Content-Based Chunk Placement Scheme for Decentralized Deduplication on Distributed File Systems

Keonwoo Kim[1], Jeehong Kim[1], Changwoo Min[1,2], and Young Ik Eom[1]

[1] College of Information and Communication Engineering, Sungkyunkwan University
Suwon, Korea
{kkw0528,jjilong,multics69,yieom}@skku.edu
[2] Samsung Electronics Co., Ltd., Suwon, Korea
multics69@skku.edu

Abstract. The rapid growth of data size causes several problems such as storage limitation and increment of data management cost. In order to store and manage massive data, Distributed File System (DFS) is widely used. Furthermore, in order to reduce the volume of storage, data deduplication schemes are being extensively studied. The data deduplication increases the available storage capacity by eliminating duplicated data. However, deduplication process causes performance overhead such as disk I/O. In this paper, we propose a content-based chunk placement scheme to increase deduplication rate on the DFS. To avoid performance overhead caused by deduplication process, we use *lessfs* in each chunk server. With our design, our system performs decentralized deduplication process in each chunk server. Moreover, we use consistent hashing for chunk allocation and failure recovery. Our experimental results show that the proposed system reduces the storage space by 60% than the system without consistent hashing.

Keywords: Deduplication, Distributed file system, Chunk placement, Consistent hashing.

1 Introduction

The amount of digital information is rapidly increasing all over the world. According to the forecast by IDC, the amount of digital information will grow by a factor of 50 over the next decade [1, 2]. Also, most of data in the digital universe is unstructured one such as image, video, audio, and document files [1]. Moreover, an influx of data is rapidly growing in cloud storage [1]. This rapid growth of data size causes several problems such as storage limitation, increment of data management cost, and network traffic congestion [3, 4]. For storing and managing massive data in cloud storage, cloud storage service provider generally uses Distributed File System (DFS). However, despite wide adoption of DFS, rapidly growing data size causes additional storage and management cost. Therefore, data size optimization techniques are required for cloud storage systems. Among

B. Murgante et al. (Eds.): ICCSA 2013, Part I, LNCS 7971, pp. 173–183, 2013.

data size optimization studies [3–5], data deduplication is a representative research issue. Data deduplication eliminates duplicated data, by which available storage space is increased and additional storage cost is reduced.

Duplicated data reduces available storage space. Most of cloud storage services use data deduplication to solve lack of available storage space. However, previous research [6–11] on DFS do not support data deduplcation because of performance overheads that are caused by additional computation cost and disk I/O. In this paper, we propose a content-based chunk placement for the data deduplication on the DFS. In order to distribute deduplication processes, we integrate *lessfs* that is an inline deduplication file system with each chunk server of MooseFS. Thus, we achieve decentralized data deduplication process, while maintaining high performance. In MooseFS [6], a file is divided into multiple chunks and each of them is stored in chunk server dispersedly. So, by using consistent hashing, the chunk placement module gather same chunks in one chunk server, by which we enhance the deduplication rate. We make following contributions in this paper:

- We introduce a new content-based chunk placement for deduplication system on MooseFS. Furthermore, we design a content-based chunk placement for deduplication system that replaces the file system of chunk server in MooseFS with *lessfs*.
- We coordinate consistent hashing with a content-based chunk placement for deduplication system for enhancing the deduplication rate. By performing deduplication process in each chunk server, our system can avoid bottleneck of the master server.
- We evaluate our chunk placement for deduplication system, and show its effectiveness.

The rest of the paper is organized as follows: We review background in Section 2. Section 3 describes the key ideas and implementation details. In Section 4, we evaluate our system and show the results. Finally, we conclude in Section 5.

2 Background and Related Work

2.1 DFS

A DFS is file system that allows access to files from multiple hosts via a communication network [12]. With DFS, a client can access remote files in the same way that it accesses local files. DFS generally consists of single master server and multiple storage servers. Master server manages file metadata and chunk server keeps chunk data of files. Typical open-source based DFSs are MooseFS [6], XtreemFS [7, 8], Ceph [10], Google file system[11], and GlusterFS [9]. We choose MooseFS as our DFS platform because it is general-purpose and good performance file system. MooseFS consists of single master server, multiple metadata backup servers, and multiple chunk servers. Master server manages whole file system and stores metadata for each file. Chunk servers store chunk data of each file. Metadata backup servers store metadata changelogs and periodically download metadata files from master server [6].

2.2 Data Deduplication in Local File System

A data deduplication technique is used to increase the storage capacity by detecting and eliminating duplicated data. With the scheme, each chunk can have only a single copy in the system. The data deduplication exists in the following types: post-process and inline data deduplication. Post-process data deduplication occurs after data has been written, whereas inline data deduplication occurs before the data has been written [5]. Therefore, inline deduplication causes more network traffic than post-process deduplication. On the other hand, post-process deduplication requires storage space to store the duplicated data before deduplication is performed. Typical open-source based deduplication file systems are *lessfs*, zfs, and sdfs [4, 5]. We use *lessfs* because it is a good performance deduplication file system. *Lessfs* is an inline block-level deduplication in Linux file system and uses data compression (e.g., LZO, QuickLZ, Snappy, bzip, and gzip). Also, block size can be defined as 4, 8, 16, 32, 64, and 128 KB. Through configuring block size, we can adjust tradeoff between throughput and deduplicated size. To store metadata, *lessfs* uses low-level FUSE API and database (e.g., BerkeleyDB, hamsterdb, and Tokyo Cabinet). Also, by using cache, *lessfs* reduces disk I/O.

2.3 Related Work

As the data size rapidly increases, the data deduplication is extensively studied to optimize storage capacity. The data deduplication technique is divided into two categories [3]: primary data deduplication and secondary data deduplication. Primary data storage directly interacts with application. In other words, application directly affects data. Thus, primary data deduplication systems are latency-aware and use RPC based protocols [13]. On the other hand, secondary data storage copies and archives data to recover from data loss and corruption. Secondary data deduplication systems are throughput-aware and use streaming protocols [13] because this storage processes large amounts of data. Mayer et al. [14] examined in primary data and secondary data. They found that block-level deduplication saves just about 10% more space than the whole-file deduplication. However, experimental result of El-Shimi et al. [15] showed that block-level deduplication saves from 2.3 times to 15.8 times more storage space than whole-file deduplcation. This difference of experimental results is due to the difference of Mayer's data set and El-Shimi's data set. HYDRAstor [16] is a secondary data storage and block-level deduplication system, and uses distributed hash table for scalability. iDedup [13] is a inline data deduplication for the primary storage, and uses in-memory indexing and metadata cache. Lillibridge et al. [17] propose a inline deduplication system and use parse indexing which is in-memory index. Wei et al. [18] use bloom filter and dual cache. Zhu et al. [19] use summary vector and locality preserved caching.

Previous research [13–20] regards deduplication process as performance degradation factor because of disk I/O. Therefore, in-memory indexing or cache is used to reduce deduplication overhead. In-memory indexing such as Bloom

filter can reduce disk I/O and quickly searches chunk or block which is a unit of deduplication. Cache is also used to reduce disk I/O.

However, most of studies [13–17, 19] perform centralized data deduplication. This causes bottleneck of the central server and increases I/O latency. For this reason, we use *lessfs* in each chunk server of MooseFS to decentralize deduplication.

3 Design and Implementation

3.1 Chunk Placement of Existing MooseFS

When the client of MooseFS executes the write operation, write operation steps of existing MooseFS are as follows: (1) The client requests a chunk server list to the master server to store chunks in chunk servers. (2) The master server returns a chunk server list that consists of information of chunk servers which have more available space than the other chunk servers. (3) The client sends the chunk data to chunk servers that correspond with a chunk server list. (4) Chunk servers receive and store chunk data.

Therefore, chunks are stored on the chunk server that has more available space than the other servers. This approach is fitted for a system requiring storage load balancing.

3.2 Deduplication on MooseFS

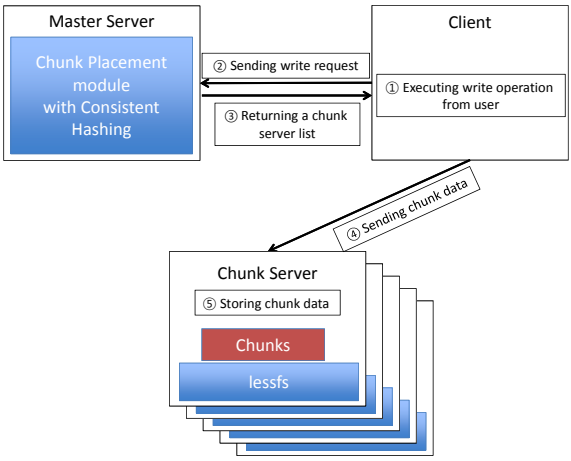


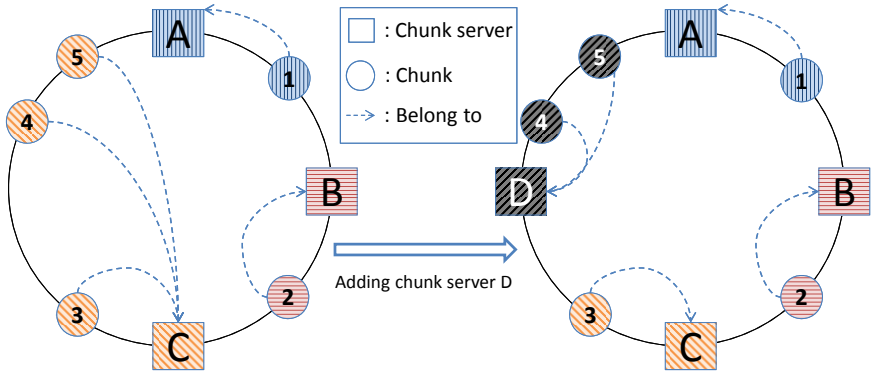
Fig. 1. Overall architecture

We propose the deduplication on MooseFS. MooseFS is chunk-based DFS. If a file size is larger than 64MB, a file is divided up into 64MB chunks. Otherwise, a chunk size becomes the same as a file size. MooseFS can consist of a single master server and the multiple chunk servers as Fig. 1 shows. Chunks are dispersedly stored in chunk servers that are distributed on the network. Chunk server can use many file systems but ext4 is generally used. To implement the deduplication on DFS, we use a *lessfs* that is a block-level and inline deduplication file system. In chunk server of Fig. 1, we mount *lessfs* in the chunk server for the data deduplication. Therefore, all chunks of chunk server are stored in a mount point of *lessfs* and the duplicated chunk data are eliminated by *lessfs*. The deduplication process is regarded as causing performance overheads such as the additional computation cost and the disk I/O. If deduplication process is performed in a master server, those overheads cause bottleneck of a master server. Therefore, the data deduplication process is performed in each chunk server so we can avoid the bottleneck to deduplicate in a master server. Also, the disk I/O can be reduced because *lessfs* uses cache and in-memory database.

3.3 Consistent Hashing for MooseFS

Consistent Hashing. Consistent hashing [21, 22] is used in a changing population of web server environment. If a sever node is added or removed, all objects of web server nodes have to be relocated. We use consistent hashing to solve this problem. Each node is mapped on a hash ring and has a hash value range. Moreover, each data object has hash value and belongs to each node. When data object is stored, the system finds a node by comparing hash value range of node with hash value of data object. After the system find a node, the data object is stored in the found node. By using consistent hashing, we can avoid all object data relocation. Consistent hashing is used in many distributed system such as Dynamo [23], Cassandra [24], and Memcached [25].

Consistent Hashing for MooseFS. When communicating with servers, Moose-FS uses 32-bit CRC that is a hash function to detect error. For identifying chunk, we utilize 32-bit CRC. In master server, we make a data structure for the chunk server node in consistent hashing. Each node also has start and end of 32-bit CRC value, size of range, and next node pointer and consistent hahsing is implemented as a circular linked list. When chunk server is registered, new node is created and included in a circular linked list. The range of consistent hashing is from 0 to 2^{32}-1 because CRC is 32-bit. Each chunk server has 32-bit CRC hash range and each chunk is stored in the corresponding chunk server. To find the chunk server, the master server compares 32-bit CRC value of chunk with a hash range of the chunk server. In Fig. 2-(a), consistent hashing include the chunk servers and chunks. Chunk 1 belongs to chunk server A, chunk 2 belongs to chunk server B, and chunk 3, chunk 4, and chunk 5 belong to chunk server C. Subsequently, in Fig. 2-(b), if chunk server D is added in consistent hashing, range of chunk server C is divided into two halves. So, range



Chunk server	Hash range	Stored chunks
A	[0, 2^{30})	1
B	[2^{30}, 2^{31})	2
C	[2^{31}, 2^{32})	3, 4, 5

(a) Before adding chunk server D

Chunk server	Hash range	Stored chunks
A	[0, 2^{30})	1
B	[2^{30}, 2^{31})	2
C	[2^{31}, $3x2^{30}$)	3
D	[$3x2^{30}$, 2^{32})	4, 5

(b) After adding chunk server D

Fig. 2. Consistent Hashing for MooseFS

of chunk server C becomes from 2^{31} to 3×2^{30}-1 and range of chunk server D becomes from 3×2^{30} to 2^{32}-1. Therefore, chunk 4 and chunk 5 are relocated from chunk server C to chunk server D because of changing range of chunk server C. When the client executes a write operation, the write operation steps of the proposed chunk placement are as follows: (1) The client generates 32-bit CRC value that corresponds with foremost 4KB of the chunk data. (2) The client sends a write request message to the master server with 32-bit CRC value. (3) To find a appropriate chunk server, the master server travels a circular linked list with comparing 32-bit CRC value with start and end value of each chunk server node. (4) MooseFS not use replication, the master server stores node information to a chunk server list. (5) Otherwise, replicas of chunk data are stored other chunk servers. Therefore, the master server chooses found node of step (4) and its next nodes. (6) The master server returns a chunk server list to the client. (7) The client sends chunk data to chunk servers that correspond with a chunk servers list. (8) Chunk servers receive and store chunk data. For content-based chunk placement, master server compares 32-bit CRC range of chunk server and 32-bit CRC value of chunk. However, the larger size data is input to 32-bit CRC, the more time is spent. Therefore, 32-bit CRC computation of all 64 MB chunks causes the bottleneck in the master server. If the master server performs 32-bit CRC computation and comparing with 32-bit CRC values, the performance of whole system is degraded. Chen et al. [26] propose content-based sampling which produces 32-bit CRC value that corresponds with the first four bytes of

each page. Moreover, they examined choosing other bytes and found that using the first four bytes performs well. For this reason, we hash a foremost 4 KB of a 64 MB chunk.

4 Evaluation

In this section, we evaluate how much our deduplication system reduces the data. We installed client, master server, and chunk server in one PC and installed chunk server in 17 PCs. Also, we mount *lessfs* on all PCs. Therefore, the total number of chunk servers is 18. We used multimedia data set (about 936 MB) which includes movie files, audio files, and document files. To identify what amount of duplicated data is eliminated, we make data *set-1*, *set-2*, and *set-3* that include same data and have different name.

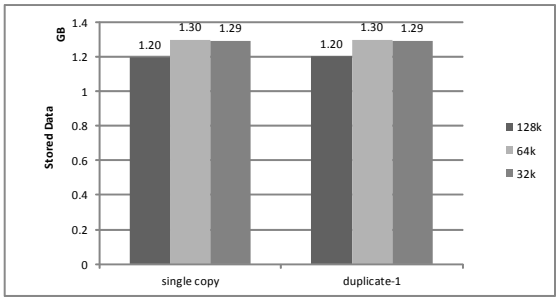


Fig. 3. Stored data size according to block size variation

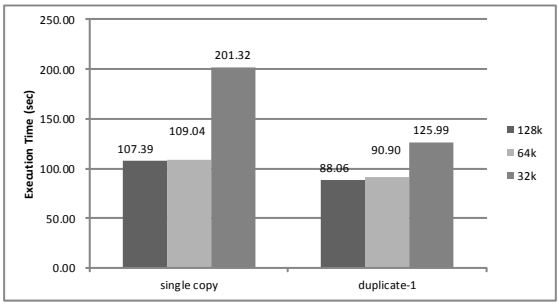


Fig. 4. Execution time according to block size variation

First of all, we experiment to find suitable block size of *lessfs*. In Fig. 3, single copy of x-axis means that unique *set-1* is stored in MooseFS, duplicate-1 of x-axis means that both *set-1* and *set-2* were stored in MooseFS. Y-axis means

stored data size on all chunk servers. 128 KB, 64 KB, and 32 KB are block size of *lessfs*. Stored data size of 128 KB is smaller than 64 KB and 32 KB block size.

In Fig. 4, execution time of 128 KB is the fastest but execution time of 32 KB is almost twice as execution time of 128 KB and 64 KB in single copy. The reason of this result is that block size is too small. 32 KB block size takes long time to execute, because small block size occurs to create more metadata and many comparision to detect data duplication. Therefore, optimal deduplication block size is 128 KB, and we experiment our system using 128 KB block size.

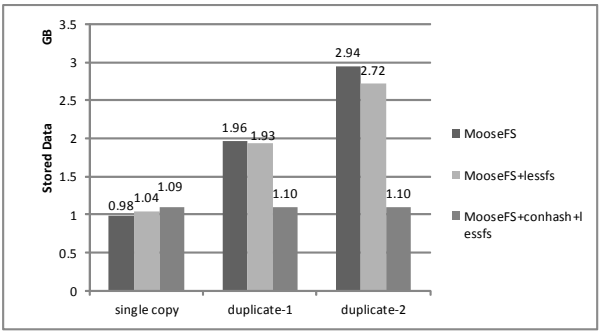


Fig. 5. Stored data size in MooseFS, MooseFS with *lessfs*, and MooseFS with consistent hashing and *lessfs*

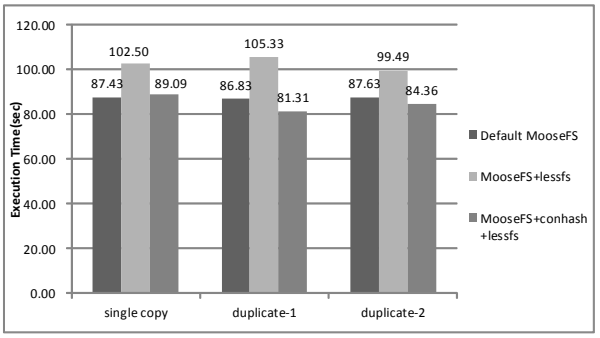


Fig. 6. Execution time in MooseFS, MooseFS with *lessfs*, and MooseFS with consistent hashing and *lessfs*

In Fig. 5, `Default MooseFS` means stored data size in default MooseFS which not use *lessfs*. `MooseFS+lessfs` means stored data size in deduplication on default MooseFS that provide default chunk placement. `MooseFS+conhash+lessfs` means stored data size in deduplication on MooseFS that provide chunk placement using consistent hashing. Fig. 5 shows that our chunk placement scheme

is more effective than chunk placement scheme of default MooseFS. Data size of single copy is more than 936 MB because *lessfs* creates metadata to manage data. We define deduplication rate as 100 - ((single copy size × a number of duplicate) ÷ actual stored data size × 100). Deduplication rate of `MooseFS+lessfs` is about 12%. In this regard, chunk placement scheme is needed for improving deduplication rate. Therefore, we experiment applying chunk placement with consistent hashing. As a result, deduplication rate is about 99%. This result shows that proposed system reduces the storage space by 60% than the system without consistent hashing. All data is not eliminated because *lessfs* creates metadata about duplicated data.

In single copy of Fig. 6, execution time of `MooseFS+conhash+lessfs` is slightly slower than `Default MooseFS` because of chunk hashing and deduplication overhead. However, duplicated file copy causes fewer write operation due to eliminating duplicated data. Because `MooseFS+lessfs` does not use content-based chunk placement, `MooseFS+lessfs` rerely performs deduplication. Therefore, `MooseFS+lessfs` is slower than the others. Duplicated file copy of `MooseFS+conhash+lessfs` is faster than single copy and duplicated copy of `Default MooseFS`.

5 Conclusion

In this paper, we study content-based chunk placement for decentralized deduplication on the DFS. We describe how we design our system in detail. We utilize open-source based DFS (MooseFS) and deduplication file system (*lessfs*). We integrate the chunk server of MooseFS with *lessfs*. Therefore, contrary to general deduplication studies, our system can avoid the bottleneck of master server because each chunk server performs decentralized deduplication processes. Moreover, in order to reduce chunk hash overhead, we design content-based chunk placement with consistent hashing that hash foremost 4 KB of chunk data. As a result, experimental results show that proposed system reduces the storage space by 60% than the system without consistent hashing and execution time of proposed system is similar to execution time of default MooseFS.

Acknowledgements. This work was supported by the IT R&D program of MKE/KEIT. [10041244, SmartTV 2.0 Software Platform].

References

1. Gantz, J., Reinsel, D.: 2011 Digital Universe Study: Extracting Value from Chaos. Technical report, IDC (2011)
2. Gantz, J., Reinsel, D.: The Digital Univers. In: 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. Technical report, IDC (2011)
3. DuBois, L., Amaldas, M.: IDC key-considerations deduplication. Technical report, IDC (2010)
4. Webb, N.: Open Source Data Deduplication. In: Linuxfest Northwest, Bellingham, WA, USA (April 2011)

5. Koutoupis, P.: Data Deduplication with Linux. 7 (2011)
6. MooseFS, http://www.moosefs.org
7. Hupfeld, F., Cortes, T., Kolbeck, B., Stender, J., Focht, E., Hess, M., Malo, J., Marti, J., Cesario, E.: The XtreemFS architecture—a case for object-based file systems in Grids. Concurrency and Computation: Practice and Experience 20(17), 2049–2060 (2008)
8. XtreemFS, http://www.xtreemfs.org
9. GlusterFS, http://www.gluster.org
10. Weil, S., Brandt, S., Miller, E., Long, D., Maltzahn, C.: Ceph: A scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), pp. 307–320 (2006)
11. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. ACM SIGOPS Operating Systems Review 37, 29–43 (2003)
12. Thanh, T., Mohan, S., Choi, E., Kim, S., Kim, P.: A taxonomy and survey on distributed file systems. In: Fourth International Conference on Networked Computing and Advanced Information Management, NCM 2008, vol. 1, pp. 144–149. IEEE (2008)
13. Srinivasan, K., Bisson, T., Goodson, G., Voruganti, K.: iDedup: Latency-aware, inline data deduplication for primary storage. In: Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST 2012), San Jose, CA (2012)
14. Meyer, D., Bolosky, W.: A study of practical deduplication. ACM Transactions on Storage (TOS) 7(4), 14 (2012)
15. El-Shimi, A., Kalach, R., Kumar, A., Oltean, A., Li, J., Sengupta, S.: Primary Data Deduplication-Large Scale Study and System Design. In: Proccedings of the USENIX Annual Technical Conference 2012 (2012)
16. Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., Welnicki, M.: Hydrastor: A scalable secondary storage. In: Proccedings of the 7th Conference on File and Storage Technologies, pp. 197–210. USENIX Association (2009)
17. Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., Camble, P.: Sparse indexing: large scale, inline deduplication using sampling and locality. In: Proccedings of the 7th Conference on File and Storage Technologies, pp. 111–123 (2009)
18. Wei, J., Jiang, H., Zhou, K., Feng, D.: Mad2: A scalable high-throughput exact deduplication approach for network backup services. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–14. IEEE (2010)
19. Zhu, B., Li, K., Patterson, H.: Avoiding the disk bottleneck in the data domain deduplication file system. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies, vol. 18 (2008)
20. Clements, A., Ahmad, I., Vilayannur, M., Li, J., et al.: Decentralized deduplication in SAN cluster file systems. In: Proceedings of the 2009 Conference on USENIX Annual Technical Conference, p. 8. USENIX Association (2009)
21. Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., Yerushalmi, Y.: Web caching with consistent hashing. Computer Networks 31(11), 1203–1213 (1999)
22. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 654–663. ACM (1997)

23. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. ACM SIGOPS Operating Systems Review 41, 205–220 (2007)
24. Cassandra, `http://cassandra.apache.org`
25. Memcached, `http://memcached.org/`
26. Chen, F., Luo, T., Zhang, X.: CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In: Proceedings of the 9th USENIX Conference on File and Stroage Technologies, p. 6. USENIX Association (2011)