Efficient Full-Text Searches on Massive Data

Sung Chae Lim

Dongduk Women’s University
sclim@dongduk.ac.kr

Abstract

The scheme for the full-text search has drawn much attention due to its popular use in web document searches and enterprises’ document searches. The full-text search leads to a large size of index files and thus may consume massive computing resources for its processing. In this paper, we present both the system architecture of a full-text search engines with a huge volume of indexed data and its multi-level cache scheme. The presented system architecture and cache scheme were implemented in a commercial search engine, which has capacity enough to process more than 5-milion queries per day and index about 70-milion web documents crawled in Korea. In economic respect, the proposed cache scheme is very crucial for our full-text search engine.

Keywords: full-text searches, search engine, caching, inverted index files

1. Introduction

Nowadays, many access methods for large-volume documents have been implemented based on the mechanism of the full-text search. Especially, in the case of web document searches almost all kinds of search engines use the technique of the full-text searches [1, 9, 10]. In addition, in many enterprises, such a full-text search scheme is widely used for document retrieving [6, 8]. For the purpose of such full-text searches on documents, a search engine has to extract all keywords included in every document and organize them as index files. Since those files saving extracted keywords have almost the same size as that of original target documents, the enlarged size of keyword index files could be a big problem when we have to guarantee reasonable response times for queries on a huge size of data. When the data size exceeds a few Giga bytes, the use of a deliberately devised architecture and a specific caching mechanism is inevitable to avoid an excessive server cost paid for a query processing system of the full-text search engine.

Much research has been done to solve technical challenges related to the efficient method for full-text searches. For example, there have been researches on index schemes [3, 4, 7], ranking algorithms [5, 7], web search engines [2, 3, 8, 11], caching schemes [6, 9, 10]. Although those researches could be useful in order to get a glimpse on basics of the full-text search engine, they may not be informative in practical respect. That is, they do not address many technical issues about the query processing system that is responsible for accepting user queries and returning query results in the fast response time. The query process system (QPS) heavily consumes computer resources for executing join operations over index files and generating short scripts summarizing result documents returned. Therefore, if anyone intends to operate a QPS having more than millions of documents, the QPS have to be organized based on a server clustering technique, if it can be realistic in economic respect [8, 10].

In this paper, we mainly focus on technical issues regarding to the QPS. In particular, we present our experiences in implementing a large-scale web search engine for full-text searches. The QPS was implemented for commercial services in Korea. The processes of the QPS runs on multiple server clusters in a distributed fashion; they communicate with others running on
different servers via high-speed LANs, and query results can be temporarily managed on multi-leveled caches. The caches are comprised of 4-leveled cache storages, which are implemented on main memory and hard disk drives. Owing to the multi-leveled cache scheme and server clustering technique, we can save 70% of server cost paid for the QPS.

The rest of this paper is organized as follows. In Section 2, the index DB structure and the system architecture of the implemented search engine are presented. In Section 3, the core mechanism of the QPS, that is, cache scheme, is described in detail. Then, we discuss the performance and effectiveness of our QPS in Section 4 and conclude in Section 5.

2. Steps for Query Processing

Figure 1 shows the steps for query processing in the QPS. After a user query is received via a web server, that query is dispatched to a coordinator server. Then, the coordinator server sends the query to four ranker servers for parallel processing. The ranker servers perform their equi-join actions over the user query so as to identify a set of DID’s containing the keywords of the query. After this equi-join, the ranker server calculates rank scores for the DID’s from the equi-join. For this, it uses additional index data such as keyword’s occurrence positions, HTML-tag related data, etc. Since the disk accesses to those data always incur frequent disk seek times, the disk usage for ranking becomes over twice that of the equi-join on average.

The rank data are sent to the coordinator server from the four ranker servers. The rank data are merged and sorted to determine DUP_K number of top ranks. To gather summary data for the DUP_K DID’s with top ranks, the coordinator server sends DID’s to SD servers. In the SD servers, summary data are generated and the data will be sent to the coordinator server.
With the data form SD servers, the coordinator server generates a result page for the user query.

3. Multi-level Caching Scheme

3.1 Idea Sketch

Since the size of data read to process a query is very huge, query processing without a cache mechanism may cause a prohibitive server cost. Against this, we design an efficient multi-level cache that can keep already processed query results on memory or disks. To get a glimpse of the benefit of cache usage, let us consider a query having two keywords of $k_1$ and $k_2$. Also let us suppose that the number of web pages containing $k_1$ (or $k_2$) is 20,000 (or 30,000) at each ranker server. Under such keyword frequency, the size of DID list to be read during a join operation in a single ranker server is around 0.4 MB, since an entry of DID list is comprised of 4-byte DID and pointer. Therefore, the disk bandwidth for reading 1.6 MB of data is needed for the join operation at least.

Additionally, IDX files, which are read for rank evaluations after the join operation, are much larger than DID list in size. To avoid reading all the large IDX files, the ranker server reads only the IDX data associate with the join-matching DIDs. Due to seek time overhead during the read, its disk usage becomes over twice that of join operation on average. Consequently, disk I/O usage by the ranker servers amounts to that for reading 4.8 MB of data in all. Such amount could be larger when the query has more join-matching web pages and keywords.

Besides I/O usage in the ranker servers, the SD server also requires a huge disk bandwidth for creating SRTs. To make an SRT, the SD server has to read at least one disk block and the data of SRTs are randomly distributed in disk. Therefore, for a single search query, a hundred of random disk accesses arises with DUP_K = 100. Due to such heavy disk use in the ranker and SD servers, the cache mechanism is unavoidable. Note that the CPU time use is not mentioned for simplicity.

The implemented cache modules resides in both of web servers and the coordinators servers, and they use four multi-level cache data. From level 1 to level 4, the former level cache is first looked up to find a matched cache data prior to the latter ones. From now on, we denote the caches of levels 1 to 4 by CL1, CL2, CL3, and CL4. The CL1 is saved on the main memory of web servers and its data are compressed for efficient memory use. For CL1, the web server allocates a specific portion of main memory and saves each cache data of a query in a fixed-size memory slot. To access the cache data, CL1 uses a hash function mapping a query URL to an address to a memory bucket, where a multiple of memory slots exist. With the hash function the web server first accesses an associated memory bucket, and then it explores the memory slots within that for the search of the matched cache data. If found, the cache data is used for query processing.

In the case of CL2 to CL4, cache data are stored in the disk of the coordinator server. The cache data of CL2 and CL3 are managed in the form of the cache record containing the resultant DIDs and their SRTs, where the DIDs are ones that are obtained after deletion of duplicated SRTs. The cache records of CL2 and CL3 save data to make multiple result pages, rather than a single result page. Therefore, the coordinator server needs to determine the data of a requested result page using the page number expressed in the query URL. By saving data for many result pages within a cache record, the CL2 and CL3 can enhance their cache hit rates with less disk space.
A difference between CL2 and CL3 exists in query sets to be cached. To CL2, all the cached queries are ones which are the most popular 30,000 queries entered earlier. Such popular query set is determined by analyzing query log, and their popularities do not change drastically in a short period. By caching such popular queries in CL2, we can guarantee a high cache hit rate. Moreover, since the cache data of CL2 can be built in batch mode, we can use the CL2 data to prevent a severe cache miss when the current web data is replaced with a new one periodically. Differently from CL2, cache data in CL3 varies according to time-varying user queries, and thus an algorithm for cache replacement is executed in the presence of cache disk shortage. For this, an LRU (Least Recently Used) style algorithm is adopted in CL3 module. The cache record of CL4 contains only DIDs sorted in the order of rank values, without saving their SRTs. Since the size of SRTs are relatively greater compared to UIDs, we can save more DIDs in a CL4 cache record. With CL4 data, the time for deleting duplicate SRTs and performing ranking operation can be eliminated in the query processing time. Since CL4 is managed in dynamic fashion, cache replacement can be done. More detail will be present in the next sections.

At the left-bottom four ranker clusters are located, each of which consists of four ranker servers. The ranker server is aimed at performing join operations to select query-matching web pages and evaluating their rank values based on query relevance. The ranker servers pertaining to the same ranker cluster have identical index files and the whole index files are partitioned into four ranker clusters. For query processing, a query is dispatched to four ranker servers belonging to different clusters respectively and then the ranking results from those ranker servers are all gathered and sorted by the coordinator server. Due to selective choice among four ranker servers in a cluster, we can easily obtain fault-tolerance against ranker server’s failure, as well as high capacity of query processing.

3.2 Proposed Algorithm

In this section, we first describe the cache algorithm of CL1 and then address the algorithms of the rest cache levels. The cache data in CL1 is emptied at the time when a new crawled data are copied into the QPS, and it will be filled up while answering user queries. Fig. 2 depicts the CL1 algorithm. In line 1 of the figure, the hash key is generated on the query URL. With the key \( h\_key \), CL1 locates a memory bucket possibly containing a memory slot that saves the cache data with key \( h\_key \). While exploring the memory slots, the CL1 algorithm decreases the popularity of the unmatched memory slots as in lines 4 to 11 for the future’s cache replacement.

If the cache data is not found in CL1, the keywords and the number of the requested result page are sent to a coordinator server in lines 13 and 14. After receiving the returned result from the coordinator server in line 15, the CL1 algorithm determines whether or not this current result page needs to be saved in CL1 using the condition of line 16. If needed, the result page is compressed and saved with popularity = 50. Of course, if a cache data is found, the data is used without any request to the coordinator server.

On the side of the coordinator server, other levels of caches are managed. In Fig. 2, we depict how such CL2, CL3, and CL4 work. With the condition check at the top of the figure, we first examine if the current query is a popular one saved in CL2. If that is true, the corresponding cache record is read from CL2, and then we check if the requested result page can be made from that cache record. If it is not possible because the requested page number, i.e., page_no, is beyond the page ranges possibly made from the cache record, the CL4 data is used. Otherwise, if the query is not a popular query for CL2, the CL3 are used. These steps are given at the top-half of Figure 2.
(1) \( h\_key \leftarrow \text{get\_key}(\text{keywords, retrieval range}) \).
(2) \( id \leftarrow h\_key \mod \text{BUCKET\_NO} \). /* bucket id */
(3) Let \( S \) be the set of cache slots in the bucket with identifier of \( id \).
(4) \( record = \text{nil} \).
(5) \( \text{foreach} \ s \in S \)
(6) if \( s\_h\_key == h\_key \) /* cache hit */
(7) \( record \leftarrow \text{pointer to} \ s \).
(8) \( s\_\text{popularity} += 1 \).
(9) \( \text{else} \)
(10) \( s\_\text{popularity} -= 1 \).
(11) \( \text{endfor} \)
(12) if \( (record \neq \text{nil}) \)
(13) Uncompress the cache record saved in \( record \) for returning a search result page.
(14) \( \text{else} \)
(15) Forward the user query to a coordinator server and receive the query result saved in \( q\_result \).
(16) Select a free slot \( s \) from \( S \) and save the compressed \( q\_result \) into \( s\_\text{data} \), and set \( s\_\text{popularity} \) to 50, if the current result is to be cached.
(17) endif

Fig. 2 Cache algorithm for CL1.

4. Conclusion

In this paper we have addressed some useful mechanisms to accelerate the processing speed of full-text searches on a large volume of data. For fast full-text searches, we designed a distributed system comprised of server clusters with different purposes so that it is possible for a full-text search to be executed in parallel. In addition, we use a hierarchical cache scheme that is very useful to reduce the computing resources needed for query processing. That is, query results are stored across caches of four different levels in a distributed fashion, and those caches are stored in main memory or hard disks in different computers. Using that caching scheme, we can reduce around 70% of the server cost. We believe that our paper gives much information to those who have an interest in implementing a large scale of full-text search engine.

References


Authors

Sungchae Lim

He received the B.S. degree in Computer Engineering from Seoul National University at 1992, and achieved the M.S. and Ph.D. degrees in Computer Science from KAIST, at 1994 and 2003, respectively. He is currently an Associate Professor in the Department of Computer Science at Dongduk Women’s University.