International Journal of Computer Applications Technology and Research
Volume 7–Issue 08, 327-330, 2018, ISSN:-2319–8656

A Strategy for Improving the Performance of Small Files in Openstack Swift

Xiaoli Zhang
School of Communication Engineering,
Chengdu University of Information Technology
Chengdu, China

Chengyu Wen
School of Communication Engineering,
Chengdu University of Information Technology
Chengdu, China

Zizhen Yuan
School of Communication Engineering,
Chengdu University of Information Technology
Chengdu, China

Abstract: This is an effective way to improve the storage access performance of small files in Openstack Swift by adding an aggregate storage module. Because Swift will lead to too much disk operation when querying metadata, the transfer performance of plenty of small files is low. In this paper, we propose an aggregated storage strategy (ASS), and implement it in Swift. ASS comprises two parts which include merge storage and index storage. At the first stage, ASS arranges the write request queue in chronological order, and then stores objects in volumes. These volumes are large files that are stored in Swift actually. During the short encounter time, the object-to-volume mapping information is stored in Key-Value store at the second stage. The experimental results show that the ASS can effectively improve Swift's small file transfer performance.

Keywords: Openstack; Swift object storage; high performance; small files; aggregated storage strategy.

1. INTRODUCTION

The popularity of the World Wide Web is largely responsible for the dramatic increase in Internet data during the past few years. Usually, social media, e-commerce, scientific experiments and other related fields will produce small files by the tens of millions every day. Global data volume is about double every two years, and will increase to 40ZB by 2020, according to IDC, a market-research firm [1][2]. It is worth noting that the largest proportion and fastest growing are small files. Typically, "a small file" refers to a file less than 1MB in size. The size of the small file ranges from a few KB to tens of KB [3-4]. Texts, pictures, and mails are often small files. The public climate system stores 450,000 climate model files. Their average size is 61 bytes [5]. Sharing photos is one of Facebook’s most popular feature. Users have uploaded over 65 billion photos by 2010 [6]. As the largest personal e-commerce website in the world, TAOBAO stores over 20 billion images, whose average size is only 15KB [7]. How to store and access large numbers of small files efficiently over time makes a new challenge to the storage architecture of the “big data era”.

Storing many small files requires a high performance, high availability, high scalability, security and manageable storage system. But although traditional RAID technology has high performance, it is not suitable for today's Internet environment due to its high cost [8]. NAS and SAN are also not suitable for storing large amounts of data because of their limited scalability [9]. The famous GFS (Global File System) consists of inexpensive PC servers and provides fault tolerance [10]. However, when the system stores small files, as the number of stored files grows rapidly, plenty of metadatas are generated on the metadata server. This results in poor file access performance. Facebook independently developed Haystack as its image storage dedicated storage system [11]. Nevertheless, it is limited in scalability because it refers to the central node design of GFS. To solve this problem, Amazon developed Dynamo storage system [12]. It adopts the method of no center node and relies on the hash algorithm to solve the file distribution problem. Similarly, Cassandra [13] and TAIR [14] are non-centralized storage system. Unfortunately, they are designed for the storage of large files and do not optimize the transfer performance of small files. In this paper, we propose the ASS for improving the transfer performance of a large number of small files in Swift. ASS has two parts. In the first stage, the ASS arranges he written objects one by one, and then merges them into large files in chronological order. Those large files are called “volumes”, which are actually stored in Swift. In the second stage, the object-to-volume mapping information (volume id, location) is stored in the key-value store.

The remainders of the paper are organized as follows: Section 2 discusses related works on improving the transfer performance of small files. In Section 3, we described the basic principles of ASS. At the same time, the ASS read algorithm and small file read/write process are introduced. At Section 4, we introduced the experimental environment and analyzed the experimental results. Section 5 concludes the paper.

2. RELATED WORKS

Many people have tried various schemes to improve the small file storage access performance. The index layout strategy can achieve efficient reading of small files by optimizing the physical layout of directory entries, inodes, and data blocks. For example, to reduce the number of IO, C-FFS [15] embeds the inodes in the directory entry and replaces the inodes pointer of the directory entry with inodes. But this strategy has the disadvantage of synchronous recovery operations in a distributed environment. The Cache structure optimization strategy reduces the access time of the storage node by using the external cache CDN and the internal cache, which effectively improves the cache hit ratio. For instance, for efficient access, the Sprite file system uses a stand-alone Cache, and each server node has its own cache space [16]. Lustre leverages the distributed cache space of each client. It uses a

collaborative caching strategy that reduces the load on a single server cache [17]. This approach improves file access efficiency. However, the multi-level cache is only effective for hotspot data accessed in the most recent period of time. Due to the small number of hotspot data, it will cause a lot of non-hotspot data access inefficiency.

At present, the combined storage solution is also widely used in the industry. Its main idea is to reduce the amount of metadatas in the metadata server. And it can improve the read/write efficiency of small files by consolidating small files into large data block storage. The consolidation of small files has many different implementations. For example, Hadoop uses its own merging file tool - HAR file archiving. The principle of HAR is to pack multiple small files into one file and then save it to a block. The archive mainly contains metadata and data files [18]. But the merging file tool that comes with the system is often to merge and archive the small files already stored in the system. This can lead to a lot of disk read and write consumption. In fact, it is also possible to merge files on the client before uploading the storage. However, the measure often stores index information locally. When a small file is requested, the system first transmits the entire data block to the client and then reads the offset. This method will result in a large number of invalid data network transmission bandwidth.

Compare with the strategies discussed above, our work differs in two ways: (1) This paper establishes a separate merge engine in Swift object storage. The merge engine combines small files into large files before storing them. It is worth noting that it applies to any small object, such as pictures, documents, etc. (2) For this merge engine, a method of merging files is proposed—ASS.

3. MERGE ENGINE

3.1 An aggregated storage strategy

Swift uses loopback devices and the VFS file system as the underlying storage. In this paper, based on the original Swift framework, a merge engine is added between the object server and the XFS file system. The merge engine uses an aggregate storage strategy. This strategy allows multiple logical files to share the same physical file. It reduces the number of files and metadatas, improves the efficiency of metadatas retrieval and query, and reduces I/O operation delays for file reads. And effectively solved Swift's small file storage problem. The keys to strategy are:

(1) Merge storage: The basic idea of the strategy is to store objects in a volume. Volumes are large files that are stored in Swift actually. This policy stores objects in a volume and separates volumes through Swift's virtual partition, which not only improves the transmission performance of small file, but also ensures Swift's data migration capabilities.
(2) Index storage: The object—volume mapping information (volume id, position) is stored in the key value store (KV server) for cluster maintenance.

The merge engine module includes an object request layer, an object merge layer, a logical map layer, and a physical map layer. When Swift's storage node receives a PUT or GET request from a proxy node, in the original case, Swift Ring uses a Consistent Hashing Algorithm to complete the “object-virtual node-device” mapping. In this paper, since a logical map layer is added, the “object-volume-virtual node-device” mapping is formed. The “volume-virtual node” mapping relationship is a logical mapping, and the “virtual node-device” mapping relationship is a physical mapping. The merge engine module uses ASS, which works as follows.

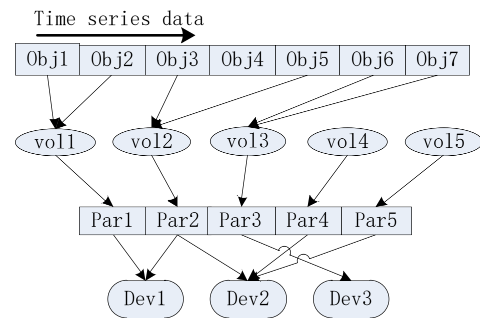

Figure 1. Basic theory of ASS.

In Figure 1, “obj” is the object, “vol” is the volume, and “Par” is Partition. As Figure 1 shows, ASS aggregates files according to the time characteristics of the objects. On the one hand, the solution translates random writes into sequential writes. It reduces the system's garbage collection overhead and data migration overhead. On the other hand, the solution merges and stores the data, which reducing the processing cost of metadatas. Both can effectively improve the transmission performance of small files in Openstack Swift.

3.2 The process of reading and writing files

In this paper, the improvement of Swift framework is embodied in the optimization of reading and writing. The flow of small file read/write operations is shown in the Figure 2.

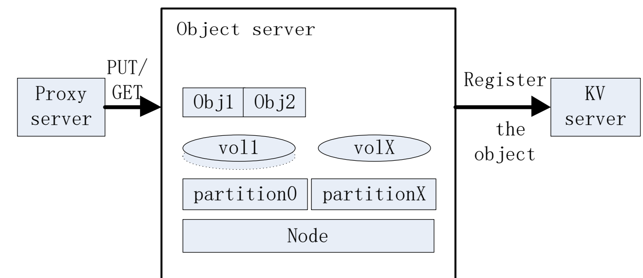

Figure 2. File read-write process.

Write: When the proxy server receives a PUT request from the client, it then forwards the PUT request to the storage nodes. Firstly, storage nodes look for an unlocked volume, or creates a new writable volume and associated lock file (if a new volume is created, it needs to be registered in the KV server). Secondly, storage nodes lock this volume. Storage nodes then appends object information (Object header、Object metadata、Object data) to the end of the volume, just like the shaded part of the figure. The next step is to synchronize the volumes. Finally, storage nodes register objects to the KV server, which is to add new entries to the key-value store.

Read: When the proxy server receives a GET request from the client, it then forwards the GET request to the storage nodes. Firstly, the storage node gets the (volume index、offset in the volume) information of the object from the KV server to locate the volume. The storage node then opens the volume files, gets the offsets, and locates the objects.The reading algorithm of the files is as follows:

Filereading(obj Name, obj Size=0)
1 Currentposition←fileposiiton(obj Name)
2 objheader←header(obj Name)
3 datasize←datasize(objheader)

4 datastartoffset←offset(obj Name)+ dataoffset(objheader)
5 dataendoffset←datasize+ datastartoffset
6 **if** Currentposition>= dataendoffset or obj Size=0
7 **then** call normal Read(c)
8 **if** obj Size is normal and obj Size>dataendoffset –
9 Currentposition
10 **then** Obj Size←dataendoffset- Currentposition
11 **else** Obj size←dataendoffset- Currentposition
12 data←read(filepositon， Obj size)
13 **return** data

4. EXPERIMENTAL ENVIRONMENT AND RESULTS

4.1 Experimental environment

To verify the effectiveness of the strategy, a small Swift cluster consisting of one proxy node and three storage nodes is built on the virtual machine. The deployment of each service is shown in Table 1.

Table 1. The deployment of each service in Swift cluster

Name	Operat-ing system	Hard-drive sizes	Memory Sizes	Major services
Contr-oller	Centos7	20GB	2GB	Swift client, keystone
Node1	Centos7	20GB	2GB	CARP, HAProxy, Swift storage
Node2	Centos7	20GB	2GB	CARP, HAProxy, Swift storage
Node3	Centos7	20GB	2GB	CARP, HAProxy, Swift storage

4.2 Experimental results

To better test the improved small files storage access performance of the improved Swift framework, many stress testing experiments have been performed on the improved framework. Swift-bench was used as test tool. The experimental test results are as follows.

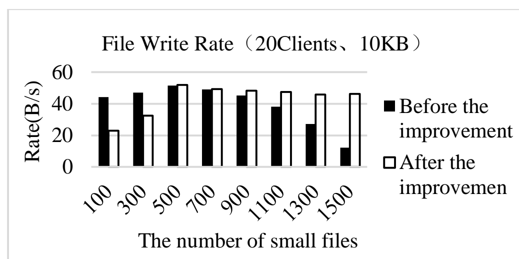

Figure 3. File write rate(20clients、10KB).

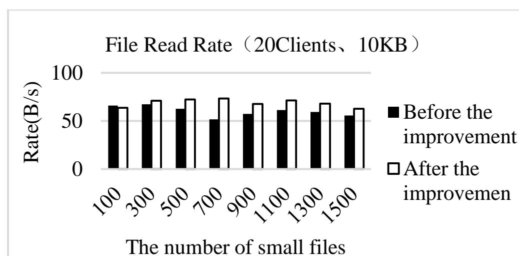

Figure 4. File read rate(20clients、10KB).

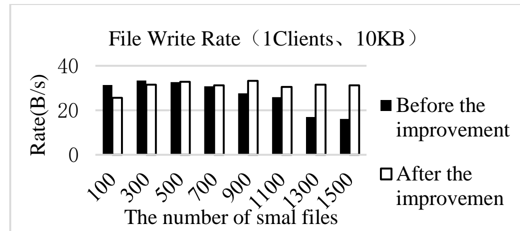

Figure 5. File write rate(1clients、10KB).

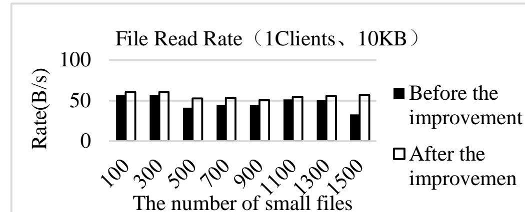

Figure 6. File reading rate(1clients、10KB).

As shown in Figure 3 and Figure 4, in the case of 20 clients writing 10KB small files concurrently, when the number of files is less than 300, the optimized system performance is lower than that of the unoptimized system. However, as the number of files increases, the transmission performance of non-optimized systems gradually decreases, and the performance advantages of optimized systems become more pronounced. In the same situation, the read performance of small files is similar to the former. The scenario where a cluster has only one client is shown in Figure 5 and Figure 6:as the number of files increases, the read/write performance of the optimized cluster is generally greater than that of no optimization. We believe that as the number of files increases, the IO of the system becomes more and more crowded. At this time, the merge strategy can reduce the number of inodes, thereby ensuring the stability of the system performance.

In order to continue to verify the effectiveness of the ASS. In the case of 20 clients, these clients uploads/download 500 small files respectively. At the same time, we record the file access rate in each case as follows. Note that the size of 500 small files is 1KB, 5KB, 10KB···, 100 KB.

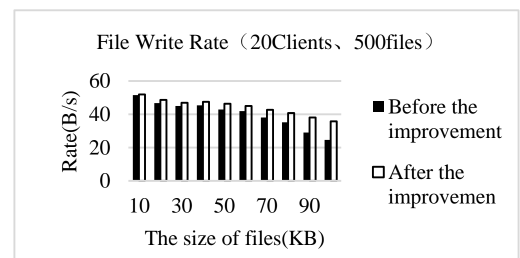

Figure 7. File write rate(20Clients、500files).

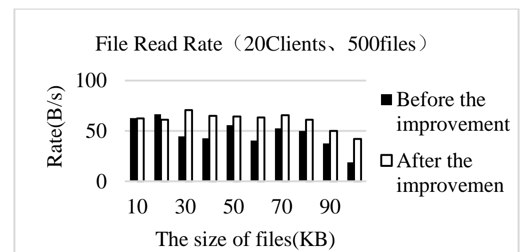

Figure 8. File read rate(20Clients、500files).

As shown in Figure 7, when 20 clients write 500 files at the same time, the improved cluster's small files transfer performance is usually higher than the unimproved cluster. In the same case, the clients read to the cluster. Although the small files transfer performance of the optimized cluster is low when the size of files is less than 20KB, the optimized system performance is more stable overall. And we believe that the improved system improves the storage and access performance of small files.

5. CONCLUSIONS

This paper describes an aggregated storage strategy that is used to improve small file storage performance in Openstack Swift. Based on the original Swift framework, we added a merge engine module between the object server and the XFS file system. This module uses ASS. Then we use ASS to merge small files into volumes. Experiments show that the improved cluster reduces IO congestion and improves the read/write performance of small files.

6. ACKNOWLEDGMENTS

The authors would like to thank the persons who review and give some valuable comments to improve the paper quality. This work was supported by Science and Technology Department of Sichuan Province, Fund of Science and Technology Planning (No. 2018JY0290).

7. REFERENCES

[1] Zwolenski, Matt, and L. Weatherill. "The Digital Universe Rich Data and the Increasing Value of the Internet of Things." Australian Journal of Telecommunications and the Digital Economy 3,2014.

[2] John Gantz, and David Reinsel. "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east." IDC iView: IDC Analyze the Future, 2007:1-16.

[3] J. R Douceur, W. J Bolosky, J. R Lorch, and N. Agrawal. " A five-year study of file-system metadata." ACM Transactions on Storage, 2007: 9-9.

[4] Meyer, T. Dutch, and W. J. Bolosky. "A study of practical deduplication." Usenix Conference on File and Stroage Technologies USENIX Association, 2011:1-1.

[5] A. Chervenak, J. M. Schopf, L. Pearlman, M. H. Su, S. Bharathi, M. D'Arcy, N. Miller, D. Bernholdt and L. Cinquini. "Monitoring the Earth System Grid with MDS4." IEEE International Conference on E-Science and Grid Computing, 2006. E-Science IEEE, 2006:69-69.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. "Finding a needle in Haystack: facebook's photo storage." Usenix Conference on Operating Systems Design and Implementation USENIX Association, 2010:47-60.

[7] Wang, Jing, and Y. Guo. "Scrapy-Based Crawling and User-Behavior Characteristics Analysis on Taobao." International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery IEEE, 2012:44-52.

[8] C. Weddle, M. Charles, J. Qian, A. I. A. Wang, P. Reiher, and G. Kuenning. "PARAID: a gear-shifting power-aware RAID." Usenix Conference on File and Storage Technologies USENIX Association, 2007:30-30.

[9] Sacks, D. "Demystifying Storage Networking DAS, SAN, NAS, NAS Gateways, Fibre Channel, and iSCSI." Ibm Storage Networking, 2001.

[10] S. Ghemawat, H. Gobioff, S. T. Leung. "The Google file system." ACM SIGOPS Operating Systems Review 37, 2003:29-43.

[11] D. Beaver, S. Doug, H. C. Li, J. Sobel, and P. Vajgel. "Finding a needle in Haystack: facebook's photo storage." Usenix Conference on Operating Systems Design and Implementation USENIX Association, 2010:47-60.

[12] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store." ACM Sigops Symposium on Operating Systems Principles ACM, 2007:205-220.

[13] Lakshman, Avinash, and P. Malik. "Cassandra:a decentralized structured storage system." Acm Sigops Operating Systems Review 44,2010:35-40.

[14] Y. han. "A brief analysis of No SQL database solution Tair. " The electronic commerce,2011:54-61.

[15] L. zhang. Research and implementation of embedded file system based on flash memory. University of Electronic Science and Technology of China, 2005.

[16] Zhong, S, J. Chen, and Y. R. Yang. "Sprite: a simple, cheat-proof, credit-based system for mobile ad-hoc networks." Joint Conference of the IEEE Computer and Communications. IEEE Societies IEEE, 2003:1987-1997.

[17] Nie, Gang, and Q. Xiu-Hua. "Research on Lustre file system based on object-based storage." Information Technology, 2007.

Website:

[18] http://hadoop.apache.org/docs/current/hadoop-archives/ HadoopArchives.html