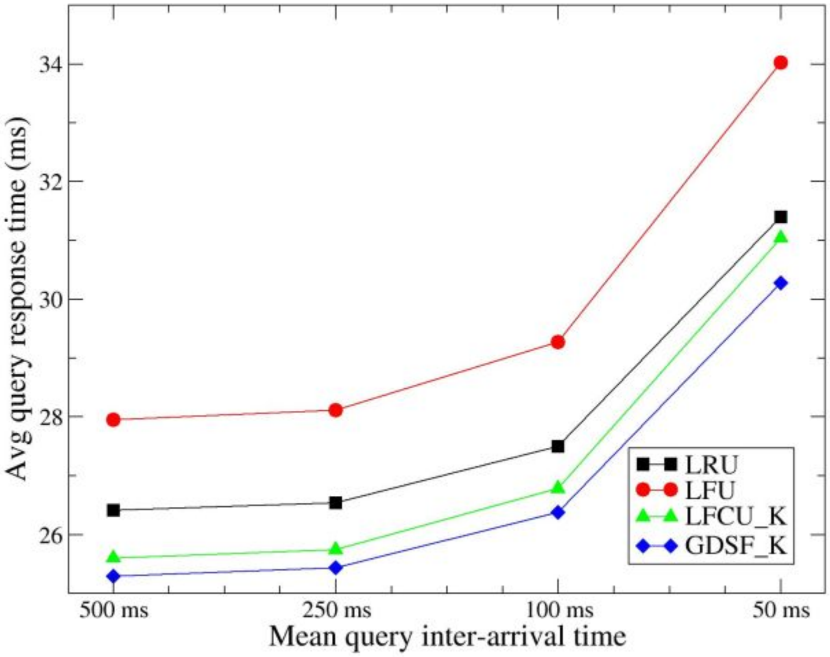


Figure 3.14: Average query response time obtained using different caching strategies for various query workloads (simulated by the different mean query inter-arrival times) of the ODP log. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Cost-Aware Strategies for Query Result Caching in Web Search Engines," ACM Transactions on the Web, Vol. 5:2, ©2011 ACM, Inc. http://dx.doi.org/10.1145/1961659.1961663. Reprinted by permission.)

dynamic, and hybrid caching cases. For static caching, we incorporate cost-awareness into the static caching policy introduced in [62]. For dynamic caching, we propose two cost-aware policies, namely LCU and LFCU_K, and show that especially the latter strategy achieves better results than its non-cost-aware counterpart and the traditional LRU strategy. We also show that cost-aware policies such as GDS and GDSF_K, as employed in other domains, perform well in query result caching. Finally, we analyze the performance of the cost-aware policies in a hybrid caching setup such that one portion of the cache is reserved for static caching and the other portion for dynamic caching. We experiment with several different alternatives in this setup and show that if both static and dynamic portions of the cache follow a cost-aware caching policy, performance improvement is highest.

We observe considerable reductions in total query processing time (i.e., sum of the CPU execution and disk access times) for all three caching modes. In the static caching mode, the reductions are up to around 3% (in comparison to the classical baseline, i.e., caching the most frequent queries). In the dynamic caching mode, the reductions are more emphasized and reach up to around 6% in comparison to the traditional LRU strategy. Finally, up to around 4% improvement is achieved in a hybrid, static-dynamic, caching mode. Thus, for all cases, the cost-aware strategies improve the state-of-the-art baselines.

Chapter 4

Query Intent Aware Result Caching

In web search engines, query results are cached and presented in terms of pages, which typically include 10 results each. In navigational queries, users seek a particular website, which would be typically listed at the top ranks (maybe, first or second) by the search engine, if found. For this type of query, caching and presenting results in the 10-per-page manner may waste cache space and network bandwidth. In this chapter, we propose non-uniform result page models with varying numbers of results for navigational queries. The experimental results show that our approach reduces the cache miss count by up to 9.17% (due to better utilization of cache space). Furthermore, bandwidth usage, which is measured in terms of number of snippets sent, is also reduced by 71% for navigational queries. This means a considerable reduction in the number of transmitted network packets, i.e., a crucial gain especially for mobile search scenarios. Our user study reveals that users easily adapt to the proposed result page model and that the efficiency gains observed in the experiments can be carried over to real-life situations.

The rest of this chapter is organized as follows. In Section 4.1, we provide the motivation for our work and list our contributions. Section 4.2 presents related work on identifying user search goals, with special emphasis on the works about navigational queries. We then define and discuss alternative result page models for navigational queries and evaluate their presentation costs in Section 4.3. The effects of non-uniform page models on caching are evaluated in Section 4.4. Section 4.5 presents the results of the user study, in which user browsing behavior is observed regarding non-uniform result pages. Finally, we conclude the chapter in Section 4.6.

4.1 Introduction

Web search engines answer millions of queries per day to satisfy the information needs of their users. In the literature [18, 72], two essential goals for web searches are identified as "informational" or "navigational." Additionally, some other classifications of user search intents including transactional [18] and resource queries [72] are also proposed. In the context of this study, we focus on the navigational queries and broadly refer to all queries that are not navigational as informational. To illustrate the difference between these categories, consider the following two queries as examples of informational and navigational types, respectively: "caching in web search engines" and "united airlines." The user asking the first query is trying to find information on caching in web search engines and (s)he does not have a specific website in mind. Therefore, (s)he may browse several result pages, read the snippets and decide to visit many different web pages. On the other hand, the user asking the second query is most probably looking for United Airlines' company website. In this case, if the search engine finds the target page, it can be expected to be among the top-ranked results since Liu et al. [50] note that information retrieval systems have much better effectiveness for navigational queries than informational queries. Thus, the user would typically glance over only a few of the top-ranked URLs and not attempt to visit alternative pages, other than the target page in his/her mind. The difference between user intention and behavior is clear: in the navigational case, the user is only interested in accessing the particular web page that (s)he knows or expects to exist (e.g., www.united.com for this example), and not any other results related to the topic of the query (e.g., financial news about United Airlines, etc.). The previous works in the caching literature [30, 47] assume that query results are cached as typical pages that include 10 results for all queries.

Web search engines, regardless of the user's search goal, present and cache the results in terms of pages, each of which includes a fixed number (typically 10) of results. By result page, we mean the URLs and snippets of the documents in the query result [30] that is necessary to construct the final HTML output. The visual content of the page (e.g., with ads, etc.) are not considered. This uniform result page model may be appropriate for informational queries, but for navigational queries, caching and presenting results in this 10-per-page manner may waste cache space and network bandwidth. In other words, for navigational queries, it may be possible to provide, say, fewer than 10 results on the first result page and still satisfy the user, while saving the valuable resources mentioned above.

In this chapter, we propose to use non-uniform result page models, i.e., pages with varying number of results, for navigational queries. By doing so, our goals are to reduce network bandwidth usage during the result page presentation, and to best utilize the cache space at search engine side. To our knowledge, this is the first work that explores the display and caching of query result pages with varying granularities (number of results). More specifically, the contributions of our work in this chapter are as follows:

1. We define a cost model for the result page models and investigate the bandwidth utilization of non-uniform result page models for presenting up to the top-20 results. We restrict ourselves to the top-20 results since, for the vast majority of web queries, at most top-20 results are inspected by users [66]. Using the cost model and a large query log, we experimentally show that non-uniform result page models considerably reduce the number of snippets transmitted to the user and hence improve bandwidth usage for navigational queries. This reduction in bandwidth usage is more crucial for mobile search scenarios with low bandwidth capacities. Considering that

a significant portion of mobile queries are navigational [24], this improvement will be an important contribution for this field. In particular, our findings reveal that presenting top-20 results in three pages containing two, eight, and 10 results, respectively, would be the most efficient model for bandwidth consumption.

2. We investigate and evaluate the gains of using non-uniform result page models in web search engine caching. We adapt a realistic framework that involves a hybrid result cache (composed of static and dynamic parts) along with an adaptive prefetching mechanism [30]. Our experimental findings show that the proposed model improves cache space utilization and subsequently reduces the number of cache misses.

3. For our proposal to make sense in practical settings, it is crucial to observe the reaction of users for the non-uniform result page models since they are accustomed to seeing a constant (e.g., 10) number of results per page. That is, in a navigational search scenario, if the user keeps on browsing successive result pages even after (s)he finds the target page, this would diminish the efficiency promises of our proposal. We explore whether the proposed model fits the real-life browsing behavior of the users by conducting a user study. The study reveals that users easily adapt to non-uniform result page models and interact with the web search engine as expected, i.e., do not look further result pages if they are satisfied with the current result page. This means that the efficiency gains observed in the experiments are realistic and can be carried into real-life situations.

4.2 Related Work on Identifying User Search Goals

User search goal identification is well explored in the literature. Early works [18, 72] analyze query logs and classify different user search goals manually. Later works [4, 5, 31, 35, 36, 43, 44, 45, 50, 53, 55, 56] in this area focus on the automatic identification of user search goals by several machine learning techniques that exploit various features. The majority of these works essentially evaluates the accuracy of determining the user search's intent and proposes some future directions for the utilization of this knowledge. That is, there are relatively few works as ours that build upon these automatic identification methods. In particular, only two of the above works [31, 43] devise different ranking algorithms based on the user's search intent. As an alternative use case, Rose [71] suggests that different user interfaces (or different forms of interaction with the users) should be provided to match users' different search goals. An illustrative study in this direction is by Kofler and Lux [44]). In their work, once users' search goals are predicted, the results (digital photos in their case) are displayed accordingly, e.g., in either thumbnail or list view.

In our work, we do not aim to propose another identification method. Therefore we adopt a simple and effective approach from Liu et al. [50] 's work. They propose two features extracted from click-through data for user intent classification. These features are N clicks satisfied (nCS) and top-n results satisfied (nRS). nCS measures the frequency of query submissions such that fewer than n clicks are performed in those cases. Similarly, nRS measures the frequency of query submissions such that all clicks in these submissions are within top-n results. A decision tree classification exploiting these features and Lee et al. [45]'s click distribution feature achieves 85% F-measure for navigational queries. Brenes et al. [17] surveys several methods for user intent classification and evaluates their accuracy using a set of 6,624 manually labeled queries. It is found that Liu et al. [50]'s nRS feature gives the best accuracy for navigational queries. As noted in this survey, a combination of various features increases the accuracy. Therefore, in this study we also decide to use a complementary method proposed by Jansen et al. [36]. In this work, navigational queries are identified by a set of heuristics such as whether it contains person/organization/company names and query length is less than three, etc.

Lu et al. [53] propose to use thousands of features for navigational query identification. They claim that for navigational queries, "presentation of the search results or the user perceived relevance can be improved by only showing the top results and reserving the rest of [the] space for other purposes." In this chapter, we explore different result page models for navigational searches to improve system efficiency, i.e., to best utilize the bandwidth and cache space.

Piwowarski and Zaragoza [67] try to predict the possibility of the user clicking on a particular result for a query. They achieve over 90% accuracy for navigational queries. They suggest that if it can be determined that it is highly likely that a user will click on a certain result, the snippet for that result could be highlighted or the user could be directly sent to the page for that result. Our approach is in the middle of these extremes. Showing the top results on the first result page is a more conservative approach than directly forwarding the user to the top result page, but it is less conservative than just highlighting the snippet.

Teevan et al. [79] examine the re-finding behavior of users by considering repeat queries in Yahoo!'s logs. Their results show that navigational queries constitute a significant portion of repeat queries. The authors suggest that the web search engine designers should take this re-finding behavior into account when designing user interfaces. They propose that a history of a user's past queries would be helpful. They also propose providing direct links labeled with the most frequent query term, to websites for users who issue a high number of navigational queries. Their findings confirm our motivation for this study, since we also propose a special treatment for navigational query types.

Finally, it is possible that some commercial search engines may already be treating different query types in different manners. For instance, Google's Browse by Name facility[1] allows users to type queries into the address bar and if the system determines the site that the user wants to go to, it directly forwards him/her to that site. Otherwise it gives the usual result page for the query. A recent work by Tann and Sanderson [78] shows that some informational queries become partly navigational nowadays. For example, many users who issue an informational query about a film actor want to see the page in internet movie database (IMDB) website or Wikipedia website. This work also shows that web

[1] http://www.google.com/support/toolbar/bin/answer.py?hl=en&answer=9267.

search engines take this user expectation into account and show the results from IMDB or Wikipedia at high ranks. However, the details of such technologies are not publicly available.

4.3 Result page models for navigational queries: Cost analysis and evaluation

Navigational queries constitute a non-trivial portion of web search queries [36]. It is hard to develop a result page model tailored to informational queries, as user behavior is unpredictable. Even if the search is successful, the user may click on several results or even request another result page to learn different aspects of the topic in question. On the other hand, those who submit navigational queries, mostly aim to obtain the address of a target site. For such a query, a successful search would return the target result at the highest ranks because Liu et al. [50] note that information retrieval systems have much better effectiveness for navigational queries than informational queries. Subsequently, the user would be expected to click on only the top few results, and no more. Therefore, it might be beneficial to show the top few results on the first result page for navigational queries. Note that this choice of result presentation for navigational queries is also confirmed by previous works on user browsing behaviors by Joachims at al. [42] and Dupret and Piwowarski [28]. In these works it is observed that in almost all cases users check the first two query results right after the result page is displayed. If these results are not satisfying, then those at the lower ranks are examined. These earlier findings imply that it may be possible to present and cache the results of navigational queries in a more efficient manner, i.e., by presenting only a few results on the first result page. In this section, we first develop a model for evaluating the presentation cost of query results in terms of the network usage and user browsing process. This cost model is used to determine whether there exists a better way of presenting the results of a navigational query to the user than the typical 10-results-per-page approach. We define some of the basic notions and introduce measures used in our cost model as follows:

Query Instance: A query submission by a user to the search engine at a specific time. This also includes activities after query submission such as browsing the result pages. It ends when the user submits another query or the query session expires, i.e., the user does not perform any activity for 30 minutes.

Click Requests: The set of clicks performed by the user after query submission. The user browses the returned results and clicks on the ones that seem to be relevant to his/her information need.

Result Page Model: A result page is an atomic item for internal (e.g., result caching) and external (e.g., result display) purposes of the web search engine. A result page model describes how the query results are placed into the pages, each of which may include a fixed (uniform model) or variable number of results (non-uniform model).

numSnippetsSent: A measure of the number of snippets sent by the web search engine to the users. It shows the network bandwidth cost incurred by the query result display. The ideal result page model must minimize this quantity.

numResultPagesBrowsed: Indicates the total number of result pages that the user browses in order to reach the target document(s) for his/her query. The ideal result page model should minimize the number of result pages browsed. This quantity also corresponds to the number of requests that must be handled by the web search engine.

Note that the underlying objectives of the last two measures conflict with each other. In an extreme case, the result page model could show one result per result page. This model would guarantee that the number of snippets sent would be minimal but it would also maximize the number of result pages browsed. At another extreme, the search engine may show all the results (e.g., top-100 or top1000) in one result page. While minimizing the number of the result pages browsed to only one, this model maximizes the number of snippets sent to the user. In what follows, we introduce a cost model based on these measures to find the optimal result page model for navigational queries.

4.3.1 Cost Analysis of the Result Page Models

In this section, we present a practical cost analysis model for the result page models with varying granularities. Various previous studies [13, 38, 39] agree on the finding that users rarely click on more than the top-20 results, so without loss of generality we restrict our analysis to the result page models for this most common case. We explore how the top-20 results can be "paged/organized" to optimize the measures (*numSnippetsSent* and *numResultPagesBrowsed*) introduced above.

Consider a query instance Q and the click requests $C = \{c_1, c_2, \ldots, c_k\}$ of k clicks for this query. Assume that click requests are at the ranks $R = \{r_1, r_2, ..., r_k\}$, where $r_i \leq r_{i+1}$ and $r_k \leq 20$. Then the result at rank r_k is defined as the *lowest-ranked clicked document* for this query instance. For simplicity, we assume that the user requested all the result pages up to the result page containing the result at rank r_k and that the user will not request more results after this rank. This assumption agrees with the "cascade model" proposed by Craswell at al. [25] for user click behavior. The cascade model assumes that users view each query result in a linear order, from top to bottom, and decide to click on it or not for each result. The users do not examine documents below a clicked result according to this model. Based on these assumptions, we collect all $< Q, r_k >$ pairs for the top-20 clicks. Then, we obtain the list $A = \{A_1, A_2, \ldots, A_{10}, A_{11}, A_{12}, \ldots, A_{20}\}$, where A_i denotes the number of query instances for which the lowest-ranked clicked document is at rank i (i.e., $r_k = i$).

Next, we can derive formulas for *numSnippetsSent* and *numResultPagesBrowsed* measures using this list. Assume we have a two-page result model for top-20 as X_Y, which denotes that the first result page contains X results and the second result page contains Y (or 20-X) results. For this case, the formulas for *numSnippetsSent* and *numResultPagesBrowsed* are presented below:

$$numSnippetsSent_{X\_Y} = X \times \sum_{i=1}^{20} A_i + Y \times \sum_{i=X+1}^{20} A_i \qquad (4.1)$$

$$numResultPagesBrowsed_{X\_Y} = \sum_{i=1}^{20} A_i + \sum_{i=X+1}^{20} A_i \tag{4.2}$$

Equation 4.1 expresses that we have to send the first result page containing X number of snippets for all query submissions in the list A (captured with the first summation) but the second result page containing Y snippets will only be sent to query submissions that also request results at the rank $X + 1$ or more (expressed with the second summation). Similarly, in Equation 4.2, the first summation accounts for the first result page that is clearly browsed for all queries and the second summation counts the number of queries that ask for the second page of results.

The objective of the result page model is to minimize these two quantities. We have to normalize these expressions in order to find a result page model that minimizes the overall cost. To this end, we use the conventional model of 10-results-per-page schema (10_10) as our baseline model and normalize the above expressions as follows:

$$numSnippetsSent\_norm_{X\_Y} = \frac{numSnippetsSent_{X\_Y}}{numSnippetsSent_{10\_10}} \tag{4.3}$$

$$numResultPagesBrowsed\_norm_{X\_Y} = \frac{numResultPagesBrowsed_{X\_Y}}{numResultPagesBrowsed_{10\_10}} \tag{4.4}$$

Note that we do not expect any two-page result model to achieve better figures than the baseline model of 10_10 for both of the above measures. That is, a model X_Y will decrease one quantity at the cost of an increase in the other quantity. However, the question is whether it is possible to find a model better than the baseline for the overall case. For this purpose, the summation of the normalized values in Equations 4.3 and 4.4 can be used as an overall measure for the result presentation models (see Equation 4.5). Note that, it is possible to have different weightings for the components in Equation 4.5 for different objectives. For example, in a mobile search scenario, the bandwidth savings might be more important and therefore the weight of *numSnippetsSent* can be higher. Since the value of such a sum for the baseline model is 2, we can use the following formulas to calculate the overall improvement percentage of the result page models over the baseline model in the experiments:

$$overall_{X\_Y} = numSnippetsSent\_norm_{X\_Y} + numResultPagesBrowsed\_norm_{X\_Y} \tag{4.5}$$

$$\%improvement_{X\_Y} = \frac{2 - overall_{X\_Y}}{2} \times 100 \tag{4.6}$$

Note that, although we restrict the discussion in this section to the top-20 results and result page models with two pages, the cost formulations can be generalized for top-K results and M pages. In the experimental evaluations, we consider models involving two and three result pages for the top-20 results.

4.3.2 Evaluation of the Result Page Models

4.3.2.1 Experimental Setup

We use the AOL Query Log [66] that contains around 20 million queries of about 650K people for a period of three months. We exclude query instances that do not have any clicks (In AOL log, it is not possible to determine exactly which result pages are viewed by the users without any knowledge of clicks). Our subset contains 10,733,457 query instances that have at least one click. Among those queries, 5,853,929 are submitted in the first 6 weeks of three months and reserved as the training set. This set is used to determine the navigational queries and fill the cache (discussed in the next section). The remaining 4,879,528 queries constitute the test set, which will be used to evaluate the cost of the result page models. We first present the characteristics of the training query log. Figure 4.1 shows the number of queries that have the lowest-ranked clicks at a given rank, i.e., it illustrates the list A described in the previous subsection. Note that in

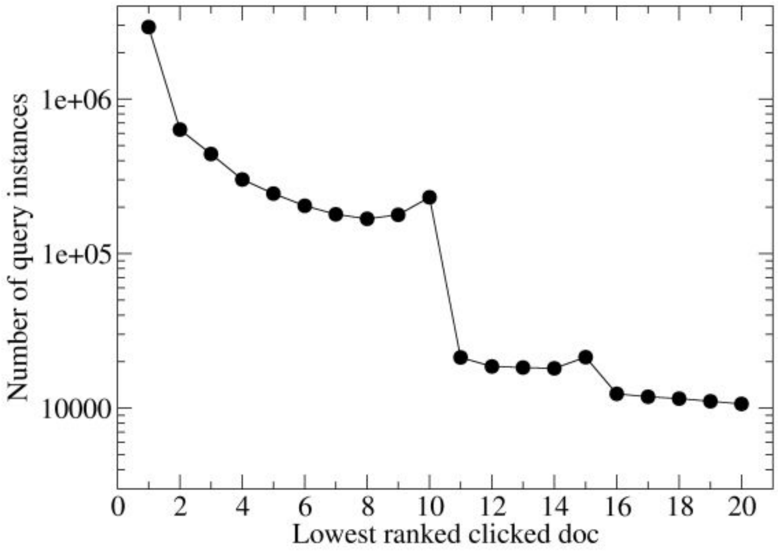


Figure 4.1: Log graph showing number of query instances of which the last click is at a given rank (for training log). (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

our query log 93.7% of all clicks are for results in the top-20, justifying our use of those results in this study. The cut between ranks 10 and 11 is clearly visible, which indicates that the number of requests for results on the second page is an order of magnitude less than the number of requests for the results on the first page.

4.3.2.2 Navigational Query Identification

As mentioned in the related work section, there are several methods proposed for automatic navigational query identification in the literature. Since the proposal of another method for this purpose is not among the objectives of this study, we adopt a simple and effective approach from Liu et al. [50]'s work and slightly extend it for more flexibility. We also combine Jansen et al. [36]'s method into our approach for higher accuracy. The first stage of our method identifies navigational queries by using the click distribution such that if 90% of the time users clicked up to rank two for that query[2], it is classified as a navigational query. However, as we will see later, this definition is too restrictive; so we have an additional heuristic as follows: For a query, we define the notion of confidence in navigational query identification as in Equation 4.7. In this formula, $f_{Q,Top2}$ denotes the frequency of query instances of query Q, in which users clicked up to rank two; the right side of the multiplication represents the log normalized frequency of query Q. Here, f_Q is the frequency of query Q in the training set and f_{MAX} is the maximum frequency value in the training set.

$$Confidence = f_{Q,Top2} \times \frac{\log(f_Q + 1)}{\log(f_{MAX} + 1)} \tag{4.7}$$

We also use the above confidence measure for identifying navigational queries. In particular, we call those queries with $f_{Q,Top2}$ greater than 80% and a confidence score greater than 0.2 navigational queries, as well. The intuition underlying the components of this confidence score, namely $f_{Q,Top2}$ and query frequency, is as follows: In navigational queries, users click on only one or two result document most of the time. Therefore, $f_{Q,Top2}$ is an important indicator in determining whether a query is navigational or not. Second, we use query frequency since it is a general rule in machine learning that as training size increases, classification accuracy also increases. Therefore, the queries with a high occurrence frequency and high $f_{Q,Top2}$ are identified as navigational queries with a higher confidence score. Finally, we exclude the queries that occur only once in the training log regardless of the above considerations, except those that include domain suffixes (www, com, edu, org, etc.), since Jansen et al. [36] report that the existence of such suffixes in the query is an important characteristic of navigational queries. In a recent study by Lee and Sanderson [46], it is found that 86% of URL queries (i.e., those that consist of full or partial URLs) are navigational.

In the second stage of our approach, we further apply Jansen et al. [36]'s method for those queries that cannot be identified as navigational by the first

[2] Note that this corresponds to the nRS feature for n=2 in Liu et al. [50].

stage. In this method, navigational queries are identified based on a set of rules derived from characteristics of those queries. Jansen et al. list the characteristics of navigational queries as following: a) containing company, organization and people names; b) having less than three terms; c) including domain suffixes (com, edu, etc.); d) viewing only the first result page. We used the freebase database[3] as also applied in [17] in order to get the list of company, organization and people names. We obtained 1,579K people names, 492K organization names, and 89K company names. We check whether the query contains any of these names and also satisfy the characteristics listed above in order to classify it as navigational. In our training log, we discover that among the 2,778,591 distinct queries, 446,026 of them are navigational queries. When we consider the occurrence frequency of navigational queries, they constitute 32.5% of the query log. Three annotators manually inspected 500 randomly chosen queries that are identified as “navigational” in order to measure the precision. It is found that 416 are correctly identified, which corresponds to 83.2% precision. There was a disagreement among three annotators for only 72 queries, and so they agree in 85.6% of the queries. When we consider the query occurrence frequency, the precision increases to 87.9%, such that out of 1,418 occurrences of 500 distinct queries, 416 correctly identified navigational queries constitute 1,246 submissions. This shows that identification is highly accurate for frequent queries but less accurate for rare queries. We further observe that there are some informational queries for which users mostly click on the top two results. We envision that the non-uniform result page models that we propose in this chapter can also serve well for such queries.

4.3.2.3 Result Page Model Experiments

In this set of experiments, our aim is to compare alternative result page models for navigational queries based on numSnippetsSent and numResultPagesBrowsed measures defined previously. We use the training set to find the navigational queries as mentioned in the previous subsection. We process the test set and

[3] http://www.freebase.com/

for those queries that are identified as navigational in the training set, we apply the result presentation model X_Y (The first page contains X results/snippets and the second page contains Y (or 20-X) results). We use the uniform result presentation model (i.e., 10_10) for informational queries.

As we mentioned in Section 4.3.1, we perform experiments for two page and three page result models for the top-20 clicks. Figure 4.2 shows the graph for the normalized cost of two page models (X_Y) ranging from 1_19 to 19_1 for navigational query types (those queries in the test set that are identified as navigational query in the training set). As expected, models left to the 10_10 baseline approach achieve lower numSnippetsSent but higher numResultPagesBrowsed values than the baseline. On the other hand, models to the right of the baseline provide improvements in terms of the browsed result pages but increase the number of snippets sent.

The best result presentation model is 3_17, which has the normalized numSnippetsSent value as 0.43 and the normalized numResultPagesBrowsed as 1.08. The overall improvement for this model is 24.5% (excluding the cost of the informational query type). If we consider the improvement in the existence of informational queries such that we use the 3_17 model for navigational queries and the baseline 10_10 model for the remaining queries, then the overall improvement becomes 8.8% (numSnippetsSent = 0.80 and numResultPagesBrowsed = 1.025, such that a 20% decrease in the number of snippets sent occurs at the cost of a 2.5% increase in the number of result pages browsed.).

Experiments with three page result presentation models are also conducted but the results for all 171 combinations are not reported due to space considerations. Here we mention only the best models achieved. In this case, the best model is 2_8_10; that achieves a normalized numSnippetsSent value of 0.29 and a normalized numResultPagesBrowsed of 1.11. The overall improvement for this model is 29.5% (excluding the cost of the informational query type). If we consider the improvement in the existence of informational queries such that we use the 2_8_10 model for navigational queries and the baseline 10_10 model for the remaining queries, then the overall improvement becomes 10.1% (numSnippetsSent

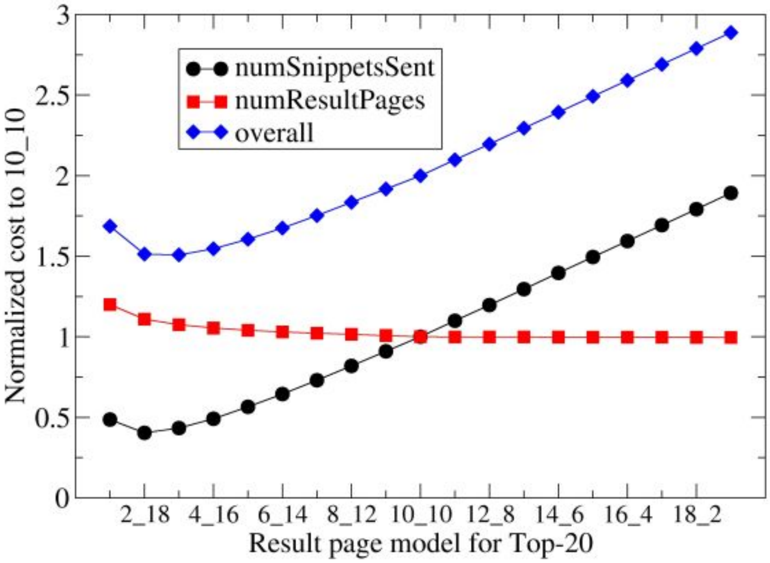


Figure 4.2: Graph showing the costs of two page result presentation models for only navigational queries. (Ozcan, R., Altingovde, I.S., Ulusoy, O., “Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines,” Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

= 0.76 and numResultPagesBrowsed = 1.039, such that a 24% decrease in the number of snippets sent occurs at the cost of a 3.9% increase in the number of result pages browsed). In the rest of this chapter, we use the 2_8_10 model for navigational queries unless stated otherwise.

4.4 Caching with the result page models and experimental evaluation

4.4.1 Employing Result Page Models for Caching

In this section, we examine the effect of using a non-uniform result page model for navigational queries on the caching mechanism in web search engines. We adapt a hybrid caching framework proposed by Fagni et al. [30], in which some part of the cache is reserved for static caching and the remaining part is reserved for dynamic caching.

The static cache is populated with the most frequent query results. In the traditional case, query results consist of result pages of 10 results. The static cache is filled with the most frequent $< query, result\_page\_no >$ pairs in the training query log. Since the size of the result pages is the same in this case, considering frequency would be enough. However, as in our case, we have pages of two results, eight results, and ten results. The size of the cached items must be taken into account in addition to the frequency. Since snippets are large enough to dominate the size, we assume that the page sizes are directly proportional to the number of results, i.e., the size of the first page of a navigational query is only 20% of the size of a 10-results page, and the second page is 80%. Based on these considerations, the query result pages are ordered by the following score formula (adapted from [6]) and the cache is populated with the items having the highest scores.

$$Score_{<query,result\_page\_no>} = \frac{FREQ_{<query,result\_page\_no>}}{SIZE_{<query,result\_page\_no>}} \tag{4.8}$$

In the dynamic part of the cache, we employ the LRU replacement policy that orders cache items based on their last usage time and when the cache is full evicts the one that has been requested least recently. Note that we do not attempt to identify navigational queries dynamically since our identification method essentially requires the click frequencies of queries, which are obtained from previous query logs in an offline manner. Therefore, for all query types, the result pages are stored in the uniform 10-per-page manner in the dynamic cache. The only exception is for those navigational queries of which the top-2 results are cached in the static cache. In this case, if the user requests the second result page, i.e., results between ranks three to 10, this page should also be inserted into the dynamic cache. Thus, the dynamic cache would include pages of size 10 or eight in our setup.

4.4.2 Experiments

4.4.2.1 Experimental Setup

We perform our experiments in the static-dynamic caching environment proposed by Fagni et al. [30]. It is important to decide what portions of the cache are reserved for static or dynamic caching for the best hit ratio. As noted in Fagni et al. [30], the best fraction value is based on the query log. The fraction of cache space reserved for the static cache is denoted as f_s; the cache with $f_s = 0$ corresponds to a purely dynamic cache and the cache with $f_s = 1.0$ corresponds to a purely static cache.

Figure 4.3 presents hit ratios of static-dynamic caches with different f_s values, ranging from purely dynamic to purely static for small (50K), medium (500K), and large (2500K) cache sizes. In the experiments reported throughout this section, the cache size is expressed in terms of the number of typical result pages in the cache. For small or medium cache sizes, most of the cache space must be reserved for the static cache, but for large cache sizes the dynamic cache portion must dominate the cache. For our query log, a static-dynamic cache with $f_s = 0.8$ gives the best hit ratio for most of the cache sizes (It has the best hit ratio for small and medium cache sizes and very close to the best ratio for the large cache size, as shown in Figure 4.3). We therefore perform caching experiments with this cache configuration.

Second, prefetching query results is an important mechanism that increases

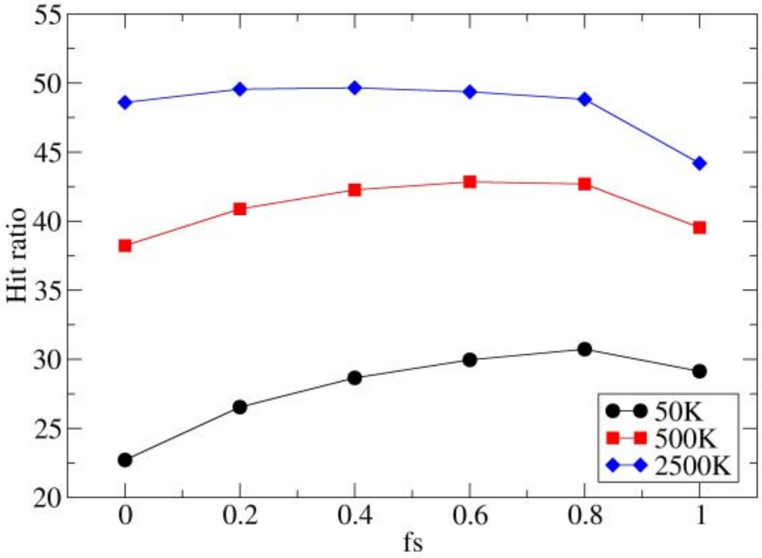


Figure 4.3: Hit ratios vs. f_s (fraction of the cache reserved for static cache); $f_s = 0$: purely dynamic, $f_s = 1$: purely static cache. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

the cache hit ratio, as noted in Fagni et al. [30]. Normally, when a user requests a page of results and it is not in the cache, only the requested page of results is brought into the (dynamic) cache. If the user asks for the next page of results and it is not in the cache, the same process is repeated. In the case of prefetching, if the requested page is not in the cache, instead of just one page, successive F (prefetching factor) result pages (including the requested page) are brought into the cache. Fagni et al. [30] proposed an adaptive prefetching policy that remarkably increases the prefetching performance. This method prefetches F successive result pages only if the miss occurs for a page other than the first result page. When a miss occurs for the first result page, only the second result page is prefetched (for details, see Table V in [30]). We applied the Fagni et al. [30]'s adaptive prefetching policy in our experiments. Figure 4.4 presents the results of the experiments with different prefetching factors for a cache with $f_s = 0.8$. Note that minimum prefetching with $F = 2$ improves the "No prefetching" case

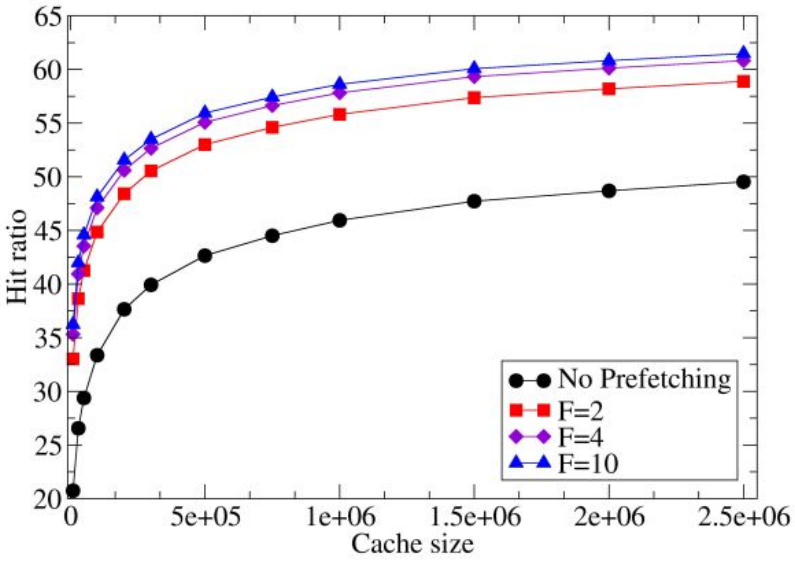


Figure 4.4: Caching performances due to various levels of prefetching (F as the prefetching factor) with $f_s = 0.8$ and the cache size as the number of cached result page entries. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

considerably. The hit ratios increase as the prefetching factor increases but the improvements are marginal after $F = 4$. It is important to note that in the case of prefetching 20 successive pages (not shown in the figure), the hit ratio starts to decrease; since after this point, the cache is filled with too many useless prefetched pages that results in eviction of pages that will likely be requested in the near future.

In the following experiments, we compare the cache performance in the case of using non-uniform result pages (2_8_10) to that of using the baseline (10_10) approach. In the light of the above discussions, we use $f_s = 0.8$ as it gives the best performance for almost all cache sizes that are experimented. For the prefetching strategy, we experiment with the prefetching factor $F = 4$ since after this point improvements in the hit ratio are marginal. The static part of the cache is filled with the queries in the training set as described in the previous subsection and the remaining queries are submitted to dynamic part of the cache. Then, the overall cache performance is evaluated by using the disjoint test set. In the literature, the hit ratio is widely used as a measure for comparing the different caching approaches. However, in our case, since the size of the cached items is not the same, using the hit ratio as an evaluation measure is not possible. For instance, assume that we have cached the first (including results at ranks one and two) and the second result pages (including results at ranks three to 10) of a navigational query in our caching scheme. Then let the conventional caching scheme cache the first result page (results of ranks one to 10) of the same query. Assume that clicks for this query in the test log are at the ranks $R = \{1, 2, 5, 12\}$. For our system, we would have hits for the first and second result pages but we would have a miss for the third result page, which results in a hit ratio of 2/3. On the other hand, the conventional system would have one hit for the first result page and one miss for the second result page, which gives a hit ratio of 1/2. As it can be seen, even though these two caching strategies cached the same amount of results for the same query, the hit ratio measure artificially differs, and thus is not appropriate to use here. Therefore, we use another approach to measure the effectiveness of the two strategies. Instead of hit ratio, we compute the absolute number of cache misses for each strategy. For instance, in the above example, there is only one cache miss for both caches, which is a fair evaluation.

4.4.2.2 Experimental Results

Our caching experiment results are shown in Table 4.1. The cache size is given as the number of 10-results-per-page cache size entries. The reduction percentages in the total miss counts are shown in a separate column. As the cache size increases, improvements decrease since this causes many of the second result pages of navigational queries to be cached also. Therefore, our method is more effective for small and medium size caches. For large cache sizes, our approach experiences higher miss counts than the baseline. This seems surprising at first, but has a sound explanation. For some of the queries that are identified as navigational in the training log and have all their clicks in the top two results,

Table 4.1: Caching performances with $f_s = 0.8$ and prefetching $F = 4$. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

Cache Size	Baseline Cache	Our cache(with 2_8_10)	
	Cache Miss Counts	Cache Miss Counts	% reduction
5K	4,072K	3,851K	5.42
10K	3,824K	3,604K	5.75
30K	3,437K	3,264K	5.04
50K	3,269K	3,124K	4.43
100K	3,059K	2,952K	3.49
200K	2,866K	2,806K	2.11
300K	2,765K	2,702K	2.27
500K	2,657K	2,613K	1.67
750K	2,546K	2,538K	0.30
1500K	2,448K	2,453K	-0.21
2500K	2,350K	2,358K	-0.33

our static caching scheme would never cache the second results page (including results three to 10), regardless of the cache size (i.e., recall that we compute the frequency of the <query, result page> pairs in the training set, thus if all clicks are for the top two results, the frequency of the second result page would be 0, and can never be cached in the static portion of the cache). They can be cached in the dynamic cache but the first occurrences of all such result pages cause cache miss in our case. To remedy this problem, we use the confidence score used during the identification of navigational queries for a smoothing operation. That is, for each query identified as a navigational query, we multiply its (first result page) frequency with $(1 - confidence)/c$, where c is an experimental constant (which is found as 2), and add this score to the frequency of the second result page. This creates a smoothing effect and allows us to cache the second result page of a navigational query with low confidence, even if results between ranks three and 10 are never clicked on the training log. In Table 4.2, we report the results with smoothing. Notice that our model does not create an increase in miss counts for any cache size, in turn of a slight decrease in the reductions for smaller cache sizes (Please compare Tables 4.1 and 4.2). The proposed result page model provides

Table 4.2: Caching performances with $f_s = 0.8$ and prefetching $F = 4$ with smoothing. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

Cache Size	Baseline Cache	Our cache(with 2_8_10)	
	Cache Miss Counts	Cache Miss Counts	% reduction
5K	4,072K	3,868K	5.00
10K	3,824K	3,622K	5.28
30K	3,437K	3,283K	4.47
50K	3,269K	3,145K	3.80
100K	3,059K	2,974K	2.78
200K	2,866K	2,805K	2.14
300K	2,765K	2,702K	2.28
500K	2,657K	2,604K	2.01
750K	2,546K	2,524K	0.87
1500K	2,448K	2,434K	0.59
2500K	2,350K	2,339K	0.46

reductions of 4-5% in absolute miss counts.

Finally, we realized that reporting our gains in terms of miss counts disfavors us in that the majority of misses in our system is for singleton queries (queries that occur only once in the test log and do not appear at all in the training log), which can never be resolved in a caching mechanism. For comparison purposes, we also report the experimental results for the test set queries excluding the singleton query miss counts in Table 4.3 (again with smoothing). For this case, reductions are more emphasized, reaching up to 9.17%.

4.5 User browsing behavior with the non-uniform result page model

Search engine users' browsing behavior is well studied in the literature [52, 25, 28] for purposes such as providing better search interfaces for users and better

Table 4.3: Caching performances by excluding costs for singleton query misses (f_s = 0.8 and prefetching F = 4). (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

Cache Size	Baseline Cache	Our cache(with 2_8_10)	
	Cache Miss Counts	Cache Miss Counts	% reduction
5K	2,445K	2,241K	8.35
10K	2,196K	1,995K	9.17
30K	1,810K	1,656K	8.53
50K	1,642K	1,517K	7.60
100K	1,431K	1,347K	5.90
200K	1,239K	1,177K	4.98
300K	1,137K	1,074K	5.50
500K	1,029K	976K	5.15
750K	919K	896K	2.48
1500K	820K	806K	1.70
2500K	722K	712K	1.44

exploitation of query logs as implicit relevance judgments. All these studies are based on the uniform result presentation model that shows 10 results per page. In this study, we propose a non-uniform result page model for search engines through the use of navigational query identification, thus it is important to investigate the effect of such a result presentation model on user browsing behavior. To this end, we conduct a user study. In this section, we first describe the experimental setup and then present our findings.

4.5.1 User Study Setup

Ten search tasks, presented in Table 4.4, are prepared for the user study[4]. Five of these tasks are navigational and the remaining five have informational intent. Navigational tasks are chosen so that the target result could be found among the top results in one of the major search engines.

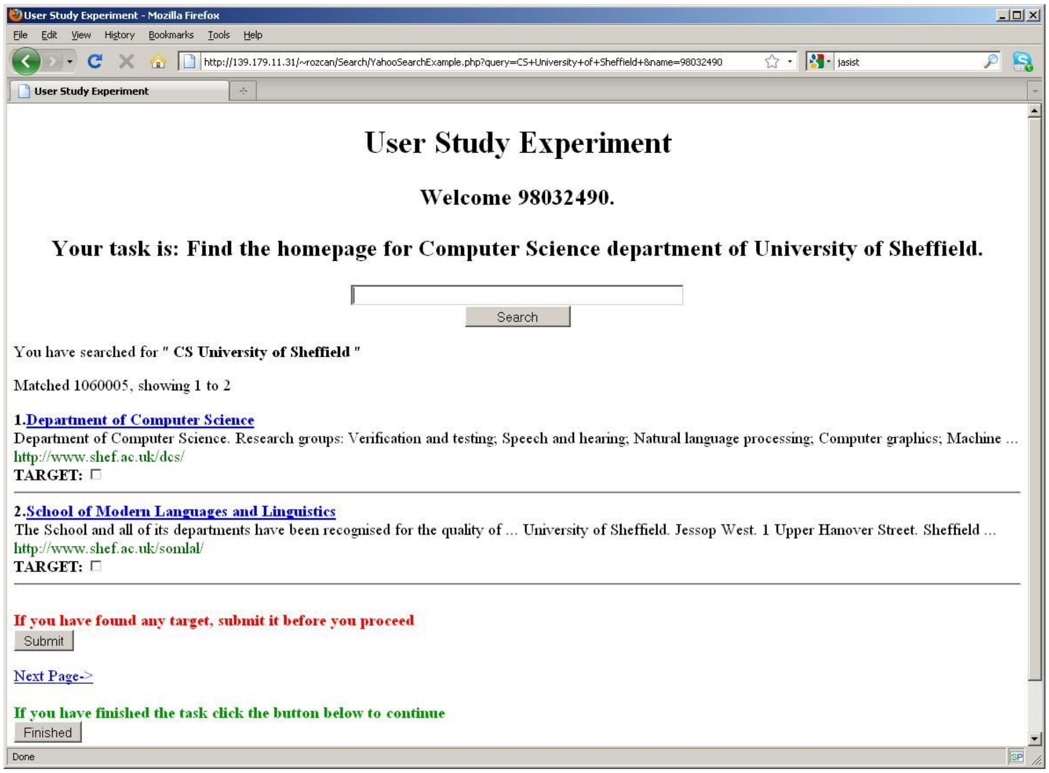
[4] Available at http://139.179.11.31/rifat/Search/

Table 4.4: Navigational and informational tasks used in the user study. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

Navigational tasks
Find the official homepage for Beijing 2008 Summer Olympics.
Find the homepage for United Airlines company.
Find the homepage for the New York Times newspaper.
Find the homepage for Computer Science department of University of Sheffield.
Find the homepage for United States Consulate in Istanbul Turkey.
Informational tasks
Find where and when the first olympics games organized.
When NASA sent a spacecraft for the first time for exploration of Mars?
Find information on the effects of global warming on polar bears.
Find information on how to quit smoking.
Find information on touristic places in Turkey.

The user study is designed as follows: When a subject logs in to the system, at the top of the usual search interface (i.e., a textbox in which to type the query) the system randomly displays a task amongst from the search tasks shown in Table 4.4 and that has never been completed by that user (see Figure 4.5). The user is asked to accomplish the given task by submitting queries and viewing the results, including URLs and snippets, as usual. We used Yahoo! search engine's "Web search" web service [89] in order to get the results of the user queries. For the navigational tasks[5] only the top two results are shown based on the non-uniform result page model (2_8_10) discussed in the previous sections. If the user clicks the "next page" link, then the top three to 10 results are shown in the second result page. On the other hand, for informational queries, the baseline result presentation is used and 10 results are shown on each page. In the tutorial stage of the user study, we requested that subjects mark the search result(s) that answer(s) the given task by clicking the "TARGET" checkboxes provided at the end of each result snippet. The subjects are allowed to click on the result URL to further see the content of the result document. Note that in this study, subjects

[5] Note that the user is not informed about navigational or informational query types and they do not know what we are measuring.


User Study Experiment - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://139.179.11.31/~rozcan/Search/YahooSearchExample.php?query=CS+University+of+Sheffield+&name=98032490
jasist
User Study Experiment
Search
Submit
Finished
Done

User Study Experiment

Welcome 98032490.

Your task is: Find the homepage for Computer Science department of University of Sheffield.

You have searched for " **CS University of Sheffield** "

Matched 1060005, showing 1 to 2

1.Department of Computer Science
Department of Computer Science. Research groups: Verification and testing; Speech and hearing; Natural language processing; Computer graphics; Machine ...
http://www.shef.ac.uk/dcs/
TARGET: □

2.School of Modern Languages and Linguistics
The School and all of its departments have been recognised for the quality of ... University of Sheffield. Jessop West. 1 Upper Hanover Street. Sheffield ...
http://www.shef.ac.uk/somlal/
TARGET: □

If you have found any target, submit it before you proceed

Next Page->

If you have finished the task click the button below to continue

Figure 4.5: The search interface of the user study. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

do not perform an evaluation of the information retrieval system; if that were the case they would be requested to scan top-100 or top-1000 results and decide whether each result is relevant or irrelevant. In this case, subjects are instructed to stop when they think they have accomplished the given task, i.e., found the web pages that answer the question in the corresponding search task.

Fifty-four subjects consisting of graduate and undergraduate students from the computer engineering and industrial engineering departments in our university perform the user study. There are 14 female subjects and remaining 40 subjects are male students. Each subject is given a brief introduction to the experiment. Our aim is to measure the browsing behavior of subjects to the non-uniform result page model for navigational queries. Specifically, we will measure the users' tendency to request the next result page even if they can identify the target answer among top-2 results in the first page. In this sense, we investigate whether the users can easily accommodate to a non-uniform result page model, or they insist on seeing all top-10 results even when they find the answer in top-2.

4.5.2 User Study Results

The result of the experiment reveals that out of 237 navigational tasks completed by the subjects (ignoring the cases where the subjects did not enter any target result for the query), in only 24 cases subjects wanted to look for the results beyond the top two. In other words, subjects were satisfied with the top two results 89.9% of the time. Table 4.5 presents additional statistics obtained in this user study. Each statistic is given as an average for navigational and informational queries. As expected, the average informational task duration is much longer than the average navigational task duration. Subjects type more queries for informational tasks. The number of target sites selected for informational queries is much higher than the number of targets selected for navigational queries. We observe that the actual URLs and/or snippets are much more helpful for navigational queries than informational queries since users click on fewer numbers of URLs in the former case. Note that this number should be close to 1 in real life case since in that case, users are expected to click on the target website because the sole purpose of a navigational query is to reach that website. However, in this experimental setting, subjects are instructed to just select the correct sites; most of the time they understand the target website from snippets and do not need to actually visit that website. The last statistic evaluates the ranks of the target sites selected for each type of task. We observe that 96% of the target sites selected for navigational queries reside in the top one or two ranks. However, this ratio decreases to 34% for informational tasks.

To justify our approach, it is important to analyze the cases when subjects want to see beyond the top two even though they found the target site(s) in the top two. We will refer to such cases as “Beyond Top 2” hereafter. On average, only in 9.3% of total navigational query instances in which users selected target sites in the top two, they also want to see beyond the top two. When we focus on such

Table 4.5: User study experiment statistics. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

Statistic	Navigational	Informational
Avg. task completion time (sec)	38.33	83.94
Avg. number of queries submitted for a task	1.09	1.27
Avg. number of results selected as TARGET for a task	1.25	4.09
Avg. number of URLs clicked for a task	0.33	1.06
Avg. % of TARGET results selected at ranks Top1-2 for a task	96%	34%

Beyond Top 2 cases, we see a pattern, shown in Figure 4.6, between the occurrence of such cases and the viewing order of the navigational task. (Recall that search tasks are assigned to subjects in random order.) It is observed that many of the Beyond Top 2 cases occur when a subject meets our result presentation model of 2_8_10 for navigational queries for the first time (i.e., 10 of 22 such cases occur when a user evaluates his/her first navigational task, as shown in Figure 4.6). As users see more examples of the non-uniform result presentation model, they get used to it; the number of Beyond Top 2 cases drops dramatically after performing a few navigational tasks. Therefore, even though, the percentage of Beyond Top 2 cases is 9.3%, users stop asking for the next result pages after they experience just a few navigational queries with our method.

The overall result of the user study can be summarized as follows: Users easily adapt to non-uniform result page models and interact with the web search engine as expected, i.e., they do not look for further result pages if they are satisfied with the current result page. This conclusion justifies our proposal of a non-uniform result model and implies that the efficiency gains obtained in the experimental setup of the previous sections could be transferred to real life search systems.

Furthermore, the work of Teevan et al. [79] examines the users' re-finding behavior and shows that navigational queries constitute a significant portion of repeat queries. This means that a user tends to use the same navigational query

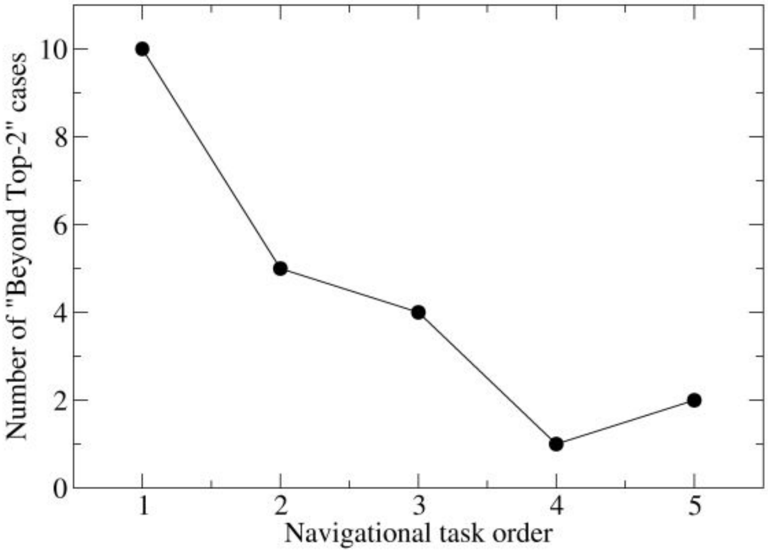


Figure 4.6: The pattern between Beyond Top 2 cases and navigational task order. (Ozcan, R., Altingovde, I.S., Ulusoy, O., "Exploiting Navigational Queries for Result Presentation and Caching in Web Search Engines," Journal of the American Society for Information Science and Technology, Vol. 62:4, 714-726. ©2011 John Wiley and Sons. http://dx.doi.org/10.1002/asi.21496. Reprinted by permission.)

to locate a particular page in his/her mind, and would be familiar with the result page. Thus, in such cases, users are even less likely to browse result pages beyond the first page. However, in the user study, the users are most probably not very familiar with the navigational tasks assigned to them. That is, our user study serves as a worst-case scenario for the number of Beyond Top 2 cases, since, in contrast to real-life, almost no task is a repeat query.

Evidence in addition to user click behavior supports our choice of the 2_8_10 result page model for navigational queries. The eye tracking study in [42] revealed that users almost always check the first two results immediately after a result page is displayed. Then, a few seconds later, other results are examined if the first two are not satisfying. Therefore, our non-uniform result model is confirmed by this type of user browsing behavior.

4.6 Conclusion

In this chapter, we propose and evaluate non-uniform result page models to display and cache the results of navigational queries. It is shown that for these types of queries, it is possible to reduce the number of snippets sent by 71%, at the cost of an 11% increase in the number of result pages browsed. In other words, assuming that an HTML result page (of top-10 results) takes 4 KB space [30] and a TCP packet stores at most 1460 bytes of data, our approach would require only one TCP packet for top-2 results, instead of three packets needed for transmitting top-10 results. This reduction in the number of packets sent will be more important in mobile search scenarios because of low bandwidth capacity and high packet loss probability. The proposed result presentation model for navigational queries is also shown to be valuable for caching mechanism, especially for small and medium cache sizes. It is possible to reduce the number of miss counts up to 9.17%. Our findings are encouraging and justify the need for the special treatment of different query types, especially navigational queries, submitted to a web search engine. Our user study with 54 subjects shows that users easily adapt to non-uniform result page models and interact with the search engine as expected, i.e., they do not look for further result pages if they are satisfied with the current result page. Therefore the efficiency gains observed in the experiments can be carried over to real-life situations.

In addition to the aforementioned gains in efficiency, the approaches presented in this chapter enable effective use of web page space [53], which implies further benefits in various scenarios. For instance, given that only two results are shown on the first (and most important) page, now it is possible to reserve the remaining page space for purposes such as advertisements, result visualization, or related query suggestions. This may have a positive impact on the number of clicks received by ads, which is an important source of revenue for commercial search engines. As another use case, search engines may want to provide more information in the result snippets especially for news site navigational queries (e.g., “new york times”) by giving direct links to several important headlines.

Chapter 5

Space Efficient Caching of Query Results

In this chapter[1], we propose an efficient storage model to cache document identifiers of query results. Essentially, we first cluster queries that have common result documents. Next, for each cluster, we attempt to store those common document identifiers in a more compact manner. Experimental results reveal that the proposed storage model achieves space reduction of up to 4%. The proposed model is envisioned to improve the cache hit rate and system throughput as it allows storing more query results within a particular cache space, in return to a negligible increase in the cost of preparing the final query result page.

The rest of this chapter is organized as follows. In the next section, we provide the motivation for our work. Section 5.2 presents the related work on clustering queries. We describe the details of our cluster-based storage approach in Section 5.3. Experimental evaluation and discussion of the results are provided in Section 5.4. Finally, we conclude this chapter in Section 5.5.

[1] ©2008 IEEE. Reprinted, with permission from Ozcan, R., Altingovde, I.S., Ulusoy, O., "Space Efficient Caching of Query Results in Search Engines," International Symposium on Computer and Information Sciences (ISCIS'08), Istanbul, Turkey, 2008. http://dx.doi.org/10.1109/ISCIS.2008.4717960

5.1 Introduction

Search engines typically cache either the query results or posting lists for query terms, or both [6]. For each case, static or dynamic caching (and even a hybrid of both) can be applied. In this chapter, we focus on the static caching of the query results. A static cache can store the query results in two ways. In a, so-called, docID cache [30] only the document identifiers of the query results are stored. The snippets and the final result page are generated each time a query request yields a cache hit. An alternative to this approach is a snippet cache, which stores the final HTML result pages (including snippets, etc.) to be displayed upon a request that yields a cache hit. Clearly, in the former approach, the cache can store more items; but a cache hit still needs some processing for preparing the result page, whereas the latter approach stores less items in the same space, but the results can be sent immediately to the user once a query is found in the cache.

We propose a storage mechanism for static docID caching. Our approach exploits the overlaps in the results of similar queries, which are identified by clustering queries. Intuitively, we presume that there may exist several query clusters that share a set of documents in their member queries' results, and we try to encode these common document identifiers in a compact form, to better utilize the static cache space and increase the hit rate. In the literature, query clustering is essentially exploited for better answer ranking and query recommendation purposes. To the best of our knowledge, our approach is the first attempt to use query clustering for efficient storage of query results.

5.2 Related Work on Query Clustering

As far as we know, none of the works in the caching literature discuss how the results (docIDs or snippets pages) are actually stored in a static or dynamic cache. Such practical details of the commercial search engines are not publicly available. Thus, we basically assume that the docID cache includes a simple list of top-K integers (as provided by the query processor) per query and the snippet cache includes HTML pages that can be directly displayed to the user for a given query. Our goal in this chapter is to devise a more compact storage scheme for storing docID's in a static cache.

Clustering of queries is addressed in several earlier studies [8, 12, 34, 58, 91]. Query clustering in the context of search engines is previously proposed for two main goals: a) query recommendation, and b) enhancing result ranking. Beeferman and Berger use clicktrough data, which consists of <query, url> pairs, in order to find related queries and related URLs by clustering [12]. The proposed approach is to see the query log as a bipartite graph between the sets of queries and sets of URLs. Then, two particular vertices (one from the queries and the other from the URLs) are connected by an edge if such a pair occurs in the query log. Finally, clustering is performed on this bipartite graph. The proposed clustering is a kind of hierarchical agglomerative clustering [34]. The similarity of two vertices in this graph is calculated by the overlap on neighbors. If two queries have more common URLs then their similarity is high. After finding query clusters, they can be used to assist users by suggesting alternative (and potentially related) queries during web search. That is, for a given user query, the system determines its cluster and suggest other queries from the same cluster. The proposed approach is evaluated by the number of user clicks on this suggested alternative query links.

In [8], queries are clustered by using the content of documents that are clicked by the users. These documents are represented by the well-known vector space model and clustered by using the k-means algorithm. After query clustering, this structure is used for 2 applications: a) Answer ranking: The popularity of the results in a query's cluster are employed to re-rank the original results of that particular query. b) Query recommendation: When a query is submitted, its query-cluster is found and queries are recommended based on their similarity to the query and their support based on the query log.

Wen et al. also perform query clustering using the logs [86]. Similar to the above approaches, they also claim that using only query keywords to do query clustering is not successful since queries are very short and words have polysemic meanings (e.g. “java”). They propose to use the common documents which are clicked for the queries to measure the similarity between those queries. This cross-reference information between the queries and clicked documents is shown to be effective for query clustering and better than using the either one of the query keywords or logs alone.

In our study, we use the result lists for forming query clusters and exploit these clusters for utilizing the storage of these results. Since we focus on storing the entire result list per query, we do not use solely the clicked documents, differing from the other works mentioned above. To the best of our knowledge, query clusters are not used previously for exploiting the cache storage space in this manner.

5.3 Cluster-based Storage of Query Results

Our approach consists of two steps: a) Query clustering, and b) Storage of query results. Each of these steps is explained in detail in the following subsections.

5.3.1 Query Clustering

We applied the single link (linkage) hierarchical clustering algorithm [34] for query clustering. The aim of the clustering in our context is to find overlaps between result lists of similar queries. Therefore, each query is represented by its n result document numbers. Clustering algorithm works as follows:

At the beginning of the clustering phase, one cluster is formed for each query in the dataset. Assume two queries Qi and Qj have the following result sets, each containing top-n result document identifiers.

$$R_i = \{r_{i1}, r_{i2}, \ldots, r_{in}\} \; R_j = \{r_{j1}, r_{j2}, \ldots, r_{jn}\}$$

Then the similarity between these two queries (actually clusters in the context of the algorithm) is computed by the following formula which considers the fraction of intersection:

$$sim\_measure_{ij} = \frac{|R_i \cap R_j|}{min(|R_i|, |R_j|)} \tag{5.1}$$

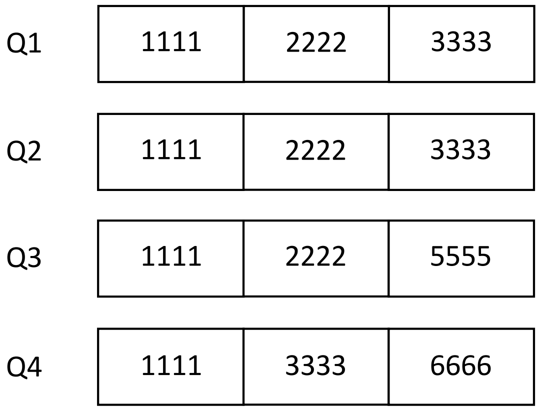
At each step of the single link clustering, most similar cluster pair is chosen whose similarity value is greater than a predefined minimum similarity threshold (T_{sim}). If such a cluster pair is found, they are merged and the union of result lists in each cluster in this pair constitutes the result list for the new cluster. This process continues until no cluster pair satisfying the minimum similarity threshold can be found or all clusters are merged to one cluster, which is practically not possible for a real query log.

5.3.2 Storage of Query Results

In this section, we present the details of our storage mechanism exploiting the query clusters obtained in the previous stage. Note that our focus is to improve the actual storage of result lists of queries in the static cache and we do not address any lookup mechanisms in this study.

We present our storage mechanism by the following simple example. Assume that we have obtained the query cluster containing queries Q1, Q2, Q3 and Q4 as given in Figure 5.1. For the sake of simplicity, let each query store only top-3 results. We also give hypothetical document ids for each result document. The shared documents in this cluster are 1111, 2222, 3333.

In Figure 5.1, we illustrate the conventional storage scheme for the results of these four queries. In this case, it is assumed that the query results are simply kept as a list of document identifiers, each requiring 4 bytes of storage. Then, the total storage space required for query results is 48 bytes (excluding the space required for lookup mechanism and the queries themselves). Note that, as it is



Q1	1111	2222	3333
Q2	1111	2222	3333
Q3	1111	2222	5555
Q4	1111	3333	6666

Figure 5.1: Conventional cache storage mechanism for queries Q1, Q2, Q3 and Q4. (©2008 IEEE. Reprinted, with permission from Ozcan, R., Altingovde, I.S., Ulusoy, O., "Space Efficient Caching of Query Results in Search Engines," International Symposium on Computer and Information Sciences (ISCIS'08), Istanbul, Turkey, 2008. http://dx.doi.org/10.1109/ISCIS.2008.4717960)

mentioned before, there is no earlier work on storage mechanisms for query result lists in the context of caching. Although commercial web search engines definitely employ static and dynamic caching mechanisms, details are not exposed. So, we use the simple storage scheme described above as the baseline in this study.

In the above scenario, it is seen that the shared document ids are stored several times in the cache. Our storage approach exploits this overlap of document ids in the query clusters. Figure 5.2 shows the general structure of our approach. A shared-documents array is constructed for the overlapping result document ids in a cluster. For each cluster, its shared-documents array includes the top-256 documents that have the highest frequency among the results of the queries in that cluster. The result list of a query stores the array index for the shared documents, which can be expressed in only 1-byte (as there are at most 256 entries in the array). For instance, in Figure 5.2, the result list of Q3 starts with the 1-byte identifier 0, which will be resolved to the first element of the array, i.e., document 1111. The result documents that are unique to each query (e.g., 5555 for Q3 and 6666 for Q4) will be stored as is, i.e., in 4-bytes. Clearly, as the degree of overlap in the results of the queries in a cluster increase, our storage scheme

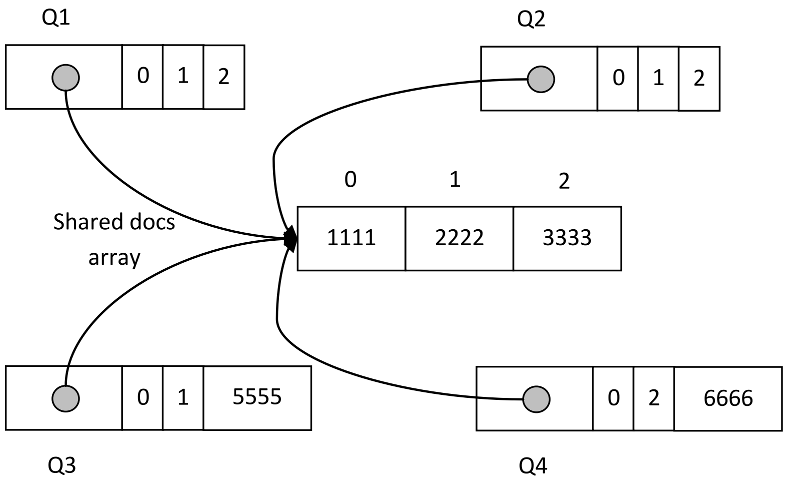


Figure 5.2: Our storage mechanism exploiting query clustering. (©2008 IEEE. Reprinted, with permission from Ozcan, R., Altingovde, I.S., Ulusoy, O., “Space Efficient Caching of Query Results in Search Engines,” International Symposium on Computer and Information Sciences (ISCIS’08), Istanbul, Turkey, 2008. http://dx.doi.org/10.1109/ISCIS.2008.4717960)

will yield more gains. That is, shared documents will be stored with 4-bytes only once, and will be pointed by 1-byte entries in each result list.

The proposed storage scheme employs a shared-documents array per cluster, which implies that each query should know the location of this array. In Figure 5.2, an extra 4-byte entry is added to the beginning of each query’s result list to store the address of this array. Furthermore, we also need a mechanism to encode whether an entry in the result list should be interpreted as a 1-byte pointer or a 4-byte document identifier, as they are in a mixed order in our scheme. In the literature, it is reported that web users very rarely see more than top-30 results ([74, 37]). Thus, we assume that for each query a result list of at most 30 entries are stored. In this case, another 4-byte entry is added to the beginning of the result list, to encode whether the succeeding entries should be interpreted as 1-byte or 4-byte values. For instance, for Q3, the corresponding bit sequence would start with 110, which means that the first two entries in the result list are pointers to the shared array, and the third entry is actually a document identifier.

As a result, our scheme incurs a cost of 8-bytes (4-bytes for the address of shared-documents array and 4-bytes for the entry interpretation mask) per query result list. For the simple scenario outlined above, the proposed storage scheme can not compensate these costs; but in real life our approach would compensate the costs and yield space gains even when a query cluster have a few documents in common among the top-30 results. For instance, in a cluster of three queries, an intersection of 5 results would be enough for compensating the additional costs. Finally, since static caching is an offline process, we can decide whether to apply our storage scheme or not, considering the cost/gain trade-off for each cluster. In Section 5.4, we provide experimental evidence supporting our claims.

5.4 Experiments

Dataset: We use a subset of the AOL Query Log [66] which contains around 20 million queries of about 650K people for a period of 3-months. Our subset contains 1,127,894 query submissions and 661,791 of them are distinct queries.

We used Yahoo! search engine's "Web search" web service [89] to get Top-100 results including titles, urls and snippets, for all distinct queries. This resulted in a 13.8 GB dataset. In our experiments, top-30 query results are cached in static cache since most users only check a few result pages [74, 37]. In [74], it is reported that in 95.7% of queries, users requested up to only three result pages.

Following the practice in the literature, static cache is populated with the most popular query results. We select most frequent K queries from our query log. Next, single link clustering algorithm is executed on this K query set. After obtaining query clusters, we distinguish clusters as "useful" and "useless" according to the space consumed by using our approach. If storing a cluster in our scheme requires more space than its baseline storage space (i.e., when there is not enough overlap in the result documents among the queries of the cluster), then it is identified as "useless" cluster and we store the queries in that cluster as in the case of baseline. Additionally, since clustering process is terminated based on a minimum similarity threshold value(T_{sim}) at some point, there may also exist single-query clusters left apart from the "useless" clusters. These single-query clusters could not be merged with any other cluster during the query clustering. Baseline storage model is also applied for those types of queries. For useful clusters that yield space gains, we apply the storage scheme proposed in this study.

Table 5.1 provides the overall reduction rates in cache sizes where the cache is filled with the most-frequent K queries. The column "baseline" denotes the case where each document id in the result lists is stored by using 4-bytes. The columns "cluster-based" denote the cases in which our storage scheme is applied as described above. We experiment with two different values of minimum similarity threshold (T_{sim}) that is used to terminate the clustering process. Finally, we also conducted an additional experiment where we kept a shared-document array for the entire set of queries in the cache. That is, instead of clustering queries we determine the top-256 most frequent result documents for all queries in the cache and store in a global shared-documents array. Again, the shared documents in the result lists are encoded with a 1-byte pointer. For this case, there is no need to store the address of array per query, since there is only one global array. The column "baseline-2" denotes this case.

We draw the following observations from Table 5.1. First of all, encoding shared documents in a compact manner is a beneficial approach even in the global case. When a global array of 256 documents is used, we observe a slight reduction in the space wasted. However, the gains are more emphasized when queries are clustered. For the case with clustering similarity threshold is set to 0.1, we obtain the space reductions up to 4%. For all values of K, cluster-based storage scheme outperforms the baseline storage and the baseline with a global shared-documents array.

Figure 5.3 shows the size distribution of query clusters (with similarity threshold 0.1) for K=40,000 queries case, for which our storage scheme achieves the highest reduction (i.e., 4.01%). As it can be seen from the graph, clusters involving two queries dominate. For a more detailed analysis, we also report the number

Table 5.1: Storage performances. (©2008 IEEE. Reprinted, with permission from Ozcan, R., Altingovde, I.S., Ulusoy, O., "Space Efficient Caching of Query Results in Search Engines," International Symposium on Computer and Information Sciences (ISCIS'08), Istanbul, Turkey, 2008. http://dx.doi.org/10.1109/ISCIS.2008.4717960)

Cache Size	Baseline	Baseline-2	% Red.	Cluster-Based (T_{sim}=0.2)	% Red.	Cluster-Based (T_{sim}=0.1)	% Red.
1K	112,212	110,868	1.20	108,954	2.90	108,616	3.20
3K	335,500	332,953	0.76	324,487	3.28	323,280	3.64
5K	558,216	555,029	0.57	539,967	3.27	537,980	3.63
10K	1,109,212	1,104,220	0.45	1,069,220	3.61	1,064,961	3.99
15K	1,664,260	1,657,619	0.40	1,603,909	3.63	1,597,907	3.99
20K	2,206,984	2,199,540	0.34	2,127,752	3.59	2,119,242	3.98
30K	3,282,072	3,273,717	0.25	3,164,765	3.57	3,152,692	3.94
40K	4,296,000	4,286,139	0.23	4,138,939	3.66	4,123,945	4.01
50K	5,325,428	5,315,378	0.19	5,133,331	3.61	5,116,419	3.92

of "useful", "useless" and single-query clusters in this case. Out of 40,000 queries, 2,840 "useful" clusters and 2,340 "useless" clusters are formed. These clusters contain 13,652 and 5,548 queries, respectively. 20,800 queries are left as single-query clusters. Average cluster size of "useful" clusters is 4.81 queries, whereas "useless" clusters have 2.35 queries per cluster on average. This is expected since more queries should be overlapping in "useful" clusters.

The queries in the useless and single-query clusters (summing up to 26,348 queries) are stored in the conventional manner whereas the remaining queries (13,652 of them) are stored by using our scheme. Thus, 34.13% of all queries are stored using our mechanism. The storage space used only for these queries drops from 1,513,212 bytes to 1,359,157 bytes; resulting an 11% reduction in the consumed space. This implies that better clustering of queries may also yield higher overall reductions.

Note that, our approach may also cause a slight increase in the preparation of final query result page in case of a cache hit, due to relatively more complicated handling of the query result lists. In turn, the gains in the storage space would improve the cache hit rate and throughput with respect to the baseline scheme,

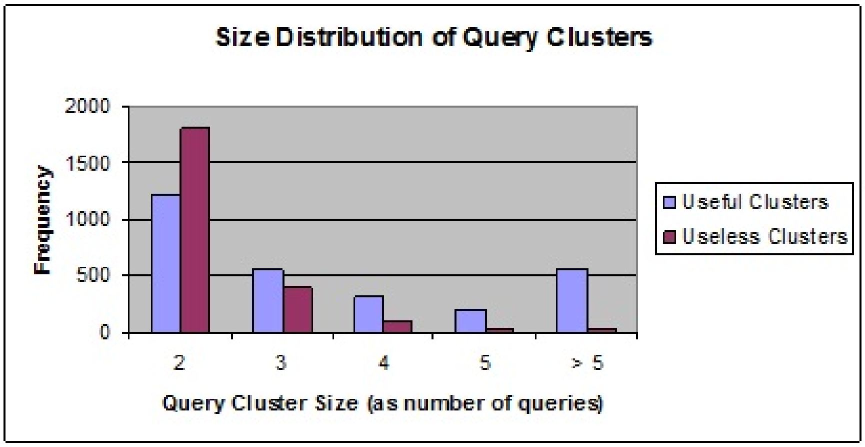


Figure 5.3: Size distribution of query clusters for most frequent 40,000 queries (clustering similarity threshold is 0.1). (©2008 IEEE. Reprinted, with permission from Ozcan, R., Altingovde, I.S., Ulusoy, O., "Space Efficient Caching of Query Results in Search Engines," International Symposium on Computer and Information Sciences (ISCIS'08), Istanbul, Turkey, 2008. http://dx.doi.org/10.1109/ISCIS.2008.4717960)

as more queries can be filled to the same cache space with our approach. As a result, we envision that the former cost of processing would be negligible and compensated by the latter gains in hit rate and throughput.

5.5 Conclusion

In this chapter, we present a storage mechanism for caching of query results by exploiting the query clustering. In particular, we store the documents identifiers that are shared by the queries in a cluster in a more compact manner and improve storage utilization. We use single link clustering algorithm to form the queries that share common result documents. Then these clusters are stored by a compact storage mechanism and we obtain the space reductions up to 4% compared to the baseline approach.

Chapter 6

A Five-Level Static Cache Architecture

In this chapter, we describe a five-level static cache architecture for web search engines, i.e., a cache that stores items of five different types (query results, precomputed scores, posting lists, intersections of posting lists, and documents' content). Moreover, we propose a greedy caching heuristic that prioritizes items for caching, based on gains computed by using items' past access frequencies, estimated computational costs, and storage overheads. This heuristic takes into account the inter-dependency between the items when making its caching decisions, i.e., caching a particular item may lead to updates on gains of items that are not yet cached. Our simulations under realistic assumptions reveal that the proposed heuristic performs better than dividing the entire cache space among particular item types at fixed proportions.

The rest of this chapter is organized as follows. In Section 6.1, we provide the motivation for our work. Section 6.2 describes query processing flow in a typical web search engine, and presents the major costs associated with each step. In Section 6.3, we present our five-level architecture and our caching strategy. We describe the dataset and our experimental setup in Section 6.4 and evaluate the proposed strategy in Section 6.5. We conclude this chapter in Section 6.6.

6.1 Introduction

Search engines may cache three different types of data items such as query results, posting lists, and document content, as mentioned in Chapter 2. Query results can be stored in the form of top-k document ids (with their scores) or as HTML result pages. These types of caches will be referred as score and result caches, respectively. Posting lists of individual terms can be cached in the list cache and posting lists of pair of terms can be cached in the intersection cache. Considering all these variations, in this chapter, we propose a five-level static cache architecture that includes result, score, list, intersection, and document caches. The literature involves a number of works [30, 47, 54, 80] on the performance of these caches separately and several other works [6, 51, 73] that forms multi-level caches using two or three different items. We are not aware of any works that consider all possible cache components in a static caching framework. To the best of our knowledge, the closest work to ours is the work of [33], which considers the interaction between different types of caches in a dynamic caching setting.

We propose a five-level static cache architecture that contains all known cache types proposed in literature. We also describe a greedy heuristic that iterates over all possible items that may be cached in one of the levels. At each iteration, the heuristic selects the item with the highest possible gain for caching such that how much processing would be needed if we did not cache this item. We calculate the caching gain of an item by considering the past access frequencies of items as well as their estimated processing costs and storage costs. In this heuristic, after an item is selected for caching, the potential gains of all related items are updated, based on the inter-dependencies between the items.

We evaluate the proposed static cache architecture in a very detailed and realistic simulation setup, where we model almost every major cost incurred in query processing. Our findings using a real-life query log and a document collection reveal that the proposed mixed-order caching heuristic performs better than assigning fixed fractions of the entire cache space to different cache types, which is the common practice in literature.

6.2 Query processing overview

Query processing involves a number of steps: issuing the query to search nodes, computing a partial result ranking in all nodes, merging partial rankings to obtain a global top-k result set, computing snippets for the top-k documents, and generating the final result page. In this study, we neglect the overheads due to network communication between the nodes and the overhead of the result merging step. These overheads are relatively insignificant. For instance, the cost of network transfer is estimated to be less than 1ms in [6], a tiny fraction of the query processing cost, if the nodes are connected through a local area network. The cost of result merging, especially for low k values (e.g., 10), would not be more than a few milliseconds either.

In our work, we take into account the following steps, which incur relatively high processing or I/O overhead:

- Step 1: For all query terms, fetch the associated posting lists from the disk. This incurs I/O cost, denoted as C_{pl}.
- Step 2: Compute relevance scores for the documents in the intersection of fetched posting lists and select the top-k documents with the highest scores. This incurs CPU overhead, denoted as C_{rank}.
- Step 3: For the top-k documents identified, fetch the document data from the disk. This incurs I/O overhead, denoted as C_{doc}.
- Step 4: For the fetched documents, compute snippets. This incurs CPU overhead, denoted as C_{snip}.

We assume that, in a search cluster, each node acts as both a master and a client. That is, there are no central brokers that collect partial results from the nodes and merge them to generate the final query result. Instead, each node plays the role of a broker for a subset of queries. The node that would serve as a broker for a particular query can be determined by using the MD5[1] hash

[1] http://en.wikipedia.org/wiki/MD5

of the query. Notably, each node in the cluster executes all five steps of query processing and needs to access and/or generate corresponding five types of data items, namely lists, intersections, scores, documents and results, in this particular order. This, in turn, means that a cache architecture in a fully-distributed search cluster should consider all types of caches at each node. Such an architecture is the focus of the rest of this chapter.

6.3 Five-level static caching

6.3.1 Architecture

We describe a five-level cache architecture for static caching in search engines. In this architecture, the space reserved to each type of item cache is not individually constrained. Instead, there is a global capacity constraint that applies to all caches, i.e., all caches share the same storage space. Therefore, individual caches can continue to grow as long as their total size is below the global capacity constraint.

Each cache stores a different type of (key, value) pair and provides saving for one or more of the query processing costs mentioned in Section 6.2. In Table 6.1, we list the five different cache types considered in this work, along with their (key, value) pairs and associated savings in query processing costs.

6.3.2 Cost-based mixed-order caching algorithm

As the caching algorithm for the above-mentioned architecture, we propose a simple greedy heuristic. In this heuristic, six different priority queues are maintained, one for each of the five caches and one for selecting the most cost-effective item currently available at each greedy choice step. The algorithm consists of an initial gain computation step for filling the five priority queues, and a selection and gain update step for determining the items to be cached. The steps of the

Table 6.1: Cache types and their cost savings. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

Cache type	Key	Value	Cost saving
Result	Query	Snippets of top k documents (output of Step 4)	$C_{\text{pl}} + C_{\text{rank}} + C_{\text{doc}} + C_{\text{snip}}$
Score	Query	Relevance scores for top-k documents (output of Step 2)	$C_{\text{pl}} + C_{\text{rank}}$
Intersection	Set of term ids	Intersection of posting lists (intermediate output of Step 2)	$C_{\text{pl}} + C_{\text{rank}}$
List	Term id	Posting lists (data fetched in Step 1)	C_{pl}
Document	Document id	Raw document content (data fetched in Step 3)	C_{doc}

algorithm can be summarized as follows:

Initial gain computation: For each item that is candidate to be cached (e.g., query result, document, term), we compute the potential gain that would be obtained by caching that item, using the statistics in a previous query log. In an earlier work [6], the gain computation is usually based on the ratio between the access frequency of the item and its size, i.e., the nominator is the observed frequency of requests for the item and denominator is the space (in bytes) that the item would occupy in the cache. In our cost-based framework [64], the gain computation is further augmented with a cost component that represents the saving achieved in query processing time (in ms) by caching the item (see Eq. (6.1)). In our case, for each item type, the corresponding cost saving is computed as shown in the fourth column of Table 6.1.

$$\text{Gain} = \frac{\text{Cost saving} \times \text{Frequency}}{\text{Size}}. \tag{6.1}$$

Here, an important assumption is that the past access frequencies of items accurately represent their future access frequencies. Although this assumption generally holds, a way of smoothing may still be needed. For instance, previous studies show that the frequency of query terms exhibit little variation in a reasonably long time period, e.g., a few months [6]. Thus, for list caching, the term access frequencies observed in earlier query logs would be a good evidence to rely on. On the other hand, for a given query stream, almost half of the queries are singleton [6] (i.e., they appear only once in the entire query log), which implies that the past query frequencies may not exactly capture the future frequencies. This is especially true for less frequent queries. In a recent study, it is mentioned that “past queries that have occurred with some high frequency still tend to appear with a high frequency, whereas queries with relatively less frequency may appear even more sparsely, or totally fade away in the future” [64]. A similar discussion about the estimation of future query frequencies for result caching is also provided in [32]. In the latter work, it is reported that, for queries with a past frequency greater than 20, the future frequencies almost precisely follow the past observations. Thus, it is practical to construct an estimation table by using a separate query log for queries with a frequency value less than 20. In this study, we follow the same practice, as discussed in Section 5.1.

After their potential gains are computed, we insert all items into their respective priority queues based on these values. The head of a priority queue at any time shows (potentially) the most cost-effective item to be cached from that particular cache type. We note that each item gain in a queue represents the total processing time saving that would be achieved if the item is cached. Therefore, gains are comparable across all queues.

Selection and gain update: The selection step finds the item with the highest expected gain by maintaining a “selection” priority queue. The priority queue keeps the current best (head) item in each “cache” priority queue, and hence its capacity is fixed to five items. The head of the selection queue is dequeued and permanently added into the cache. A new item of the same type (the new best item in the queue) is inserted into the selection queue. The procedure is repeated until the cache is full or no more items are left in the cache queues.

During this procedure, caching of an item may affect the frequency or cost saving of other items. In Figure 6.1, we illustrate the dependencies among items

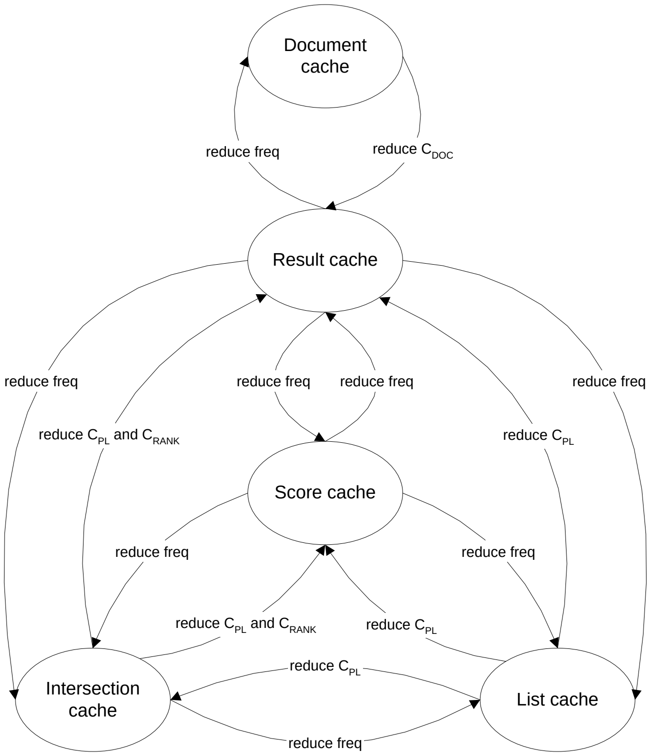


Figure 6.1: Update dependencies in the mixed-order static caching algorithm. Each arc decreases the value of a variable used in the gain computation. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

of different types. In the figure, a directed edge from type T_i to T_j indicates that whenever an item of the former type is cached, the update operation on this edge (as shown on the label) should be executed for the related items of latter type, T_j. The update operation may reduce either the frequency or the cost saving. For example, if the result set of a query is cached, the frequency of the lists for the terms appearing in the query is reduced. In contrast, whenever a list of a term is cached, the cost saving for the results of the queries including that term is reduced by the cost of fetching that list, i.e., C_{pl} for that list. Similar trade-offs exist between different items and cache types.

6.4 Dataset and Setup

As the dataset, we use a collection of around 2.2 million web pages crawled from the open directory project web directory[2]. The dataset is indexed without stemming and stopword removal. The uncompressed index file, which includes only a document identifier and a term frequency per posting, takes 3.2 GB on disk.

As the query log, we use a subset of the AOL query log [66], which contains around 20 million queries issued during a period of 12 weeks. Our training query set, which is used to estimate items' access frequencies, include one million queries from the first 6 weeks. The test query set contains an equal number of queries from the second 6 weeks. Note that, we verify the compatibility of our dataset and query log in Section 3.3.2 of Chapter 3.

Query terms are normalized by case-folding and sorted in alphabetical order. We also removed punctuation and stopwords. Queries are processed in conjunctive mode, i.e., all query terms appear in documents matching the query. In performance evaluations, we omitted queries that do not match any documents.

Queries in the training set are used to compute the item frequencies for results (equivalently, scores), lists and intersections. Without loss of generality, we consider intersections only for the term pairs that appear in queries. Access frequencies for documents are obtained from query results.

In our simulation runs, we consider a distributed search cluster where each search node caches all five types of data items. However, while computing costs of data items, we restrict our experiments to a single search node. This choice does not cause any issue since we neglect network and result merging overheads, as discussed in Section 6.2. Furthermore, since query processing in a search cluster is embarrassingly parallel, query processing times on a 1-node system with Q queries and D documents are comparable to those on a K-node system with $K \times Q$ queries and $K \times D$ documents. In this sense, our choice of a 3.2

[2] Open directory project, available at http://www.dmoz.org.

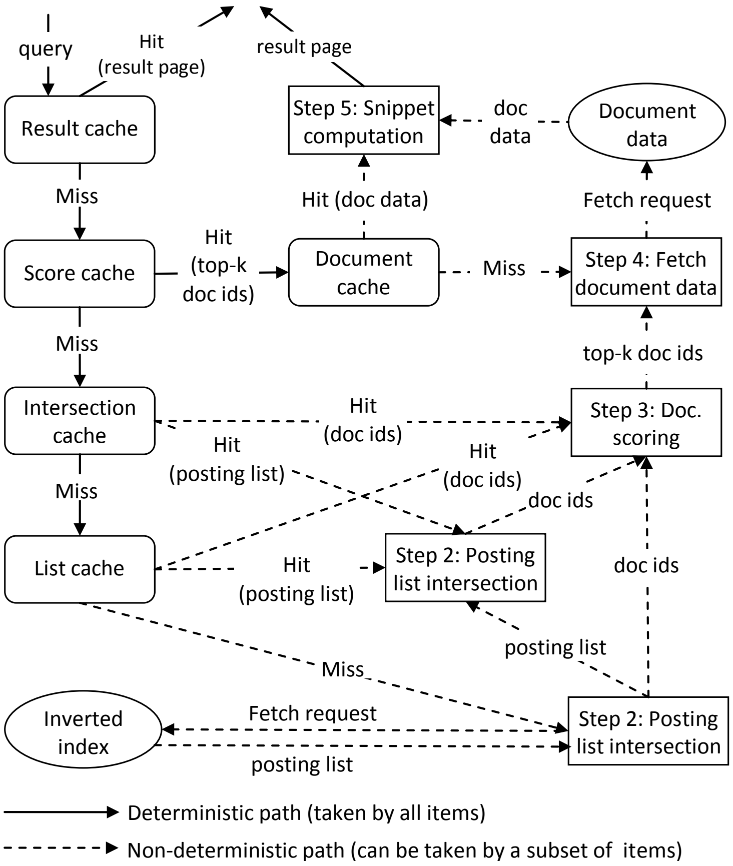


Figure 6.2: The workflow used by the simulator in query processing. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

GB dataset is not arbitrary, but intentional. We anticipate that the data size we use in our simulations roughly corresponds to the size of data that would be indexed by a single node in a typical search cluster. Nevertheless, in what follows, we discuss how the costs associated with the aforementioned four steps in query processing (see Section 6.2) are computed to model the workflow of a node in a realistic manner. Figure 6.2 illustrates the workflow used by our simulator and also illustrates the interaction between different cache components.

In a real-life search cluster, the processing cost of a query depends on all nodes, since partial results (i.e., steps 1 and 2) have to be computed at each node. However, under a uniform workload assumption (i.e., no load imbalance), the execution time on a node would almost be the same as the time on other nodes. Thus, in the simulation runs, a node considers its own processing time for the query and assumes that it would not need to wait for other nodes, i.e., partial results from those would also be ready at the same time[3]. Subsequently, in our setup, computation of C_{pl} and C_{rank} values for finding the partial top k results in a single node is a realistic choice[4].

Once partial results are computed and sent to the broker node for a particular query, this node fetches the documents and generates snippets to create the final result. For this stage of processing (i.e., steps 3 and 4), again each node considers its local execution time. However, during snippet generation, a node itself may not need to take into account the time for creating snippets for all top 10 results. This is because the documents are partitioned into nodes and, for a practically large number of servers in a cluster (e.g., in the order of hundreds [27]), it is highly likely that each document in the final top-10 set will be located in a different node. Thus, we can model the cost of document access and snippet generation steps for only one document, i.e., presumably for the highest ranking document in its partial result set. In other words, we assume that a node would contribute only its top-scoring partial result to the final top 10 results. This is a reasonable assumption, given the high number of nodes in a cluster and search engines' desire for providing diversity in their top-ranked results.

In the simulations, we assume that the node at hand experiences the cost of producing partial results for top 10 documents and then producing the snippets for the highest scoring document (d_{top}). We believe that this setup reflects the costs that would be experienced by each node in a search cluster as close as possible. The cost values associated with each query step is computed using the formulas shown in Table 6.2. Note that there is a subtle detail in the computation of C_{rank} for an intersection item. For an intersection of two lists I_1 and I_2, the total posting count of a query is computed as $|I_1| + |I_2| - |I_1 \cap I_2|$ since the gain

[3] In practice, a search engine may enforce an upper bound on the execution time at the nodes so that the execution terminates when this threshold is reached [21].

[4] Note that search engines usually produce 10 results for each result page [47]. Hence, we essentially focus on generating 10 results at a time.

Table 6.2: Cost computations in the cache simulation. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., “A Five-level Static Cache Architecture for Web Search Engines,” Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

Cost	Notation	Computation
Posting lists access	C_{pl}	$D_{\text{seek}} + D_{\text{rotation}} + D_{\text{read}} \times \lceil \frac{\lvert I_i\rvert \times S_p}{D_{\text{block}}} \rceil$
Ranking	C_{rank}	$\text{CPU}_{\text{scoring}} \times \sum_{t_i \in q} (\lvert I_i\rvert \times S_p)$
Document access	C_{doc}	$D_{\text{seek}} + D_{\text{rotation}} + D_{\text{read}} \times \lceil \frac{\lvert d_{\text{top}}\rvert}{D_{\text{block}}} \rceil$
Snippet generation	C_{snip}	$\text{CPU}_{\text{snippet}} \times \lvert d\rvert$

in this case avoids to process lists $|I_1|$ and $|I_2|$ in entirety.

The parameters used in the simulations are listed in Table 6.3. The default parameters are determined by either consulting the literature or through experimentation. In particular, parameters regarding the disk access times are figures for a modern disk [70]. Storage parameters are based on typical assumptions in the literature, i.e., a posting size is usually assumed to take 8 bytes (4 bytes for document id and term frequency). The scoring time is computed by running experiments with the publicly available Terrier system [59] on our dataset. Finally, we assume a rather simplistic snippet generation mechanism (e.g., highlighting the first appearance of the query words, as well as a couple of words surrounding them in a document) and set the snippet computation time to a fraction of the scoring time.

6.5 Experiments

In this section, we evaluate the performance of several cache architectures in terms of the total query processing time. In particular, we first compare the performance of each cache type separately. Next, we discuss performance of some previously proposed two- and three-level cache architectures, where each cache type is reserved a fixed portion of the available cache space. Finally, we

Table 6.3: Simulation parameters. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

Parameter	Type	Notation	Default value
Result item size	Storage	S_r	512 bytes
Score item size	Storage	S_s	8 bytes
Posting size	Storage	S_p	8 bytes
Disk seek	Disk	D_{seek}	8.5 ms
Rotational latency	Disk	D_{rotation}	4.17 ms
Disk block read	Disk	D_{read}	4.883 ns
Block size	Disk	D_{block}	512 bytes
Cache lookup cost	CPU	$\text{CPU}_{\text{lookup}}$	40 ns
Scoring cost per posting	CPU	$\text{CPU}_{\text{scoring}}$	200 ns
Snippet generation cost per byte	CPU	$\text{CPU}_{\text{snippet}}$	10 ns
Requested query results	Other	k	10

evaluate the performance of our five-level cache architecture with mixed-order caching strategy and show that it is superior to others.

6.5.1 Performance of single-level cache architectures

In Figure 6.3, we show the total query processing time versus cache size (ranging from 1% to 50% of the full index size in bytes), separately for each cache type. As expected, for the smallest cache size (i.e., 1% of the index), the performance of the score cache is the best, and the result cache is the runner-up. In this case, it is not possible to cache all non-tail queries by the result cache. The processing time achieved by the intersection cache is better than that of the list cache, and the document cache is the worst. However, as the cache size grows, the list cache becomes the most efficient choice as the terms are shared among many queries. The intersection cache, which is shared by fewer number of queries, performs better at the beginning, but then becomes inferior to the list cache as the cache size exceeds 10% of the index. For the larger cache sizes, the result cache is better than the score cache, but cannot compete with the list cache, as also discussed in [6]. Finally, our findings show that caching only documents is not a feasible

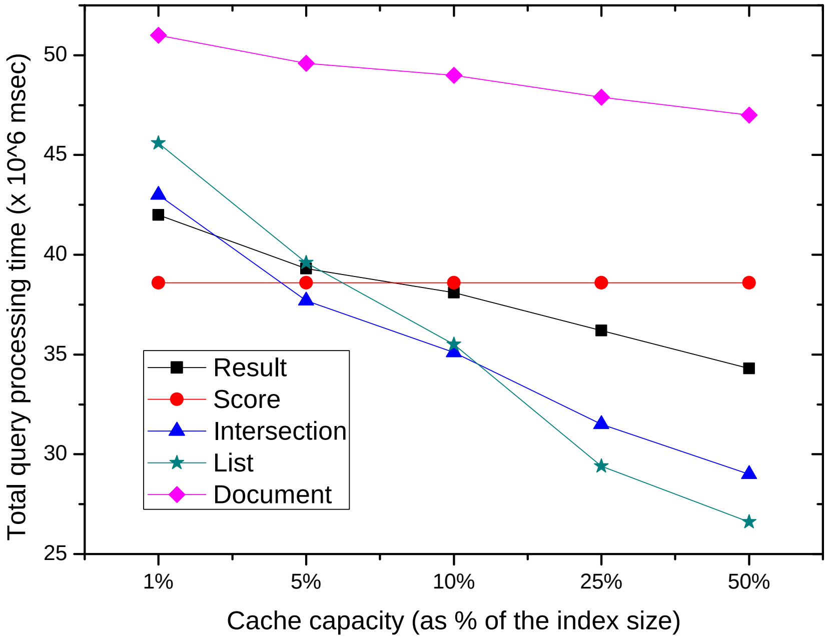


Figure 6.3: Performance of one-level cache architectures. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., “A Five-level Static Cache Architecture for Web Search Engines,” Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

choice at all for a search engine, as fetching posting lists are much more expensive than the former.

In the experiments mentioned above, the caching decision for a particular item is given by using the gain function (Eq. 6.1), where the frequency of an item is simply set to its past access frequency observed in the training query log. However, the frequency values observed in the training log are not necessarily true indicators of the future frequencies for some of the item types, like result pages [64]. This can be explained by the fact that, for all item types relevant to the query processing process, the access frequencies follow a power-law distribution[5].

[5] This is also verified for our query logs in a separate set of experiments that are not reported here.

Table 6.4: Future frequency values for past frequencies smaller than 5. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

Past query (or term) frequency	Future query frequency	Future query term frequency
1	0.15	0.73
2	0.66	1.46
3	1.53	2.27
4	2.47	3.15
5	3.48	3.93

Thus, while relatively few items are repeated many times in the log, majority of the items are asked rarely, or only once. This is especially true for result pages (and scores), as previous works show that almost 50% of the queries in a given query stream are singleton. For such rarely encountered items, future access frequencies may significantly differ than the training frequencies. This may diminish the performance of the static cache.

As a remedy to this problem, following the practice of [32], we construct a table to experimentally obtain future frequency values corresponding to a small set of past frequency values, i.e., up to 20 (it is shown that queries with past access frequency higher than 20 achieve almost the same future frequency in [32]). To this end, we use another subset of the AOL log different from the training and test logs employed throughout this section. This validation log is split into two parts, representing past and future observations. For each frequency value $f < 20$, we check the future frequency values of those queries that have frequency f in the past. The average of such frequency values is recorded as the future frequency value for f. The same idea is applied for the frequency of query terms (i.e., list items).

In Table 6.4, to save space, we only report the frequency values obtained for past frequencies smaller than 5, for both queries and query terms. We observe that the frequency values for query terms are more stable in comparison to query

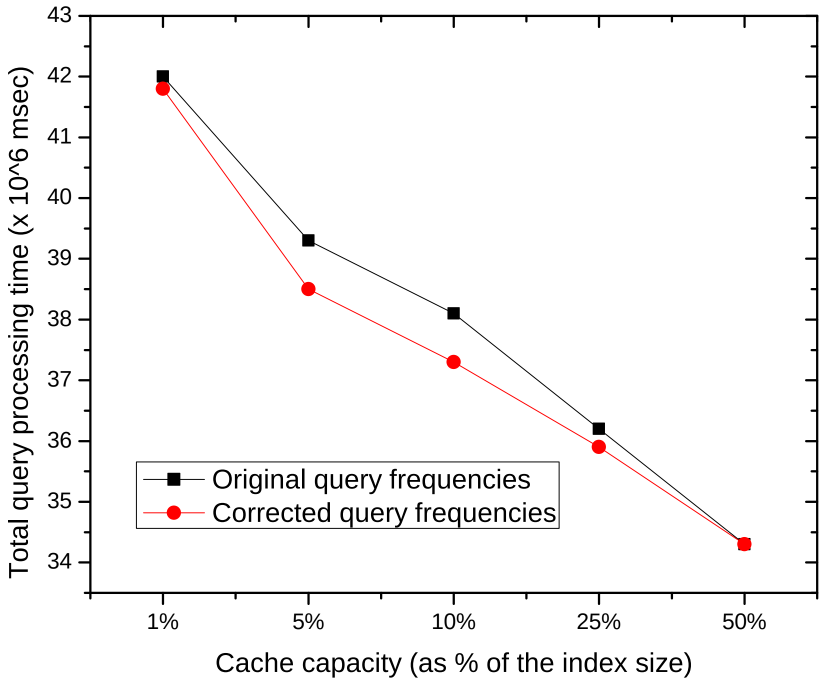


Figure 6.4: The effect of frequency correction on the result cache performance. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

frequencies. This suggests that frequency correction is more crucial for query result caching than posting list caching.

In Figure 6.4, we show the effect of frequency correction on result caching performance. Here, while computing the gains, we use the estimated future frequency value whenever the training frequency of the query is found to be less than 20. According to the figure, the improvement is higher for medium cache sizes (5% or 10% of the index size). As the cache size grows, the gains become smaller since the majority of result items can fit into the cache (for instance, a static cache with a capacity equal to 50% of the full index size can store more than 80% of all results). In this case, the remaining result pages are for the

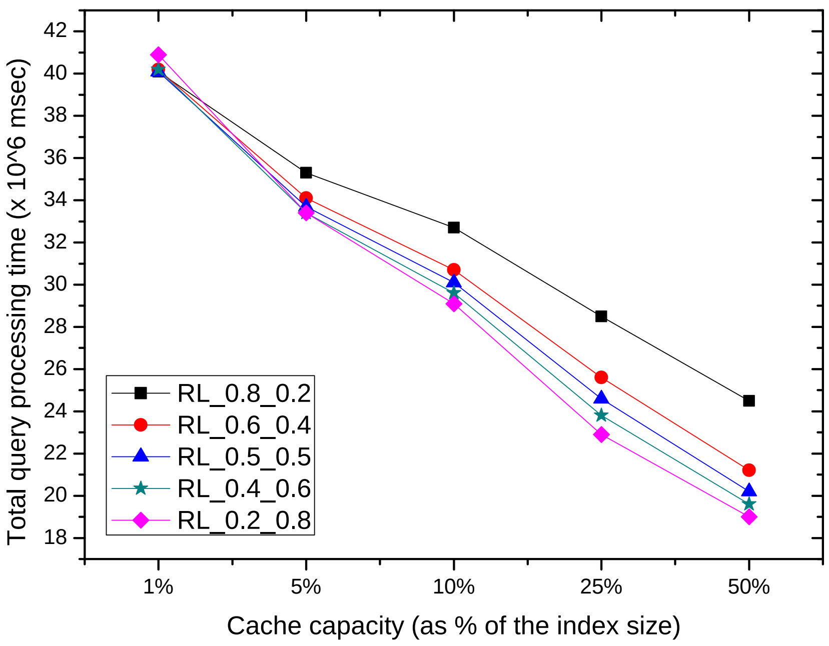


Figure 6.5: The performance of two-level caches for varying split ratios of cache space between result (R) and list (L) items. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

singleton queries, all of which would have the same estimated future frequency (i.e., 0.15 in Table 6.4) and the same relative caching order, as before. On the other hand, our experiments using corrected frequencies for list items do not yield any improvement. Hence, in the rest of the experiments, we employ frequency correction only for result items.

6.5.2 Performance of two-level and three-level cache architectures

In most work in literature, a two-level cache architecture that involves a result and a list cache is considered. An extension of this architecture is three-level caching, where an intersection cache is introduced. Note that, in [51], the intersection cache is assumed to be on disk, whereas we assume that all caches are in the memory.

In Figure 6.5, we demonstrate the performance of two-level caching for various divisions of cache space between the two caches. Our findings confirm those of [6] in that the minimum query processing time is obtained when almost 20% of the cache space is devoted to results and the rest is used for the lists. Note that, while filling the cache, the frequencies of the cached results are reduced from those of the lists, as recommended in [6].

We show the comparison of the best performing two-level cache with two-level mixed-order cache in Figure 6.6. It is seen that mixed-order caching considerably improves with the frequency estimation. On the other hand, the performance improvement in the baseline two-level cache due to frequency estimation is minor. Nevertheless, mixed-order two-level cache with frequency estimation achieves as good performance as the baseline cache. Note that, there is no tuning overhead in mixed-order caching and the division ratio of cache space among results and lists is decided by the algorithm adaptively, which is an advantage over the baseline.

In Figure 6.7, we show the same plot for three-level caches. Here, the best performance for three-level caches is obtained when 20% of the cache space is reserved for result items, 20% for intersection items, and the remaining 60% for list items. Note that, determining the best division of the cache capacity among these item types requires experimenting with a large number of combinations; and these are not reported here to save space. On the other hand, as before, our mixed-order caching approach achieves the same performance as the best case of the baseline algorithm without any tuning.

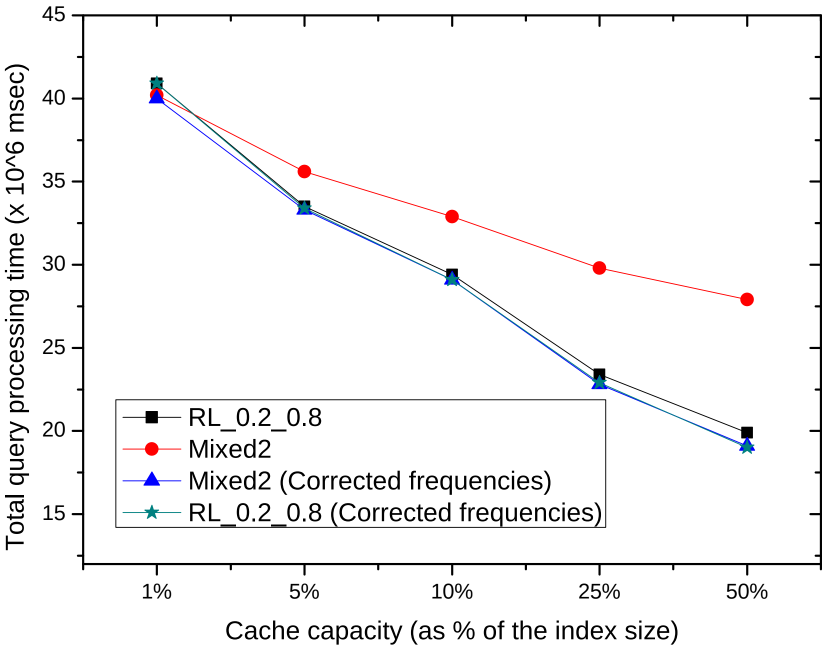


Figure 6.6: The comparison of baseline two-level cache with two-level mixed-order cache. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

6.5.3 Performance of five-level cache architecture with mixed-order algorithm

In this section, we analyze the performance of our five-level cache architecture. To the best of our knowledge, there is no work in the literature that investigates static caching of these five item types (results, scores, intersections, lists and documents) in a single framework. As in previous sections, we reserve a fixed fraction of the cache space for each item type in the baseline five-level cache. However, as it might be expected, the tuning of the cache space splitting becomes a very tedious job in the five-level architecture.

In the five-level mixed order caching experiments, we realized that score items

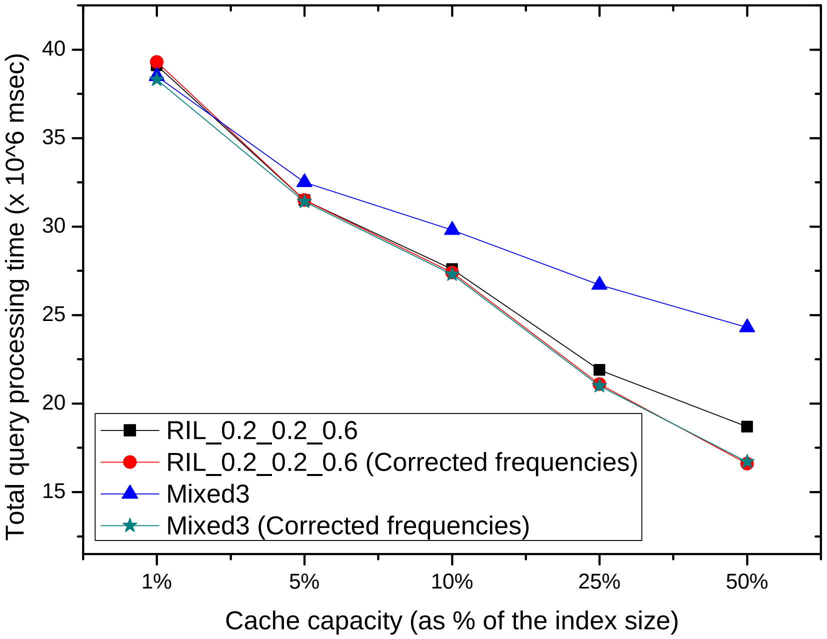


Figure 6.7: The comparison of baseline three-level cache with three-level mixed-order cache. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

turn out to be the most profitable item type according to the greedy selection approach. This is because a score item provides significant cost savings (i.e., all query processing costs, other than the snippet generation cost, are eliminated) while consuming very small storage space (see Table 6.3). Thus, even for very small cache sizes (e.g., %1 of the index), the majority of the score items are selected for caching before any other item types. This causes a devastating effect on the item types that are related to scores (see Figure 6.1). Since almost all score items are brought into the cache, the frequency values of the result items, intersection items, and list items are significantly reduced. This makes documents the only profitable item type to be cached after the scores. Clearly, a cache configuration of mostly scores and documents would yield a poor performance.

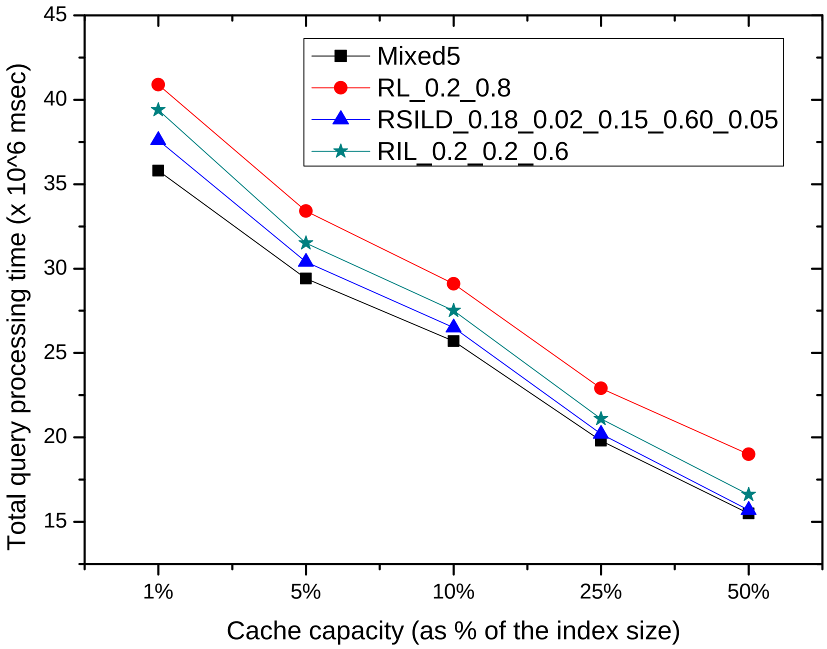


Figure 6.8: The comparison of baseline two-, three-, and five-level caches with five-level mixed-order cache. (Ozcan, R., Altingovde, I.S., Cambazoglu, B. B., Junqueira, F. P., Ulusoy, O., "A Five-level Static Cache Architecture for Web Search Engines," Information Processing & Management, In Press. ©2011 Elsevier. http://dx.doi.org/10.1016/j.ipm.2010.12.007. Reprinted by permission.)

As a remedy, we still allow the greedy approach to cache the score items before the others, but do not permit those score items to affect frequencies of related items. Our decision is based on the observation that all possible score items require a tiny fraction of the cache space, i.e., in our case, all score items take about only 1% of the index size. Hence, it is reasonable to keep them in the cache as long as they are not allowed to totally reset frequencies of related items.

In Figure 6.8, we provide the comparison of our five-level mixed-order caching approach with the best results obtained for the baseline two-level and three-level caches, proposed in literature. We also obtain a five-level baseline cache by experimentally determining the fraction of the cache space that should be devoted to each item type. For this latter case, the best results are obtained when 18% of cache space is reserved for result items, 2% for score items, 15% for intersection items, 60% for list items, and 5% for document items. Our findings reveal that mixed-order caching outperforms the baselines at all cache sizes. In particular, our approach yields up to 18.4% and 9.1% reduction in total query processing time against the two-level and three-level caches, respectively. Furthermore, the highest improvement against the baseline five-level cache is 4.8%.

6.6 Conclusion

In this chapter, we present a hybrid static cache architecture that brought together different types of caches that are independently managed in web search engines. We also propose a greedy heuristic for prioritization of different data items for placement in the cache. This heuristic takes into account the inter-dependencies between processing costs and access frequencies of data items. The proposed approach is shown to provide efficiency improvements compared to architectures in which caches are manipulated independent of each other. In particular, cost-based mixed-order cache yields up to 18.4% and 9.1% reduction in total query processing time against the two-level and three-level caches, respectively. Furthermore, the highest improvement against the baseline five-level cache is 4.8%.

Chapter 7

Conclusion

Large scale search engines try to cope with rapidly increasing volume of web content and increasing number of query requests each day. Caching is one of the crucial methods that can help reducing this burden on search engines. Query result caching in the context of search engines has become a popular research topic in the last decade. In this thesis, we focus on this problem and propose solutions that can contribute to the performance improvement of caching in web search engines.

We first present cost-aware caching policies both for static and dynamic caching of query results. We show that query costs vary significantly and cost aware policies outperform its non-cost-aware counterparts. We also exploit navigational queries for caching. It is shown that result page browsing behavior of users for navigational queries is very different from that for informational queries. We propose result page models and evaluate their performance using real query logs. We show that it is possible to obtain reductions in cache miss counts using the result page model proposed for navigational queries. It is shown through a user study that the proposed result page model does not affect users' browsing behavior in a negative manner.

As another contribution of this thesis, we propose a storage mechanism for query results by exploiting the queries with similar results. We cluster queries with common result documents and provide a compact storage model for these clusters.

Finally, we propose a five-level static cache architecture that consists of five different cache items such as query results (in the form of HTML result pages and document identifiers), posting lists of terms, and document contents. We show the inter-dependency among cache items and provide a greedy approach.

Caching for search engines is a recent research topic and there are still open problems that can be considered as future work. We envision that processing and caching requirements may differ for different types of queries (e.g., navigational queries may have different requirements than informational queries, as discussed in [63]). In our future studies, we plan to extend the cost-aware strategies to take into account the characteristics of such query types.

As a future work for query intent aware caching, we first plan to investigate the use of non-uniform result page models for query types other than navigational. Furthermore, transactional query identification might be exploited for more relevant sponsored search links. Indeed, it is even possible to have a separate cache specialized for each query type, i.e., each cache employs not only different result page models but also different caching policies. We also plan to exploit navigational queries to reduce the cost of some other major tasks, such as the actual query processing, in a web search engine.

Our work on five-level static cache architecture can be extended in two ways. First, the inter-dependencies between different caches need to be investigated for the dynamic caching setting. Although this has already taken some attention [33], it is still not clear how cost updates can be performed on-the-fly and what data structures are needed to perform these updates efficiently. Another research direction is to consider our hybrid cache architecture in a setting where cache entries are refreshed.

Bibliography

[1] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy. Timestamp-based cache invalidation for search engines. In *Proceedings of the 20th International Conference Companion on World Wide Web (WWW '11)*, pages 3–4, 2011.

[2] 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 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 973–982, 2011.

[3] M. F. Arlitt, R. J. F. L. Cherkasova, J. Dilley, and T. Y. Jin. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Perform. Eval. Rev.*, 27(4):3–11, 2000.

[4] A. Ashkan, C. Clarke, E. Agichtein, and Q. Guo. Classifying and characterizing query intent. In *Proceedings of the 31th European Conference on IR Research on Advances in Information Retrieval*, pages 578–586, 2009.

[5] R. Baeza-Yates, L. Calderon-Benavides, and C. Gonzalez-Caro. The intention behind web queries. In *Proceedings of String Processing and Information Retrieval*, pages 98–109, 2006.

[6] 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*, pages 183–190, 2007.

[7] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *ACM Transactions on the Web*, 2(4):1–28, 2008.

[8] R. Baeza-Yates, C. Hurtado, and M. Mendoza. Improving search engines by query clustering. *Journal of the American Society for Information Science and Technology*, 58(12):1793–1804, 2007.

[9] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. F. Witschel. Admission policies for caches of search engine results. In *Proceedings of 14th International Symposium on String Processing and Information Retrieval, Lecture Notes in Computer Science (Springer Verlag), Vol. 4726*, pages 74–85, 2007.

[10] R. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *Proceedings of 10th International Symposium on String Processing and Information Retrieval, Lecture Notes in Computer Science (Springer Verlag), Vol. 2857*, pages 56–65, 2003.

[11] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE Computer Society*, 23(2):22–28, 2003.

[12] D. Beeferman and A. Berger. Agglomerative clustering of a search engine query log. In *Proceedings of the Sixth ACM SIGKDD international Conference on Knowledge Discovery and Data Mining*, pages 407–416, 2000.

[13] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topically categorized web query log. In *Proceeding of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 321–328, 2004.

[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 19th International Conference on World Wide Web*, pages 1065–1066, 2010.

[15] 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 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 82–89, 2010.

[16] E. Bortnikov, R. Lempel, and K. Vornovitsky. Caching for realtime search. In *Proceedings of 33rd European Conference on Information Retrieval, Lecture Notes in Computer Science (Springer Verlag), Vol. 6611*, pages 104–116, 2011.

[17] D. J. Brenes, D. Gayo-Avello, and K. Perez-Gonzalez. Survey and evaluation of query intent detection methods. In *Proceedings of the 2009 Workshop on Web Search Click Data*, pages 1–7, 2009.

[18] A. Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.

[19] F. Cacheda and A. Vina. Experiencies retrieving information in the world wide web. In *Proceedings of the 6th IEEE Symposium on Computers and Communications*, pages 72–79, 2001.

[20] B. B. Cambazoglu. *Models and algorithms for parallel text retrieval.* PhD thesis, Bilkent University, 2006.

[21] B. B. Cambazoglu, F. 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*, pages 181–190, 2010.

[22] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 18–18, 1997.

[23] L. Cherkasova and G. Ciardo. Role of aging, frequency and size in web caching replacement strategies. In *Proceedings of the 2001 Conference on High Performance Computing and Networking (HPCN'01), Lecture Notes in Computer Science (Springer Verlag), Vol. 2110*, pages 114–123, 2001.

[24] K. Church, B. Smyth, K. Bradley, and P. Cotter. A large scale study of european mobile search behaviour. In *Proceedings of the 10th International*

Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '08), pages 13–22, 2008.

[25] N. Craswell, O. Zoeter, M. Taylor, and B. Ramsey. An experimental comparison of click position-bias models. In *Proceedings of the International Conference on Web Search and Web Data Mining*, pages 87–94, 2008.

[26] E. S. de Moura, C. F. dos Santos, D. R. Fernandes, A. S. Silva, P. Calado, and M. A. Nascimento. Improving web search efficiency via a locality based static pruning method. In *Proceedings of the 14th International Conference on World Wide Web*, pages 235–244, 2005.

[27] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 1–1, 2009.

[28] G. E. Dupret and B. Piwowarski. A user browsing model to predict search engine click data from past observations. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 331–338, 2008.

[29] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, 1984.

[30] 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*, 24(1):51–78, 2006.

[31] A. Fujii. Modeling anchor text and classifying queries to enhance web document retrieval. In *Proceedings of the 17th International Conference on World Wide Web*, pages 337–346, 2008.

[32] 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*, pages 431–440, 2009.

[33] S. Garcia. *Search engine optimisation using past queries.* PhD thesis, RMIT University, 2007.

[34] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, Inc., 1988.

[35] B. J. Jansen and D. Booth. Classifying web queries by topic and user intent. In *Proceedings of the 28th of the International Conference Extended Abstracts on Human Factors in Computing Systems 2010*, pages 4285–4290, 2010.

[36] B. J. Jansen, D. Booth, and A. Spink. Determining the informational, navigational, and transactional intent of web queries. *Information Processing & Management*, 44(3):1251–1266, 2008.

[37] B. J. Jansen and A. Spink. An analysis of web documents retrieved and viewed. In *Proceedings of the 4th International Conference on Internet Computing*, pages 65–69, 2003.

[38] B. J. Jansen and A. Spink. How are we searching the world wide web? a comparison of nine search engine transaction logs. *Information Processing & Management*, 42(1):248–263, 2005.

[39] B. J. Jansen, A. Spink, C. Blakely, and S. Koshman. Web searcher interactions with the dogpile.com meta-search engine. *Journal of the American Society for Information Science and Technology*, 58(4):1875–1887, 2006.

[40] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *Proceedings of Ninth International Symposium on High-Performance Computer Architecture*, pages 327–337, 2003.

[41] J. Jeong and M. Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, 2006.

[42] T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately interpreting clickthrough data as implicit feedback. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 154–161, 2005.

[43] I. Kang and G. Kim. Query type classification for web document retrieval. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 64–71, 2003.

[44] C. Kofler and M. Lux. Dynamic presentation adaptation based on user intent classification. In *Proceedings of the Seventeen ACM international Conference on Multimedia*, pages 1117–1118, 2009.

[45] U. Lee, Z. Liu, and J. Cho. Automatic identification of user goals in web search. In *Proceedings of the 14th International Conference on World Wide Web*, pages 391–400, 2005.

[46] W. M. Lee and M. Sanderson. Analyzing URL queries. *Journal of the American Society for Information Science and Technology*, 61(11):2300–2310, 2010.

[47] 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*, pages 19–28, 2003.

[48] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *Proceedings of the 6th International Conference on Web Information Systems Engineering, Lecture Notes in Computer Science (Springer Verlag), Vol. 3806*, pages 470–477, 2005.

[49] S. Liang, K. Chen, S. Jiang, and X. Zhang. Cost-aware caching algorithms for distributed storage servers. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*, pages 373–387, 2007.

[50] Y. Liu, M. Zhang, L. Ru, and S. Ma. Automatic query type identification based on click through information. In *Proceedings of the Asia Information Retrieval Symposium*, pages 593–600, 2006.

[51] 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*, pages 257–266, 2005.

[52] L. Lorigo, B. Pan, H. Hembrooke, T. Joachims, L. Granka, and G. Gay. The influence of task and gender on search and evaluation behavior using Google. *Information Processing & Management*, 42(4):1123–1131, 2006.

[53] Y. Lu, F. Peng, X. Li, and N. Ahmed. Coupling feature selection and machine learning methods for navigational query identification. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 682–689, 2006.

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

[55] M. Mendoza and R. Baeza-Yates. A web search analysis considering the intention behind queries. In *Proceedings of the Latin American Web Conference*, pages 66–74, 2008.

[56] M. Mendoza and J. Zamora. Identifying the intent of a user query using support vector machines. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, pages 131–142, 2009.

[57] A. Ntoulas and J. Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 191–198, 2007.

[58] E. Omiecinski and P. Scheuermann. A parallel algorithm for record clustering. *ACM Transaction on Database Systems*, 15(4):599–624, 1990.

[59] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and D. Johnson. Terrier information retrieval platform. In *Proceedings of the 27th European Conference on Information Retrieval*, pages 517–519, 2005.

[60] R. Ozcan, I. S. Altingovde, B. B. Cambazoglu, F. P. Junqueira, and O. Ulusoy. A five-level static cache architecture for web search engines. *Information Processing & Management*, In Press.

[61] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Space efficient caching of query results in search engines. In *Proceedings of the 23rd Int. Symposium on Computer and Information Sciences*, pages 1–6, 2008.

[62] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Static query result caching revisited. In *Proceedings of the 17th International Conference on World Wide Web*, pages 1169–1170, 2008.

[63] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Utilization of navigational queries for result presentation and caching in search engines. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM)*, pages 1499–1500, 2008.

[64] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Transactions on the Web*, 5(2):Article 9, 2011.

[65] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Exploiting navigational queries for result presentation and caching in web search engines. *Journal of the American Society for Information Science and Technology*, 62(4):714–726, 2011.

[66] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems*, page 1, 2006.

[67] B. Piwowarski and H. Zaragoza. Predictive user click models based on click-through history. In *Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management (CIKM)*, pages 175–182, 2007.

[68] S. Podlipnig and L. Boszormenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

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

[70] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* Mc Graw Hill, 2003.

[71] D. E. Rose. Reconciling information-seeking behavior with search user interfaces for the web. *Journal of the American Society for Information Science and Technology*, 57(2):797–799, 2006.

[72] D. E. Rose and D. Levinson. Understanding user goals in web search. In *Proceedings of the 13th International Conference on World Wide Web*, pages 13–19, 2004.

[73] P. C. Saraiva, E. S. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 51–58, 2001.

[74] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[75] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[76] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 175–182, 2007.

[77] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 2007.

[78] C. Tann and M. Sanderson. Are web based informational queries changing? *Journal of the American Society for Information Science and Technology*, 60(6):1290–1293, 2009.

[79] J. Teevan, E. Adar, R. Jones, and M. A. Potts. Information re-retrieval: repeat queries in Yahoo!'s logs. In *Proceedings of the 30th Annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 151–158, 2007.

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

[81] Y. Tsegay, S. J. Puglisi, A. Turpin, and J. Zobel. Document compaction for efficient query biased snippet generation. In *Proceedings of the 31st European Conference on Information Retrieval, Lecture Notes In Computer Science (Springer Verlag), Vol. 5478*, pages 509–520, 2009.

[82] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. In *Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management (CIKM)*, pages 987–990, 2007.

[83] A. Turpin, Y. Tsegay, D. Hawking, and H. E. Williams. Fast generation of result snippets in web search. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 127–134, 2007.

[84] WebBase. Stanford University Webbase Project. www-diglib.stanford.edu/ testbed/doc2/WebBase, 2007.

[85] W. Webber and A. Moffat. In search of reliable retrieval experiments. In *Proceedings of the Tenth Australasian Document Computing Symposium*, pages 26–33, 2005.

[86] J. Wen, Y. Jian, and H. Zhang. Query clustering using user logs. *ACM Transactions on Information Systems*, 20(1):59–81, 2002.

[87] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 1999.

[88] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings of the 21st Annual joint Conference of the IEEE Computer and Communication Societies*, pages 1238–1247, 2002.

[89] Yahoo! Web search API. http://developer.yahoo.com/search/, 2009.

[90] N. E. Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.

[91] C. T. Yu. *Theory of indexing and classification*. PhD thesis, Cornell University, 1973.

[92] Zettair. The Zettair search engine. http://www.seg.rmit.edu.au/zettair/, 2007.

[93] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):Article 6, 2006.