A New Distributed Caching Technique for Accelerating the Web Query Processing

Sungchae Lim and Chang-Sup Park
Dept. of Computer Science, Dongduk Women’s University
Seoul 136-714, South Korea
{sclim, cspark}@dongduk.ac.kr

Abstract
Because of the fast growing volume of web documents during the past decades, the efficiency of the web search engine has become more crucial than ever. Such efficiency can be estimated with both factors of the query relevance of search results answered and the financial cost for query processing. Between them, the ways for improving query relevance of web searches have been intensively studied in the research topics like hyperlink-based ranking, topic-sensitive document classifications, and semantic-awareness in rank evaluations. However, there have been not studies that provide an efficient solution to cut the financial cost of query processing, while retaining high query relevance. In this light, we propose a distributed cache scheme and a server-clustering technique that can be used to reduce the query processing cost. With the help of such techniques for accelerating the web query processing, we saved around 70% of the server cost of a commercial web search engine implemented in South Korea. We believe that our experiences can give a valuable insight to anyone who wants to develop a large-scale search engine.

Key Words: Web search engine, distributed system, cache scheme, inverted index files

1. Introduction
Due to a fast growing number of documents in the web, web search engines capable of retrieving them in a quick and cheap way have become more crucial than ever [1, 5]. However, the web search engine (WSE) nowadays faces a dilemma between the necessity of huge web coverage and the financial limit on its server cost. For a WSE to yield query results with high query relevance, it is essential to crawl web documents as widely as possible. That is, web coverage of the WSE can be a major factor for the high quality of search results [2, 3]. Unfortunately, however, such enlarged web coverage is apt to increase its server cost because of increasing I/O demands and CPU time usage during the query processing time [4]. To solve the dilemma, it is required to devise a cost-effective mechanism that can evade such heavy use of computing resources without impairing the quality of search results.

In this context, many studies have been done for saving the server cost of the WSE. For instance, efficient techniques for handling compressed index files were proposed to reduce the amount of disk reads needed for joining and ranking operations [1, 5, 11]. Meanwhile, other studies [6, 8, 12] scrutinize query logs to pick popular user queries and cook cache data for them. The cache data from the query log analysis can be managed in a static manner or a dynamic manner, based on LRU-like replacement algorithms [8]. Besides, the techniques of server clustering were proposed to shorten the query response times through the server parallelism in query processing [5, 10]. In these techniques, a user query is dispatched to a group of servers in parallel, where the servers concurrently perform ranking operations using their own partitions of the whole index files. Due to the server parallelism, the response time can be kept less than a specific time limit. Such server parallelism is also employed for doing repetitive computations required for hyperlink-based evaluations of web documents as in PageRank [4, 7].

Although the earlier literature above covers various challenging issues concerned with the WSE, there was not enough information on the realistic technique able to accelerate query
processing. By contrast, we place our research focus on the distributed cache scheme that is implemented for a commercial WSE at the aim of a cheap server cost. Since the query processing system of the WSE is comprised of sever clusters, our cache scheme is also designed in the consideration of the distributed architecture of sever clusters. That is, our cache data are divided into multi-levels, and different levels of data are stored over different server clusters. Owing to such a distributed cache scheme, we can save 70% of server cost paid for the query processing system in the WSE.

The rest of this paper is organized as follows. In Section 2, we present the index structure and the procedure for query processing. In Section 3, the main contribution of the cache mechanism is described. Then, we discuss the performance and effectiveness of the cache scheme in Section 4, and conclude the paper in Section 5.

2. System Overview

Since the WSE services users’ full-text searches based on keyword matching in general, its index files are usually organized as inverted files for efficiently saving index tokens of keywords and performing join operations using them [1, 5, 12]. A single index token has the form of (keyword ID, Document ID, a keyword description). Here, the entries of the keyword description are comprised of various indexing information such as the locations of keyword’s occurrences in the document and various contextual data, that is, keyword’s font style, morpheme analysis data, and a flag data notifying whether the keyword is included in its document title. Those indexing information are utilized to determine query relevance orders among indexed documents in the time of rank evolutions.

![Figure 1. The multi-leveled structure of the stored index files. Three levels of index files are used to efficiently process a web query in parallel](image)

When the web coverage of a WSE becomes greater than the scale of ten millions, the file size for saving index tokens easily grows too large to handle them in a naive way. Therefore, there is a need to organize the index token files into multi-level sub-files for the purpose of parallel query processing. Figure 1 gives the simplified view of our index files that have been organized in three levels in all. At the top level, the keyword directory stores a group of disk addresses to the DID lists. For fast look-ups, the keyword directory is managed within main memory and it saves the address of the DID list for each keyword. At the middle level, the files of DID lists exist so that its entries save the disk addresses of the associated IDX file and DIDs. For this, with respect to each keyword k, the DID list saves DIDs of the web documents containing keyword k and the IDX file addresses involved with k. At the bottom level, the IDX data is created to store the before-mentioned keyword descriptions of the index tokens on disks.
During the time of query processing, our WSE first executes join operations using the keyword directory. From the joining, a set of DIDs are selected so that the corresponding documents have all the keywords found in the user query. Consecutively, the WSE performs rank evaluations for the selected DIDs, in order to give query relevance orders to them. We call such two steps of join operations and rank evaluations the ranking operation.

![Diagram of the implemented WSE architecture.](image)

**Figure 2. The architecture of the implemented WSE. The servers of the system are inter-connected via high-speed LANs**

Our WSE was designed to process over 5 millions of users’ queries a day. The real data size of indexed documents is around 700 GB in all, and their inverted index files of Figure 1 are around 600 GB in size. To fulfill the requirement of short response times, we design our query processing system such that a single web query is processed over multiple servers that are inter-connected through high-speed LANs. According to this distributed architecture, our index files are partitioned and stored in the servers in a distributed manner.

Figure 2 illustrates the overall architecture of the WSE that was implemented for a commercial purpose. When a user query arrives at the particular port allocated by the IDC (Internet Data Center), that query is dispatched to a web server through the firewall and the load balancer. Then, the web server begins to parse the accepted query, and checks if it can answer the query by using the cache data on itself. If it is not possible, the web server will send the parsed query to a coordinator server for processing. Consecutively, the coordinator server completes the query processing in the corporation with LAN-connected ranker servers and summary servers.
Since our systems should be operational in spite of abrupt hardware failures among the involved servers, our system has duplicated clusters of servers for redundancy. With the technique of failovers among the duplicated redundancy servers, our system can avoid any catastrophic system crash in the presence of failures on several servers. Moreover, if any failure arises, that will be sensed by the performance monitor and an alert message will be automatically sent to an associated server administrator.

If every user query has to be processed from scratch without using cache, the costs for I/Os and CPU times become too huge. Therefore, we need to use an efficient cache scheme to save such server costs. To present our idea about that, we use Figure 3 that illustrates the procedure for processing a user query. In the figure, steps 1 and 2 are executed by a web server that has accepted the user query. During those steps, the web server first parses the accepted user query to extract keywords from it. The parsed keywords are dispatched towards a cluster of ranker servers via TCP packets. Correspondingly, each ranker server performs its join operations and rank evaluations consecutively. From this ranking operation, the ranker server can get a set of DIDs that have been sorted according to their query relevancies. Then, the resultant DIDs are sent to the coordinator server and the coordinator server merges those DIDs sent by the ranker server cluster in step 3.

In the steps 5 to 6, then, the coordinator server sends DIDs and keywords of the user query to a cluster of summary servers for summary generation. The corresponding summary server generates appropriate excerpts for each DID through string matching of the keywords over DIDs’ plain texts. That is, string matching is applied to locate the keywords’ occurrences in the web document. Then, these keyword-containing excerpts are back sent to the coordinator server for the use of highlighting in resultant pages, along with URLs of the DIDs. Such summary generation is performed during the steps 7 and 8 in Figure 3.
3. Scheme for Accelerating Query Processing

3.1. Basic Ideas

If all the steps in Figure 3 are repetitively performed for every user query from scratch, then the query processing could be prohibitively costly because of its too many disk reads and long CU times. Against that, we employ a distributed caching scheme to use the previous processing results of repeated user queries. That is, we manage three categories of cache data. On the side of the web server, its cache storage saves the top five resultant pages of a cached user query. The cache data of the web server are compressed and managed in main memory for fast accesses. Since almost all users’ clicks on the query results arise within the first to the third page, we just save the top five pages out of the whole resultant pages. Owing to the web server cache, a user query can be answered without any disk accesses, it hit on the cache. Since the web server saves cache data in memory, only a small set of queries can be cached in the web server.

Besides the web server cache, our WSE has other types of cache data, that is, the summary cache and the DID cache. The summary cache is for saving the resultant data that have been obtained through the steps of ranking operations and summary generations. This type of cache saves the list of pairs of (DID d, summary S_d for d), where the pairs of (d, S_d) are sorted according to the query relevance of d’s. Using this summary cache, we can omit both of the ranking operations and summary generations during query processing. That is, if a user query is hit on the summary data, our WSE can make the query result of the user query without any expensive operations such as ranking operations and summary generating operations.

The summary cache data are created in the coordinator servers. As mentioned before, the coordinator server is responsible for receiving a parsed user query from a web server and dispatching the user query to a cluster of ranker servers via TCP packets. Then, the coordinator server collects the results of ranking operations from the ranker cluster and saves the results in its summary cache on disk. When a user query arrives at the coordinator server, the coordinator server looks up the summary cache. If the cached data is found for that query, this query is answered using the cache data. Therefore, operations for joining and summary generation can be omitted using this cache.

As the third cache data, the coordinator server stores DIDs that are gathered from ranker servers. The DID cache is used to save the results of previous ranking operation performed in ranker servers. The DIDs in this cache have been sorted in the order of query relevance. In general, since the data size of the DID cache is much less than that of the summary cache, we can cache a larger number of user queries in the DID cache than in the summary cache. Using the DID cache, the coordinator server can send DIDs to summary servers without requesting ranking operations. From this, we can remove expensive disk reads required for joins and rank evaluations.

3.2. Algorithm for Cache Management

For exposition convenience, we refer to the web server cache, the summary cache, and the DID cache as WSC, SC, and DC, respectively, from now on. As before mentioned, the WSC is managed in main memory and others are manipulated on hard disk drives. As a basic algorithm for managing cache entries in those caches, we employ the LRU-styled algorithm [8, 9]. For the fast search of matched cache entries, their cached entries are accessed through hash tables. In the hash tables, a hash key is created with the combination of query’s keywords and the requested rank range of query results.

In the case of the SC, we have two different types of data in order to maintain the cache hit rate of SC at a high and stable level. To pick out hot queries in advance, our system scans the log data and counts repetitive user queries. From this log analysis, our WSE creates SC data of the hot queries in advance and saves them in a static manner. We denote such static SC data by \( SC_{hot} \). The other kind of the SC is referred to as \( SC_{rep} \), and the entries in \( SC_{rep} \) are dynamically managed based on an LRU-styled algorithm.
Figure 4 presents the algorithm for the cache management of WSC, SC_hot, SC_rep, and DC. At step 1, the user query is parsed for the search in the cache storages. If no entry for the query is found, the parsed query is sent to a coordinator server. Then, the ranking operations for the query are performed during steps 4 to 8, and the results of the ranking operations are gathered by the coordinator server. Using the sorted ranking results at step 9, the coordinator server divides the gathered DIDs evenly and sends each of them to a set of summary servers for parallel processing. After the query processing, the resultant data are properly stored across four kinds of cache storages.

4. Performance Analysis

We now analyze the performance gains obtained from the use of our distributed cache scheme. To this end, we introduce a simple probabilistic model of cache hit rates. Let \( p_1, p_2, p_3, p_4 \) be the cache hit rates of WSC, SC_hot, SC_rep, and DC, respectively. Therefore, the overall hit rate, i.e., \( P_{\text{hit}} \), can be calculated as follows:

\[
P_{\text{hit}} = p_1 + (1 - p_1) \cdot p_2 + (1 - p_1) \cdot (1 - p_2) \cdot p_3 + (1 - p_1) \cdot (1 - p_2) \cdot (1 - p_3) \cdot p_4
\]

Since the hit rates of multi-level caches can vary with the characteristic of the user query sets to be processed, the cache hit statistics in our WSE may not be the case to all of other search engines. However, we believe that our experience can give an insight to anyone who wants to run a similar type of the search engine. In our WSE, the values of \( p_1, p_2, p_3, \) and \( p_4 \) are the same as 0.62, 0.19, 0.24, and 0.13, respectively. From those values, we have \( P_{\text{hit}} = 0.79 \). That is, around 80% of user queries can be answered by using the cache data without expensive disk reads and CPU time consumption.

If a user query can be answered from the cache data of WSC, the operation for generating the resultant page is very simple. That is, the involved web server only has to uncompress the memory cached data after searching on the cache. For the cache memory of WSC, we allocate 75% of main memory exclusively out of the whole memory. Using WSC, we can cache about 20,000 search result pages for a web server equipped with 2 GB memory. The cache hits on WSC has the largest benefits, compared with cache hits on other types of cache storages.

In the cases of SC_hot, SC_rep, and DC, although they need disk bandwidth for cache reads, their cost for query processing is less than 3% of the cost that should be paid for query
processing without caches. From our experiments, we can say that our distributed cache scheme improves the throughput of the implemented system by about 300%. Due to such performance enhancement, we can reduce 70% of the sever cost of the query processing system in our WSE.

Before the use of our cache scheme, we predicted that the hit rate of $SC$ will be much larger than that of $WSC$, because of the large amount of cached queries in $SC$ (i.e., $SC_{hot}$ and $SC_{rep}$). However, the hit rate of $WSC$ is larger than that of $SC$ in reality, differently from our expectation. From the analysis on the query logged in later time, we found that such a higher hit rate of $WSC$ mainly comes from a significant number of repetitive user queries. That is, a small set of queries has a great portion among the whole user queries because of its high popularity.

Since the most of queries cached in $WSC$ are saved in $SC$ as well, however, it is the case that the potential hit rate of $SC$ is much better than that of $WSC$. In particular, since the cache records of $SC_{hot}$ can be baked in advance by scanning the earlier user queries logged, we can dump the data of $SC_{hot}$ into the query processing system of the WSE before we copy a new set of index data into the system at the data refresh time. With the data dumping of $SC_{hot}$ on the cache storage, we can easily preclude an undesirable situation where system overloads arise at the time of data refreshing. Since the popularity of the user queries of $SC_{hot}$ retains during a rather long time, we do not manage the data entries of $SC_{hot}$ in a dynamic manner.

5. Conclusion

When we service ad-hoc user queries on a large volume of web documents, the search technique of keyword matching is usually employed. That is, the search engine picks out a collection of web documents where the keywords of the given query are all found. For the keyword matching-based searches, the inverted file is usually created due to its advantages in both the ranking operation and the join operation. Since the size of inverted files easily goes so big with an increasing number of indexed documents, however, the cost for query processing could become prohibitive in the case of web searches.

To solve such a problem, we have designed a distributed cache scheme suitable to a large scale of web search engines. In our cache scheme, the resultant data from query processing are stored across different levels of caches according to their expected hit rates. With the cache scheme, we can reduce around 70% of the server cost. We believe that those who have an interest in implementing a large scale of full-text search engines can also use our technique for saving their server costs.

References
