Original-page small file oriented EXT3 file storage system

Zhang Weizhe , Hui He, Zhang Qizhen

School of Computer Science and Technology, Harbin Institute of Technology, Harbin
E-mail: wzzhang@hit.edu.cn

Abstract. This paper analyses the disadvantages of the existing EXT3 file system in accessing small files, and designed an original-page oriented large file organization structure and large file related read-write query tree, based on the small, many and no modifications after being written characters of small files.

Keywords: search engine, small file storage, storage time, storage space

1 Introduction

The accessing speed and utilization ratio of storage space of search engine are two essential performance indicators. If accessing speed is too low, storage will be the bottleneck of search engine performance, while the crawling speed of crawlers will be limited because crawlers are filled with the obtained pages, if storage speed is too low. If read speed is too low, it will affect the speed of analysis of search engine, while the cost of data storage will sharply increase if utilization ratio is too low, and resource will be wasted.

Contents of search engine storage comprise by original pages, content pages and indexes. In these, original pages own the largest data volume, while the number of content pages and indexes is much smaller. The data volumes of content pages and indexes are more or less the same. The proportion of these three data volumes is almost 100:1:1. Original pages are the WEB pages crawled by web crawlers, whose sizes range from several KB to several hundred KB, dozens of KB normally. Content pages are pages that are extracted by original pages, the sizes of which are smaller than original pages, half of them generally. The establishment, update, storage and locating of indexes are in the charge of third-party software, and storage systems only need to provide the storage directory to these third-party software. Thus, original-page small files are the main objects for storage nodes to handle.

Great defects exist in the existing file system accessing the original-page small files. In this paper, we do the compare of common compressing algorithms firstly, and choose a proper compressing algorithm to compress and store the page data. Then, we design a large file storage format for original pages, at the same time, we design a large file storage related snapshot query tree to optimize the speed of reading

snapshots. These measures reduce the storage nodes' accessing response time and the usage of disk space.

2 The strategy on storing small file of EXT3 file system

In this paper, based on the ext3 file system in linux operating system, and drawing lessons from the log-structured file system thoughts, we cache a large number write-operation of original pages to the storage nodes memory, and organize the original pages to a large file in the cache. Then the large file is written to disk, greatly reducing the disk seek and data modification operation, at the same time, reducing the disk fragments and the disk metadata occupancy

2.1 File format designing

In this paper, we design a compact large file format consists of small files, named LOG_COMPACT file. The large file is divided into three sections. As shown in figure 1, the first part is the file header, recording the whole file information, such as the number of WEB documents and large file bytes and some other information. The second part is the WEB document information unit array, used to quickly locate the WEB document position. Each unit records the WEB document URL (if URL is too long, then storing the hash value of the URL) and the offset in the large file and its occupants bytes length. The third part is the WEB document array, in which each element is a WEB page and can be located rapidly through the second part. Compressing every WEB document to save the storage space, this storage method can save the metadata storage as much as possible, and also can quickly locate the content of the large file. Because the size of WEB information unit array is generally dozens of KB, when reading file, the WEB information unit array of large files is first read. Through the array, it can find the offset of the given URL and the length of data, and then read out the file data.

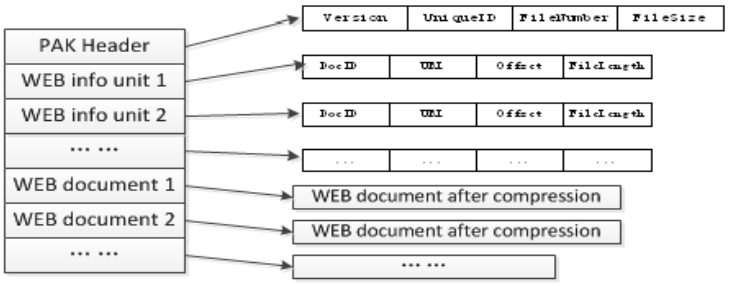

Fig. 1. The files layout inner LOG_COMPACT file.

2.2 Large file read-write designing

When the storage node receives a snapshot fetch request, it gets some URL and date, and it needs some mechanism to quickly locate the URL and date to the position in large files where there is the original page, and then reads the page snapshot from the large file.

In this article, we map the URL and date and the path of the large file to a database table. The table has three rows, and respectively, they are URL, date and the path, of which URL and date together make up the unique identity of each column. When the pages are written, add lines to the database. When reading the file, locate the large file through URL and date, and then, from the large file, read the page. Because of the huge amount of data, with the increase in the number of stored pages, the table will become quite large, so the table is often added, which makes query speed very slow. Therefore, the method is not suitable for page retrieval.

Referencing the TRIE tree, we design snapshot retrieval mechanism. Corresponding relationship of URL, date and file path is designed into a tree. The date is the first division level, and then in accordance with the URL natural path, stretch to the bottom of the tree. Leaf nodes store the path, represented by the date and URL corresponding to the original page where the large file path exists. In this way, when querying a snapshot, just need to find the leaf node along the query tree through data and URL. The path, where the leaf node stores, is the path of the large file. Then the page can be read from the path. Huge number of pages, the tree can't be completely stored in memory. To solve this, the upper layers can be stored in memory, and the content of under layers can be stored on hard disk. When read the path of under layers, the needed sub-tree is read into memory. Then operate the sub-tree in memory.

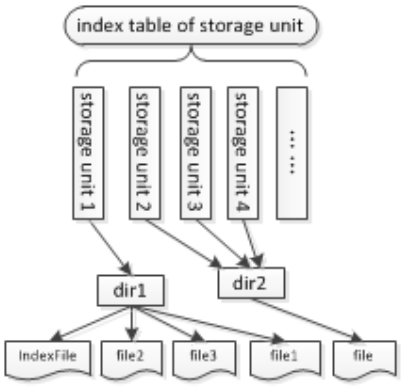

Fig. 2. The storage structure of raw pages in store machine.

Although query tree is able to quickly finish modification and query, but operations are complicated and the number of layers is varying in depth. These make the query tree very uneven. The seeking and modification are remarkably different between files. In order to reduce the complexity of the operation, at the same time, be able to quickly query page snapshot, we conduct necessary simplification on the

query tree in this paper, leaving only two layers. The first layer is the storage unit, on which the information is always kept in memory. The second layer is index table of storage unit, whose information is stored in the hard disk. In the last chapter, we chose the day channel as the load distribution unit, by which day channel in the storage node is as the unit of storage division. Because the data amount of day channel is moderate, and the number of day channel is small, the storage unit layer in query tree can always kept in memory. When query Information arriving, through URL and date, the storage unit can be found by query tree. And through storage unit, the index table of this storage unit can be acquired. And through the index table, the large file where the page is stored can be quickly located, and then the page can be extracted from large file.

3 Experiment

To compare the accessing efficiency and storage volume of different storage schemes for original pages, we design and realize five storage schemes for original pages.

1) Compressing document by document storage, short for Raw_Store. Compressing the pages firstly when writing pages, and storing the pages, with the same day channel, in the same directory. When do snapshot query, day channel directory is firstly located. Then, querying the page files in this directory. Finally, the pages are decompressed and returned. Because the EXT3 file system uses hash locating to handle with large directories, so the file search in some directory can be very fast. This scheme is simple to realize, but there will be a number of small files existing in system.

2) Archiving compressing storage, short for AC_Store. Archiving original pages in memory and assembling into a large file when writing pages, and then, compressing the large file and storing it in disk. When do snapshot query, the large file is found through the query tree firstly, and the large file is read to memory and decompressed. Then, the archiving item wanted by the query is read finally. This scheme remarkably decreases the small files in system storage, but the whole large file needs to be read to the memory.

3) Compressing archiving storage, short for CA_Store. Compressing original pages when writing pages, and then, archiving the compressed page files to a large file and storing to the disk. When do snapshot query, the large file is found through the query tree. Then, the data is read to memory by archiving item and decompressed until the page files wanted by the query are obtained. This scheme just reverses the sequence of compressing and archiving in scheme 2, but the average data amount of reading and decompressing is lesser that scheme 2 when querying pages.

4) LOG_COMPACT large file storage, short for Log_Store. Compressing the pages in memory when writing pages, and then, the compressed files are organized as LOG_COMPACT file and written to disk. When do snapshot query, the large file is found through the query tree. Then, reading LOG_COMPACT header and WEB information unity array, and getting the offset and length of wanted page in the large file. Next, reading and decompressing the page. This scheme doesn't need to operate the whole file when extract the snapshot, only little information of the large file

needed, thus the speed of snapshot is higher.

5) Content duplication storage, short for CI_Store. Conducting repeatability test first when writing pages. If the content of the page is the first appearance in system, then compress and store it. Locating page file through inquiring the query tree, and decompressing the page file after finding it. Due to maintaining the reference count of content and being beneficial to deleting when expired, this scheme conducting the storage document by document.

We assume in experiment that there's no limitation for the receiving speed of network card, to test the maximum of read-write speed. Reading the previously crawled original pages dataset to memory, and then, sending the original pages to storage module through memory. The size of experimental dataset is 832M, and number of pages is 26646, and average page size is 31.2KB. Because the most operation on original pages in storage system is write-operation, while snapshot extraction operation is relatively less, we set the ratio of frequency of two operations as 100:1 which means that there about one snapshot extraction operation during 100 times write-operations. The time of writing pages and snapshot extraction makes up the total time of accessing time of a storage node. The read-operation and write-operation on the same storage node ought to be synchronized, so we choose the total accessing time as the standard to measure the accessing efficiency of a storage node. Compressing data is exchanging the accessing speed for less disk space occupation, so we compared the accessing efficiency and disk occupation of different methods of lossless compression and loss compression in this paper before, and we finally chose gzip as our proper compression algorithm.

From the method (2), (3), (4) in Table 6, we know that it can remarkably reduce writing file time if the small file buffers are organized to large file. But the speeds of method (2) and method (3) are very low that because they have to read the most content of archiving large files to find the wanted small page file. The read speed of method (4) is lower than method (1), because it has to read much more data amount from large files than method 1. Write-operation in method (5) needs to conduct duplication test, so the write cost is high, while the read operation is the same as method (1), thus the speed is high.

Table 1. The non-compressed file read-write velocity and disk occupied.

No	storage method	volume(MB)	write(ms)	read(ms)	read+write(ms)
1	Raw_Store	945	30847	290	31137
2	AC_Store	845	5160	13800	18960
3	CA_Store	845	5155	13650	18805
4	Log_Store	833	5087	772	5859
5	CI_Store	518	75525	314	75839

6 Conclusion

To improve the accessing efficiency of storage node to improve storage throughput and reduce storage disk space occupation, so as to reduce system deployment cost. In this paper, we contrasted the compression ratio and compression speed of common compression algorithms to web data firstly. Then, we analyzed the problems in accessing small files of EXT3 file system and designed the LOG_COMPACT large file format based on the characters of original pages which are much write and little read and almost no modifications after written, and designed the accessing process of it. Then, we conducted an experiment on different accessing methods supplemented by compression algorithm, and the result of the experiment showed that LOG_COMPACT related storage methods performed best in the comprehensive evaluation of accessing efficiency and disk space occupation.

References

1. RFC1952. GZIP file format specification version 4.3. http://www.ietf.org/rfc/rfc1951.txt
2. RFC1950.ZLIB Compressed Data Format Specification version 3.3. http://www.ietf.org/rfc/rfc1950.txt
3. T.A.Welch. A Technique for High-Performance Data Compression. Computer In Computer. 1984,17(6):8-19
4. Ziv J/Lempel A. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory,1977,23(3):337-343
5. S.C.Tweedie. Journaling the Linux ext2fs Filesystem. Proceedings of the 4th Annual LinuxExpo, Durham, NC. 2007,10(4):42-50
6. Namesys web site. http://www.namesys.com/
6. JFS for linux project website. http://jfs.sourceforge.net/
   The SGI XFS project website. http://oss.sgi.com/projects/xfs/