Implement and compare caching paradigms in Search Engines

Ananya Das
Computer Science and
Engineering
University of California
Santa Cruz
adas13@ucsc.edu

ABSTRACT

In this project I aim to compare the different existing caching paradigms like LRU, LFU etc and evaluate the performance of web search engines to efficiently process user queries. Large scale search engines have to cope up 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.

This project presents a comparative analysis of the the different existing caching eviction policies on the search engine that I have built and how the caching policies perform . Also for searching I have used BitFunnel in compare to B+ tree indexes and the performance have increased after using BitFunnel.

Categories and Subject Descriptors

• [**Information Storage and Retrieval**] : Information Search and Retrieval— *Search Process*

KEYWORDS

Web Search Engines, Caching Strategies, Query Processing, Cache Eviction Policies, Searching and Indexing, BitFunnel

1 Introduction

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, and it includes a wide range of different web sites such as giant enterprise web sites, multimedia sharing web-sites, blogs and social sites. It is a challenge to find relevant information from this huge amount of data.

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 collection. It is shown that inverted index file is the state-of-the-art data structure for efficient retrieval [1] . 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 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 initial stage of processing a query must be forwarded to index servers containing the partial inverted index file since it is not possible to accommodate the full index file on a 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.

Another 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, caching search engines employ a number of key mechanisms to cope with these efficiency and scalability challenges. Caching is one of them. 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[2].

Contributions: In this project, I focus on comparing the existing caching eviction policies on my search engine and evaluating the performance of them on how they perform.
I have also used BitFunnel over B+ tree while indexing and the performance have increased.

2 Background and Related Work

In this part I am going to provide some background information for a large scale search engine and related work about caching mechanisms in these systems. Large scale web search engines like Google 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 topdown architecture for a large scale web search engine consisting of geographically distributed search clusters [1].

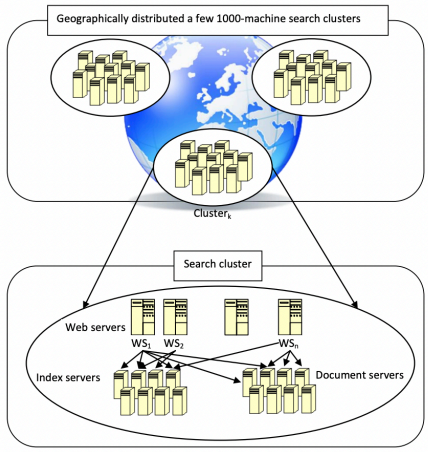


Figure 2.1: Large scale search engine consisting of geographically distributed search clusters.

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.

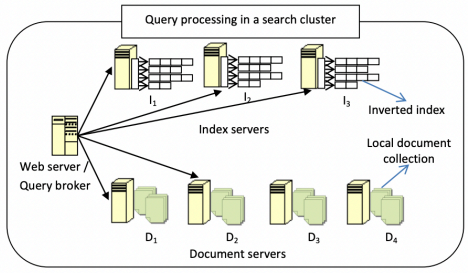


Figure 2.2: Query processing in a search cluster.

Each document server store a subset of the crawled web pages. Web server (query broker) nodes send 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.

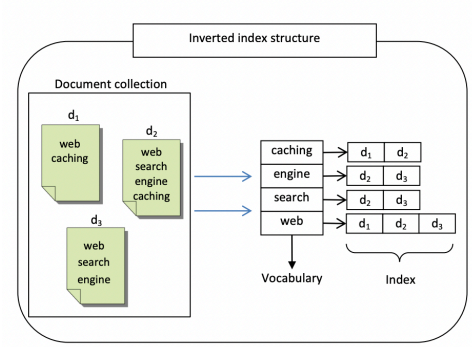


Figure 2.3: Inverted index data structure.

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) [1] in the query broker machine or posting lists for query terms (list cache) in the index servers, or both [1] . 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.

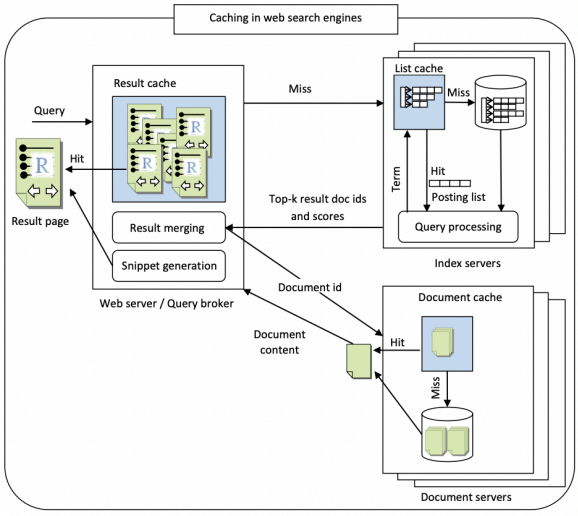


Figure 2.4: Caching in a large scale search engine architecture.

List cache stores the full (or a portion of) posting list of terms. Intersection cache [1] 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 [7, 8] 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 [3]. 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.

2.1 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

2.1.1 **Frequency Based Caching:** 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 [3]. Static cache content is periodically updated. Dynamic result caches can also employ frequency based methods such as the widely-known LFU method [2,3]. 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.

2.1.2 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.

2.1.3 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

2.2 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.

3 Experimental Setup

Datasets: In this study, we use two datasets which have been already provided for HW1.

4 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 [8] 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 [8].

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.

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 Cq and result page size Sq, the H-value is computed as follows:

$$H\ value(q) = Cq\ Sq + L \quad (3.4)$$

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.

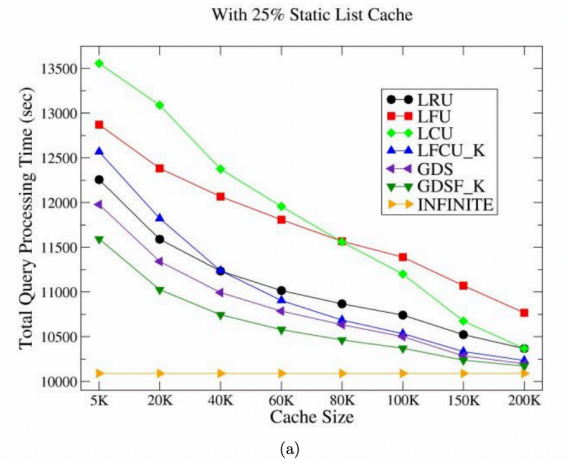
Greedy Dual Size Frequency (GDSF K). This method is a slightly modified version of the GDS replacement policy [3]. In this case, the frequency of cache items is also taken into account. The corresponding H-value formula is presented below.

$$H\ value(q) = F\ K\ q \times Cq\ Sq + L \quad (3.5)$$

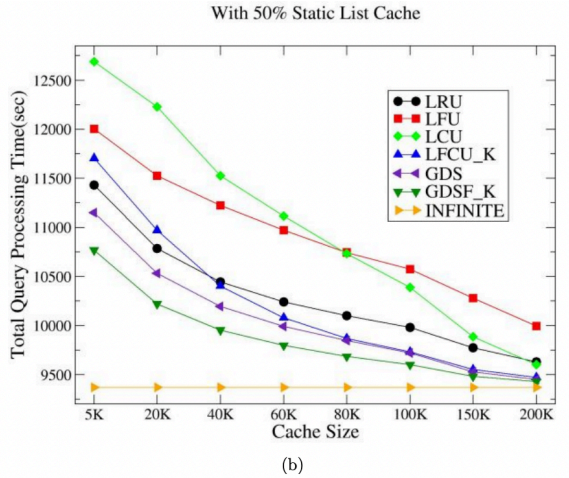
In this strategy, the frequency of the cache items are also kept and updated after each request. As discussed in Section 3.4.1, again we favor the higher frequencies by adding an exponent K(> 1) to the frequency component. Note that with this extension, the formula also resembles the generalized form 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, Cq, and apply weighting only for the frequency values.

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

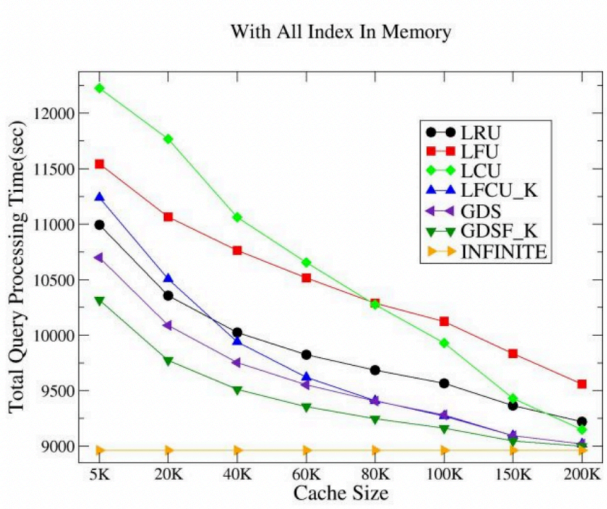
4.1 Results for Dynamic Caching



(a)



(b)



(c)

CONCLUSION

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 dynamic cases.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.

FUTURE WORK

I was about to implement caching based on reinforcement learning but due to time constraint I couldn't complete it. That can be taken up as the next part of the project.

REFERENCES

[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 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.

[7] 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.

[8] 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.

[9] 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.