國立交通大學

資訊學院 資訊學程

碩士論文

雲端儲存的檔案去重複

File Deduplication with Cloud Storage File System

研究生：古展易

指導教授：袁賢銘 教授

中華民國一百零二年三月

雲端儲存的檔案去重複

File Deduplication with Cloud Storage File System

研究生：古展易　Student：Chan-I Ku

指導教授：袁賢銘　Advisor：Shyan-Ming Yuan

國立交通大學

資訊學院 資訊學程

碩士論文

A Thesis

Submitted to College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science
Mar 2013

Hsinchu, Taiwan, Republic of China

中華民國一百零二年三月

雲端儲存的檔案去重複

研究生：古展易 指導教授：袁賢銘 教授

國立交通大學 資訊學院 資訊學程碩士班

摘要

Hadoop Distributed File System (HDFS)被運用在解決大量的資料儲存問題，但是並未提供對重複檔案的處理機制，此研究以 HBASE 架構虛擬中介層檔案系統(Middle layer file system)，在 HDFS 中達到 File Deduplication 的功能，依照應用需求的可靠度要求不同提出兩種架構，一者為不許可有任何錯誤的 RFD-HDFS(Reliable File Deduplicated HDFS)，另一者為可容忍極少錯誤的 FD-HDFS (File Deduplicated HDFS)兩種解決方案，除了空間複雜度上的優勢，也探討比較其帶來之邊際效益。

假設一個內容完全相同的熱門影片被一百萬個用戶上傳到 HDFS，經過 Hadoop replication 成三百萬個檔案來儲存，這是非常浪費磁碟空間的做法，唯有雲端除去重複才能有效裝載，經此將只占用 3 個檔案空間，也就是達成百分百去除重複檔案的效用。

實驗架構為一個雲端文獻系統，類似 EndNote Cloud 版，模擬研究生將資料與雲端同步時，與海量數據庫的群聚效應。

關鍵字：HDFS, Data Deduplication, Cloud Computing, Single instance storage.

File Deduplication with Cloud Storage File System

Student：Chan-I Ku　　Advisor：Dr. Shyan-Ming Yuan

Degree Program of Computer Science

National Chiao Tung University

ABSTRACT

The Hadoop Distributed File System (HDFS) is used to solve the storage problem of huge data, but does not provide a handling mechanism of duplicate files. In this study, the middle layer file system in the HBASE virtual architecture is used to do File Deduplicate in HDFS, with two architectures proposed according to different requires of the applied requirement reliability, therein one is RFD-HDFS (Reliable File Deduplicated HDFS) which is not permitted to have any errors and the other is FD-HDFS (File Deduplicated HDFS) which can tolerate very few errors. In addition to the advantage of the space complexity, the marginal benefits from it are explored. Assuming a popular video is uploaded to HDFS by one million users, through the Hadoop replication, they are divided into three million files to store, that is a practice wasting disk space very much and only by the cloud to remove repeats for effectively loading. By that, only three file spaces are taken up, namely the 100% utility of removing duplicate files reaches. The experimental architecture is a cloud based documentation system, like the version of EndNote Cloud, to simulate the cluster effect of massive database when the researcher synchronized the data with cloud storage.

Key Words: HDFS, Data Deduplication, Cloud Computing, Single instance storage.

Acknowledgements

首先要感謝袁賢銘教授對我的指導，因為在職的關係，比較沒有機會和分散式實驗室的小同學交流，所以就不斷勞煩袁教授，教授不但一一解決了我的問題，更提供許多明確的指導，讓我在研究時快速的釐清問題和方向。

論文口試審查期間王尉任老師，梁凱智博士，林縣城副總提出許多寶貴的意見，讓我更有效的闡述研究的效益。博士班羅國亨學長積極快速的協助和檢視下，讓我發現研究的關鍵問題，也感謝洪大均同學在口試期間的幫忙，謝謝我的爸爸媽媽家人朋友同事同學們給我的支援和幫助。

年輕時因為排斥填鴨式教育和家庭因素未曾用心在學業，出了社會才確信自己的天賦和能力，程式和軟體設計不但成為我的職業，也成了興趣，碩士學歷對我而言只是在補償自己過往的遺憾，三十好幾才完成碩士學位並無法挽回過往我錯失的機會，也沒有值得驕傲的地方,只是實現自己對自己的諾言,人生很多事情都是沒有意義徒勞無功的，相對的任何事物也都能找出它的價值，經過這些歷練就是我最大的收穫。

List of Contents

Acknowledgements III

List of Contents IV

List of Figure VI

List of Tables VII

1 Introduction 1

  1.1 Motivation 1

  1.2 Objectives 3

  1.3 Importance and Contribution 4

  1.4 Outline of the thesis 5

2 Background And Related Work 6

  2.1 Data Deduplication 6

    2.1.1 Detection of the Same Data / Similar Data 6

    2.1.2 Source Deduplication(Client)/ Target Deduplication (Storage Devices) 7

    2.1.3 In-line Processing/Post Processing 7

    2.1.4 Keep old data/new data 8

    2.1.5 Whole File Deduplication vs. Chunk Data Deduplication 8

    2.1.6 Cloud Storage 9

  2.2 HDFS 9

  2.3 Hash-SHA2 10

  2.4 HBase 11

3 System Design and Implementation 12

  3.1 Overview 12

3.1.1 HDFS 12
3.1.2 RFD-HDFS 12
3.1.3 FD-HDFS 13
3.2 Architecture 14
3.2.1 HDFS 14
3.2.2 RFD-HDFS 16
3.2.3 FD-HDFS 19
3.3 Algorithm 21
3.3.1 File Writing 21
3.3.2 File reading 22
3.3.3 Background Worker 22
3.3.4 FD-HDFS Function Presentation 23
4 Experiments 26
4.1 Environment, and Setting 26
4.2 Experiment 1 27
4.3 Result of Experiment 1 28
4.4 The setting of Experiment 2 30
4.5 Result of Experiment 2 31
4.6 The setting of Experiment 3 33
4.7 Result of Experiment 3 34
4.8 Overhead 35
4.9 Multi-Threading(Multi-Users) 37
5 Conclusion 38
6 Future Work 40
7 Reference 41

List of Figure

Figure 1: The DFS Usage in duplicate files 3

Figure 2 : Use Case of HDFS 15

Figure 3: HDFS Write 15

Figure 4: HDFS Read 16

Figure 5: The architecture of RFD-HDFS 17

Figure 6: Use case of RFD-HDFS 17

Figure 7: RFD-HDFS Write 18

Figure 8: RFD-HDFS Read 18

Figure 9: Use case of FD-HDFS 19

Figure 10: FD-HDFS Write 20

Figure 11: FD-HDFS Read 20

Figure 12: The writing of RFD-HDFS 21

Figure 13: The write of FD-HDFS 21

Figure 14: The reading of RFD-HDFS 22

Figure 15: The reading of FD-HDFS 22

Figure 16: The background worker of RFD-HDFS 23

Figure 17: File List Preview 24

Figure 18: The actual files in HDFS 24

Figure 19: The file information in HBASE table 25

Figure 20: Two rows in table and a single file stored 25

Figure 21: The column Family was delete, but not file in HDFS 26

Figure 22: Ex1-DFS usage 30
Figure 23: Ex2-DFS usage 33
Figure 24: Ex3-DFS usage 35
Figure 25: Time usage for Hash and Comparison 37

List of Tables

Table 1: Hadoop Nodes 26
Table 2: The schema and samples of HBase Table 27
Table 3: Ex1-User file list 27
Table 4: Ex1-DFS usage 29
Table 5: Ex2-User file list 30
Table 6: Ex2-DFS usage 32
Table 7: Ex3-HDFS List 33
Table 8: Ex3-DFS usage 34
Table 9: Sample files for overhead testing 36
Table 10: Result of overhead test 36
Table 11: Compare with HDFS 38

1 Introduction

As to solving the problems of massive data storage and computing, Google's distributed Google file system (GFS) at 2003[1], proposed a large number of data processing model MapReduce at 2004, and a large structured data storage system based on GFS: BigTable at 2006[2], Corresponds to Google, an open-sourced organization- Apache Also continued to establish incorporating the corresponding GFS's HDFS[3], corresponding BigTable of HBase[4].

1.1 Motivation

The enterprise has gradually begun to test and apply analytical uses Hadoop as the storage of large amounts of data and information. However, we found that the growth rate of the data is much larger than the default value of the 3 time DFS Replication.

The Data Deduplication technology is widely used in business File Server, Database, NAS (RAID), Backup Devices or lots storage devices, but there is no any implement in Hadoop.

Hadoop is widely used in the kinds of distributed computing and massive data storage, through the following simple experiment that, HDFS did not apply any Common Data Deduplication technology.

When we uploaded three files to the Hadoop Distributed File System (HDFS), which the file’s name is different but content is the same..

To exclude the deliberately manufactured to ensure reliability(HDFS Replication) of the data copy of file duplication, three different file names with the same file contents are generated in a duplicate copy of files. However, the confirmed HDFS does not have any mechanism to deal with file duplication and it wastes storage space. Here is the Hash sum by SHA-2[5] from the file apache-solr-4.0.x.tgz, in which we can deduce the three files are identical, and only their file names are different since the three values of hash sum are the same.

```
[root@namenode ~]# sha512sum ~/Downloads/apache-solr-4.0.0.tgz
e37c36f910f922a35877431e44b77f3a035c4ce47bdf78ffc88afcf6c97f42ad6c54cec3f
01b3093f886099688a3e90603f5c879d03cf629621861817973c631
/root/Downloads/apache-solr-4.0.0.tgz
[root@namenode ~]# sha512sum ~/Downloads/apache-solr-4.0.1.tgz
e37c36f910f922a35877431e44b77f3a035c4ce47bdf78ffc88afcf6c97f42ad6c54cec3f
01b3093f886099688a3e90603f5c879d03cf629621861817973c631
/root/Downloads/apache-solr-4.0.1.tgz
[root@namenode ~]# sha512sum ~/Downloads/apache-solr-4.0.2.tgz
e37c36f910f922a35877431e44b77f3a035c4ce47bdf78ffc88afcf6c97f42ad6c54cec3f
01b3093f886099688a3e90603f5c879d03cf629621861817973c631
/root/Downloads/apache-solr-4.0.2.tgz
```

Copy these 3 files to HDFS by shell command:

```
Hadoop dfs -copyFromLocal ~/Downloads/apache-solr-4.0.0.tgz test
Hadoop dfs -copyFromLocal ~/Downloads/apache-solr-4.0.1.tgz test
Hadoop dfs -copyFromLocal ~/Downloads/apache-solr-4.0.2.tgz test
```

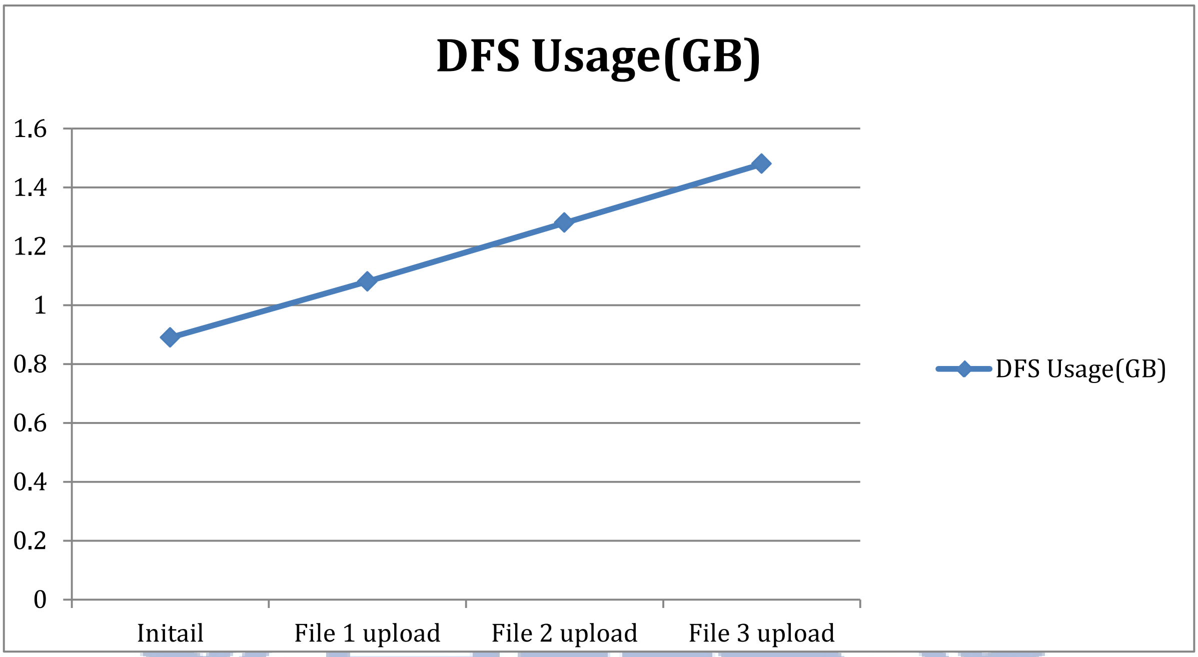


Figure 1: The DFS Usage in duplicate files

By DFS Usage, the three each 0.2G identical-contented files occupy a 0.6G DFS use of space, Obviously HDFS is occupied by duplicate files.

1.2 Objectives

In this thesis, we will use the tree view of the file system architecture based on HBASE Table, In the following, we will create the Mapping table between virtual path and Hash sum, The underlying Key-Value(NoSQL) storage[6] query mechanism has been packaged into the tree file folder access concept, by which the users can access FD-HDFS by the original tree folder concept file.

However, due to the presence of Hash, there are possibly the Hash Table balance problems and birthday attack[7]. MD5 and SHA1 can develop the same HASH file by violent calculation[8]. However, the SHA2 and SHA3 method of artificially repeating the HASH value are still not found but the Collision cannot be avoided. Therefore, two architectures are proposed in accordance with the reliability of application requirements: One is the

RFD-HDFS (Reliable File Deduplicated HDFS) which does not allow any error. The other is FD-HDFS (File Deduplicated HDFS) which can tolerate very few errors. In addition to reduce of space complexity, the marginal benefits will be explored and compared.

In the normal distribution of user file holdings; there is inevitably the intersection in the files held by different users and even multiple intersections. In a variety of environments, the probability of different intersections is higher for videos, music, documents, and e-books, that user repeat cross-holdings, and the unique probability for the machine-generated files is generally lower.

For example, if time is frozen, there is limited number of video files all over the world to be load by Hadoop, such as Google YouTube using GFS. As for the technical documents, reports, patents, journals, research reports, test reports, assessment data, procurement reference documents, pictures and other electronic files are centralized and provide query interface. Or, the cloud based EndNote can be constructed. When the graduate student synchronizes the paper to the cloud terminal, through the Deduplication technology, massive data storage becomes possible and efficient that the MapReduce computing capability can be used more to construct the data document Meta-Data so as to achieve the efficacy of quick searches and optimize Data-Mining. The premise of these benefits, however, is framed in the effectively-used storage space system, which is the purpose to achieve in this study.

1.3 Importance and Contribution

Although the architectures in the traditional hard disk array has the higher characteristics of reducing data and the use of cutting and version stack, these algorithms cannot be applied to the Hadoop since Hadoop has its unique cut demand. The traditional hard disk array cannot be elastic expanded and has no clustering effect of mass data, therefore, this framework maintains the Replication that HDFS maintains the reliability and there is no side effects sacrificing reliability in exchange for space-saving and no the Over Head brought by complicated algorithm.

Assuming a popular video is uploaded to HDFS by one million users and stored by cutting into three million files through Hadoop replication, it is a practice very wasting disk space. Only after the cloud removes duplicate files can all files be effectively loaded. Through this system, only three file spaces are occupied, namely reaching the utility of completely removing duplicate files.

In the FD-HDFS architecture, the Client terminal can achieve the effect of Source Deduplication by HASH sum, so as to save upload time and bandwidth. The RFD-HDFS can completely ensure data accuracy and achieve the benefits of de-duplication.

1.4 Outline of the thesis

Chapter 1 has presented a brief introduction to the concepts of uncertainty, and the importance of uncertainty analysis in the context of transport-transformation models. It has also discussed the steps involved in a systematic uncertainty analysis, and the associated limitations. Additionally, Chapter 1 summarizes the objectives and presents a brief overview of this thesis.

Chapter 2 presents the relevant background information in the following order: Data Deduplication, HDFS, Hash-SHA2, Hash Collision and HBase.

Chapter 3 shows the original HDFS model architecture, the RFD-HDFS model architecture, and the FD-HDFS model architecture.

Chapter 4, the experiment environment set with: 1. 20 users upload the folders when there is no data in the cloud. 2. 20 users upload data when the database has a certain amount of data. 3. The crossover experiment, in which the same user samples 10-100% of the precursor in the database.

Chapter 5 presents the conclusions of this thesis, and recommendations for future work. This is followed by bibliography.

Chapter 6 future works of the thesis.

2 Background And Related Work

2.1 Data Deduplication

The hard disk drive, Disk RAID, NAS, Type storage or Storage Server…, we can do broadly the data deduplication job to do the following classification.

2.1.1 Detection of the Same Data / Similar Data

The same data detection mainly includes two levels, the same File and the same Data Block. In the technology of the whole file detection[9], the Data Mining is conducted through the hash technology[5]. In the same Data Block detection, the fingerprint is checked through the fixed-sized partition or the check and deletion of duplicate data are conducted through the detection technology of content-defined chunking and the sliding block technology.

The similar data detection uses similar data characteristics, through the shingle technology and the bloom filter technology, the duplicate data which the same data detection cannot detect is found out. For similar data, the delta technology is used to encode, minimize, and compress similar data, further reducing the storage space and the network bandwidth usage.

2.1.2 Source Deduplication(Client)/ Target Deduplication (Storage Devices)

In typical storage applications such as backup and replication, data is moved from Source to the target storage devices through the network. Therefore, the source end is the front-end application Host or the Backup server to produce the raw data. The target end is the ultimate storage equipment, such as VTL or disk arrays.

But at the front end where the data deletion calculus is conducted, the deletion computing is first conducted before the data is sent to the network. Therefore, it has the advantage of saving the network bandwidth and the upload time, but the deletion computing will occupy the computing resources in the front end of the host. As for the advantages and disadvantages of Target Deduplication, although the network bandwidth cannot be saved, the resources of the front-end host will not be consumed.

2.1.3 In-line Processing/Post Processing

Online In-line processing means the deletion computing is synchronously executed when the data is performed with backup, copy or writing to the disk. In other words, when the data is copied for the preparation of sending it to the Destination through the network, or the back-end storage device receives the Source data via the Internet and prepare to write to the disk, the De-Dupe system will conduct data content comparison and deletion computing at the same time. Processing after writing in means after the data is written to the disk, it is started by instruction, or the deletion computing is conducted for the data stored on the disc in the customized scheduled startup De-Dupe system.

The advantages and disadvantages of online In-line processing and processing after writing are just at the opposite. Since data comparison and deletion computing quite consume processor resources, if the online real-time processing architecture is adopted, the system performance will be clearly temporized so that the backup speed will be delayed. But relatively, since the deletion computing has been conducted before the data is written to the disc, it occupies less space. In comparison, although the system performance will not be affected in processing after writing in, we can choose off-peak hours to start the De-Dupe. But when the data is written to the disc, the original form without deletion is maintained, so that the same storage space is occupied like the front-end and the reduction effect is shown after the De-Dupe is started up. Therefore, the De-dupe products in processing after writing in; the users must prepare a larger temporary storage space.

2.1.4 Keep old data/new data

De-Dupe technology will delete duplicate data, and there are two deletion ways:

One is to retain old data. When new information is determined to be the repeat of the old data, the system will remove new data and create an index pointing to the old data. The other is to retain new information. When the new data is determined as the repeat with the old data, the old data will be deleted and the index will be pointed to the new location.

2.1.5 Whole File Deduplication / Chunk Data Deduplication

DUTCH T. MEYER[10] find that whole-file deduplication together with sparseness is a highly efficient means of lowering storage consumption, even in a backup scenario. It approaches the effectiveness of conventional deduplication at a much lower cost in performance and complexity. The environment we studied, despite being homogeneous, shows a large diversity in file systems and file sizes. These challenges, the increase in unstructured files and an ever-deepening and more populated namespace pose significant challenge for future file system designs. However, at least one problem – that of file fragmentation, appears to be solved, provided that a machine has periods of inactivity in which defragmentation can be run.

There should be more trouble in reliability and compatibility at the peer project of Hadoop: Nutch, Hive, Pig …

2.1.6 **Cloud Storage**

These above-mentioned technologies are widely used in the storage devices of enterprise File Server, Database, NAS (RAID), and Backup Devices. However, there is no related implementation in Hadoop since HDFS has its special block mode and network Topology[11] demand to ensure the reliability of the copy. So, the similar data detection is not the conditions to remove duplicate files. In too much emphasizing on the space use of algorithm, the questions of reliability and the hard error reversion will be relatively occur and the overhead will be produced in effect. Therefore, here the same data detection is used to apply the detection technology for the same file data in HDFS, and the File Level Deduplication is conducted in the methods of HASH and Stream Compare.

2.2 HDFS

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware[3], and be used to replace the high-priced server; HDFS is highly Fault-tolerant and designed to be deployed on low-cost hardware. Used to replace the high-priced Disk Raid, HDFS provides application data with high throughput access, to replace the hardware routing shunt dispersed bandwidth and server load. There are automatic propagation and flexibility to increase or decrease for mass storage.

MapReduce may analyze the data and create the meta-data of file for file searching, and HDFS is base storage for MapReduce.

HDFS File access can be achieved through the native Java API, the Thrift API to generate a client in the language of the users' choosing (C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, and OCaml), the command-line interface, or browsed through the HDFS-UI webapp over HTTP.

2.3 Hash-SHA2

In cryptography, SHA-2 is a set of cryptographic hash functions (SHA-224, SHA-256, SHA-384, SHA-512) designed by the National Security Agency (NSA) and published in 2001 by the NIST as a U.S. Federal Information Processing Standard[5]. SHA stands for Secure Hash Algorithm. SHA-2 includes a significant number of changes from its predecessor, SHA-1. SHA-2 consists of a set of four hash functions with digests that are 224, 256, 384 or 512 bits.

In 2005, security flaws were identified in SHA-1, namely that a mathematical weakness might exist, indicating that a stronger hash function would be desirable. Although SHA-2 bears some similarity to the SHA-1 algorithm, these attacks have not been successfully extended to SHA-2.

In computer science, a collision or clash is a situation that occurs when two distinct pieces of data have the same hash value, checksum, fingerprint, or cryptographic digest.[12]

Collisions are unavoidable whenever members of a very large set (such as all possible person names, or if this was sent to other people, or all possible computer files) are mapped to a relatively short bit string. This is merely an instance of the pigeonhole principle.

The impact of collisions depends on the application. When hash functions and fingerprints are used to identify similar data, such as homologous DNA sequences or similar audio files, the functions are designed so as to maximize the probability of collision between distinct but similar data. Checksums, on the other hand, are designed to minimize the probability of collisions between similar inputs, without regard for collisions between very different inputs

How big is SHA2-512? How often does the hash collision happen?

Total Bits of $SHA512 = 2^{512} \approx 10^{155} Bit$

If we store 1 Million files into hash table, the chance of hash collision for next store operation should be $10^{6}/10^{155}$=10^{-149}

2.4 HBase

Apache HBase: random, real-time read/write access to your Big Data[4]. This project's goal is the hosting of very large tables -- billions of rows X millions of columns -- atop clusters of commodity hardware. Apache HBase is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable: A Distributed Storage System for Structured Data by Chang et al. Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS.

HBase uses a data model very similar to that of Bigtable. Users store data rows in labeled tables. A data row has a sortable key and an arbitrary number of columns. The table is stored sparsely, so that rows in the same table can have crazily-varying columns, if the user likes. At its core, HBase/BigTable is a map. Depending on your programming language background, you may be more familiar with the terms associative array (PHP), dictionary (Python), Hash (Ruby), or Object (JavaScript).

A table's column families are specified when the table is created, and are difficult or impossible to modify later. It can also be expensive to add new column families, so it's a good idea to specify all the ones you'll need up front.

Fortunately, a column family may have any number of columns, denoted by a column "qualifier" or "label".

All data is versioned either using an integer timestamp (seconds since the epoch), or another integer of your choice. The client may specify the timestamp when inserting data.

3 System Design and Implementation

3.1 Overview

3.1.1 HDFS

In uploading files directly to HDFS, it is the original architecture of HDFS[13]. The Hadoop API provides a Shell command and Java API as a file management interface.

The version of Hadoop is 1.03; the version of HBase is 0.94.3.

3.1.2 RFD-HDFS

In the applications which require precise calculation without any errors, such as financial computing, errors are definitely not allowed in the computing system. Due to the SHA Hash Collision, the conflict probability still cannot be ignored even though it is very low (depending on the algorithm). In binary comparison, in order to ensure data accuracy, time and resources will be wasted for the file comparison. Therefore, the binary comparison circuit or the MapReduce cluster computing capacity[14] can be used to speed up file comparison. At the same time, the Post Processing method is adopted to reduce user's time for waiting- files are first uploaded to the temporary storage pool, waiting for the background worker for the implementation of file comparison. In this study, the Stream Comparison is used to partially retrieve data fragments to conduct Binary Compare in the sequential serial method.

Three phases are divided to determine whether the files are the same:

1. If the Hash value exists.
2. If the File Sizes are the same.
3. Gradually and sequentially executing Stream Comparison.

Once any difference is found in the comparison work in each phase, the comparison will be immediately stopped so as not to consume computing power and occupy resources. For the Collision policy of the Duplicate files, if the SHA value of the file is the same, the file size is the same. At the time, it is necessary to first put the files in the Storage Pool, waiting for the Background Process to conduct the Stream comparison to decide whether the hash collision policy is started. Therein, the used file path is appended after the file name as a handling strategy of hash collision.

3.1.3 **FD-HDFS**

In the application where little errors can be tolerated, such as Web information extraction (for search engine use) and vocabulary and semantic analysis, the repeat Collision less likely occurs and the judgment result application will not be affected. The HASHs with the same fingerprints are regarded as the duplicate files. Thus, the effect of Source Deduplication can be reached, the effect of reducing the network bandwidth and saving upload time is achieved, and even the burden of NameNode and HBase can be reduced.

For example, web crawler software Nutch[15] daily needs to capture page files to HDFS. In accordance with the comparison of the HASH value and the HBASE database, it can quickly learn if the website content changes so that the time for Binary Compare can be saved and the target can be retrieved, such as directly generating the SHA value from the source end. If the source SHA does not change, the time of uploading Full Content will be saved further. If Hash generates program to implant the Host from source, the loading of NameNode and HMaster can be eased so that a crawl for a website becomes the crawl for new added and changed files, which saves not only upload time and the bandwidth but server side Loading.

3.2 Architecture

3.2.1 HDFS

In the use case of HDFS, users may access file by the Hadoop shell command or Hadoop API.

HDFS: Hadoop Distributed file system.

Write: The API of file write into HDFS (Upload).

Read: The API of file read from HDFS (Download).

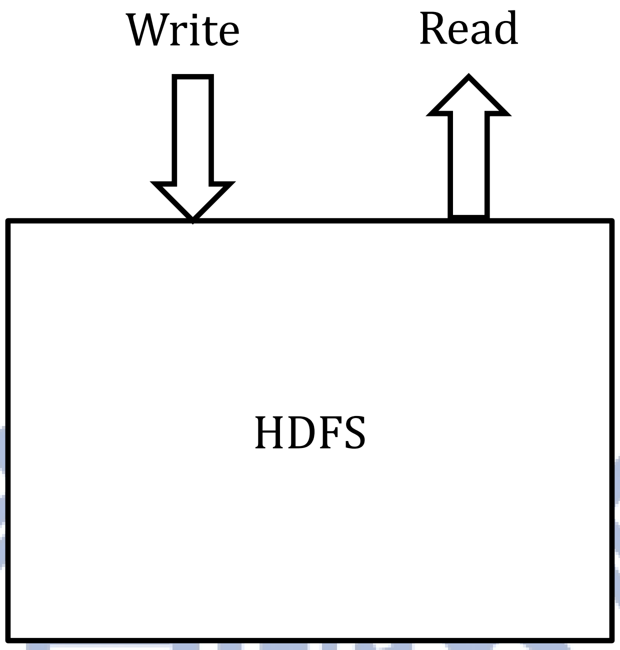


Figure 2 : Use Case of HDFS

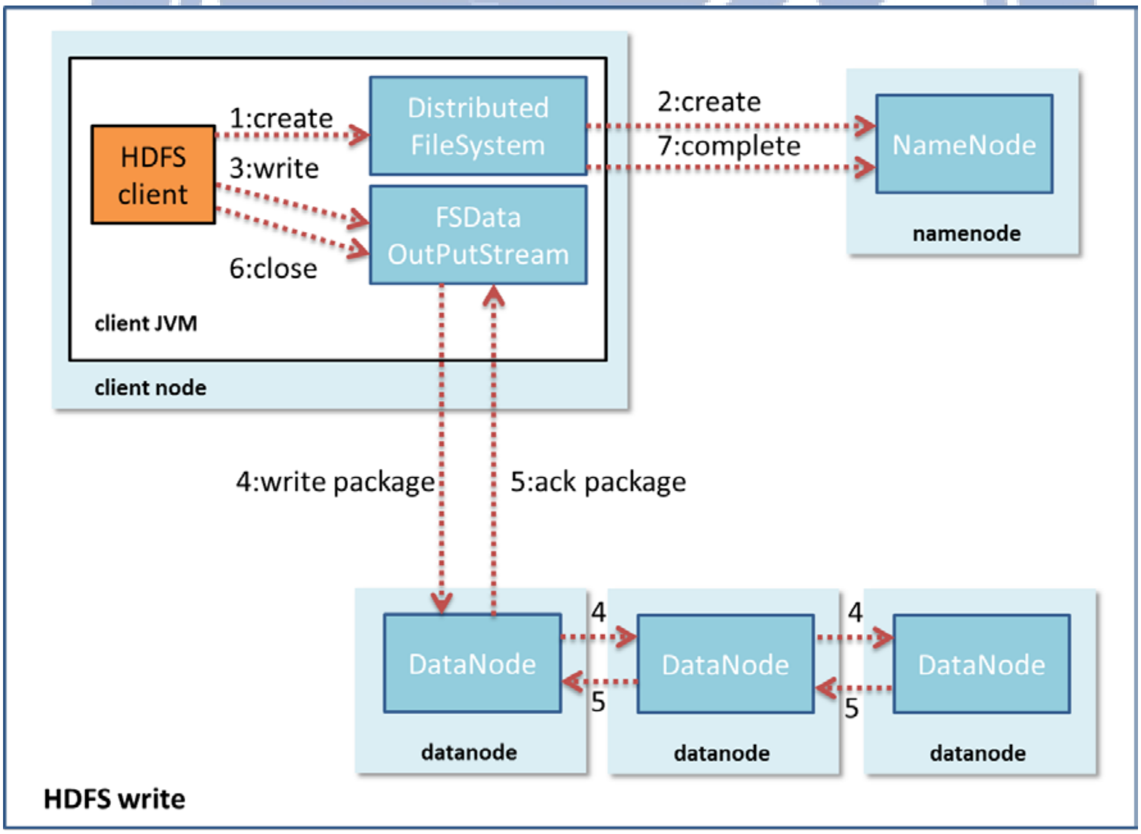


Figure 3: HDFS Write

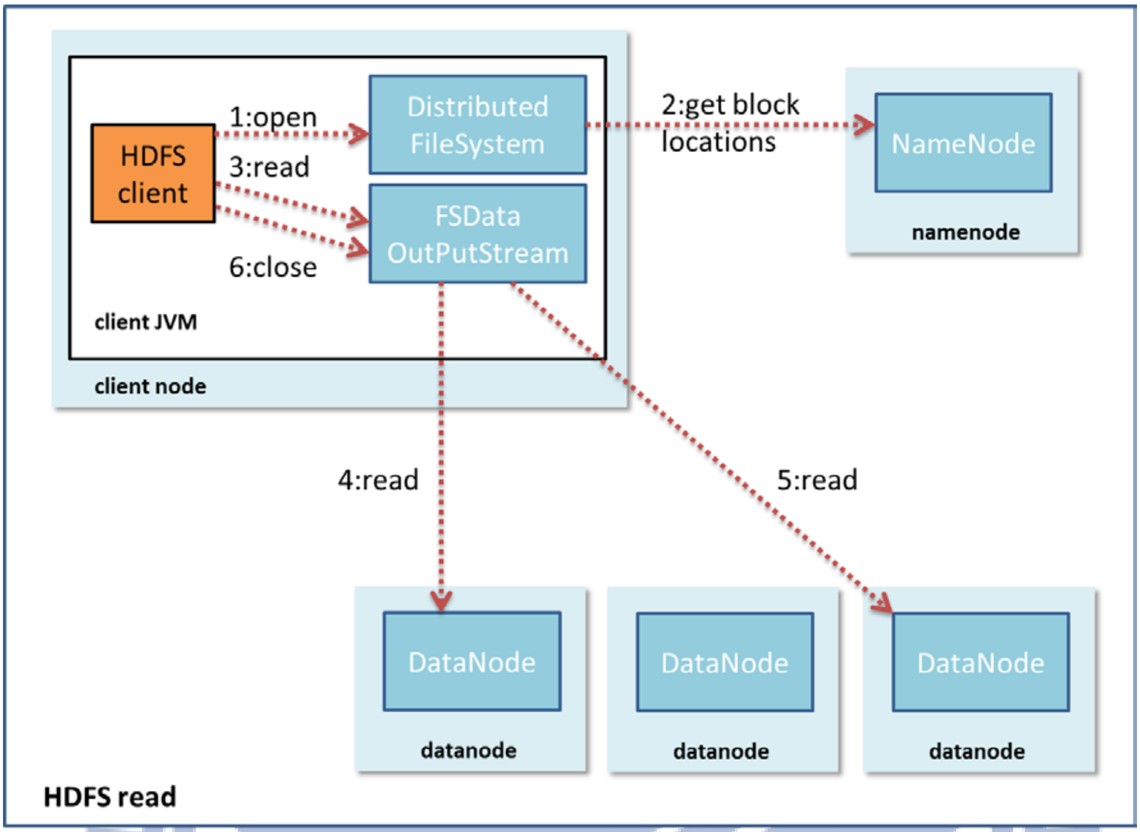


Figure 4: HDFS Read

3.2.2 **RFD-HDFS**

RFD-HDFS: A middle layer between the client and HDFS, Provide a viewpoint of visual HDFS, if client try to access the file over RFD-HDFS, the files Deduplication is enabled.

HBase: The HBase table records the mapping between Hash key and Full Path of file.

Hash Generator: In the hash key generator function for file, it could be present by SHA2, SHA3…

Binary comparer: The comparer could be a circuit of hardware or MapReduce function of Hadoop.

Storage Pool: A Temporary file pool for post processing, the binary comparer will load the file and do comparisons in background. All files store into pool will be log for tracking, and file could be roll back for fail over.

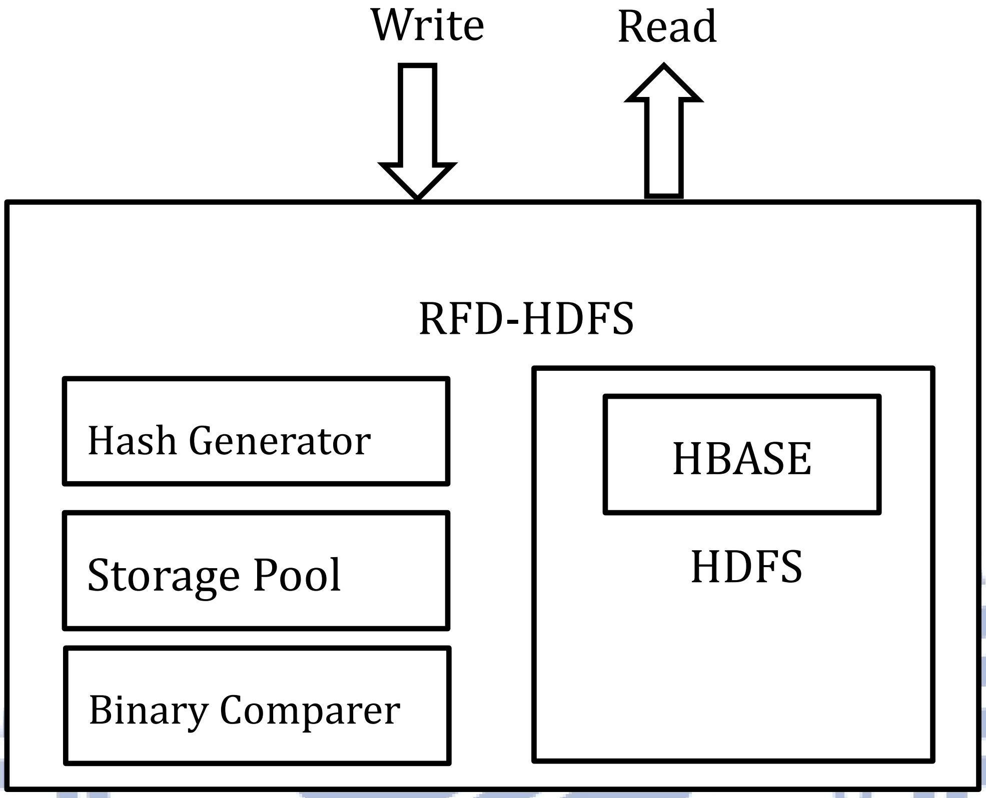


Figure 6: Use case of RFD-HDFS

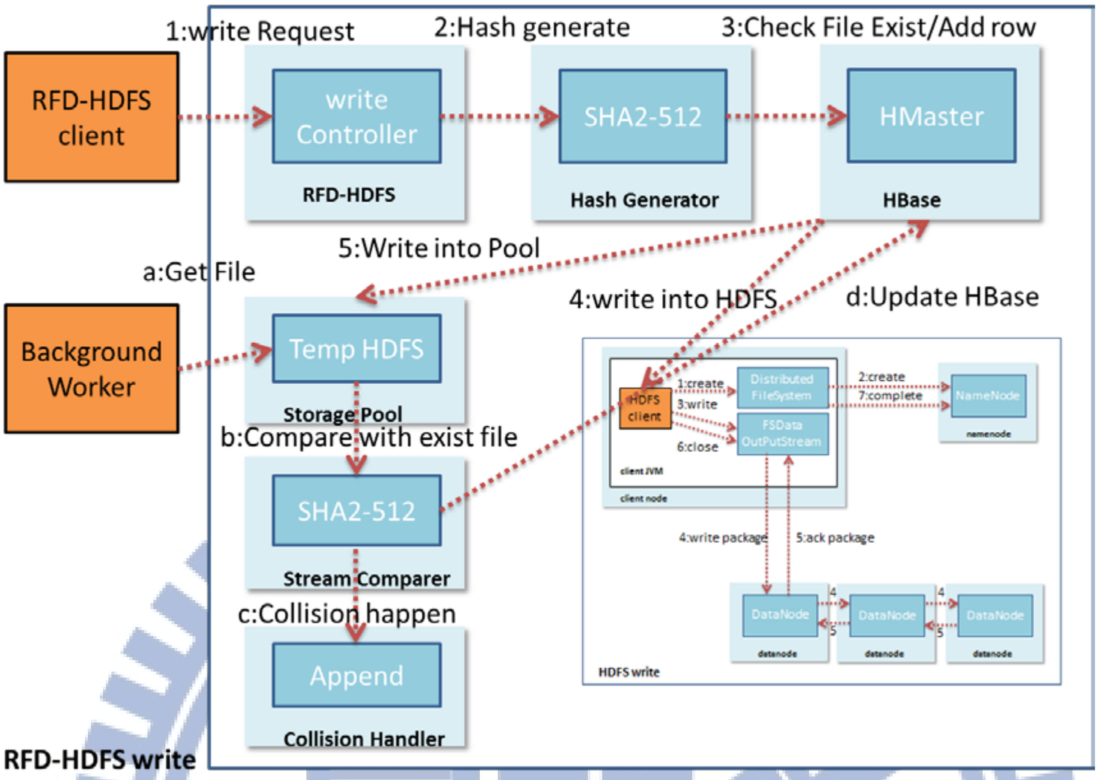


Figure 7: RFD-HDFS Write

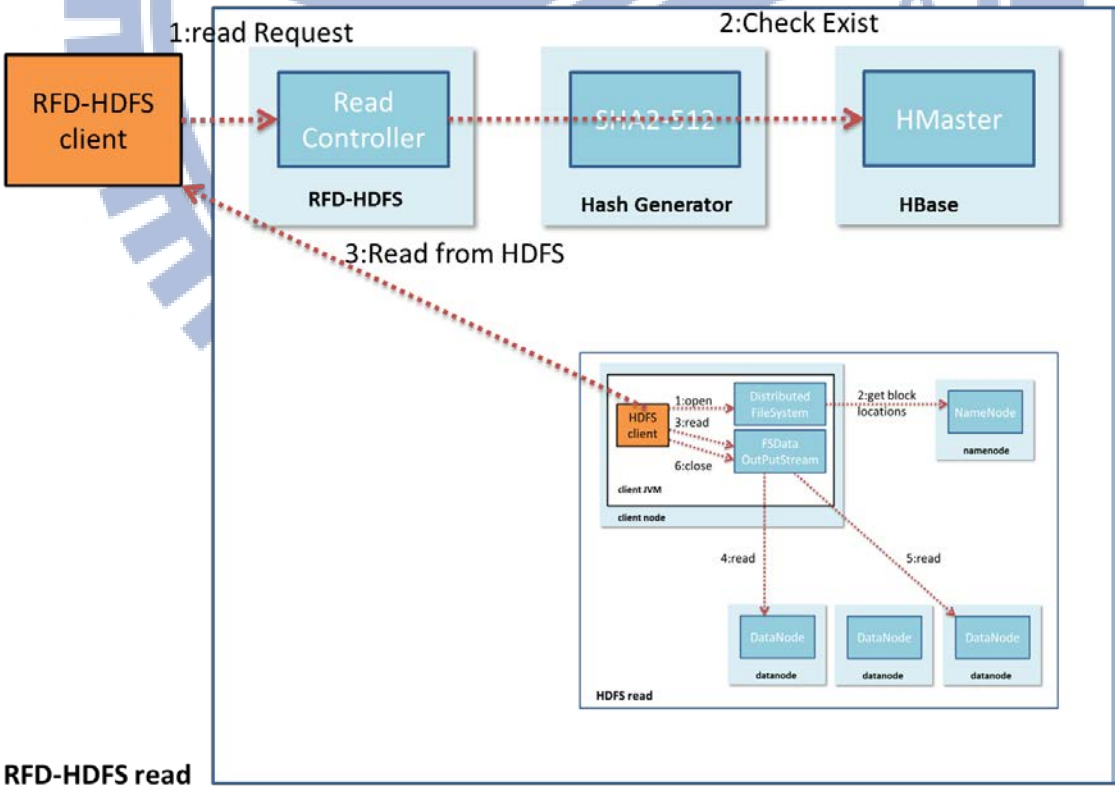


Figure 8: RFD-HDFS Read

3.2.3 **FD-HDFS**

FD-HDFS: If the Error Torrance is acceptable, we can ignore the collision of Hash; let's move the Hash generator to the side of client. And the binary comparer and storage pool is removed.

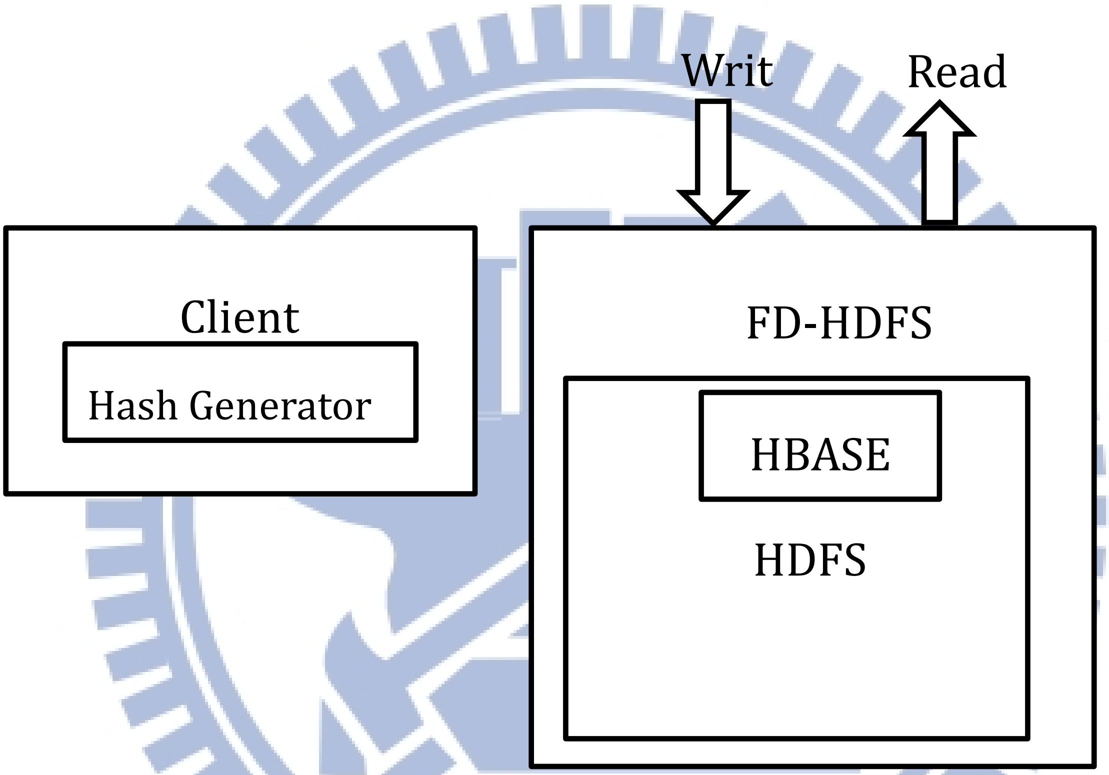


Figure 9: Use case of FD-HDFS

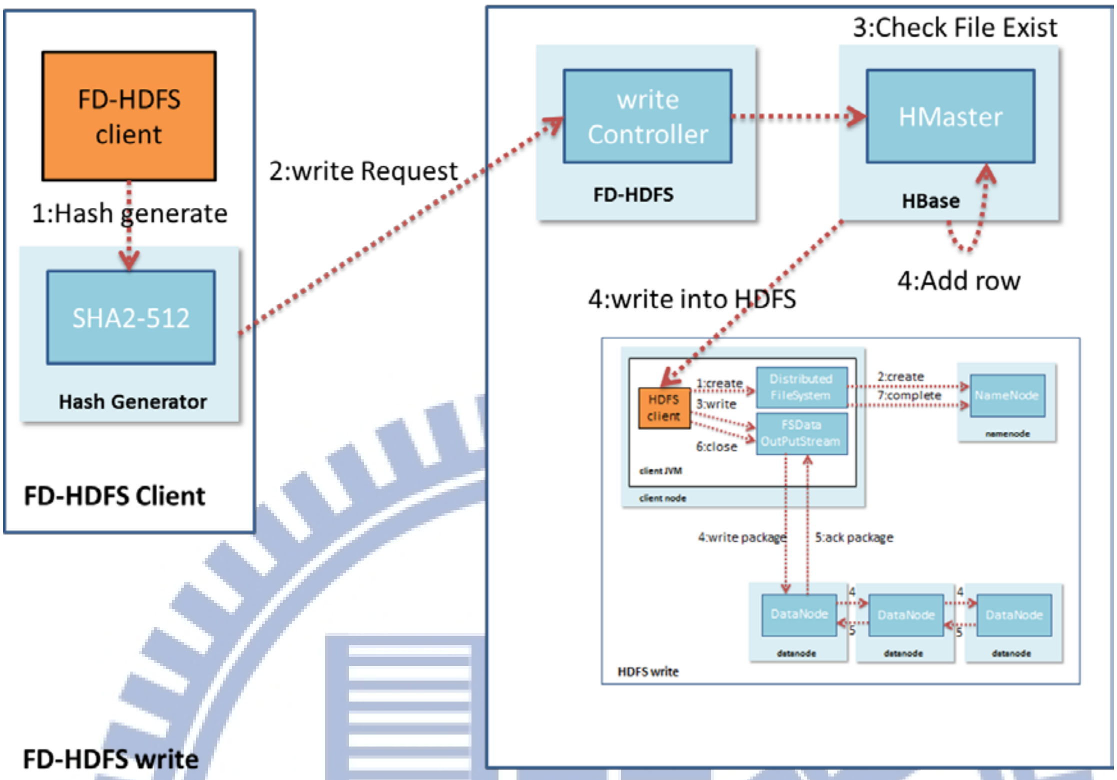


Figure 10: FD-HDFS Write

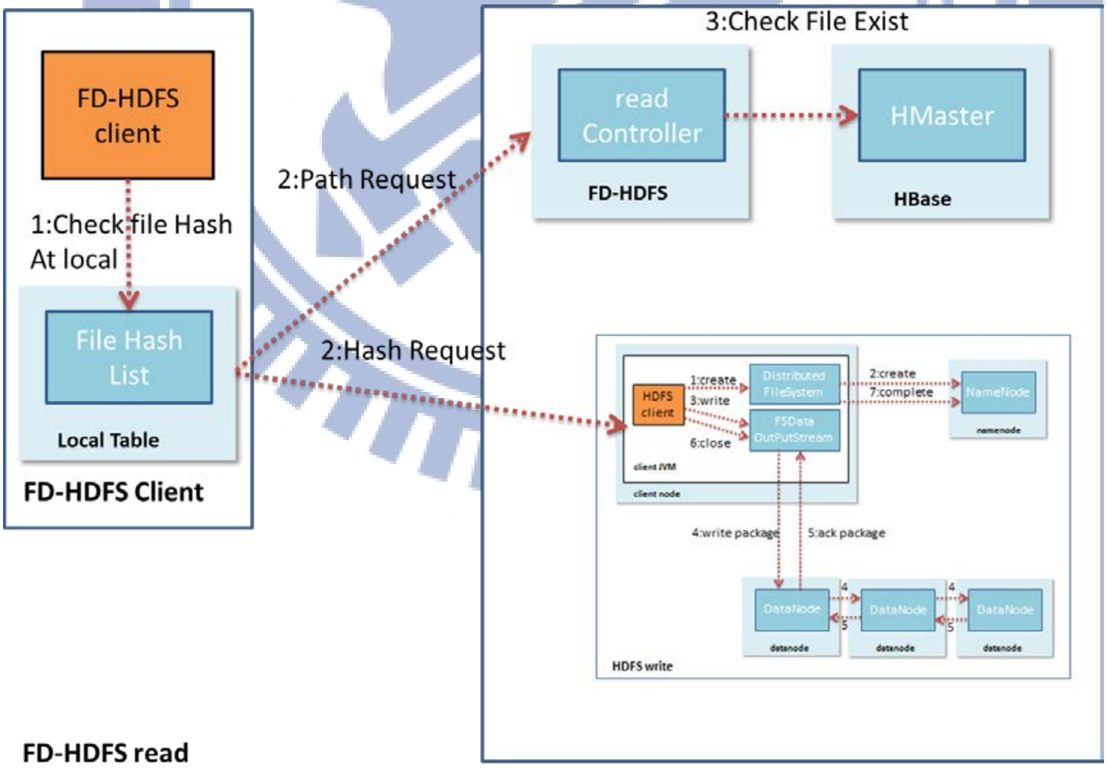


Figure 11: FD-HDFS Read

3.3 Algorithm

3.3.1 File Writing

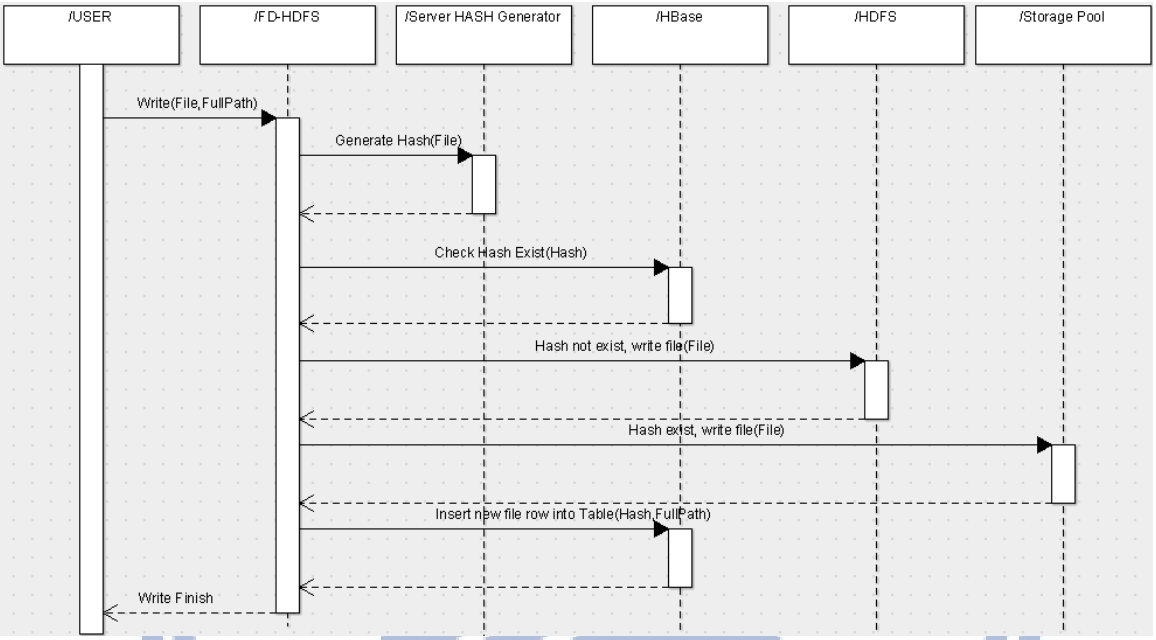


Figure 12: The writing of RFD-HDFS

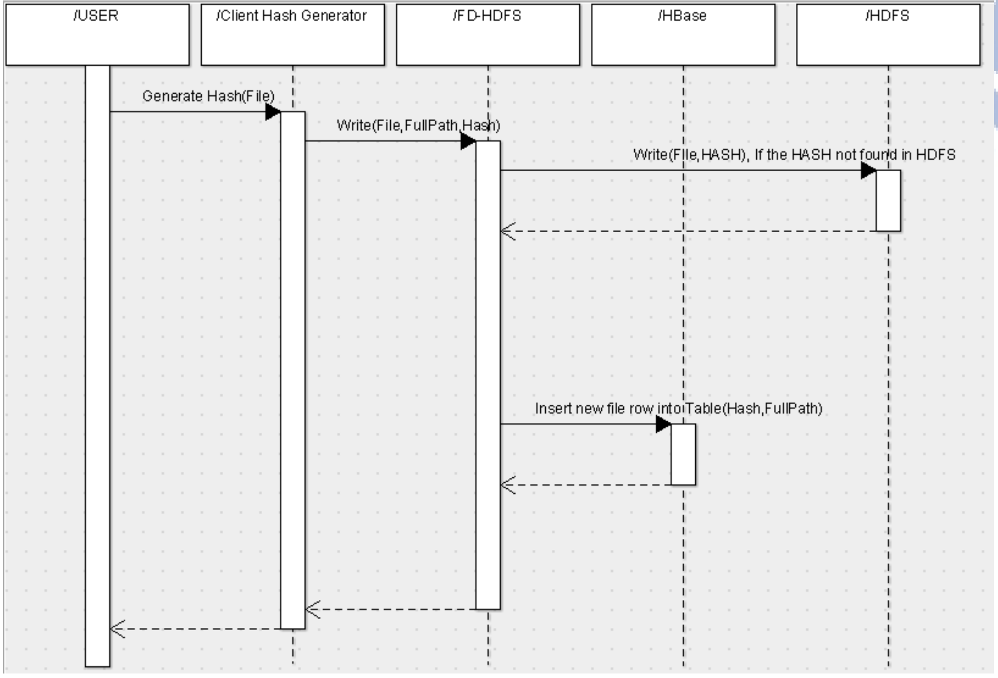


Figure 13: The write of FD-HDFS

3.3.2 File reading

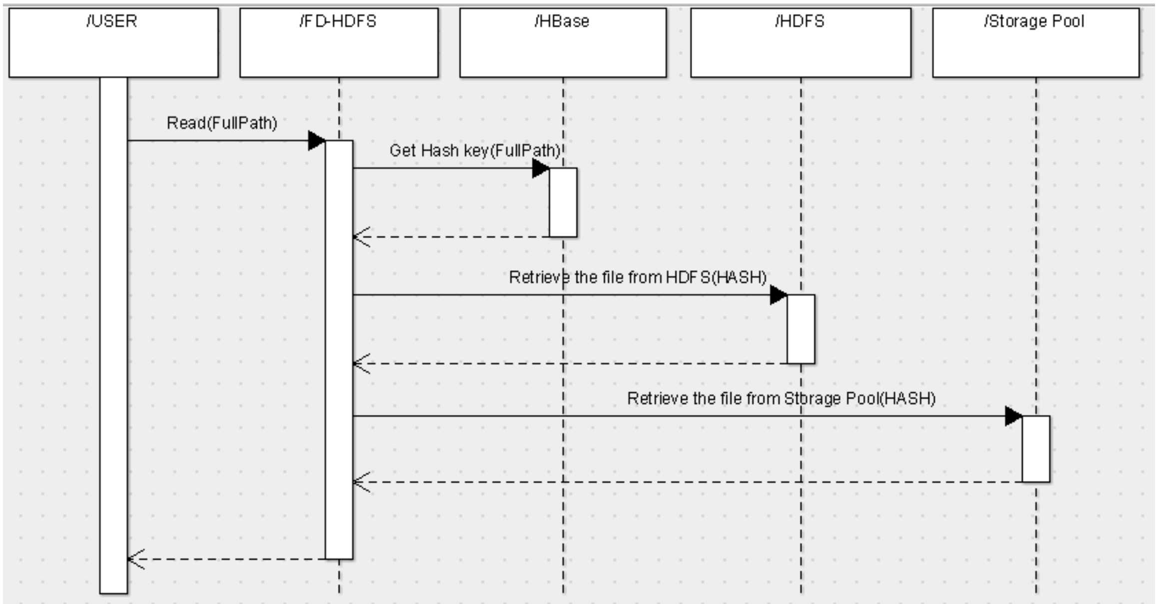


Figure 14: The reading of RFD-HDFS

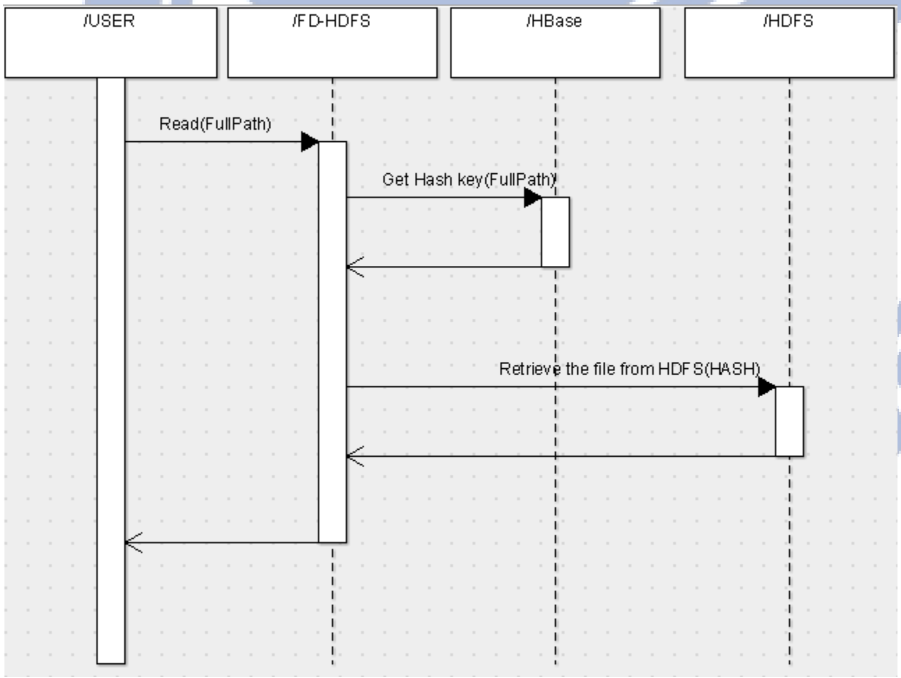


Figure 15: The reading of FD-HDFS

3.3.3 Background Worker

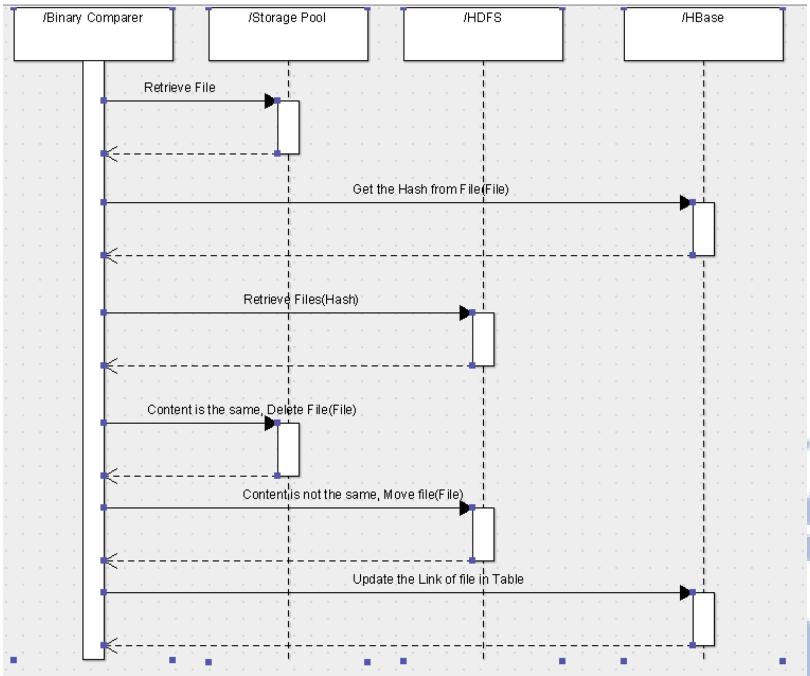


Figure 16: The background worker of RFD-HDFS

3.3.4 **FD-HDFS Function Presentation**

Upload folder preview

command: ls /datadisk/pic

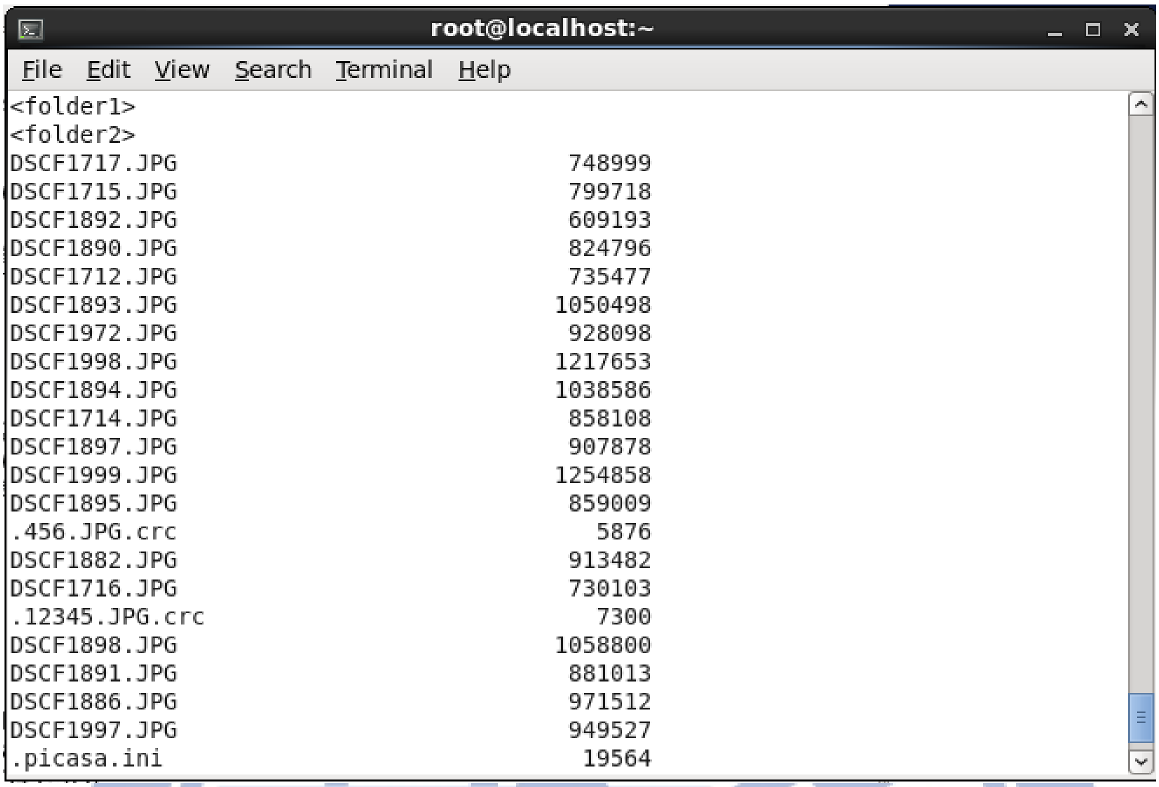
```
[root@namenode ~]# ls /datadisk/pic
DSCF1712.JPG  DSCF1717.JPG  DSCF1891.JPG  DSCF1896.JPG  DSCF1997.JPG
DSCF1713.JPG  DSCF1875.JPG  DSCF1892.JPG  DSCF1897.JPG  DSCF1998.JPG
DSCF1714.JPG  DSCF1882.JPG  DSCF1893.JPG  DSCF1898.JPG  DSCF1999.JPG
DSCF1715.JPG  DSCF1886.JPG  DSCF1894.JPG  DSCF1971.JPG  folder1
DSCF1716.JPG  DSCF1890.JPG  DSCF1895.JPG  DSCF1972.JPG  folder2
```

Upload folder to FD-HDFS

command: java -jar FD.jar writefolder /datadisk/pic pic

List Files in FD-HDFS

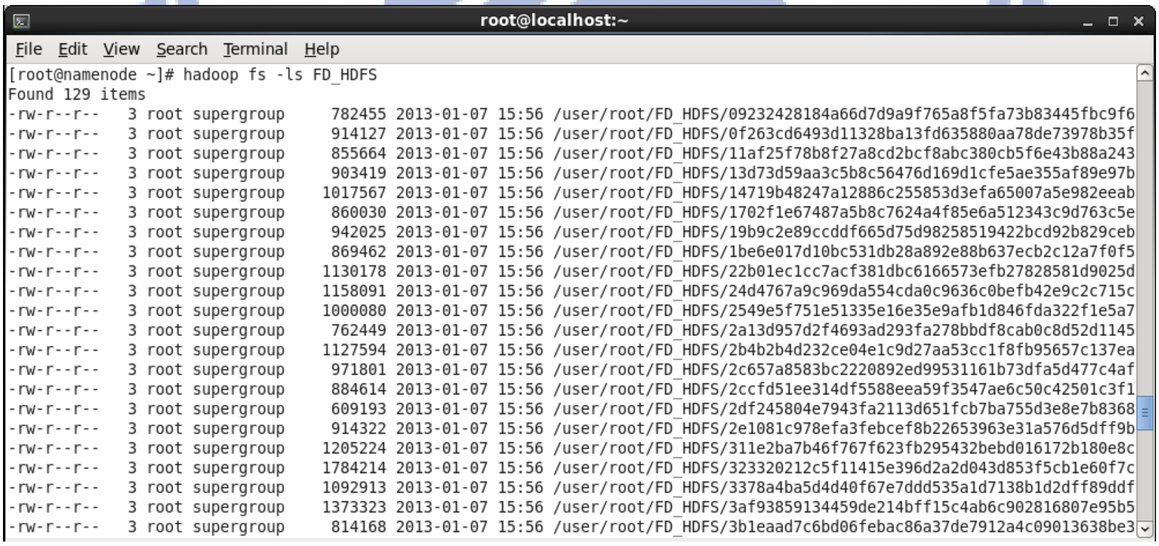
command: java -jar FD.jar ls pic



```
<folder1>
<folder2>
DSCF1717.JPG                       748999
DSCF1715.JPG                       799718
DSCF1892.JPG                       609193
DSCF1890.JPG                       824796
DSCF1712.JPG                       735477
DSCF1893.JPG                      1050498
DSCF1972.JPG                       928098
DSCF1998.JPG                      1217653
DSCF1894.JPG                      1038586
DSCF1714.JPG                       858108
DSCF1897.JPG                       907878
DSCF1999.JPG                      1254858
DSCF1895.JPG                       859009
.456.JPG.crc                         5876
DSCF1882.JPG                       913482
DSCF1716.JPG                       730103
.12345.JPG.crc                       7300
DSCF1898.JPG                      1058800
DSCF1891.JPG                       881013
DSCF1886.JPG                       971512
DSCF1997.JPG                       949527
.picasa.ini                         19564
```

Figure 17: File List Preview

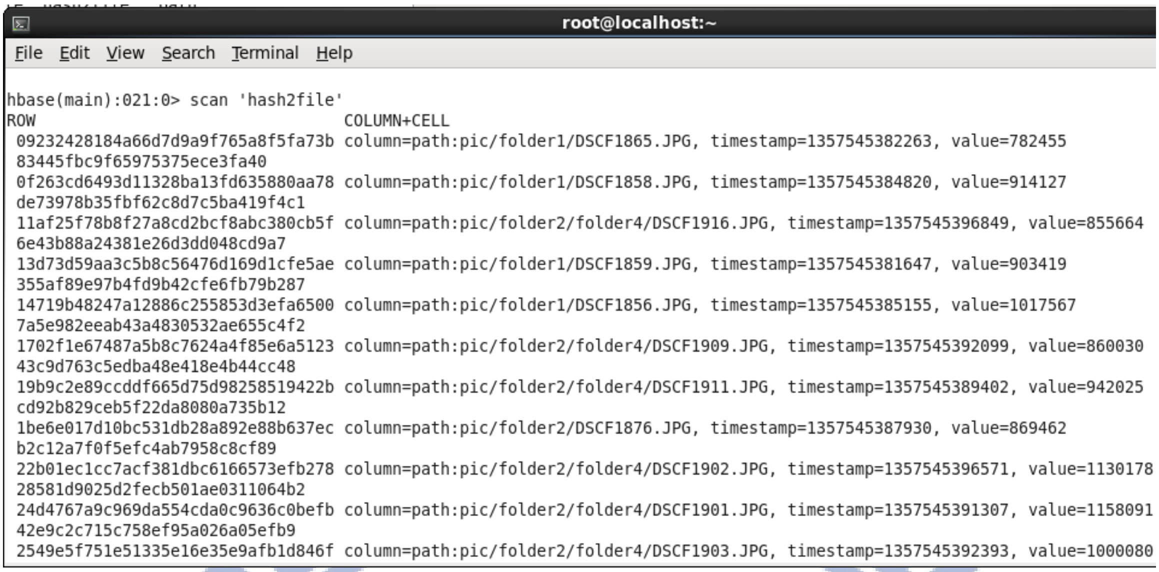
The actual files in HDFS



```
[root@namenode ~]# hadoop fs -ls FD_HDFS
Found 129 items
-rw-r--r--   3 root supergroup     782455 2013-01-07 15:56 /user/root/FD_HDFS/09232428184a66d7d9a9f765a8f5fa73b83445fbc9f6
-rw-r--r--   3 root supergroup     914127 2013-01-07 15:56 /user/root/FD_HDFS/0f263cd6493d11328ba13fd635880aa78de73978b35f
-rw-r--r--   3 root supergroup     855664 2013-01-07 15:56 /user/root/FD_HDFS/11af25f78b8f27a8cd2bcf8abc380cb5f6e43b88a243
-rw-r--r--   3 root supergroup     903419 2013-01-07 15:56 /user/root/FD_HDFS/13d73d59aa3c5b8c56476d169d1cfe5ae355af89e97b
-rw-r--r--   3 root supergroup    1017567 2013-01-07 15:56 /user/root/FD_HDFS/14719b48247a12886c255853d3efa65007a5e982eeab
-rw-r--r--   3 root supergroup     860030 2013-01-07 15:56 /user/root/FD_HDFS/1702f1e67487a5b8c7624a4f85e6a512343c9d763c5e
-rw-r--r--   3 root supergroup     942025 2013-01-07 15:56 /user/root/FD_HDFS/19b9c2e89ccddf665d75d98258519422bcd92b829ceb
-rw-r--r--   3 root supergroup     869462 2013-01-07 15:56 /user/root/FD_HDFS/1be6e017d10bc531db28a892e88b637ecb2c12a7f0f5
-rw-r--r--   3 root supergroup    1130178 2013-01-07 15:56 /user/root/FD_HDFS/22b01ec1cc7acf381dbc6166573efb27828581d9025d
-rw-r--r--   3 root supergroup    1158091 2013-01-07 15:56 /user/root/FD_HDFS/24d4767a9c969da554cda0c9636c0befb42e9c2c715c
-rw-r--r--   3 root supergroup    1000080 2013-01-07 15:56 /user/root/FD_HDFS/2549e5f751e51335e16e35e9afb1d846fda322f1e5a7
-rw-r--r--   3 root supergroup     762449 2013-01-07 15:56 /user/root/FD_HDFS/2a13d957d2f4693ad293fa278bbdf8cab0c8d52d1145
-rw-r--r--   3 root supergroup    1127594 2013-01-07 15:56 /user/root/FD_HDFS/2b4b2b4d232ce04e1c9d27aa53cc1f8fb95657c137ea
-rw-r--r--   3 root supergroup     971801 2013-01-07 15:56 /user/root/FD_HDFS/2c657a8583bc2220892ed99531161b73dfa5d477c4af
-rw-r--r--   3 root supergroup     884614 2013-01-07 15:56 /user/root/FD_HDFS/2ccfd51ee314df5588eea59f3547ae6c50c42501c3f1
-rw-r--r--   3 root supergroup     609193 2013-01-07 15:56 /user/root/FD_HDFS/2df245804e7943fa2113d651fcb7ba755d3e8e7b8368
-rw-r--r--   3 root supergroup     914322 2013-01-07 15:56 /user/root/FD_HDFS/2e1081c978efa3febcef8b22653963e31a576d5dff9b
-rw-r--r--   3 root supergroup    1205224 2013-01-07 15:56 /user/root/FD_HDFS/311e2ba7b46f767f623fb295432bebd016172b180e8c
-rw-r--r--   3 root supergroup    1784214 2013-01-07 15:56 /user/root/FD_HDFS/323320212c5f11415e396d2a2d043d853f5cb1e60f7c
-rw-r--r--   3 root supergroup    1092913 2013-01-07 15:56 /user/root/FD_HDFS/3378a4ba5d4d40f67e7ddd535a1d7138b1d2dff89ddf
-rw-r--r--   3 root supergroup    1373323 2013-01-07 15:56 /user/root/FD_HDFS/3af93859134459de214bff15c4ab6c902816807e95b5
-rw-r--r--   3 root supergroup     814168 2013-01-07 15:56 /user/root/FD_HDFS/3b1eaad7c6bd06febac86a37de7912a4c09013638be3
```

Figure 18: The actual files in HDFS

The file information in HBASE table



```
hbase(main):021:0> scan 'hash2file'
ROW                                COLUMN+CELL
 09232428184a66d7d9a9f765a8f5fa73b column=path:pic/folder1/DSCF1865.JPG, timestamp=1357545382263, value=782455
 83445fbc9f65975375ece3fa40
 0f263cd6493d11328ba13fd635880aa78 column=path:pic/folder1/DSCF1858.JPG, timestamp=1357545384820, value=914127
 de73978b35fbf62c8d7c5ba419f4c1
 11af25f78b8f27a8cd2bcf8abc380cb5f column=path:pic/folder2/folder4/DSCF1916.JPG, timestamp=1357545396849, value=855664
 6e43b88a24381e26d3dd048cd9a7
 13d73d59aa3c5b8c56476d169d1cfe5ae column=path:pic/folder1/DSCF1859.JPG, timestamp=1357545381647, value=903419
 355af89e97b4fd9b42cfe6fb79b287
 14719b48247a12886c255853d3efa6500 column=path:pic/folder1/DSCF1856.JPG, timestamp=1357545385155, value=1017567
 7a5e982eeab43a4830532ae655c4f2
 1702f1e67487a5b8c7624a4f85e6a5123 column=path:pic/folder2/folder4/DSCF1909.JPG, timestamp=1357545392099, value=860030
 43c9d763c5edba48e418e4b44cc48
 19b9c2e89ccddf665d75d98258519422b column=path:pic/folder2/folder4/DSCF1911.JPG, timestamp=1357545389402, value=942025
 cd92b829ceb5f22da8080a735b12
 1be6e017d10bc531db28a892e88b637ec column=path:pic/folder2/DSCF1876.JPG, timestamp=1357545387930, value=869462
 b2c12a7f0f5efc4ab7958c8cf89
 22b01ec1cc7acf381dbc6166573efb278 column=path:pic/folder2/folder4/DSCF1902.JPG, timestamp=1357545396571, value=1130178
 28581d9025d2fecb501ae0311064b2
 24d4767a9c969da554cda0c9636c0befb column=path:pic/folder2/folder4/DSCF1901.JPG, timestamp=1357545391307, value=1158091
 42e9c2c715c758ef95a026a05efb9
 2549e5f751e51335e16e35e9afb1d846f column=path:pic/folder2/folder4/DSCF1903.JPG, timestamp=1357545392393, value=1000080
```

Figure 19: The file information in HBASE table

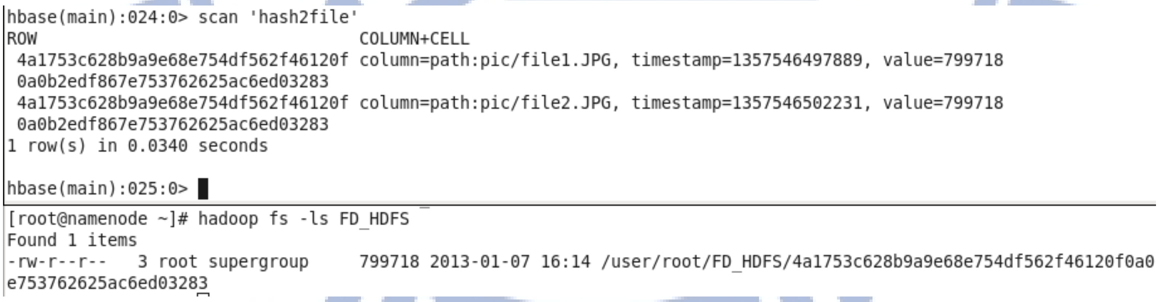
Let’s truncate table and clear hdfs

Upload 2 files which the content of files both the same

command: java -jar FD.jar write /datadisk/pic/DSCF1715.JPG pic/file1.JPG

command: java -jar FD.jar write /datadisk/pic/DSCF1715.JPG pic/file2.JPG

We’ll found two rows in table of hbase and a single file stored in HDFS.



```
hbase(main):024:0> scan 'hash2file'
ROW                                COLUMN+CELL
 4a1753c628b9a9e68e754df562f46120f column=path:pic/file1.JPG, timestamp=1357546497889, value=799718
 0a0b2edf867e753762625ac6ed03283
 4a1753c628b9a9e68e754df562f46120f column=path:pic/file2.JPG, timestamp=1357546502231, value=799718
 0a0b2edf867e753762625ac6ed03283
1 row(s) in 0.0340 seconds

hbase(main):025:0> █
```

```
[root@namenode ~]# hadoop fs -ls FD_HDFS
Found 1 items
-rw-r--r--   3 root supergroup     799718 2013-01-07 16:14 /user/root/FD_HDFS/4a1753c628b9a9e68e754df562f46120f0a0
e753762625ac6ed03283
```

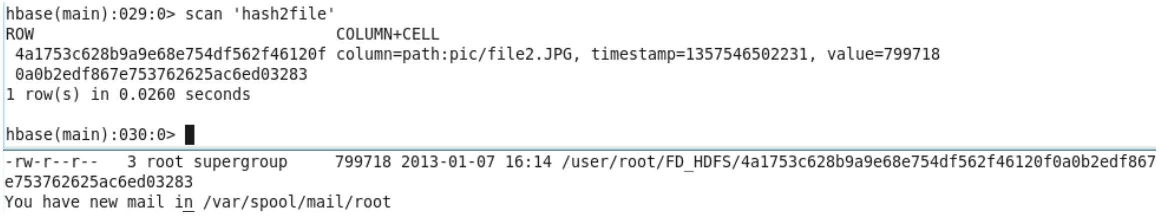
Figure 20: Two rows in table and a single file stored

Read file from FD-HDFS

command: java -jar FD.jar read pic/file1.JPG /datadisk/pic/file1.JPG

Delete file1 from FD-HDFS

command: java -jar FD.jar delete pic/file1.JPG



```
hbase(main):029:0> scan 'hash2file'
ROW                                COLUMN+CELL
 4a1753c628b9a9e68e754df562f46120f column=path:pic/file2.JPG, timestamp=1357546502231, value=799718
 0a0b2edf867e753762625ac6ed03283
1 row(s) in 0.0260 seconds

hbase(main):030:0> █
-rw-r--r--   3 root supergroup     799718 2013-01-07 16:14 /user/root/FD_HDFS/4a1753c628b9a9e68e754df562f46120f0a0b2edf867
e753762625ac6ed03283
You have new mail in /var/spool/mail/root
```

Figure 21: The column Family was delete, but not file in HDFS

Delete file2 from FD-HDFS

command: java -jar FD.jar delete pic/file2.JPG

If the setting of delete option is true, the file will be removing from HDFS, and nothing exists in table and hdfs.

But the default value is false, the file is no use right now, but it may be uploading by the other user in the future.

4 Experiments

Try to build up an EndNote[16] storage over cloud, import the collection of papers from 20 student, and simulate the user upload all the files to cloud.

The setting of Hadoop dfs.replication is 3.

4.1 Environment, and Setting

Table 1: Hadoop Nodes

Host Name	OS	IP	HBase	Process
NameNode	CentOS 6.3	192.168.74.100	HMaster	FD.jar& RFD.jar
DataNode1	CentOS 6.3	192.168.74.101	HRegion	

DataNode2	CentOS 6.3	192.168.74.102	HRegion	
DataNode3	CentOS 6.3	192.168.74.103	HRegion	

Table 2: The schema and samples of HBase Table

Row Key	Time Stamp	FullPath	File Attributes		
			Permissions	Size	Update Time
File HASH	T1	Test/apache-solr-4.0.0.tgz	-rwxr-xr-x	200M	T11
File HASH	T2	Test/apache-solr-4.0.1.tgz	-rwxr-xr-x	200M	T12
File HASH	T3	Test/apache-solr-4.0.2.tgz	-rwxr-xr-x	200M	T13

4.2 Experiment 1

The experiment 1, FD-HDFS is empty, there is a little duplicated file between user and user, and no duplicate between user and FD-HDFS before user upload.

The flow of experiment 1:

a. Truncate HBase table and delete all files from FD-HDFS.

b. Upload user# files.

c. Write down the space usage.

Repeat b and c.

Table 3: Ex1-User file list

	File Size(MB)	File Count	Duplicate (%)	File Size*3

USER01	7.2	18	0	21.6
USER02	14.1	25	0	42.3
USER03	6.3	30	0	18.9
USER04	8.3	22	0	24.9
USER05	68.9	24	0.29	206.7
USER06	10	22	0	30
USER07	23.2	38	10.3	69.6
USER08	101.6	184	2.2	304.8
USER09	70.3	80	2.5	210.9
USER10	15.6	33	23.7	46.8
USER11	41.9	62	3.8	125.7
USER12	94.3	164	6.3	282.9
USER13	61.1	108	7.6	183.3
USER14	25,3	38	11.8	75.9
USER15	34.7	62	8.9	104.1
USER16	24.4	42	2	73.2
USER17	3.6	15	0	10.8
USER18	20.1	31	8.4	60.3
USER19	12.1	20	28	36.3
USER20	3.7	18	13.5	11.1
Total	646.7	1036	2% per user	1904.1

4.3 Result of Experiment 1

Table 4: Ex1-DFS usage

	HDFS(MB)	FD-HDFS(MB)	RFD-HDFS(MB)
Initial	2.25	9.05	8
USER01	24	24.59	24.12
USER02	66.63	67.22	66.75
USER03	85.66	86.25	85.78
USER04	110.9	111.5	111.45
USER05	319.22	319.22	318.75
USER06	351.27	349.46	348.99
USER07	421.29	414.15	413.68
USER08	728.57	714.81	714.34
USER09	941.12	921.94	921.47
USER10	988.21	958.16	957.69
USER11	1090	1060	1060
USER12	1370	1320	1320
USER13	1550	1490	1490
USER14	1690	1550	1550
USER15	1730	1650	1650
USER16	1800	1720	1720
USER17	1810	1730	1730
USER18	1870	1790	1790
USER19	1910	1810	1810
USER20	1920	1820	1820

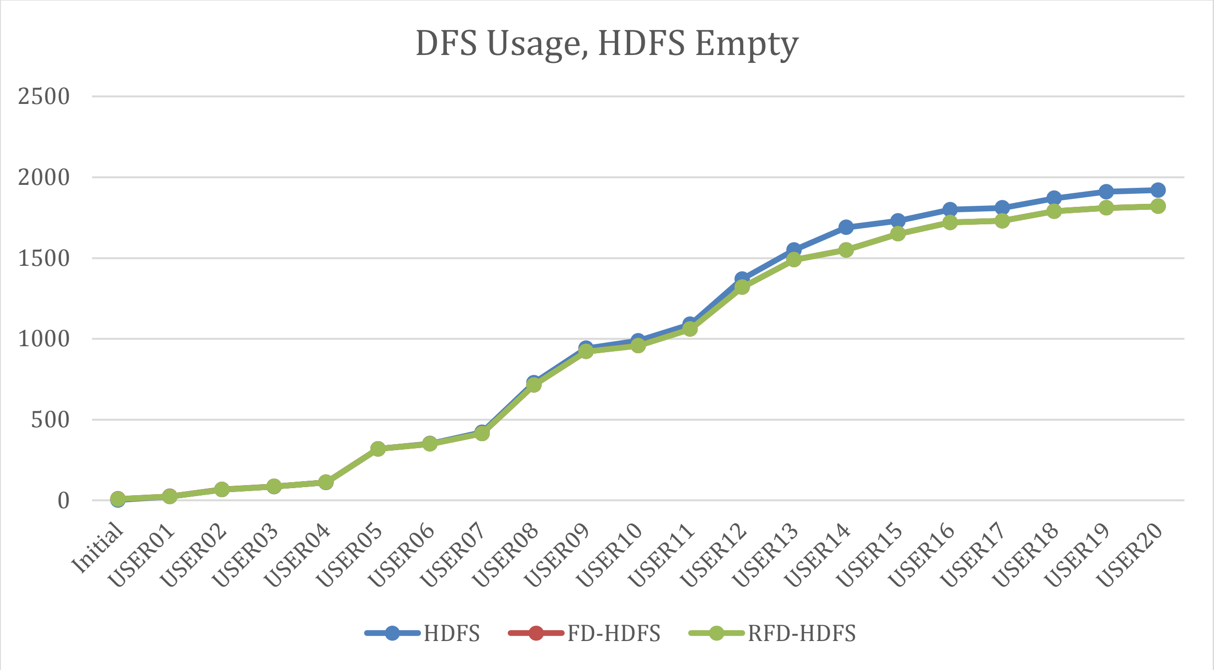


Figure 22: Ex1-DFS usage

4.4 The setting of Experiment 2

Let's fill all files which download from ACM and JSTOR. (About 20,000 Files)

Simulating the most of files was uploaded to HDFS.

The flow of experiment 2:

a. Truncate HBase table and delete all files from FD-HDFS.

b. Upload user# files.

c. Write down the space usage.

Repeat b to c.

Table 5: Ex2-User file list

	File Size(MB)	File Count	Duplicate (%)	File Size*3

USER01	7.2	18	66	21.6
USER02	14.1	25	76	42.3
USER03	6.3	30	30	18.9
USER04	8.3	22	59	24.9
USER05	68.9	24	58	206.7
USER06	10	22	13	30
USER07	23.2	38	42	69.6
USER08	101.6	144	41	304.8
USER09	70.3	120	50	210.9
USER10	15.6	33	57	46.8
USER11	41.9	62	38	125.7
USER12	94.3	164	17	282.9
USER13	61.1	108	26	183.3
USER14	25,3	38	73	75.9
USER15	34.7	62	74	104.1
USER16	24.4	42	83	73.2
USER17	3.6	15	33	10.8
USER18	20.1	31	61	60.3
USER19	12.1	20	75	36.3
USER20	3.7	18	72	11.1
Total	646.7	1036	40.9	1904.1

4.5 Result of Experiment 2

Table 6: Ex2-DFS usage

	HDFS(GB)	FD-HDFS(GB)	RFD-HDFS(GB)
Initial	44.81	44.82	44.82
USER01	44.83	44.83	44.83
USER02	44.87	44.83	44.83
USER03	44.89	44.84	44.84
USER04	44.91	44.84	44.84
USER05	45.12	44.86	44.86
USER06	45.15	44.89	44.89
USER07	45.21	44.93	44.93
USER08	45.51	45.10	45.10
USER09	45.72	45.19	45.19
USER10	45.77	45.21	45.21
USER11	45.88	45.24	45.24
USER12	46.02	45.47	45.47
USER13	46.35	45.62	45.62
USER14	46.43	45.65	45.65
USER15	46.53	45.66	45.66
USER16	46.60	45.67	45.67
USER17	46.61	45.68	45.68
USER18	46.67	45.70	45.70
USER19	46.72	45.71	45.71
USER20	46.73	45.71	45.71

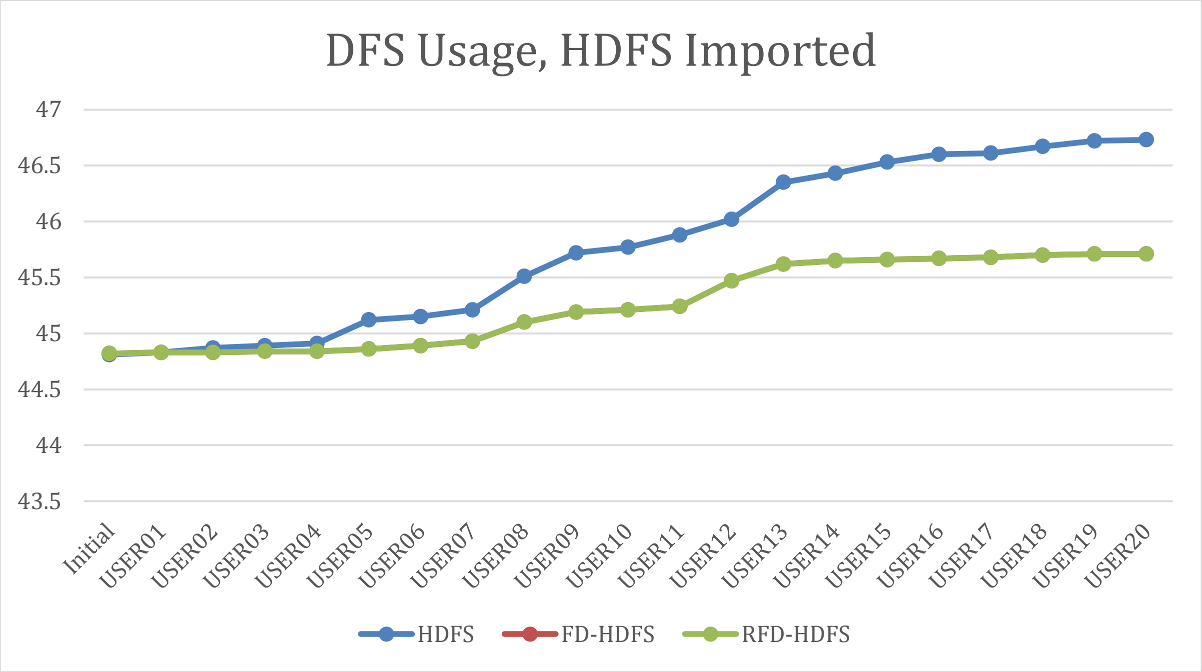


Figure 23: Ex2-DFS usage

4.6 The setting of Experiment 3

Let's simulate the FD-HDFS import all files from 10% to 100%, and User08 uploads the same 184 files to each FD-HDFS.

The flow of experiment 3:

a. Truncate HBase table and delete all files from FD-HDFS.

b. Upload files to FD-HDFS, file duplicate is 10%~100%.

c. Upload user8 files.

d. Write down the increase of space usage.

d. Repeat b to d.

Table 7: Ex3-HDFS List

	Duplicat	File Size(MB)	File Count	File

	e (%)			Size*3
HDFS00%	0	101.6	184	304.8
HDFS10%	10	101.6	184	304.8
HDFS20%	20	101.6	184	304.8
HDFS30%	30	101.6	184	304.8
HDFS40%	40	101.6	184	304.8
HDFS50%	50	101.6	184	304.8
HDFS60%	60	101.6	184	304.8
HDFS70%	70	101.6	184	304.8
HDFS80%	80	101.6	184	304.8
HDFS90%	90	101.6	184	304.8
HDFS100%	100	101.6	184	304.8

4.7 Result of Experiment 3

Table 8: Ex3-DFS usage

	HDFS	FD-HDFS	RFD-HDFS
HDFS00%	304.81	299.61	299.61
HDFS10%	304.82	269.34	269.34
HDFS20%	304.77	241.18	241.18
HDFS30%	304.80	210.81	210.81
HDFS40%	304.81	174.03	174.03
HDFS50%	304.81	151.99	151.99

HDFS60%	304.78	121.7	121.7
HDFS70%	304.80	91.46	91.46
HDFS80%	304.82	62.14	62.14
HDFS90%	304.80	32.20	32.20
HDFS100%	304.81	0	0

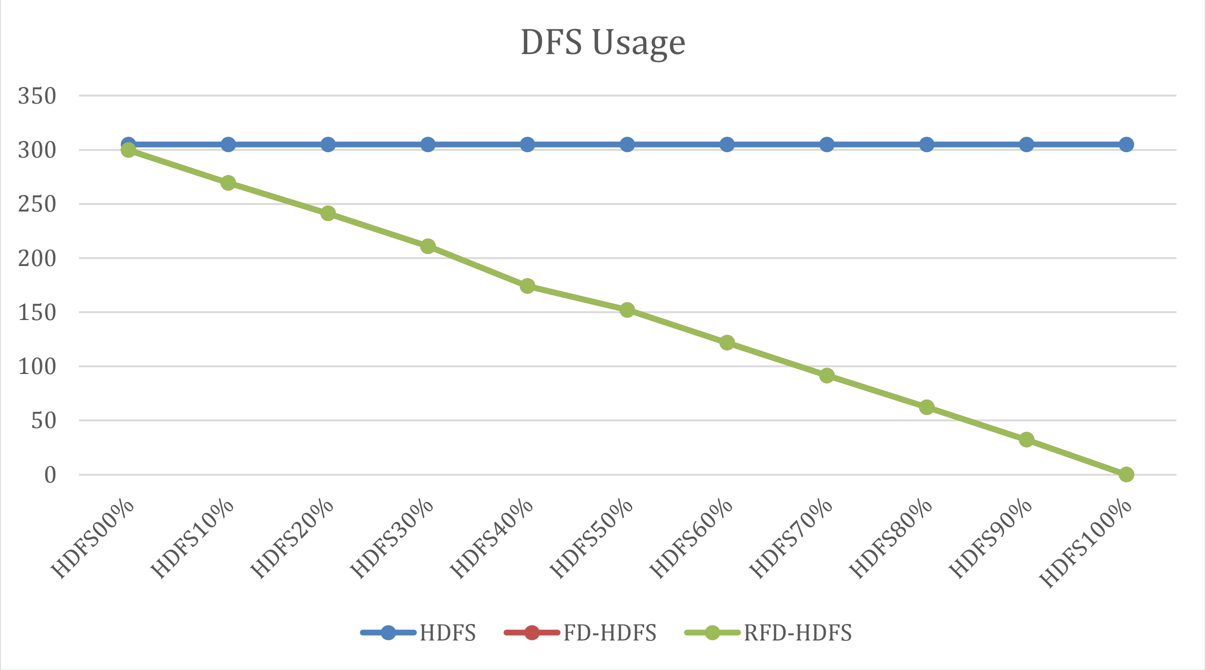


Figure 24: Ex3-DFS usage

4.8 Overhead

The experiment of overhead was running at VMware WorkStation 9, the hard disk drive is 1T 5400rpm.

6 file will be upload to FD-HDFS, let's record the time span in log file.

The flow of experiment overhead:

a. Truncate HBase table and delete all files from FD-HDFS.

b. Upload file #.

c. Write down the log for time usage.

Repeat b to d.

Table 9: Sample files for overhead testing

	File Size(MB)
File0	131
File1	1001
File2	2051
File3	3024
File4	3992
File5	4989

Table 10: Result of overhead test

	FD-HDFS(RFD-HDFS Post process)	RFD-HDFS(In-Process)	HDFS Write
File0	10.4	29.5	9
File1	270.6	434.6	260
File2	548.3	835	520
File3	881	1318	774
File4	1073	1724	1020
File5	1383	2156	1305

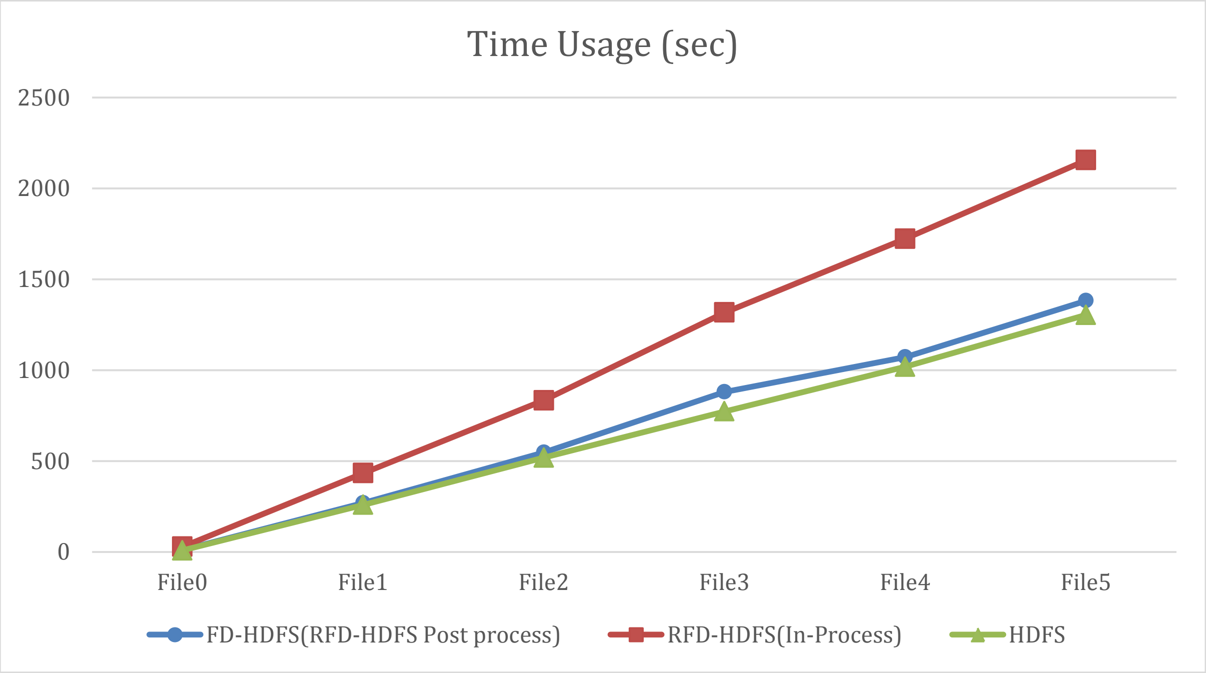


Figure 25: Time usage for Hash and Comparison

4.9 Multi-Threading(Multi-Users)

20 User upload all file to cloud at the same time. The system have no issue about Multi-Threading.

The flow of experiment Multi-Users:

a. Truncate HBase table and delete all files from FD-HDFS.

b. Upload all files from 20 users one by one.

c. Truncate HBase table and delete all files from FD-HDFS.

d. Upload all files from 20 users at the same time.

	File Size(MB)	Log
Single thread	2 minutes 28 sec	04:42:59-985-HASH start 04:45:27-989-WRITEFOLDER End
Multi thread	2 minutes 9 sec	04:31:37-985-HASH start

		04:33:46-195-WRITEFOLDER End

We’ll see the multi-threading is a little faster than the single one. But the user count is much bigger than 20, the effective of nodes cluster should not a few second of process time.

5 Conclusion

Table 11: Compare with HDFS

	HDFS	RFD-HDFS	FD-HDFS
File Deduplication	No	Yes	Yes
File Reliability	O	O	X(SHA2-512=2^{512})
The Loading of comparison	No	Hash generate, Stream compare	Hash generate
The Cost of Upload time and Bandwidth	Full	Full	The duplicated file upload/download time and bandwidth are almost zero
Loading of NameNode and HMaster	Normal	Normal	May be reduced
Same Data / Similar Data	Not support.	Same Data	Same Data
Source Dedup/Target	Not support.	Target Dedup	Source Dedup

Dedup			
In Line Process/Post Process	Not support.	Post Process	In Line Process
Keep old data/new data	Not support.	Keep old data	Keep old data

SHA1 needs 2^{63} to find the file of different content but equal Hash Sum[8]. It is obvious that the SHA2-512 collision probability is very low, so the experimental results of RFD-HDFS and FD-HDFS are identical. That is, although the Hash Collision Policy has been concluded, there is no opportunity to use. The chance of the hash collision should be really small, but how about the Nuclear Science and financial application? That's the reason about the model of RFD-HDFS, which will be individually listed since many systems still cannot afford one-astronomical number error, many systems have the requirements of saving transmission time and the bandwidth, and the both have suitable occasions and applications.

From first experimental results, we can observe that the probability of holding duplicate files between User and User is not high.

The reason why the repeat exists is because in the experiment, the students electing the same course participated and needed to report the paper report, that the opportunity interactively holding the same file. But as long as the clouds do not have any information, the rate of repetition is low.

In the second experiment, we had simulated the cloud HD-HDFS operation for a certain time and have housed many papers. Many documents already exist in FD-HDFS, under which we can observe the space complexity decreased.

In the third experiment, the same user uploaded the same paper at different points of time.

We can find that the FD-HDFS system housed more papers have less space occupancy rate. In ideal, when storage references reach one hundred percent, then all new existing files will no longer consume space.

In conclusion, a centralized cloud system can gradually lead to the increase of file repetition when the users upload data. Through file de-duplication, the storage space required can be greatly reduced.

Through this system, incorporating some type of file population becomes possible. What is needed is only constant space.

RFD-HDFS system suitable for use in commercial information, nuclear engineering, if the application cannot allow any error; FD-HDFS could implement at most kind of files, if a small error is acceptable.

6 Future Work

The MapReduce is useful to determine how to split the file to data blocks (Chunk), For example: a Longest Common Subsequence (LCS) for determine the diff of the file content, but if the file was split, there should be data fragmentation issue. DUTCH T. MEYER[10] find that whole-file deduplication together with sparseness is a highly efficient means of lowering storage consumption, even in a backup scenario. It approaches the effectiveness of conventional deduplication at a much lower cost in performance and complexity.

Hadoop provide compressed stream write in, but it’s optional and should be handle by programmer or user. If the middle layer file system exists, the compress function could apply to each file of FD-HDFS, and user does not have to care about it at all.

It should be easy to port the architecture to IBM GPFS[17] and Amazon S3(Dropbox)[18] easily, everything we have to do is change the API of cloud storage.

MapReduce, CUDA or a specific hardware may reduce the overhead of hash generate and stream comparison.(RFD-HDFS)

How about a BitTorrent Proxy server for speed up the download time and reduce the bandwidth requirement of ISP.

The replica of file is reduced. For data localization, the topology is useful at the hotspot files.

7 Reference

[1] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *ACM SIGOPS Operating Systems Review*, 2003, pp. 29-43.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows*, et al.*, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS),* vol. 26, p. 4, 2008.

[3] T. White, "Hadoop : the definitive guide," ed: Sebastopol, Calif. : O'Reilly Media, Inc., 2009.

[4] L. George, "HBase : the definitive guide," ed: Sebastopol, CA : O'Reilly, 2011.

[5] D. Eastlake and P. Jones, "US secure hash algorithm 1 (SHA1)," ed: RFC 3174, September, 2001.

[6] M. Stonebraker, "SQL databases v. NoSQL databases," *Communications of the ACM,* vol. 53, pp. 10-11, 2010.

[7] M. Bellare and T. Kohno, "Hash function balance and its impact on birthday attacks," in *Advances in Cryptology-Eurocrypt 2004*, 2004, pp. 401-418.

[8] B. Schneier, "New cryptanalytic results against SHA-1," *Weblog: Schneier on Security,* 2005.

[9] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur, "Single instance storage in Windows 2000," in *Proceedings of the 4th USENIX*

Windows Systems Symposium, 2000, pp. 13-24.

[10] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *Trans. Storage,* vol. 7, pp. 1-20, 2012.

[11] J. J. Rao and K. V. Cornelio, "An Optimized Resource Allocation Approach for Data-Intensive Workloads Using Topology-Aware Resource Allocation," in *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*, 2012, pp. 1-4.

[12] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of computer and system sciences,* vol. 18, pp. 143-154, 1979.

[13] D. Borthakur, "HDFS Architecture Guide," 2008.

[14] L. Kolb, A. Thor, and E. Rahm, "Dedoop: efficient deduplication with Hadoop," *Proc. VLDB Endow.,* vol. 5, pp. 1878-1881, 2012.

[15] B. Nutch, "Open source search," *Queue. v2 i2,* pp. 54-61, 2004.

[16] M. Reis and G. Resiss, "EndNote 5 reference manager—functions—improvements—personal experiences," *Schweiz Rundsch Med Prax,* vol. 91, pp. 1645-50, 2002.

[17] J. Barkes, M. R. Barrios, F. Cougard, P. G. Crumley, D. Marin, H. Reddy, *et al.*, "GPFS: a parallel file system," *IBM International Technical Support Organization,* 1998.

[18] J. Varia, "Cloud architectures," *White Paper of Amazon, jineshvaria. s3. amazonaws. com/public/cloudarchitectures-varia. pdf,* 2008.

Appendix A: C# Code of the Paper Graber

```
public void ThreadProc()
{
    //Grab Paper from website
    int i = 0;
    //SendEmail sendEmail = new SendEmail();
    StreamWriter sw = new StreamWriter("C:\\ouput" + loopEnd + ".txt");
    try
    {
        //form.buttonGrab.BackColor = Color.Red;
        CookieCollection cookies = new CookieCollection();
        HttpWebResponse response;
        //String TimeFormat = "HH:mm:ss";
        DateTime startTime = DateTime.Now;
        DateTime CurrentTime = DateTime.Now;
        DateTime EndTime = DateTime.Now;
        TimeSpan timeSpan = TimeSpan.Zero; ;
        //FormPaperGrabing.textBoxStartTime.Text = startTime.ToString(TimeFormat);
        //FormPaperGrabing.textBoxStartTime.Refresh();
        Regex regex = new Regex(@"\/stable\/pdfplus/(.*).pdf");
        //Regex regex = new Regex(@".*ft_gateway.cfm[?]id=(.*pdf.*)[""] title.*");
        MatchCollection mc;
        System.IO.StreamReader reader;
        int retryCount = 0;
        for (i = loopStart; i < loopEnd; i += loopStep)
        {
            int endNum = tagUrl.IndexOf("&", 79);
            tagUrl = tagUrl.Remove(79, endNum - 79);
            tagUrl = tagUrl.Insert(79, i.ToString());
            string respHTML;
            try
            {
                response = HttpWebResponseUtility.CreateGetHttpResponse(tagUrl, null,
null, cookies);
                reader = new System.IO.StreamReader(response.GetResponseStream(),
```

```
System.Text.Encoding.UTF8);
                        respHTML = reader.ReadToEnd();
                    }
                    catch (Exception e)
                    {
                        sw.WriteLine("ERROR " + i + " in " + loopEnd + e.Message);
                        sw.Flush();
                        if (retryCount < 2)
                        {
                            retryCount += 1;
                            i -= 1;
                        }
                        else
                        {
                        retryCount = 0;
                        }
                        continue;
                    }

                    mc = regex.Matches(respHTML);
                    foreach (Match item in mc)
                    {
                        String line = "http://www.jstor.org/stable/pdfplus/" +
item.Groups[1].Value + ".pdf?acceptTC=true";
                        sw.WriteLine(line);

                    }
                    FormPaperGrabing.textBoxIndex.Text = i.ToString();
                    FormPaperGrabing.textBoxIndex.Refresh();
                    CurrentTime = DateTime.Now;
                    timeSpan = CurrentTime.Subtract(startTime);
                    double avgSeconds = timeSpan.TotalSeconds / i;
                    EndTime = CurrentTime.AddSeconds((loopEnd - i) * avgSeconds);
                    FormPaperGrabing.textBoxTimeSpan.Text = timeSpan.Hours.ToString() + "Hours" +
timeSpan.Minutes.ToString() + "Minutes" + timeSpan.Seconds.ToString() + "Seconds";
                    FormPaperGrabing.textBoxTimeSpan.Refresh();
                    FormPaperGrabing.textBoxTimeEnd.Text = EndTime.ToString(TimeFormat);
```

```
                    FormPaperGrabing.textBoxTimeEnd.Refresh();
                }
                FormPaperGrabing.buttonGrab.BackColor = Color.Green;
                sendEmail.Send("PaperGrabing pass" + loopStart + " to " + loopEnd + " finish in "
+ i, "C:\\ouput" + loopEnd + ".txt");
            }
            catch (Exception e)
            {

            }
            finally
            {
                //sendEmail.Send("PaperGrabing End", "");
                sw.Close();
            }
        }
    }
```

Appendix B: Java Code of the main, command controller

```
public enum EnumCommand {
    LS, DELETE, WRITE, READ, HASHREAD, HASHWRITE, WRITEFOLDER, WRITEHDFS, TRUNCATE,
TEST

}

public static void main(String[] args) throws Exception {
        List<String> argsSB = Arrays.asList(args);
        EnumCommand enumCommand = null;
        if (argsSB.size() > 0) {
            enumCommand = EnumCommand.valueOf(argsSB.get(0).toUpperCase());

            switch (enumCommand) {
            case LS:
                if (argsSB.size() != 2)
                    throw new Exception("Source and Destination Require");
                LS(argsSB.get(1));
                break;
            case WRITE:
                if (argsSB.size() != 3)
                    throw new Exception("Source and Destination Require");
                WRITE(argsSB.get(1), argsSB.get(2));
                break;
            case WRITEFOLDER:
                if (argsSB.size() != 3)
                    throw new Exception("Source and Destination Require");
                WRITEFOLDER(argsSB.get(1), argsSB.get(2));
                break;
            case WRITEHDFS:
                if (argsSB.size() != 3)
                    throw new Exception("Source and Destination Require");
                WRITEHDFS(argsSB.get(1), argsSB.get(2));
                break;
            case READ:
                if (argsSB.size() != 3)
                    throw new Exception("Source and Destination Require");
```

```
                READ(argsSB.get(1), argsSB.get(2));
                break;
            case DELETE:
                if (argsSB.size() != 2)
                    throw new Exception("DELETE target require");
                DELETE(argsSB.get(1));
                break;
            case HASHREAD:
                if (argsSB.size() != 3)
                    throw new Exception("Source and Destination Require");
                HASHREAD(argsSB.get(1), argsSB.get(2));
                break;
            case TEST:
                TEST();
                break;
            default:
                throw new Exception("Command not found");
            }
        }
private static void TEST() throws IOException {
        HDFSAPI hdfsAPI = new HDFSAPI();
        hdfsAPI.Test();
    }

    public static void DELETE(String source) throws IOException {
        //System.out.println(DateTime.getTimeStamp() + "DELETE Start");
        HBaseAPI hbaseAPI = new HBaseAPI();
        HDFSAPI hdfsAPI = new HDFSAPI();
        if (hbaseAPI.scanQualifier(source) == 0)
            throw new IOException("Source file not found");
        String HexSHA = hbaseAPI.GetHashByPath(source);
        hbaseAPI.DelRowByPath(source);
        if (hbaseAPI.scanRow(HexSHA) == 0)
            hdfsAPI.delFilebyHash(HexSHA);
        //System.out.println(DateTime.getTimeStamp() + "DELETE End");
    }
```

```
public static void READ(String source, String destination) throws Exception {
        //System.out.println(DateTime.getTimeStamp() + "READ Start");
        HBaseAPI hbaseAPI = new HBaseAPI();
        HDFSAPI hdfsAPI = new HDFSAPI();
        if (hbaseAPI.scanQualifier(source) == 0)
                throw new IOException("Source file not found");
        String HexSHA = hbaseAPI.GetHashByPath(source);
        hdfsAPI.getFilebyHash(HexSHA, destination);
        //System.out.println(DateTime.getTimeStamp() + "READ End");
}

private static void HASHREAD(String hash, String destination)
                throws Exception {
        //System.out.println(DateTime.getTimeStamp() + "HASHREAD Start");
        HDFSAPI hdfsAPI = new HDFSAPI();
        hdfsAPI.getFilebyHash(hash, destination);
        //System.out.println(DateTime.getTimeStamp() + "HASHREAD End");
}

private static void WRITEHDFS(String sourceF, String destinationF)
                throws IOException {
        //System.out.println(DateTime.getTimeStamp() + "WRITEHDFS Start");
        File sourceFolder = new File(sourceF);
        HDFSAPI hdfsapi = new HDFSAPI();
        int i = 0;
        for (File file : sourceFolder.listFiles()) {
                if (file.isFile()) {
                        hdfsapi.AddFile(sourceF + "/" + file.getName(), destinationF
                                        + "/" + file.getName());
                        System.out.println(i++);
                }
        }
        //System.out.println(DateTime.getTimeStamp() + "WRITEHDFS End");
}

private static void LS(String folder) {
        // System.out.println(DateTime.getTimeStamp() + "LISTFOLDER Start");
        try {
```

```
        HBaseAPI hbaseAPI = new HBaseAPI();
        hbaseAPI.ListFolder(folder);
    } catch (Exception e) {
        System.out.println(DateTime.getTimeStamp() + "LISTFOLDER Fail:"
                + e.getMessage());
    } finally {
        // System.out.println(DateTime.getTimeStamp() + "LISTFOLDER End");
    }
}

private static void WRITEFOLDER(String sourceF, String destinationF) {
    System.out.println(DateTime.getTimeStamp() + "WRITEFOLDER Start");
    try {
        File sourceFolder = new File(sourceF);
        if (sourceFolder.isFile())
            throw new IOException("Source is not a folder");
        WriteFilesInFolder(sourceFolder, sourceF, destinationF);

    } catch (Exception e) {
        System.out.println(DateTime.getTimeStamp() + "WRITEFOLDER Fail:"
                + e.getMessage());
    } finally {
        System.out.println(DateTime.getTimeStamp() + "WRITEFOLDER End");
    }
}

private static void WriteFilesInFolder(File folder, String sourceF,
        String destinationF) throws IOException {
    if (folder.isFile()) {
        WRITE(sourceF, destinationF);
        return;
    } else {
        for (File file : folder.listFiles()) {
            if (file.isFile()) {
                WRITE(sourceF + "/" + file.getName(), destinationF + "/"
                        + file.getName());
            } else {
                WriteFilesInFolder(file, sourceF + "/" + file.getName(),
```

```
                            destinationF + "/" + file.getName());
                }
            }
        }

    }

    public static void WRITE(String source, String destination)
            throws IOException {
        //System.out.println(DateTime.getTimeStamp() + "WRITE Start");
        HBaseAPI hbaseAPI = new HBaseAPI();
        HDFSAPI hdfsAPI = new HDFSAPI();
        File file = new File(source);
        try {

            String HexSHA = SHACheckSum.SHA256(file);
            boolean rowExist = hbaseAPI.scanRowSize(HexSHA, file) > 0;
            boolean pathExist = hbaseAPI.scanQualifier(destination) > 0;
            if (!rowExist & !pathExist) {
                // row和size相同的紀錄和檔案不存在,路徑不存在
                //System.out.println(DateTime.getTimeStamp() + "加入新路徑,加入檔案");
                hbaseAPI.AddRow(HexSHA, destination,
                        String.valueOf(file.length()));
                hdfsAPI.AddFile(source, HexSHA);
            }
            if (!rowExist & pathExist) {
                // row和size相同的紀錄和檔案不存在,路徑已存在
                //  刪除舊路徑(也許Row和檔案也刪),加入新路徑,加入檔案
                //System.out
                //      .println(DateTime.getTimeStamp() + "刪除舊路徑,加入新路徑,加入檔案
");
                hbaseAPI.DelRowByPath(destination);
                hbaseAPI.AddRow(HexSHA, destination,
                        String.valueOf(file.length()));
                hdfsAPI.AddFile(source, HexSHA);
            }
            if (rowExist & !pathExist) {
```

```
            // row和size相同的紀錄和檔案已存在,路徑不存在,加入路徑後移除暫存檔
            //System.out.println(DateTime.getTimeStamp() + source);
            hdfsAPI.AddFile(source, "pool/" + HexSHA);
            if (hdfsAPI.StreamCompare(HexSHA)) {
                hbaseAPI.AddRow(HexSHA, destination,
                        String.valueOf(file.length()));
                hdfsAPI.delFilebyHash("pool/" + HexSHA);
                System.out.println( source+" "+file.length());
            } else {
                hbaseAPI.AddRow(HexSHA + "-" + destination, destination,
                        String.valueOf(file.length()));
                hdfsAPI.AddFile(source, HexSHA + "-" + destination);
            }
        }
        if (rowExist & pathExist) {
            // row和size相同的紀錄和檔案已存在,路徑已存在,移除暫存檔
            //System.out.println(DateTime.getTimeStamp() + source);
            hdfsAPI.AddFile(source, "pool/" + HexSHA);
            if (hdfsAPI.StreamCompare(HexSHA)) {
                hdfsAPI.delFilebyHash("pool/" + HexSHA);
                System.out.println( source+" "+file.length());
            } else {
                hbaseAPI.AddRow(HexSHA + "-" + destination, destination,
                        String.valueOf(file.length()));
                hdfsAPI.AddFile(source, HexSHA + "-" + destination);
            }
        }

    } catch (Exception e) {
        System.out.println(DateTime.getTimeStamp() + "WRITE Fail:"
                + e.getMessage());
    } finally {
        //System.out.println(DateTime.getTimeStamp() + "WRITE End");
    }

}
```

Appendix C: Java Code of HBASE API

```java
public class HBaseAPI {
    HTable table;

    public HBaseAPI() {
        Configuration HBASE_CONFIG = HBaseConfiguration.create();
        HBASE_CONFIG.set("hbase.zookeeper.quorum", "datanode2");
        try {
            table = new HTable(HBASE_CONFIG, "hash2file");
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public int scanRow(String rowKey) {

        try {

            Scan sc = new Scan();

            // sc.addFamily("myfamily".getBytes());
            RowFilter filter1 = new RowFilter(CompareOp.EQUAL,
                    new BinaryComparator(Bytes.toBytes(rowKey)));

            FilterList flist = new FilterList(FilterList.Operator.MUST_PASS_ALL);

            flist.addFilter(filter1);

            sc.setFilter(flist);

            ResultScanner rsan = table.getScanner(sc);

            Result rs = rsan.next();

            if (rs == null) {
```

```
            //System.out.println("scan row no record");
        } else {

            while (rs != null) {
                //System.out.println("scan row record:");
                List<KeyValue> kvList = rs.list();

                for (@SuppressWarnings("unused") KeyValue kv : kvList) {

                    //System.out.println(new String(kv.getRow(), "UTF-8")
                    //              + "->" + new String(kv.getQualifier(), "UTF-8")
                    //              + ":" + new String(kv.getValue(), "UTF-8"));
                    rs = rsan.next();
                }
                return kvList.size();
            }
        }
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return 0;
}

public int scanRowSize(String rowKey, File file) {
    try {

        Scan sc = new Scan();
        String size = String.valueOf(file.length());
        // sc.addFamily("myfamily".getBytes());
        RowFilter filter1 = new RowFilter(CompareOp.EQUAL,
                new BinaryComparator(Bytes.toBytes(rowKey)));

        FilterList flist = new FilterList(FilterList.Operator.MUST_PASS_ALL);
        ValueFilter filter2 = new ValueFilter(CompareOp.EQUAL,
                new RegexStringComparator(size));

        flist.addFilter(filter1);
```

```
            flist.addFilter(filter2);
            sc.setFilter(flist);

            ResultScanner rsan = table.getScanner(sc);

            Result rs = rsan.next();

            if (rs == null) {
                //System.out.println("scan row & size no record");
            } else {

                while (rs != null) {
                    //System.out.println("scan row & size record:");
                    List<KeyValue> kvList = rs.list();

                    for (@SuppressWarnings("unused") KeyValue kv : kvList) {

                        //System.out.println(new String(kv.getRow(), "UTF-8")
                        //              + "->" + new String(kv.getQualifier(), "UTF-8")
                        //              + ":" + new String(kv.getValue(), "UTF-8"));
                        rs = rsan.next();
                    }
                    return kvList.size();
                }
            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return 0;
    }

    public String GetHashByPath(String path) throws IOException {

        try {

            Scan sc = new Scan();
```

```
        sc.addFamily("path".getBytes());
        QualifierFilter filter1 = new QualifierFilter(CompareOp.EQUAL,
                new RegexStringComparator("^" + path + "(.*)"));

        FilterList flist = new FilterList(FilterList.Operator.MUST_PASS_ALL);

        flist.addFilter(filter1);

        sc.setFilter(flist);

        ResultScanner rsan = table.getScanner(sc);

        Result rs = rsan.next();

        if (rs == null) {
            throw new IOException("path not found");
        } else {

            while (rs != null) {
                List<KeyValue> kvList = rs.list();

                for (KeyValue kv : kvList) {
                    return new String(kv.getRow(), "UTF-8");
                }
            }
        }
    } catch (IOException e) {
        throw new IOException("path not found");
    }
    throw new IOException("path not found");
}

public int scanQualifier(String path) {

    try {

        Scan sc = new Scan();
```

```
        sc.addFamily("path".getBytes());
        QualifierFilter filter1 = new QualifierFilter(CompareOp.EQUAL,
                new RegexStringComparator("^" + path + "(.*)"));

        FilterList flist = new FilterList(FilterList.Operator.MUST_PASS_ALL);

        flist.addFilter(filter1);

        sc.setFilter(flist);

        ResultScanner rsan = table.getScanner(sc);

        Result rs = rsan.next();

        if (rs == null) {
            //System.out.println("scan Qualifier no record");
        } else {

            while (rs != null) {
                //System.out.println("scan Qualifier record:");
                List<KeyValue> kvList = rs.list();

                for (@SuppressWarnings("unused") KeyValue kv : kvList) {

                    //System.out.println(new String(kv.getRow(), "UTF-8")
                    //              + "->" + new String(kv.getQualifier(), "UTF-8")
                    //              + ":" + new String(kv.getValue(), "UTF-8"));
                    rs = rsan.next();
                }
                return kvList.size();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return 0;
}
```

```
public void AddRow(String Rowkey, String Path, String Size)
        throws IOException {
    Put put = new Put(Bytes.toBytes(Rowkey));
    put.add(Bytes.toBytes("path"), Bytes.toBytes(Path), Bytes.toBytes(Size));
    table.put(put);
}

public void DelRowByPath(String path) {

    try {

        Scan sc = new Scan();

        sc.addFamily("path".getBytes());
        QualifierFilter filter1 = new QualifierFilter(CompareOp.EQUAL,
                new RegexStringComparator("^" + path + "(.*)"));

        FilterList flist = new FilterList(FilterList.Operator.MUST_PASS_ALL);

        flist.addFilter(filter1);

        sc.setFilter(flist);

        ResultScanner rsan = table.getScanner(sc);

        Result rs = rsan.next();

        if (rs == null) {
            //System.out.println("scan Qualifier no record");
        } else {
            List<KeyValue> DelList = new ArrayList<KeyValue>();
            HDFSAPI hdfsAPI = new HDFSAPI();

            while (rs != null) {
                //System.out.println("scan Qualifier record:");
                List<KeyValue> kvList = rs.list();

                for (KeyValue kv : kvList) {
```

```
                    DelList.add(kv);
                    rs = rsan.next();
                }
            }
            for (KeyValue kv : DelList) {
                DELETEByQualifier(new String(kv.getRow(), "UTF-8"),
                        new String(kv.getQualifier(), "UTF-8"));
                if (this.scanRow(new String(kv.getRow(), "UTF-8")) == 0)
                    hdfsAPI.delFilebyHash(new String(kv.getRow(), "UTF-8"));
            }

public void ListFolder(String folder) {
    try {

        Scan sc = new Scan();

        sc.addFamily("path".getBytes());
        QualifierFilter filter1 = new QualifierFilter(CompareOp.EQUAL,
                new RegexStringComparator("^" + folder + "(.*)"));

        FilterList flist = new FilterList(FilterList.Operator.MUST_PASS_ALL);

        flist.addFilter(filter1);

        sc.setFilter(flist);

        ResultScanner rsan = table.getScanner(sc);

        Result rs = rsan.next();

        if (rs == null) {
            System.out.println("Folder is empty");
        } else {
            HashMap<String, FileAtt> hmFile = new HashMap<String, FileAtt>();
            HashMap<String, FileAtt> hmFolder = new HashMap<String, FileAtt>();
            while (rs != null) {
                List<KeyValue> kvList = rs.list();
```

```
        for (KeyValue kv : kvList) {
            FileAtt fileAtt = new FileAtt();
            fileAtt.Path = new String(kv.getQualifier(), "UTF-8");
            fileAtt.Size = new String(kv.getValue(), "UTF-8");
            fileAtt.FileName = fileAtt.Path
                    .replaceFirst(folder, "");
            if (fileAtt.FileName.startsWith("/")) {
                fileAtt.FileName = fileAtt.FileName.substring(1);
            }
            int slashIndex = fileAtt.FileName.indexOf("/");

            if (slashIndex > 0) {
                fileAtt.FileName = fileAtt.FileName.substring(0,
                        slashIndex);
                fileAtt.isFile = false;
                if (!hmFolder.containsKey(fileAtt.FileName))
                    hmFolder.put(fileAtt.FileName, fileAtt);
            } else {
                fileAtt.isFile = true;
                hmFile.put(fileAtt.FileName, fileAtt);
            }
            rs = rsan.next();
        }
    }
    for (FileAtt fileAtt : hmFolder.values()) {
        System.out.println("<" + fileAtt.FileName + ">");
    }
    for (FileAtt fileAtt : hmFile.values()) {
        System.out.println(String.format("%-30s %15s",
                fileAtt.FileName, fileAtt.Size));
    }
    System.out.println("Folder count:" + hmFolder.size()
            + "File count:" + hmFile.size());
}
```

Appendix D: Java Code of HDFS API

```java
public class HDFSAPI {
	FileSystem hdfs;
	static Configuration config;

	public HDFSAPI() throws IOException {
		config = new Configuration();
		config.set("fs.default.name", "hdfs://namenode:9000/");
		hdfs = FileSystem.get(config);
	}

	public void AddFile(String Source, String Destination) throws IOException {
		Path src = new Path(Source);
		Path dst = new Path("FD_HDFS/" + Destination);
		hdfs.copyFromLocalFile(false, true, src, dst);

	}

	@SuppressWarnings("deprecation")
	public static boolean deleteHDFSFile(String dst) throws IOException {

		FileSystem hdfs = FileSystem.get(config);

		Path path = new Path(dst);

		boolean isDeleted = hdfs.delete(path);

		hdfs.close();

		return isDeleted;
	}

	@SuppressWarnings("deprecation")
	public static void deleteDir(String dir) throws IOException {

		FileSystem fs = FileSystem.get(config);
```

```
        fs.~~delete~~(new Path(dir));

        fs.close();
}

public void getFilebyHash(String Source, String Destination)
                throws Exception {
        {
                Path src = new Path("FD_HDFS/" + Source);
                Path dst = new Path(Destination);
                hdfs.copyToLocalFile(src, dst);
        }
}

@SuppressWarnings("deprecation")
public void delFilebyHash(String hexSHA) throws IOException {
        Path src = new Path("FD_HDFS/" + hexSHA);
        hdfs.~~delete~~(src);
}

public boolean StreamCompare(String hexSHA) throws IOException {
        System.out.println(DateTime.getTimeStamp() + "Stream Compare start");
        FileSystem fs = FileSystem.get(config);
        Path newFile = new Path("FD_HDFS/pool/" + hexSHA);
        Path oldFile = new Path("FD_HDFS/" + hexSHA);
        FSDataInputStream newIS = null;
        FSDataInputStream oldIS = null;
        BufferedReader newReader = null;
        BufferedReader oldReader = null;
        try {
                newIS = fs.open(newFile, 64 * 1024 * 1024);
                oldIS = fs.open(oldFile, 64 * 1024 * 1024);
                newReader = new BufferedReader(new InputStreamReader(newIS));
                oldReader = new BufferedReader(new InputStreamReader(oldIS));
                String newline;
                String oldline;
                while ((newline = newReader.readLine()) != null
```

```
                && (oldline = oldReader.readLine()) != null) {
            if (newline.compareTo(oldline) != 0) {
                System.out.println(DateTime.getTimeStamp() + "Stream Compare not the same");
                return false;
            }

        }
    } catch (Exception e) {
        System.out.println(DateTime.getTimeStamp() + "Stream Compare Fail" + e.getMessage());
        return false;
    } finally {
        newIS.close();
        oldIS.close();
        newReader.close();
        oldReader.close();
    }
    System.out.println(DateTime.getTimeStamp() + "Stream Compare the same");
    return true;
}
```