An Efficient and Metadata-Aware Big Data Storage Architecture

Rize Jin[1(✉)], Joon-Young Paik[1], and Yenewondim Biadgie[2]

[1] School of Computer Science and Technology, Tiangong University, Tianjin 300160, China
jinrize@tiangong.edu.cn
[2] Department of Software and Computer Engineering, Ajou University,
Suwon 16499, Republic of Korea

Abstract. This paper introduces a hash partitioning-based file compaction design to improve the efficiency of storing and accessing small files in big data storage systems. The proposed approach consists of a file compaction tool and an access interface. The compaction tool merges a group (usually a directory) of small files into a set of "big files" to reduce the metadata required to be maintained in the on-chip memory. The data locality and tree structure of those small files are preserved. The access interface is designed to provide transparent access to the small files in the big files. Experimental results confirm that the proposed approach lead to a significantly enhancement in terms of namespace usage and access speed.

Keywords: Big data · Small file · File compaction · Metadata management

1 Introduction

A file is called small [1] when its size is substantially less than the block size of big data platforms [2, 3] which typically is 128 KB. Files and blocks are metadata objects and occupy namespace (mapping space) of the on-chip memory. When there are numerous small files stored in the system, the massive metadata information can occupy a large memory space. The storage space may be underutilized because of the namespace limitation. For example, 100 million files with an average size of 3 K can consume 70% of the namespace capacity of 256 MB RAM, yet occupy only 1% of a 4 TB storage. Moreover, massive small files generate small and random writes, which incurs an increase of write amplification [4]. This paper proposes a file compaction method as an effective solution to the problem of managing massive amounts of small files in big data storage systems. The proposed approach increases the scalability of the system by reducing the namespaces usage and decreasing the operation load in the on-chip memory by distributing namespace management.

Our primary contribution is to explore empirically a hash partitioning-based file compaction format to organize a group of small files into a compact file consisting of multiple larger files. The file compaction reduces the amount of file metadata residing in RAM. The compact file retains data locality and data intact by maintaining a local index and the metadata information of the original small files. Our second contribution

Y. Nah et al. (Eds.): DASFAA 2020 Workshops, LNCS 12115, pp. 146–152, 2020.
https://doi.org/10.1007/978-3-030-59413-8_12

is to provide a set of principles to implement a multi-thread merge tool to improve the efficiency of creating "big files" from many small files. Each big file is a self-manageable key-value document.

The remainder of this paper is organized as follow. Section 2 defines the problem of managing massive amounts of small files in big data storage. Section 3 presents the design details of the proposed solution. Section 4 reports the performance evaluation. Section 4.1 discusses related work and Sect. 5 concludes this paper.

2 Problem Definition

Several studies [5–7] indicate that small and random writes are slower than sequential writes. Small files tend to incur many small and random writes. We call a write small when the request size is equal to or less than the block size (i.e., <128 KB). The controller performs additional work to maintain the metadata necessary for mapping small writes. Further, an enormous number of small files requires significantly more namespace to maintain the extremely large amount of metadata and this is clearly not efficient as the big data storage systems have a relatively limited on-chip memory space [8, 9]. Read performance is a consequence of the write pattern. When writing a large chunk of data, it is likely to be stored in locations that are contiguous in the physical space. However, small files whose addresses are contiguous in the logical space may refer to addresses that are not contiguous in the underlying storage system to the dynamic mapping performed by the buffer management.

The on-chip memory is relatively a limited resource. The OS maintains namespace information for each file buffered in the on-chip memory. When the number of files becomes extremely large, the usage of the memory increases sharply. For example, assume that there are 100 million small files and the metadata of each file occupies 100 bytes of memory space; then, they would consume 1 GB memory space. As user data increases, it is not difficult to exceed the namespace limit. The actual namespace stores considerably more information including the metadata of directories and blocks.

3 The Proposed Method

3.1 Design Considerations

Considering the abovementioned issues, the proposed file compaction approach includes five design principles: 1) **Buffer hot files**. Files that change frequently are considered as hot [10]. Hot files and their metadata should be buffered in the on-chip memory as much as possible and written to the underlying storage infrequently. To address this, the proposed method uses the least recently used (LRU) buffer replacement policy. **2) Collect small writes**. To maximize throughput and minimize write amplification, whenever possible, small writes should be maintained in the buffer in the on-chip memory and only written once when the buffer is full. **3) Clean-first buffer management** [11]. The buffer manager should consider not only the buffer hit ratio but also the heavy write cost of the logging mechanism employed by the majority of big data systems. **4) In-block update**. It uses an in-block update (IBU) strategy that divides one physical block into

data blocks and log blocks to maintain the updates of small files both separately from and locally to the original files. **5) Prefetching** [12]. It is a widely used storage optimization technique to reduce buffer misses by exploiting access patterns and fetching data into a buffer before they are requested. We employ a two-level prefetcher, which consists of local index file prefetching and correlation-based file prefetching.

3.2 Compaction File Format

Figure 1 illustrates a method to locate a small file in the proposed method. The small file is stored in a compact file that consists of multiple big files. Small files within big files are indexed by a hash index to maintain the original separation of data. In detail, small files are stored as a set of <key, value> pairs, the file name (or the path of the file in a host system) is the key and the content is the value. A metadata file records the original directory tree structure.

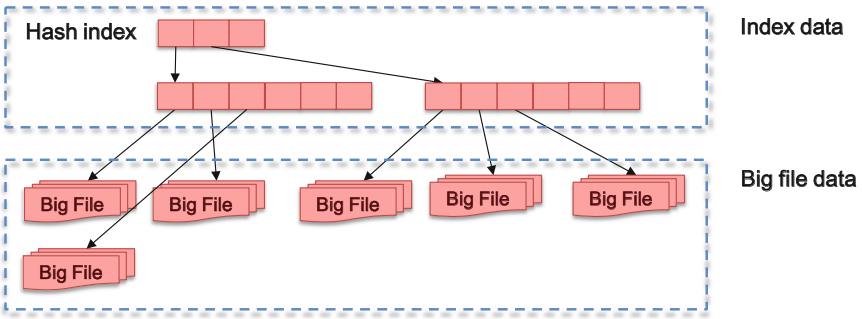

Fig. 1. Locate small files using hash index.

3.3 Transparent Access

The proposed method is designed to fit into the underlying file system and hence, can provide transparent access to the upper applications. When accessing a directory, the user can expose small files and the directory tree structure. Individual small files can also be accessed directly without being extracted from the big file. Accessing a file is slower than a usual access. We first must locate the index of the small file and then read the data from the big file. The application or host uses the absolute path of the small file as the identifier (*sID*) when accessing. The hash function calculates a *bID* value, which is the identifier of its corresponding big file, from *sID* and the system can then locate the local index file by finding a record that associates with *bID* in the namespace. Figure 2 presents the process.

3.4 File Compaction Tool

To create a big file efficiently, this paper implements a file compaction tool in a multi-thread fashion: a list of small files is generated by traversing the source directories

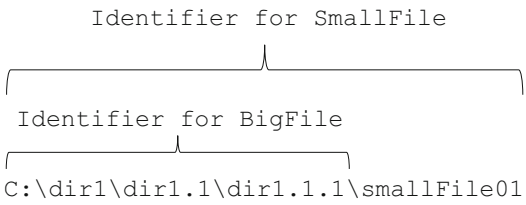

Fig. 2. Resolve *sID* to *bID*.

recursively and then the list is divided into several ranges using a hash function. The small files within each range are stored into a big file (approximately 8 MB, configurable). Finally, index and metadata files are generated. In detail, each big file consists of a metadata file and a local index file, which records the offset and length for each original small file it includes, and a set of <key, value> pairs, where key is the file name and value is the content of that file. The reason the index and the metadata file are maintained within the big file is that we want these data stored physically close together to facilitate prefetching.

Compared to maintaining numerous metadata objects as a global index in the namespace, the proposed solution does not result in additional overhead to the on-chip memory. If the sum of the small files to be compacted exceeds the predefined size for a big file, the list of small files is divided into multiple big files. In Fig. 3, a directory containing many small files is combined as a directory with several big files and indexes.

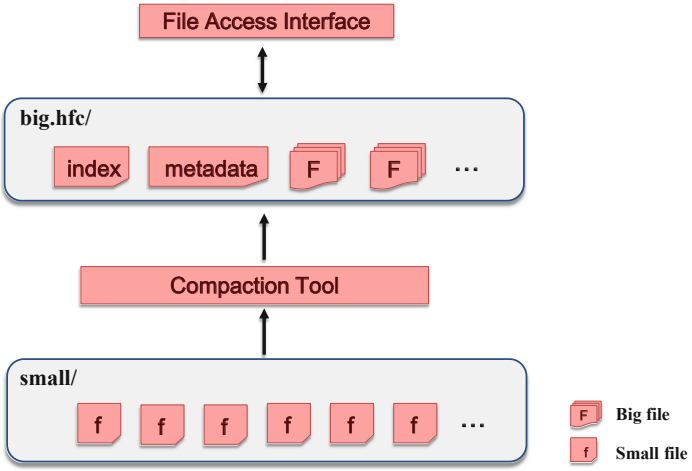

Fig. 3. Compact small files into a big file.

As mentioned earlier, the big data storage functions well for a sequential access pattern. However, small files usually do not provide sequential accessing, even for a batch processing requested to them. The reason is that they are likely to lose data locality in the presence of out-of-place updates. Consequently, the host cannot provide a prefetching

function. File compaction ensures that related small files are stored sequentially and in-block updates preserve the maximum possible data locality.

4 Performance Evaluation

4.1 Experiment Settings

The test platform was built on a Core i5–2500 machine (3.30 GHz, 16 GB DDR3 RAM) running Ubuntu 16.04 with Hadoop version 2.7.

Figure 4 compares the memory usage of the proposed method (HFC) and the original approach when the system stored multiple file sets. We varied the number of small files, *#ofSmallFiles*, along with the sizes of the big files, *sizeOfBigFile*. As expected, the proposed approach achieved considerably improved efficiency storing small files compared to the original method, increasing approximately 510 times for *sizeOfBigFile* = 1.75 MB and approximately 3,800 times for *sizeOfBigFile* = 14 MB. The proposed approach required less namespace because of its local index file design.

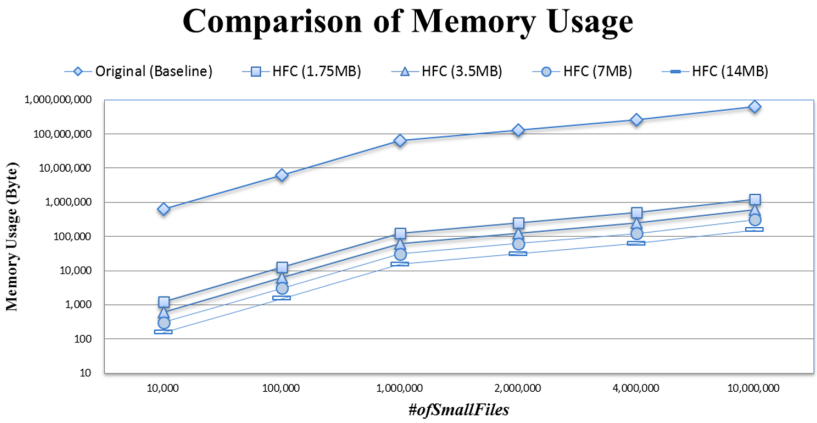

Fig. 4. Comparison of memory usage varying the number of small files and the size of big files.

4.2 Varying Read/Write Ratio

HFC was evaluated by varying the read/write ratio, *RWR*, of a workload made of a mix of interleaved reads and writes (the update ratio: 50%). The test assumed a namespace of 32 MB and buffer size of 64 MB. At the beginning of this test, the system was populated with two million small files. The performance decreased with an increase in the write ratio. The performance degradation of HFC was relatively less than that of the other approach, as indicated in Fig. 5. The reason is that the in-block update of HFC reserves the data locality of the original file and its logs and therefore, HFC performs fewer disk writes. Conversely, the existing approach can suffer performance degradation owing to the increased garbage collection ratio. In particular, the execution time of the original approach increased 100 times as the *RWR* increased from 100/0 to 0/100; HFC had an increase of 36%.

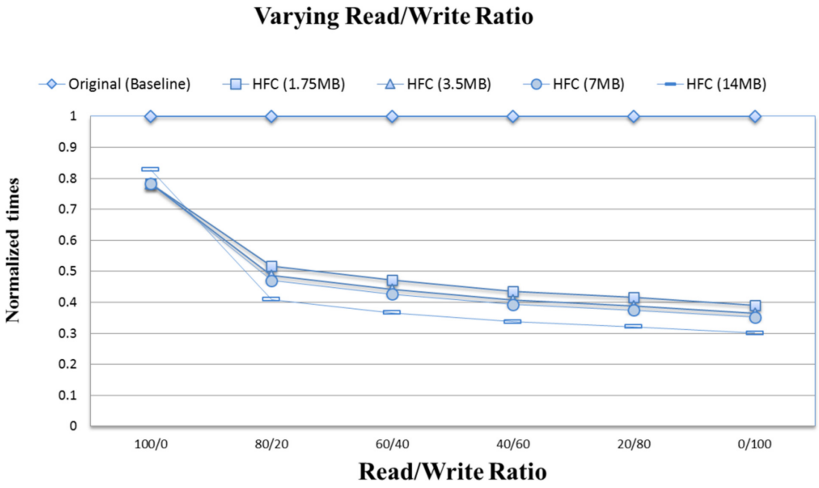

Fig. 5. Comparison of execution times by varying read/write ratio.

5 Conclusions

The increasing capacity of big data storage systems has resulted in significant overhead for managing the in-RAM metadata of massive amounts of small files. To ensure the scalability and efficiency of accessing these small files, this paper proposed a hash partitioning-based file compaction: a novel file compaction interface that provides transparent access to individual small files within a large file using a local index design. In detail, massive amounts of small files are compacted into a small number of big files; each big file consists of a local index file that records the offset and length for each original small file included and a set of <key, value> pairs, where key is the file name and value is the content of that file. Compared to maintaining many metadata objects as a global index in the namespace, the proposed solution reduces significantly the lookup and access overhead to big data storage systems.

Acknowledgments. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61806142, in part by the Natural Science Foundation of Tianjin under Grant 18JCYBJC44000.

References

1. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. ACM SIGARCH Comput. Architect. News **18**, 364–373 (1990)
2. McAfee, A., Brynjolfsson, E., Davenport, T.H., Patil, D.J., Barton, D.: Big data: the management revolution. Harvard Bus. Rev. **90**(10), 60–68 (2012)
3. Labrinidis, A., Jagadish, H.V.: Challenges and opportunities with big data. Proc. VLDB Endowment **5**(12), 2032–2033 (2012)

4. Hu, X.Y., Eleftheriou, E., Haas, R., Iliadis, I., Pletka, R.: Write amplification analysis in flash-based solid state drives. In: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, p. 10 (2009)
5. Lee, S.W., Moon, B.: Design of flash-based DBMS: an in-page logging approach. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 55–66 (2007). https://doi.org/10.1145/1247480.1247488
6. Chen, F., Koufaty, D.A., Zhang, X.: Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: Proceedings of ACM SIGMETRICS Performance Evaluation Review, pp. 181–192 (2009). https://doi.org/10.1145/2492101.1555371
7. Kang, J.U., Jo, H., Kim, J.S., Lee, J.: A superblock-based flash translation layer for NAND flash memory. In: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software, pp. 161–170 (2006). https://doi.org/10.1145/1176887.1176911
8. Bende, S., Shedge, R.: Dealing with small files problem in hadoop distributed file system. Procedia Comput. Sci. **79**, 1001–1012 (2016)
9. ElKafrawy, P.M., Sauber, A.M., Hafez, M.M.: HDFSX: big data distributed file system with small files support. In: 2016 12th International Computer Engineering Conference (ICENCO), pp. 131–135. IEEE, 28 December 2016
10. Park, D., Du, D.H.: Hot data identification for flash-based storage systems using multiple bloom filters. In: 2011 IEEE 27th Symposium on Proceedings of Mass Storage Systems and Technologies (MSST), pp. 1–11 (2011). https://doi.org/10.1109/msst.2011.5937216
11. Jin, R., Cho, H.J., Chung, T.S.: LS-LRU: a lazy-split LRU buffer replacement policy for flash-based B+-tree index. J. Inf. Sci. Eng. **31**(3), 1113–1132 (2015)
12. Joseph, D., Grunwald, D.: Prefetching using Markov predictors. In: Proceedings of ACM SIGARCH Computer Architecture News, pp. 252–263 (1997). https://doi.org/10.1109/12.752653