Distributed and High Performance Big-File Cloud Storage Based On Key-Value Store *

Thanh Trung Nguyen,[1,2] **Minh Hieu Nguyen**[2]

[1] *Le Quy Don Technical University ,*
No 236 Hoang Quoc Viet street, Cau Giay District,
Ha Noi, Viet Nam
E-mail: thanhnt@vng.com.vn

[2] *Research and Development Department, VNG Corporation,*
Trung Kinh Street, Cau Giay District,
Ha Noi, VietNam
E-mail: minhnh@mta.edu.vn

Abstract

This research proposes a new Big File Cloud (BFC) with its architecture and algorithms to solve difficult problems of cloud-based storage using the advantages of key-value stores. There are many problems when designing an efficient storage engine for cloud-based storage systems with strict requirements such as big-file processing, lightweight meta-data, low latency, parallel I/O, deduplication, distributed, high scalability. Key-value stores have many advantages and outperform traditional relational database in storing data for heavy load systems. This paper contributes a low-complicated, fixed-size meta-data design, which supports fast and highly-concurrent, distributed file I/O, several algorithms for resumable upload, download and simple data deduplication method for static data. This research applies the advantages of ZDB - an in-house key-value store which was optimized with auto-increment integer keys for solving big-file storage problems efficiently. The results can be used for building scalable distributed data cloud storage that support big-files with sizes up to several terabytes.

1. Introduction

Cloud-based storage services commonly serve millions of users with storage capacity for each user can reach to several gigabytes to terabytes of data. People use cloud storage for the daily demands, for example backing-up data, sharing files to their friends via social networks such as Facebook [14], Zing Me [4]. Users also probably upload data from many different types of devices such as computer, mobile phone or tablet. After that, they can download or share them to others. System load in a cloud storage is usually really heavy. Thus, to guarantee a good quality of service for users, the system has to face many difficult problems and requirements: Serving intensity data service for a large number of users without bottle-neck; Storing, retrieving and managing big-files in the system efficiently; Parallel and resumable uploading and downloading; Data deduplication to reduce the waste of storage space caused by

* The preliminary version of this paper was presented in the 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2015) [20] June 2015

storing the same static data from different users. In traditional file systems, there are many challenges for service builder when managing a huge number of big-file: How to scale system for the incredible growth of data; The challenges in **distributing** data in a large number of nodes; Effective methods to **replicate** data for load-balancing and fault-tolerance; How to **cache** frequently accessed data for fast I/O, minimize the latency, etc. In many Distributed File Systems and Cloud Storages, a common method for solving these problems is splitting big file into multiple smaller chunks, storing them on disks or distributed nodes and then managing them using a meta-data system [7,6,29,1]. Storing chunks, meta-data efficiently and designing a lightweight meta-data are significant problems that cloud storage providers have to face. After a long time of investigating, we realized that current cloud storage services have a complex meta-data system, at least the size of meta-data is linear to the file size. Therefore, the space complexity of these meta-data system is $O(n)$. For big-files which have sizes of serveral Gigabytes or Terabytes can lead to big meta-data sizes. And it is not well scalable for big-files.

In this research, we propose a new big-file cloud storage architecture and a better solution to reduce the space complexity of meta-data.

Key-Value stores have many advantages for efficiently storing data in data-intensive services. They often outperform traditional relational databases in the ability of heavy load and large-scale systems. In recent years, key-value stores have an unprecedented growth in both academic and industrial fields. They have low-latency response time and good scalability with small and medium key-value pair size.

Current key-value stores are not designed for directly storing big-values or big-files in our case. We executed several experiments in which we put whole file-data to key-value store, the system did not have good performance as usual for many reasons: firstly, the latency of put/get operation for big-values is high, thus it affects other concurrent operations of key-value store service and multiple parallel accesses to different value reach limited. Secondly, when the value is bigger, there is no more space to cache other objects in main memory for fast access operations. Finally, it is difficult to scale-out system when the number of users and data increased. This research is implemented to solve those problems when storing big-values or big-file using key-value stores. It brings many advantages of key-value store in data management to design a cloud-storage system called Big File Cloud (BFC).

These are our contributions in this research:

- Propose a light-weight meta-data design for big file. Every file has nearly the same size of meta-data. BFC has $O(1)$ space complexity of meta-data of a file, while the size of meta-data of a file in Dropbox[7], HDFS[1] has space complexity of $O(n)$ where n is size of the original file. See Fig. 13
- Propose a logical contiguous chunk-id for chunk collections of files. That makes it easier to distribute data and scale-out the storage system.
- Bring the advantages of key-value store into big-file data store which is not supported by default for big-value.
- Study the key-value stores to find the most appropriate key-value store for BFC.

These contributions are implemented and evaluated in Big File Cloud (BFC) that serves storage for Zing Me [4] users . Disk Image files of CSM Boot-Diskless [†]system are stored in Big File Cloud. The rest of this paper is organized as follow: Section 2 is about related works analysis and some comparison between them and BFC. Section 3 presents Big-File Cloud model, detail architecture, design of BFC, data logical layout and algorithms for parallel upload and download big files. Section 4 evaluates to find the best fit key-value store for proposed cloud storage architecture. Then, it compares BFC and other popular cloud storage systems to emphasize the advantages of the proposed metadata design. It also has

[†] http://csmboot.zing.vn

benchmarks to compare BFC and other open-source solution such as HDFS, Blob of MySQL. Finally, section 5 presents the conclusion of this research.

2. Related Works

CEPH[29] is a distributed storage platform designed to store object, block and file. Ceph is designed to provide excellent performance, reliability and scalability. Ceph removes file allocation tables and replaces them with generating functions, so it separates metadata and data operations. This is a significant improvement. Because this allows it to distribute the complex data access operations, update serialization, replication and reliability, failure detection, and recovery. CEPH uses special-purpose data distribution function called CRUSH [30], CRUSH is a pseudo-random data distribution algorithm. Its advantage is that it can independently calculate the location of any object by any party in a large system. In addition, the size of metadata is small and mostly static unless devices are added or removed. CEPH breaks files into 8 MB chunks and stores them.

Hadoop Distributed File System (HDFS) [25] is a distributed file system designed to run on low cost commodity hardware. The significant advantage of HDFS is fault-tolerant ability. Typically, an HDFS instance consists of hundreds or thousands nodes which to store an extremely large amount of data. There are two types of node in a HDFS instance: NameNode and DataNode which are the components to construct a master/slave architecture. An HDFS cluster includes a single NameNode and a number of DataNodes. The first one manages the file system namespace and serves the file access request from clients. It executes file system namespace operations such as opening, closing or renaming a file or directory. DataNodes are responsibility for storing application data and serving file system read/write requests. When a client creates a new file in an HDFS instance, that file will be splited to several blocks with a configurable size per file (typically 64MB). After that, HDFS replicates those blocks and stores them to DataNodes. All of metadatas and transaction logs will be stored at NameNode. When a client request to read a file in the file system, it send the request to NameNode to get file metadata, then get all blocks of that file from DataNodes. Every block of a file has a checksum value, these value are stored in a separate hidden file in HDFS namespace, when client retrieves a block, it will check this checksum value, if not match, if will find another replica of that block at another DataNode. This is the way HDFS avoid corrupted data from disk failure. Beside these advantages, HDFS has a single point of failure - the NameNode. It has only a single NameNode in a cluster, if it crashes, the whole system will down. Files in HDFS are write-once and have strictly one writer at any time. This limits the writing throughput while applications can write data in parallel.

LevelDB [9] is an open source key-value store developed by Google Fellows Jeffrey Dean and Sanjay Ghemawat, originated from BigTable [2]. LevelDB implements LSM-tree [21] and consists of two MemTable and set of SSTables on disk in multiple levels. When a key-value pair is written, it firstly is appended to commit log file, then it is inserted into a sorted structure called MemTable. When MemTable's size reaches its limit capacity, it will become a read-only Immutable MemTable. Then a new MemTable is created to handle new updates. Immutable MemTable is converted to a level-0 SSTable on disk by a background thread. SSTables which reach the level's limit size, will be merged to create a higher level SSTable. We already evaluated LevelDB in our prior work [19] and the results show that LevelDB is very fast for small key-value pairs and data set. When data growing time-by-time and with large key-value pairs, LevelDB become slow for both writing and reading.

Zing-database (ZDB) [19] is a high performance key-value store that is optimized for auto increment Integer-key. It has a shared-memory flat index for fast looking-up position of key-value entries in data files. ZDB supports sequential writes, random read. ZDB is served in ZDBService using thrift protocol and distribute data using consistent-hash method. In BFC, both file-id and chunk-id are auto increment integer keys, so it is very good to use ZDB to store data. The advantage of ZDB is lightweight memory index and performance for big data. When data grow, it still has a low latency for read and write operations. Many other researches try to optimize famous data structures such as B+tree [15] on SSD, HDD or hybrid storage device. It is also useful for building key-value stores on these data structures. With the design and architecture of BFC, the chunkId of a file has a contiguous integer range, ZDB is still the most effective to store chunk data.

Distributed Storage Systems (DSS) are storage systems designed to operate on network environment including Local Area Network(LAN) and the Internet. In DSS, data is distributed to many servers with ability to serve millions of users [22]. There are many type of Distributed Storage System which can be categorized by its functions, architecture. According to [22], DSS can have these functional requirements: Archival, General purpose Filesystem, File Sharing, Performance, etc. Systems under archival category provide user backup and retrieve data functions. BFC fully supports these functions. DSS also provide services as a general purpose file system such as NFS [24] or other Distributed File System (DFS) such as GFS [10,11]. BFC is a persistent non-volatile cloud storage, so it can provide this function in Linux by using FUSE[27] and BFC client protocol. Applications store data on BFC can take advantages of its high performance and parallel processing ability.

UDT[12] is a high performance data transport protocol built on top of UDP. It has reliability control and congestion control. The congestion control algorithm can be customize to utilize the high network bandwidth. UDT support socket API similar to OS socket interface. In BFC, UDT is an option for desktop client application, it can choose between TCP or UDT for exchange data with BFC servers.

3. Big File Cloud Model and Architecture

3.1. General Big File Model

After examining many popular cloud storage system such as DropBox,[‡] Google Drive,[§] OneDrive.[¶] We can model the general big file system (BFS) as follow:

$BFS = \{F, C, M, A_w, A_r\}$ where:

- BFS: Big file system
- F : original file of user in System. It can be identified by a $fileId$. BFS has ability to store large number of file F.
- $C = (c_0, c_1, ..., c_n)$: chunks split from original file F , chunks are often stored in key-value store or a persistent storage. Each chunk can be identify by a $chunkId$.
- M : Metadata that described the organization of F by its chunks. We need to care about space complexity of this Metadata.
- A_w : Algorithm to write file F into BFS.
- A_r: Algorithm to retrieve content of F from BFS by $fileId$.

Using this model we can easily analyze pros and cons of a specific architecture of big file system.

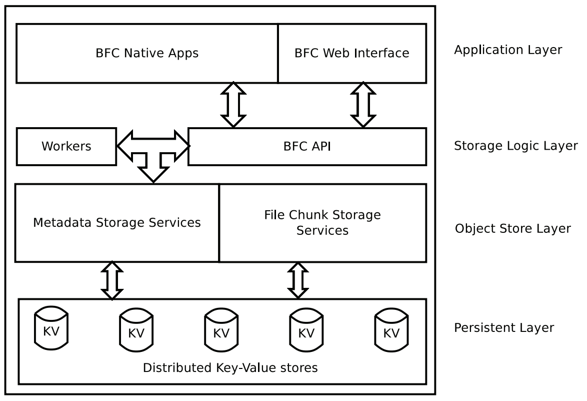

Fig. 1. BFC Architecture

‡ http://dropbox.com

§ http://drive.google.com

¶ https://onedrive.live.com

3.2. Architecture Overview

The BFC system includes four layers: *Application Layer*, *Storage Logical Layer*, *Object Store Layer* and *Persistent Layer*. Each layer contains several coordinated components. They are totally shown in Fig 2. *Application Layer* consists of native software on desktop computers, mobile devices and web-interface, which allow user to upload, download and share their own files. This layer uses the API provided by *Storage Logical Layer* and applies several algorithms for efficient downloading and uploading process which are described in subsections 3.7 and 3.8.

Storage Logical Layer consists of many queuing services and worker services, ID-Generator services and all logical API for Cloud Storage System. This layer implements business logic part in BFC. The most important components of this layer are upload and download service. In addition, this layer provides a high scalable service named CloudAppsService which serves all client requests. When the number of clients reaches a certain limited ones, we can deploy CloudAppsService into more servers for scaling. Clients do not directly request to CloudAppsService, but through a dispatcher which provides public APIs for clients. The dispatcher checks user session before forwarding the client request to CloudAppsService. Moreover, the dispatcher also checks the number of connections from a client, if there are too many concurrent connections from a client, the dispatcher can block requests from that client. *Storage Logical Layer* stores and retrieves data from *Object Store Layer*.

Object Store Layer is the most important layer which has responsibility for storing and caching objects. This layer manages information of all objects in the system, including user data, file information data, and especially meta-data. In BFC system, meta-data describes a file and how it is organized as a list of small chunks. We implemented some optimizations to make low-complicated meta-data. *Object Store Layer* contains many distributed backend services. Two important services of *Object Store Layer* are FileInfoService and ChunkStoreService.

FileInfoService stores information of files. It is a key-value store mapping data from fileID to FileInfo structure.

ChunkStoreService stores data chunks which are created by splitting from the original files that user uploaded. The size of each chunk is fixed(the last chunk of a file may have a smaller size). Splitting and storing a large file as a list of chunks in distributed key-value store bring a lot of benefits. First of all, it is easier to store, distribute and replicate chunks in key-value stores. Small chunks can be stored efficiently in a key-value store. It is difficult to do this with a large file directly in local file system. In addition, this supports uploading and downloading file parallel and resumable. All data on this layer are persisted to *Persistent Layer* based on ZDB [19] key-value store. There are multiple ZDB instances which are deployed as a distributed service and can be scaled when data growing. Components in these layer are coordinated and automatically configured using Zookeeper [13]. Fig 1 shows the overview of BFC Architecture.

3.3. Logical Data layout

Fig 3 shows the layout of big file data. Every file consists of one or more fixed-size chunks. Each chunk has an unique integer ID, and all of chunks which were generated from a file have a contiguous range of chunk-id. This is a different point to many other Cloud Service such as DropBox[6] which uses SHA-2[23] of chunk as chunk-ID.

3.4. Chunk Storage

Basic data units in the BFC cloud storage system are chunks. A chunk is a data segment generated from a file. When a user uploads a file, if the file size is bigger than the configured size, it will be split into a sequence of chunks. All chunks which are generated from a file except the last chunk have the same size (the last chunk of a file may have an equal or smaller size). After that, the ID generator will generate id for the file and the first chunk with auto-increment mechanism. Next chunk in the chunk set

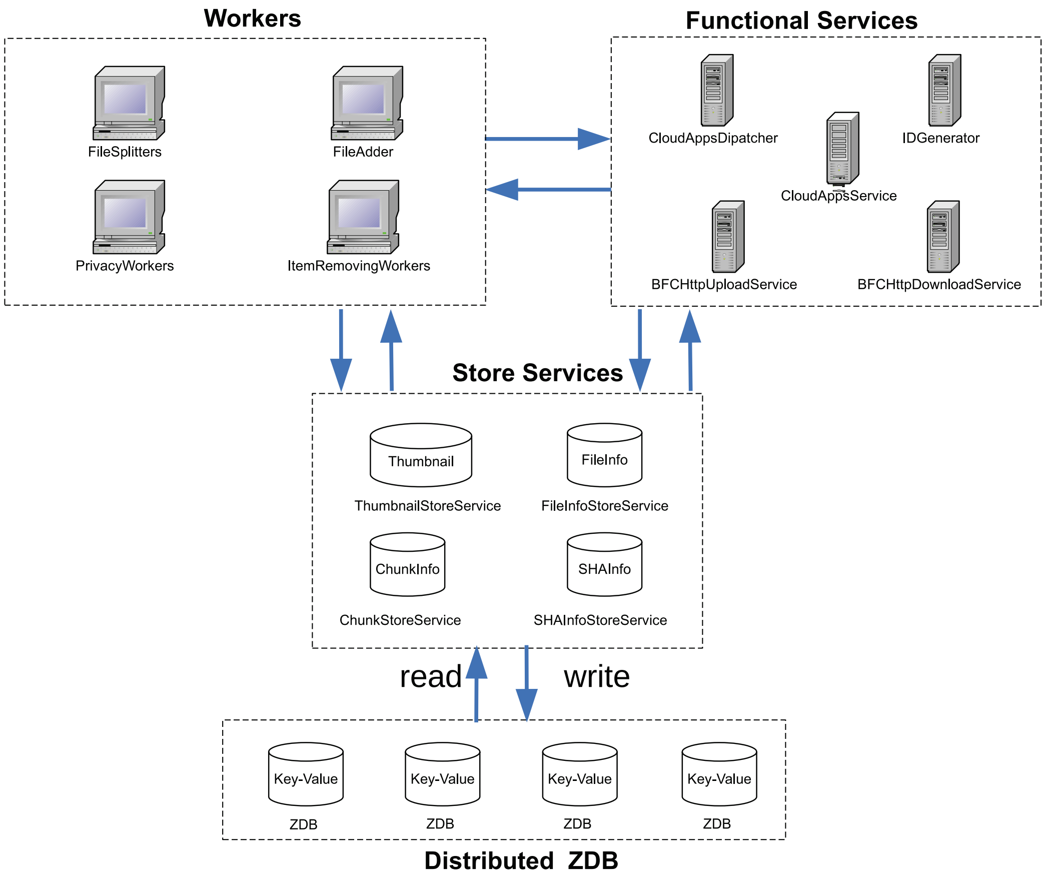

Fig. 2. BFC Main Backend Components

will be assigned an ID gradually increased until the final chunk. A FileInfo object is created with information such as file-id, size of the file, id of the first chunk, the number of chunks and stored in a key-value store implemented in ZDBService [19]. Similarly, the chunk will be stored in key-value store as a record with *key* is the id of chunk and *value* is chunk data. Chunk storage is one of the most significant technique of BFC. By using chunks to represent a file, we can easily build a distributed file storage system service with replication, load balancing, fault-tolerant and recovery. Fig 4 describes Chunk Storage System of BFC.

3.5. Metadata

Typically, in the cloud storage system such as Dropbox [6], CEPH [29], the size of meta-data will respectively increase with the size of original file, it contains a list of elements, each element contains information such as chunk size, hash value of chunk. Length of the list is equal to the number of chunk from file. So it becomes complicated when the file size is big. BFC proposed a solution in which the size of meta-data is independent of number of chunks with any size of file, both a very small file or a huge file. The solution just stores the id of first chunk, and the number of chunks which is generated by original file. Because the id of chunk is increasingly assigned from the first chunk, we can easily calculate the i^{th} chunk id by the formula:

$$chunkid[i] = fileInfo.startChunkID + i \qquad (1)$$

Meta-data is mainly described in FileInfo structure consist of following fields: $fileName$ - the name of

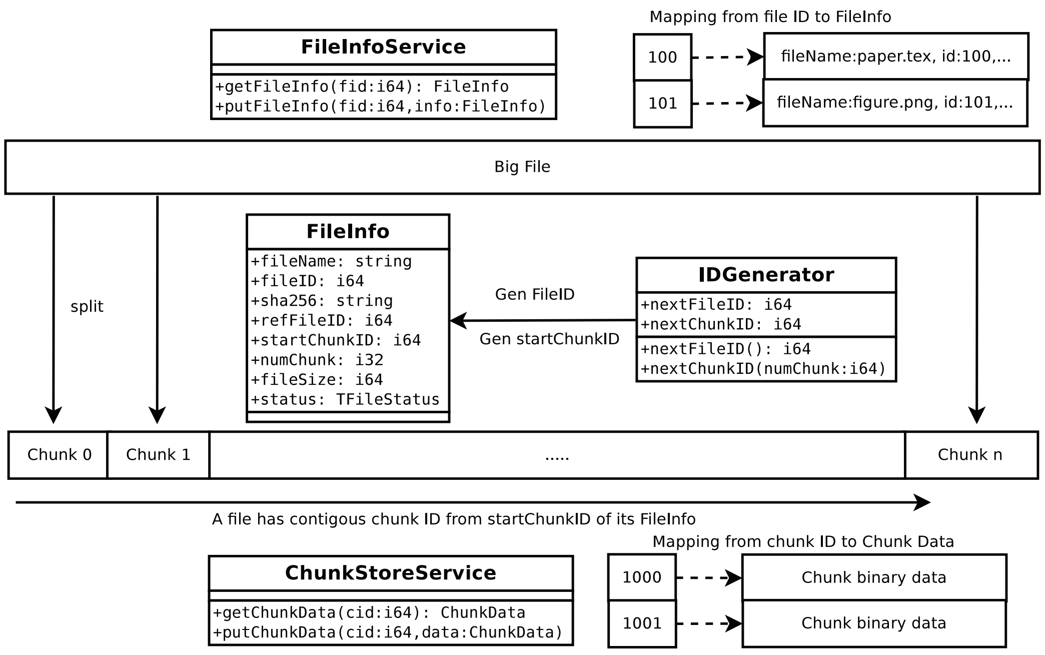
FileInfoService
+getFileInfo(fid:i64): FileInfo
+putFileInfo(fid:i64,info:FileInfo)
Mapping from file ID to FileInfo
100
101
fileName:paper.tex, id:100,...
fileName:figure.png, id:101,...
Big File
split
FileInfo
+fileName: string
+fileID: i64
+sha256: string
+refFileID: i64
+startChunkID: i64
+numChunk: i32
+fileSize: i64
+status: TFileStatus
Gen FileID
Gen startChunkID
IDGenerator
+nextFileID: i64
+nextChunkID: i64
+nextFileID(): i64
+nextChunkID(numChunk:i64)
Chunk 0
Chunk 1
.....
Chunk n
A file has contigous chunk ID from startChunkID of its FileInfo
Mapping from chunk ID to Chunk Data
ChunkStoreService
+getChunkData(cid:i64): ChunkData
+putChunkData(cid:i64,data:ChunkData)
1000
1001
Chunk binary data
Chunk binary data

Fig. 3. Data layout of Big File in system

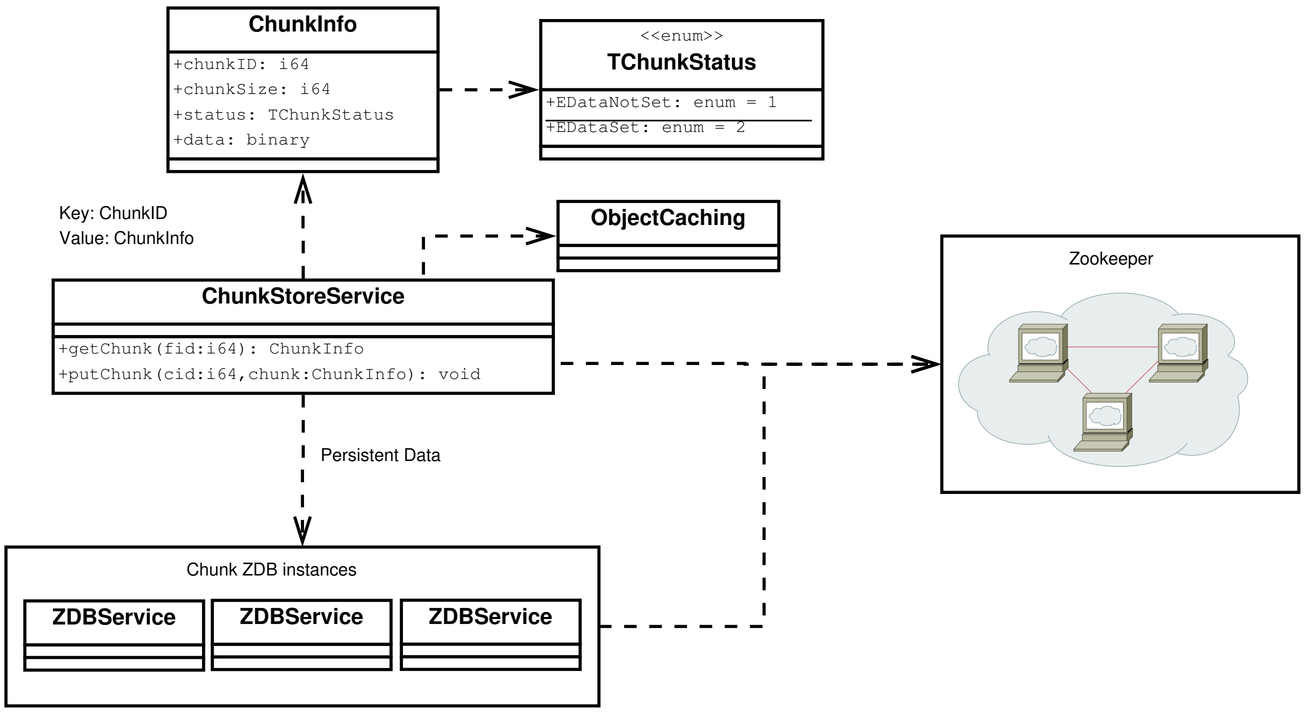

Fig. 4. Chunk storage system

file; *fileID*: 8 bytes - unique identification of file in the whole system ; *sha*256 : 32 bytes - hash value by using sha-256 algorithm of file data; *refFileID*: 8 bytes - id of file that have previous existed in

System and have the same $sha256$ - we treat these files as one, $refFileID$ is valid if it is greater than zero; $startChunkID$: 8 bytes - the identification of the first chunk of file, the next chunk will have id as $startChunkID + 1$ and so on; *numChunk*: 8 bytes - the number of chunks of the file; $fileSize$: 8 bytes - size of file in bytes; $status$: enum 1 bytes - the status of file, it has one in four values named: $EUploadingFile$ - when chunk are uploading to server, $ECompletedFile$ - when all chunk are uploaded to server but it is not check as consistent, $ECorruptedFile$ - when all chunk are uploaded to server but it is not consistent after checking, $EGoodCompleted$ - when all chunk are uploaded to server and consistent checking completed with good result.

Thus the size of FileInfo object - the meta-data of a file will be nearly the same for all file in the system, regardless of how large or small the file size is (the only difference meta-data of files is the length of $fileName$). By using this solution, we created a lightweight meta-data design when building a big file storage system. Fig 5 describes meta-data store system of BFC.

3.6. *Data distribution and replication*

Because BFC is built based on ZDB - a distributed key-value storage system. It is obvious that the meta-data of BFC is stored distributed and can be replicated for fault-tolerance and load-balancing. Store Services such as FileInfoService, ChunkStoreService distribute data using consistent-hashing which is proposed in[16]. Chain replication [28]is used to replicate key-value data. Each type of store service has its own distributed ZDB instances. Each ZDB instance has a range $[h_{lowerbound}, h_{upperbound})$ which is used to determine the range of key to store. If $hash(key)$ is in the range, it is stored in that instance. In BFC, file-id and chunk-id are auto increment integer keys. We can use simple hash function $hash(key) = key$ for consistent hashing. It is very easy to scale-out system in this case.

Fig 6 shows how data is distributed and replicated in the BFC.

3.7. *Uploading and deduplication algorithm*

Fig 7 describes an algorithm for uploading big file to BFC. Data deduplication is supported in BFC. There are many types and methods of data deduplication [26] which can work both on client-side or server-side. In BFC, we implemented it on server-side. We use a simple method with key-value store and SHA2 hash function to detect duplicate files in the whole system in the flow of uploading. A comparison between BFC and other cloud storage systems in deduplication is shown in Table 1 in Section 4

The upload flow on BFC cloud storage system has a little different between mobile client and web interface. On mobile client, after a file to upload is selected, we call it A, the client computes the SHA hash value of content of this file. After that, the client creates a basic information of file including file name, file size, SHA value. This basic information will be sent to server. At server-side, if data de-duplication mode is enabled, SHA value will be used to lookup associated fileID, if there is a fileID in the system with the SHA-value we call it B, this means that file A and file B are exactly the same. So we simply refer file A to file B by assigning the id of file B to $refFileID$ property of file A - a property to describe that a file is referenced to another file. The basic information will be sent back to client , and the upload flow complete, there is no more wasteful upload. In the case there is no fileID associated with SHA-value of file A or data de-duplication is disabled, the system will create some of new properties for the file information including the id of file, the id of first chunk using *IDGenerator* and number of chunk calculated by file size and chunk size. The client will use this information to upload file content to the server. Then, all chunks will be uploaded to the server.

This process can be executed in parallel to maximize speed. Every chunk will be stored in the storage system as a key-value pair, with the key is the id of chunk, and the value is data content of the chunk. When all chunk are uploaded to the system, there is a procedure to verify uploaded data such as verifying the equation of SHA-value calculated by client

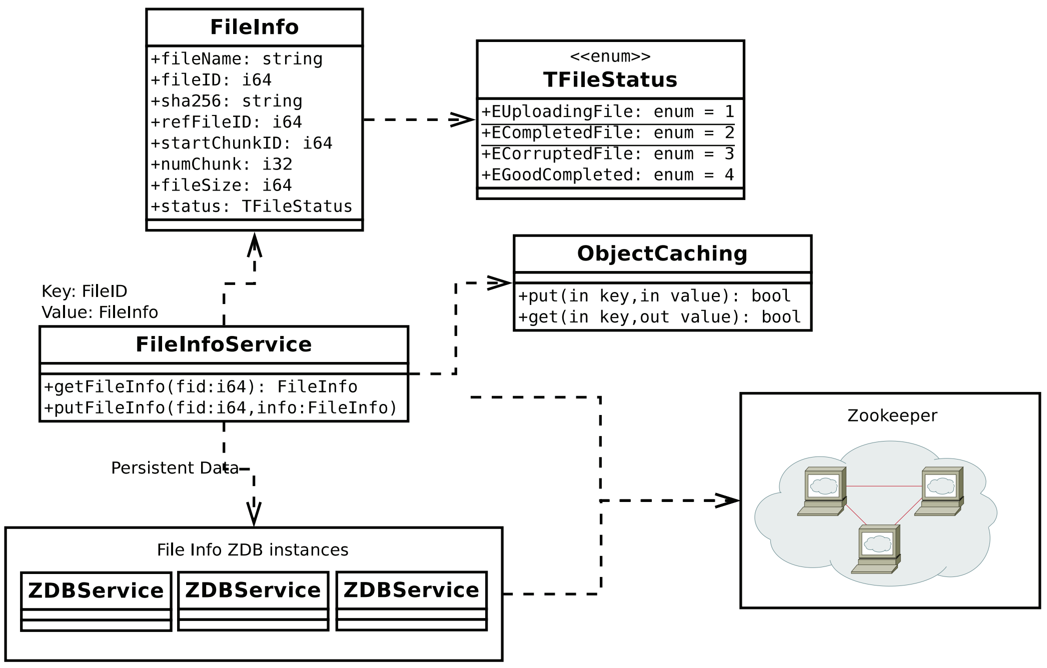

Fig. 5. Metadata storage system

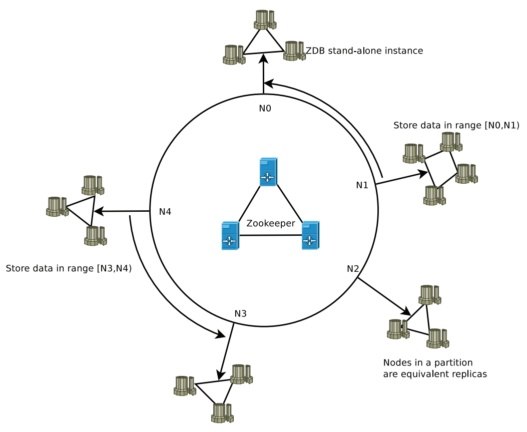

Fig. 6. Data partitioning and replication from [19]

and SHA-value of file created by uploaded chunk in server. If everything is good, the *status* field of File-Info is set to EGoodCompleted.

In web-interface client upload process, the client always uploads the file to server and saves it in a temporary directory. Then the server computes SHA hash value of the uploaded file. If there is any file in the system which has the same SHA value with it. Server will refer the uploaded file with this file and remove the file at temporary directory. Otherwise, a worker service called *FileAdder* will upload file to the system using similar algorithm of the mobile application client.

3.8. Downloading algorithm

Mobile clients of BFC have download algorithms described in Fig 8. Firstly, the client sends the id of file that will be downloaded to the server. The dispatcher server will check the session and number of connection from the client. If they are

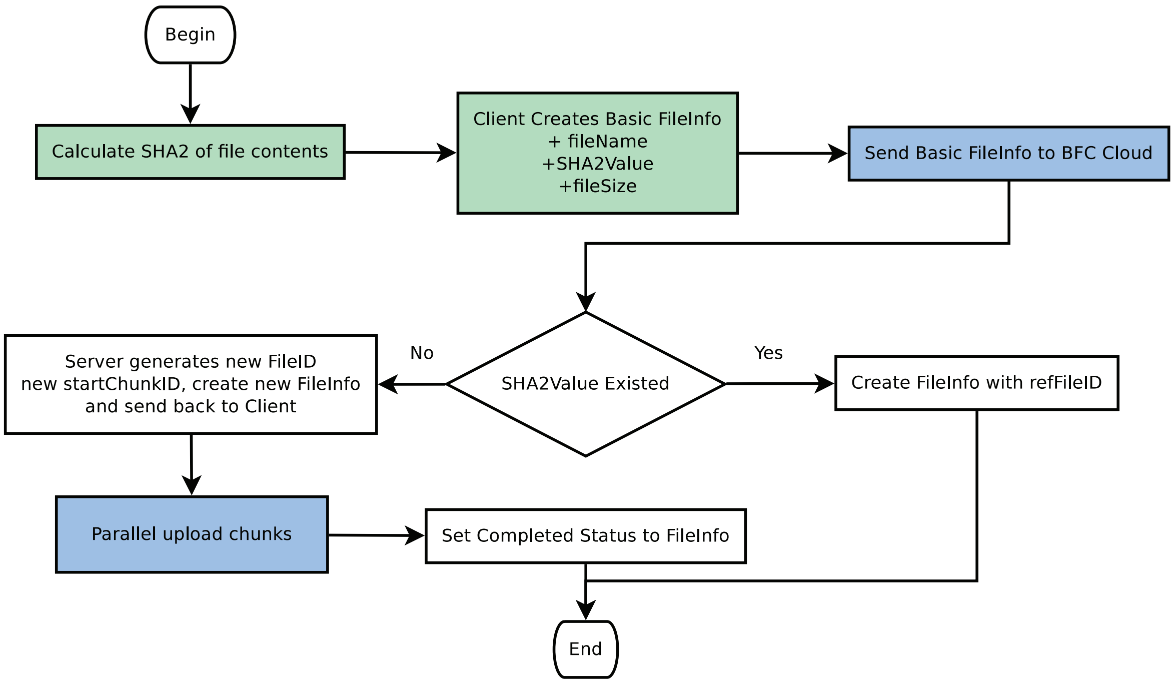
Begin
Calculate SHA2 of file contents
Client Creates Basic FileInfo + fileName +SHA2Value +fileSize
Send Basic FileInfo to BFC Cloud
SHA2Value Existed
No
Yes
Server generates new FileID new startChunkID, create new FileInfo and send back to Client
Create FileInfo with refFileID
Parallel upload chunks
Set Completed Status to FileInfo
End

Fig. 7. Uploading Algorithm of Application

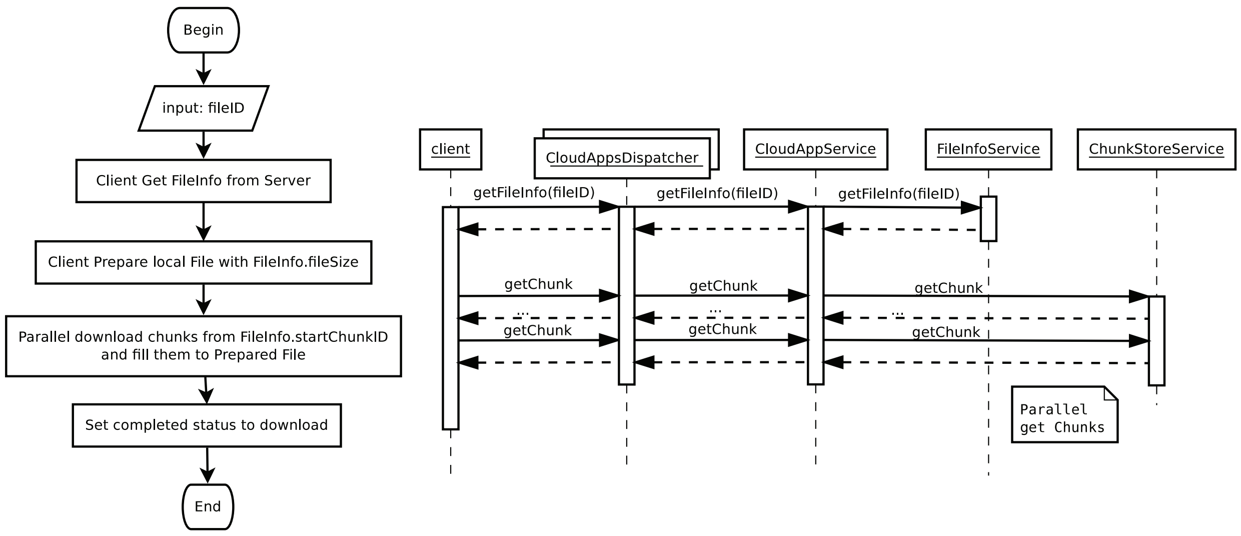

Fig. 8. Downloading Algorithm of Application

valid, the dispatcher sends download request to the *CloudAppsService* server, then it will lookup the file information in the FileInfoService which stores meta-data information with file-ID as a key. If *FileInfo* is existed with the requested file-ID, this information will be sent back to the client. The most important information of the file from FileInfo structure includes: first id of chunk (*chunkIdStart*), number of chunk (*chunkNumber*), size of chunk (*chunkSize*) and size of File ($fileSize$). The client uses these information to schedule the download process.

After that, the mobile client downloads chunks of files from ChunkStoreService via CloudAppDispatcher and CloudAppService, chunks with range ID from $chunkIdStart$ to $chunkIdStart + numberChunk - 1$ are concurrently downloaded in several threads, each chunk has a size of *chunkSize*, except last chunk. Native application will pre-allocate file in local filesystem with file-size specified in $fileSize$ field of FileInfo. Every downloaded chunk will be save directly to its position in this file. When all chunks are fully downloaded successful, the download process is completed.

3.9. Secure Data Transfer Protocol

Data confidentiality is one of strict requirements of cloud storage system. To ensure quality of service, a light-weight and fast network protocol for transfer data is also required. For web-interface and restful APIs, we support http secure protocol (https) to protect the connection from catching packets in all operations. In both desktop and mobile native applications, BFC Data transfered over Internet between client and server are encrypted using AES[8] algorithms with simple key exchange between client and server. We also use UDT [12] - an UDP-based protocol to use network bandwidth efficiently. This is detail of simple key exchange method:

- When an user login via https restful API, client receive Session-ID, User-ID, Secret-AES key, Public Key-Index. Secret-AES key is secret between client and server, it can be generated by client or server and stored on server as a key-list. It is used to encrypt chunks for transferring between client and server.
- In every operation such as uploading or downloading chunks, the data is encrypted using secret-AES key and transferred via network using UDT or TCP as client selected. Public Key-index is binded with encrypted packets for peer to determine secret AES key to decrypt received packets.

4. Evaluation

In this section, we firstly evaluate key-value stores with proposed design of BFC to examine which is the most appropriate key-value store for BFC. Then, we benchmark BFC with some opensource solutions for storing Big-File such as HDFS, Cassandra, MySQL's Blob in VNG's data center. Finally we present several scenarios to evaluate BFC with other personal cloud storage systems. It consists of theoretically comparing Metadata, evaluating deduplication ability. In a paper of Idilio Drago et al [5], many personal cloud storages were benchmarked in a black-box evaluation method. The test cases in [5] used files with size: 10kB, 100kB, 1MB to compare Dropbox, SkyDrive, Cloud Drive, Google Drive and Wuala. In this research, we deployed an instance of BFC system in Amazon EC2 to compare with Dropbox which uses Amazon EC2 and Amazon S3. Clients of both BFC and Dropbox run from Viet-Nam. According to paper [6] about some aspects inside Dropbox, we compared BFC's metadata with Dropbox. Then, we did experiments for comparing deduplication ability of BFC and other cloud storages such as Google Drive, Dropbox, OneDrive.

4.1. Evaluate Key-Value store for BFC

We setup a benchmark to evaluate performance of key-value stores that work in backends of BFC. This evaluation is useful for choosing the most appropriate key-value store for the design of BFC. We deployed BFC with difference backends in the same hardware and network environment described below:

- 1 Gigabit Network
- 04 Servers, each server has 32GB Ram, 4TBs Raid-5 HDD for storing chunks
- 02 Servers, each server has 64GB Ram, 500GB HDD for storing metadata

BFC deployment contains multiple instances of ZDBService [19]. They can be configured to use different key-value store engine: LevelDB, KyotoCabinet, ZDB, etc. In each key-value configuration , we evaluate throughput when upload, download file with different read/write load ratio. The throughput information is monitored when the data set growing by the time. The result is shown in Fig 9. It shows that BFC using ZDB has the best throughput. The throughput of BFC - ZDB is more stable when the data set grow. ZDB is designed to optimize storing and retrieving key-value pair with auto increment integer keys. And it fits with the specification of the keys in BFC's storage backends.

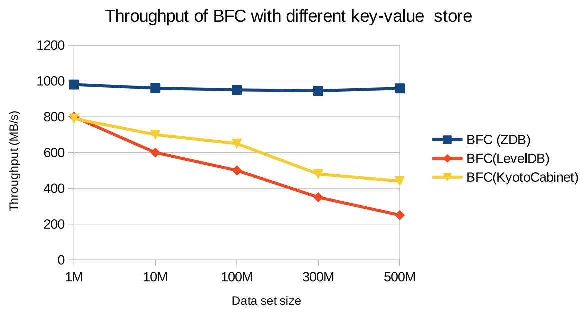

Fig. 9. Throughput of BFC when using different key-value store

4.2. *Locally performance comparison*

We setup a benchmark to compare BFC with some other opensource solution such as HDFS [25], Blob of MySQL [18] and Cassandra [17]. All these solutions in the benchmark were deployed in VNG data center with the same hardware configurations, then we measured average time to complete reading and writing files with different size.

To store big-files, we had to re-configure MySQL to change default maximum field size of tables (typically each field is configured to limit size of 16MBs). Cassandra often time out when we write big-value (big files data) into its columns and we removed it from the comparison. The result of the benchmark is shown in Fig 10, Fig 11 and Fig 12.

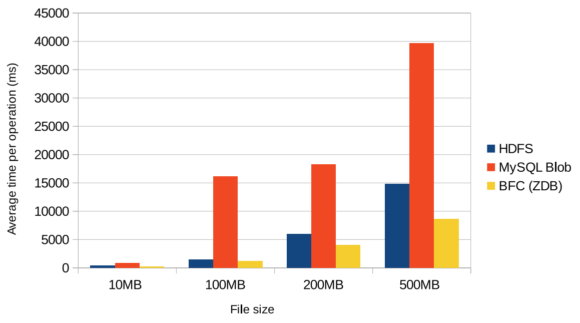

Fig. 10. Latency of Reading

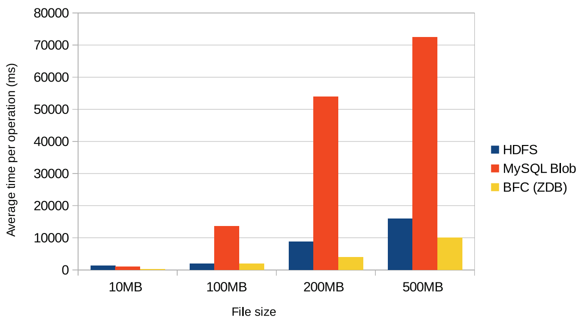

Fig. 11. Latency of Writing

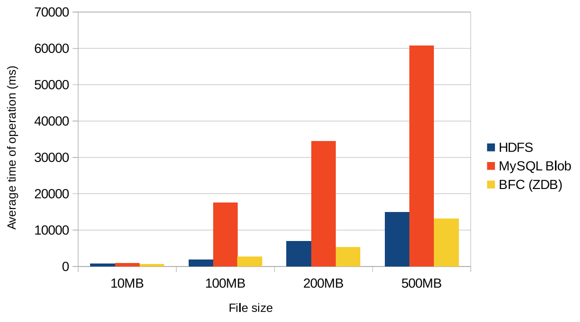

Fig. 12. Latency of concurrent R/W

4.3. *Metadata comparison*

Dropbox[6] is a cloud-based storage system that allows users to store documents, photos, videos and other files. Dropbox is available on Web interface, and many types of client softwares on desktop and mobile operating systems. The client supports synchronization and sharing between devices with personal storage. Dropbox were primarily written in Python. The native client uses third parties libraries such as wxWidgets, Cocoa, librsync. The basic object in the Dropbox system is a chunk of 4MB data. If a file is larger than this configured size, it will be split in several of basic objects. Each basic object is an independent element, which is identified by a SHA256 value and stored in Amazon Simple Storage Service (S3). Metadata of each file in Dropbox contains a list of SHA256 of its chunks [7,6]. Therefore, its size is linear to the size of file. For big-file, it has a big metadata caused by many of chunks and a long list of SHA256 values from them. In our research BFC has a fixed-size metadata of each file, so it is easier to store and scale storage system for big file. It reduces the amount of data for exchanging metadata between clients and servers. The comparison is shown in Fig 13.

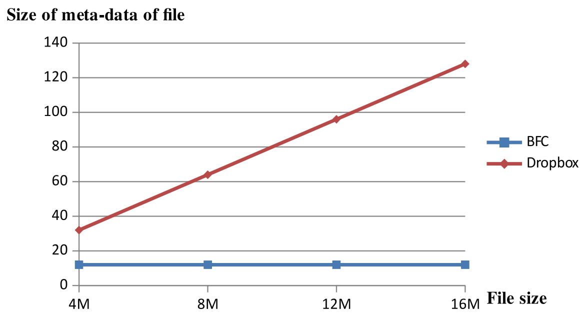

Fig. 13. Metadata comparison of BFC and DropBox

4.4. Deduplication

This comparison was done to study the deduplication ability of BFC and other cloud storages: Dropbox, OneDrive and Google Drive. We used WireShark [3] to capture network flow of cloud storage client application. To estimate the deduplication ability, we did following test cases: (1) A file is multiply uploaded to different folders by a User; (2) A file is multiply uploaded by different users. The result in Table 1 showed that Dropbox supports deduplication per user accounts, it could be done in client applications. BFC support a global deduplication mechanism, it saves the network traffic and internal storage space when many users store the same file content. Google Drive and OneDrive do not support deduplication.

5. Conclusion

BFC designed a simple meta-data to create a high performance Cloud Storage based on ZDB key-value store. Every file in the system has a same size of meta-data, regardless of file-size. Every big-file stored in BFC is split into multiple fixed-size chunks (may except the last chunk of the file). The chunks of a file have a contiguous ID range, thus it is easy to distribute data and scale-out storage system, especially when using ZDB. This research also brings the advantages of key-value store into big-file data store which is not supported big-value by default. ZDB[19] is used for supporting sequential write, small memory-index overhead. The data deduplication method of BFC uses the SHA-2 hash function and a key-value store to fast detect data-duplication on server-side. It is useful to save storage space and network bandwidth when many users upload the same static data. In the future, we will continue to research and improve our ideas for storing big data structure in a larger domain of applications, especially in the ”Internet of things” trend.

1. D. Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.
2. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
3. L. Chappell and G. Combs. *Wireshark network analysis: the official Wireshark certified network analyst study guide*. Protocol Analysis Institute, Chappell University, 2010.
4. V. Corporation. Zing me. `http://me.zing.vn`. Accessed October 28, 2014.
5. I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking personal cloud storage. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 205–212. ACM, 2013.

Table 1. Deduplication Comparison.

Deduplication	Dropbox	OneDrive	Google Drive	BFC
Single user	yes	no	no	yes
Multi-user	no	no	no	yes

6. I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.
7. Dropbox. Dropbox tech blog. `https://tech.dropbox.com/`. Accessed October 28, 2014.
8. P. FIPS. 197: the official aes standard. *Figure2: Working scheme with four LFSRs and their IV generation LFSR1 LFSR*, 2, 2001.
9. S. Ghemawat and J. Dean. Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values. `https://github.com/google/leveldb`. Accessed November 2, 2014.
10. S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.
11. S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.
12. Y. Gu and R. L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Computer Networks*, 51(7):1777–1799, 2007.
13. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.
14. F. Inc. Facebook. `http://facebook.com`, 2014.
15. P. Jin, P. Yang, and L. Yue. Optimizing b+-tree for hybrid storage systems. *Distributed and Parallel Databases*, pages 1–27, 2014.
16. D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203–1213, 1999.
17. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
18. A. MySQL. Mysql the world's most popular open source database. *Internet WWW page, at URL: http://www. mysql. com*, 2015. Accessed April 28, 2015.
19. T. Nguyen and M. Nguyen. Zing database: high-performance key-value store for large-scale storage service. *Vietnam Journal of Computer Science*, pages 1–11, 2014.
20. T. T. Nguyen, T. K. Vu, and M. H. Nguyen. Bfc: High-performance distributed big-file cloud storage based on key-value store. In *Proceedings of the 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2015)*, pages "1–6", Takamatsu, Kagawa, Japan, 2015. 'IEEE/ACIS'.
21. P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
22. M. Placek and R. Buyya. A taxonomy of distributed storage systems. *Reporte técnico, Universidad de Melbourne, Laboratorio de sistemas distribuidos y cómputo grid*, 2006.
23. F. PUB. Secure hash standard (shs). *FIPS PUB 180-4*, 2012.
24. S. Shepler, M. Eisler, D. Robinson, B. Callaghan, R. Thurlow, D. Noveck, and C. Beame. Network file system (nfs) version 4 protocol. *Network*, 2003.
25. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
26. J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl. A secure data deduplication scheme for cloud storage. In *International Conference on Financial Cryptography and Data Security*, pages 99–118. Springer, 2014.
27. M. Szeredi et al. Fuse: Filesystem in userspace, 2010. Accessed April 28, 2015.
28. R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.
29. S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.
30. S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.