Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2014, Article ID 561340, 11 pages
http://dx.doi.org/10.1155/2014/561340

Research Article

Design and Implementation of File Deduplication Framework on HDFS

Ruey-Kai Sheu,[1] Shyan-Ming Yuan,[2] Win-Tsung Lo,[1] and Chan-I Ku[3]

[1] *Department of Computer Science, Tung Hai University, Taichung, Taiwan*
[2] *Institute of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan*
[3] *Computational Intelligence Technology Center, Industrial Technology Research Institute, Hsinchu, Taiwan*

Correspondence should be addressed to Shyan-Ming Yuan; smyuan@cs.nctu.edu.tw

Received 10 January 2014; Accepted 2 March 2014; Published 10 April 2014

Academic Editor: Yue Shan Chang

File systems are designed to control how files are stored and retrieved. Without knowing the context and semantics of file contents, file systems often contain duplicate copies and result in redundant consumptions of storage space and network bandwidth. It has been a complex and challenging issue for enterprises to seek deduplication technologies to reduce cost and increase the storage efficiency. To solve such problem, researchers proposed in-line or offline solutions for primary storages or backup systems at the subfile or whole-file level. Some of the technologies are used for file servers and database systems. Fewer studies focus on the cloud file system deduplication technologies at the application level, especially for the Hadoop distributed file system. It is the goal of this paper to design a file deduplication framework on Hadoop distributed file system for cloud application developers. The architecture, interface, and implementation experiences are also shared in this paper.

1. Introduction

File systems are designed to control how files are stored and retrieved. Due to the unawareness of structure and semantics of file contents, a file system often contains duplicate copies of files which will result in redundant consumptions of storage space and network bandwidth. As the progress of internet social applications in recently years, such as Facebook [1], Youtube [2], and Dropbox [3], the frequency of file duplication also grows at an explosive rate when users share and synchronize files between each other.

It has been a complex and challenging issue for enterprises to reduce the redundant cost of storage spaces. According to the IDC [4], more than 80% of enterprises are seeking deduplication technologies to reduce cost and increase the storage efficiency. File deduplication is the task of identifying entities of the same real-world object [5]. In a company, files are usually stored in local disk or primary storages, network file repositories, document management systems, and secondary storages, such as backup storage or tapes. For internet service providers, files are geographically dispersed locations. Taking Google as an example, files are stored in Google file system [6]. Generally speaking, most companies consolidate the storage architecture in a mixture manner, and it further increases the complexity of file deduplication solutions design.

Recent advances of deduplication technologies are proposed based on the characteristics of applications. Maddodi et al. [7] proposed data deduplication techniques and analysis on data warehouse applications and tried to reduce the storage consumptions for database or online transaction processing. There are also literatures that tried to indicate the considerations while choosing file deduplication solution. For the granularity considerations, deduplication can work at either the subfile [8, 9] or whole-file [10] level. This would raise the trade-off evaluation issue in space savings and performance impacts. The more fine-grained deduplication creates more opportunities of space savings which may cause significant performance impacts.

While shifting the paradigm to the file deduplication of cloud systems, the trade-off of space savings and performance impacts are still major issues for data deduplication techniques. Chun-Ho Ng's LiveDFS focused on the deduplication of VM image in the Open-Source cloud and achieved at least

40% of storage saving for VM images storage with reasonable performance [11]. SAM [12] is a semantic-aware multitiered source deduplication framework for cloud backup system. It gets a high deduplication ratio which is as better as global chunk-based deduplication and very low overhead than that of global chunk-based deduplication. Shang and Li [13] pointed out several shortcomings of existing works and discussed the corresponding possible solutions for data deduplication for cloud systems. The challenging issues for cloud data deduplication are still the balance of trade-off between storage efficiency and performance.

There are still challenging issues missing in the previous studies, and they are the processing of large volume of hash data and the computation and search of hash index. It would be natural for cloud systems that the file sizes are larger than traditional storage and the number of files is enormously increasing from day to day. Less studies paid attentions to the practical cloud file system. And there is a strong demand for cloud application developers to have a simple and feasible data deduplication solution bundled with existing cloud file system. It is the purpose of this paper to propose a simple and feasible application framework of data deduplication for developers based on the most-popular Hadoop [14] system.

The reminder of this paper is as follows. Section 2 provides background and motivation of the work; Section 3 describes the design of our deduplication framework; Section 4 describes the implementation; Section 5 evaluates the implementation; and, finally, we give concluding remarks and future works in Section 6.

2. Background

2.1. Data Replication in HDFS. Hadoop is widely used and plays the role of mass data storage in the cloud environments equipped with no data deduplication technologies. Take the following simple experiment as example. Once three identical files are stored to the HDFS using different file names, the HDFS will at least reserve three disk spaces for each file, respectively. That means that no file deduplication technologies are leveraged in the HDFS even the contents of these three files are the same. Considering the scenario of the sharing of a popular Youtube video between millions of subscribers, without the deduplication mechanism, it could waste very large volume of disk storage. To meet the requirements of HDFS applications, two data deduplication schemes are proposed based on the application characteristics. They are the FD-HDFS which stands for File Deduplicated HDFS and the RFD-HDFS which stands for Reliable File Deduplicated HDFS. RFD-HDFS guarantees the data reliability with deduplication capability. FD-HDFS provides high performance for data deduplication with negligible overhead if the application can tolerate minor errors.

2.2. Detection of Similar Data. Generally speaking, the process of detection of identical files is mainly based on two levels of granularities, and they are the file level and the data block level. For whole-file level detection, the data mapping is conducted through the hash technology [10, 15]. As for the data block level detection, the fingerprint is checked through the fixed-sized partition or the check and deletion of duplicate data is conducted through the detection technology of content-defined chunking and the sliding block technology. Most of the detection of the same data tries to identify the similarity of data characteristics by the shingle and the bloom filter techniques [16, 17]. 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.

2.3. Source and Target Deduplication. Typically, data-intensive applications, such as backup and replication, files are moved from source to the target storage devices through the network. The source end is the front-end application and the Host or the Backup server to process and store the raw data. The target end is usually ultimate storage equipment, such as VTL or disk arrays. If unnecessary or redundant data is indicated and deleted in the front-end application, it will reduce the network transmission bandwidth and time. The disadvantage of data cancelling at the front-end application is that it will cost more for duplicate data detection and deletion. If the data deduplication task is done at the target end, it will spend more resources in redundant data processing, data transmission over networks, and computing resource consumption for redundant data detection. Therefore, it is recommended that source deduplication is more feasible than the target one.

2.4. In-Line and Postprocessing. In-line data deduplication is the process that redundant data deletion is executed synchronously once the data of instructions of backup, copy, or writing is sent to the disk. In other words, when the data is copied for the preparation to the target end through the network or once the back-end storage device receives the source data via the internet and prepares to write to the disk, the data deduplication process will start the data content comparison and deletion tasks at the same time. Postprocessing of data deduplication means that the deletion of redundant data job starts after the data is completely written to the disk. It might be triggered by specific criteria or scheduled periodically by some instructions.

The advantages and disadvantages of online in-line processing and postprocessing are just at the opposite. Data comparison and deletion computing consume lot of processor resources. If the online real-time processing architecture is adopted, the system performance will be clearly temporized and the backup speed will be delayed. On the other hand, for in-line processing, since the deletion computing has been conducted before the data is written to the disk, it needs less data space than postprocessing. Contrarily, the system performance will not be affected for postprocessing, and we can start the deduplication tasks at off-peak hours. Before the deduplication tasks start, all data have to be stored in the disk for postprocessing. It means that postprocessing mechanism will occupy more disk storage than the in-line processing.

2.5. Whole-File or Chunk-Based Data Deduplication. Meyer and Bolosky [18] proposed 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. In our study, despite environment homogeneity, Hadoop system has characteristics of diversity of file volumes and sizes as well as the deep hierarchical namespace. This would lead more challenges to the chunk-based data deduplication techniques. Based on this consideration, in this paper, the whole-file data deduplication techniques are used for our proposed framework.

2.6. HDFS Hash Functions. The Hadoop distributed file system is a distributed file system designed to run on commodity hardware and to be used to replace the high-priced servers. 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 storages. Based on HDFS, MapReduce is the most popular mechanism which may analyze the data and create the metadata of file for file searching. It might be used to detect the similarity of different files in the future. SHA-2 (Secure Hash Algorithm 2) is a set of cryptographic hash functions (SHA-224, SHA-256, SHA-384, and SHA-512) designed by the National Security Agency (NSA) and published in 2001 by the NIST as a US Federal Information Processing Standard [15]. 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, 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 [19].

The impact of collisions depends on functions used by applications to identify similar data sets. For example, hash functions and fingerprints are used by applications to identify similar 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 to collisions between very different inputs.

The aforementioned technologies are widely used in the storage devices of enterprise File Server, Database, NAS (RAID), and Backup Devices. However, there is no related implementation on Hadoop since HDFS has its special block mode and network topology demanding to ensure the reliability of the copy instruction [20]. That is, the detection of similar or the same data is not just the conditions of removing duplicate files. Besides the space or time consumption considerations, the proposed algorithm or framework should also take the reliability and overhead of recovery process for frequently happened hard errors that is the reason we suggest to use Stream Compare function for whole-file level for the same data detection in HDFS.

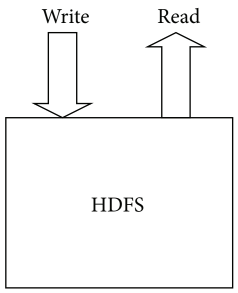

Figure 1: Programming model of HDFS write and read.

3. System Design

3.1. Original HDFS. Figure 1 shows the original HDFS programming model for data read and write. Users may access file by the Hadoop shell command or Hadoop API. HDFS is the Hadoop Distributed file system. User uses the Write API to upload files to HDFS and the Read API to download files from HDFS.

An HDFS client can invoke the create instruction in Figure 2 to HDFS, and then the HDFS will create a NameNode for it. Once the NameNode is got, the client can write data to FSData OutputStream which will write data packages to DataNode with acknowledgements. Similar to the write instruction, an HDFS client can use the open instruction shown in Figure 3 to get data block locations and perform the read instruction from DataNode.

3.2. 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 [21] can be used to speed up file comparison. At the same time, the postprocessing method is adopted to reduce user's waiting time. While files are first uploaded, they will be stored into a temporary Storage Pool. A data similarity detector process will then start the file comparison tasks to check whether the files are duplicated or not. In this paper, the stream comparison is used to partially retrieve data fragments to conduct Binary Compare in the sequential serial method.

As shown in Figure 4, we use the following three steps to determine whether the files are the same:

(a) check whether the Hash value exists or not;

(b) if cache hit in the Hash table, then check whether the file size is the same;

(c) perform the stream comparison.

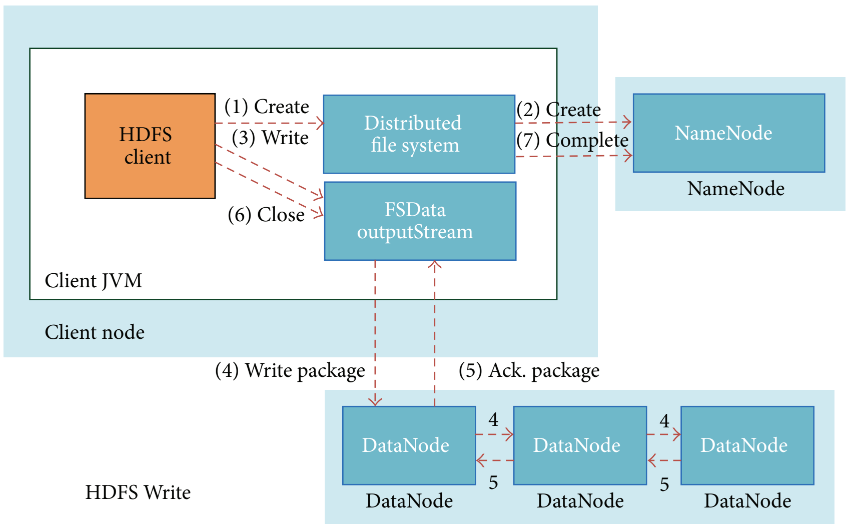

Figure 2: Instruction invoking path of HDFS write.

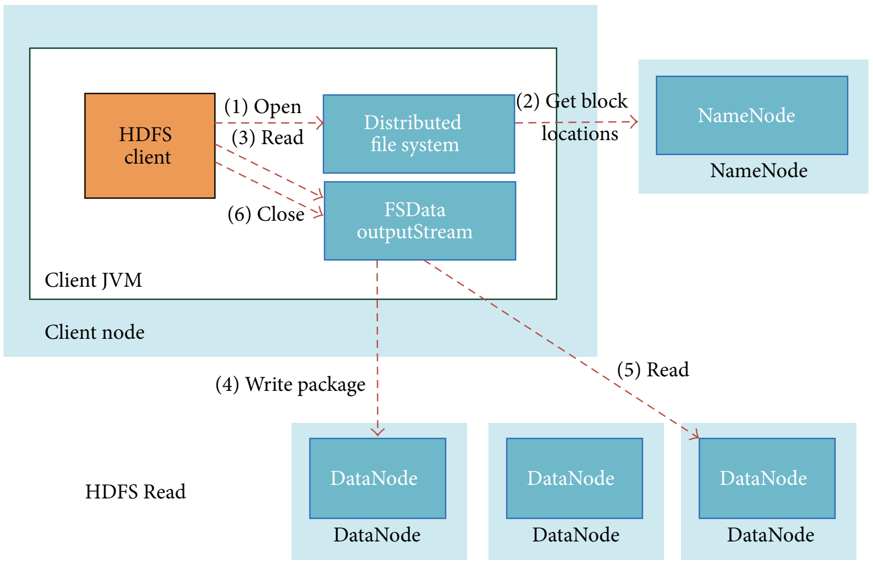

Figure 3: Instruction invoking path of HDFS read.

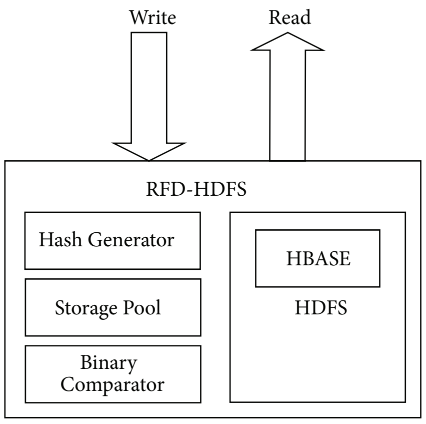

Figure 4: RFD-HDFS framework.

Once any difference is found in the comparison work in each phase, the comparison will be immediately stop to save computing and memory 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.

3.3. FD-HDFS. In the application where little errors can be tolerated, such as web information extraction 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, can reduce the network bandwidth, can save upload time, and even can reduce the burden of NameNode and HBase. Taking the web crawler software for example, Reiss and Resiss [22] daily need to capture page files to HDFS. In accordance with the comparison of the HASH value and the HBASE database, it can quickly learn whether the website content changes or not. Then, it can save the time for Binary Compare and the target can be retrieved by 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 newly added and changed files, which saves not only upload time and the bandwidth but also server side loading.

For applications which can tolerate some errors, such as video sharing or mass file sharing, the FD-HDFS provides a high-performance and low-storage consumption solution for file deduplication. As shown in Figure 5, the Hash Generator is used to compute the hash values for firstly coming files and save storage spaces once hash collision happened.

4. System Implementation

4.1. Implementation of RFD-HDFS. Figure 4 shows the proposed RFD-HDFS framework. A wrapper interface is defined for HDFS clients, and it will provide reliable data deduplication functions for clients and invoke the HDFS write instruction in the background processes. Besides the wrapper interface, there are still three major components in the framework, and they are the Hash Generator, Storage Pool, and the Binary Comparator. The HBase table records the mapping between Hash key and Full Path of file. Hash Generator can be implemented using SHA2 and SHA3. Binary Comparator could be a circuit of hardware or MapReduce function of Hadoop. Storage Pool is the temporary file pool for post processing; the Binary Comparator will load the file and do comparisons in background. All files stored into pool will be logged for tracking and could be roll back for fail over purposes.

Figure 6 shows the detail of RFD-HDFS Write algorithm. There are two parallel tasks in the RFD-HDFS Write operation. The first one is controlled by the write controller and is responsible for storing files into HDFS by invoking the HDFS *AddFile* API. It first calculates the Hash value by HASH Generator and then stores the value into HBase by HMaster component for further lookup. After that, two physical files are stored into HDFS and a temporary file pool, respectively. The file stored in HDFS is the one of original call path of HDFS. The one in the temporary file pool will be used for collision detection which is done by the second task in the background. A Background Worker process is designed to calculate the hash value for each files in the temporary file pool. Once more than one file has the same hash value, the Background Worker will then update the recode of HBase and link the physical HDFS file for those files of the same hash value. In this implementation, after the processes of Background Worker, the files in the temporary file pool will be still stored for further usage once a collision occurred. Once two files of the same hash value are retrieved, the RFD-HDFS will return those files to user with correct file content. It will be selective for users to invoke RFD-HDFS Write or FD-HDFS Write if they can prioritize the storage capacity consumption and accuracy of file contents.

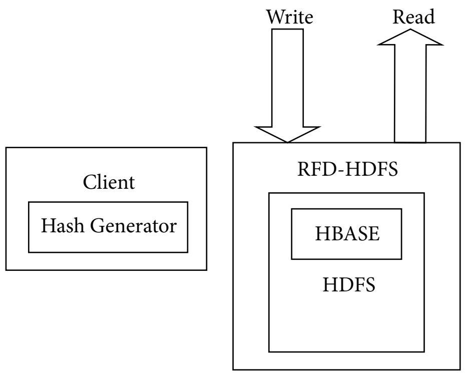

Figure 5: FD-HDFS framework.

The detailed implementation of RFD-HDFS Read algorithm is shown in Figure 7. Similar to the RFD-HDFS Write instruction, the RFD-HDFS Read instruction wrapped the original HDFS *getFilebyHash* API by calculating the hash value of a file with some exception handling logics. The Read Controller is firstly invoked by a RFD-HDFS client and triggers the Hash Generator to calculate the hash value of the target file. The key of the implementation of the RFD-HDFS Read instruction is that the reliability service guarantee is transparent to users, and once a hash collision happened, the HDFS files or the files in the temporary file pool will be returned to users for reliability service guarantee purpose.

4.2. Implementation of FD-HDFS. If the minor error is acceptable, we can ignore the collision of hash value. As shown in Figure 5, we can revise the RFD-HDFS architecture by moving the Hash Generator to the client side and removing the Binary Comparator and Storage Pool from the server side.

The FD-HDFS Write algorithm is shown in detail in Figure 8. The FD-HDFS Write instruction wrapped the HDSF *AddFile* API by firstly calculating the hash value of the given file by invoking the Hash Generator component and then checking whether it has been stored in the HDFS or not by invoking the HMaster component. Once it is hit in the HBase table, the request will be returned right away no matter how large the file size is because no data transmission is needed for this case. Once the hit is missed; then a file transmission from the client side to HDFS is then begun, and the hash value is recorded into HBase table after the file is completely written to the HDFS. Comparison to the RDF-HDFS Write, there is no need for the temporary file pool, and the performance for writing files into HDFS is much faster. But, due to the existence of possibility hash collision between files, the FD-HDFS is not as reliable as the RFD-HDFS.

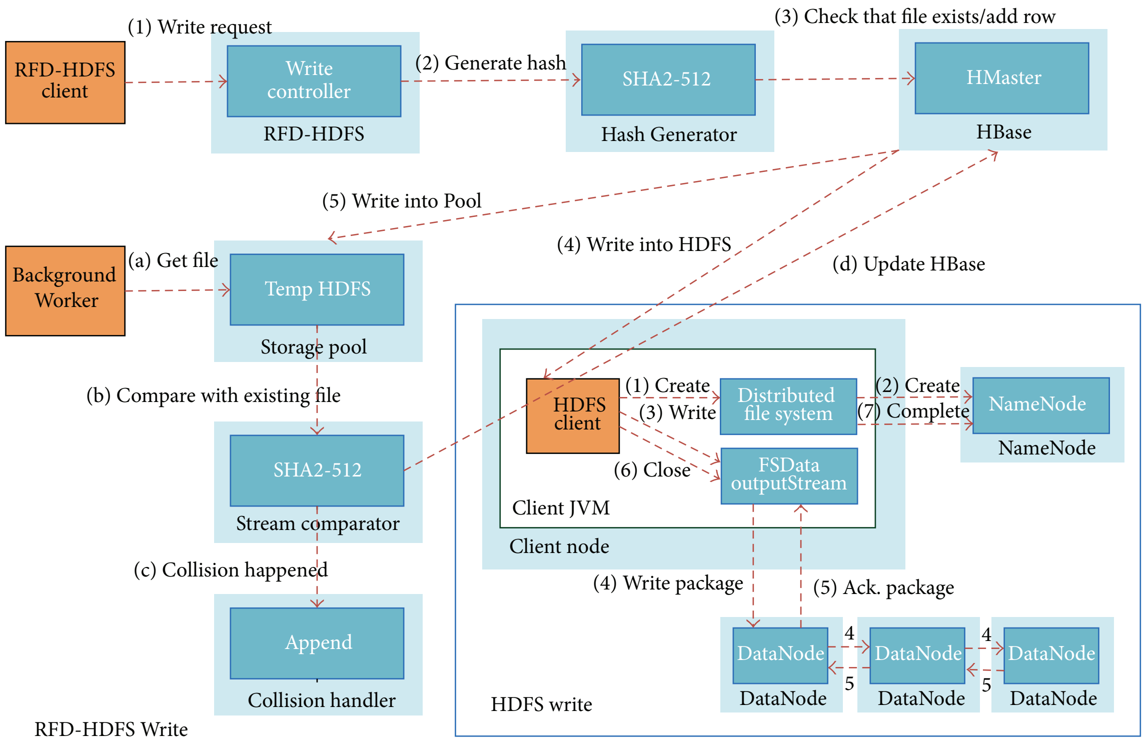

Figure 6: Instruction invoking path of RFD-HDFS Write.

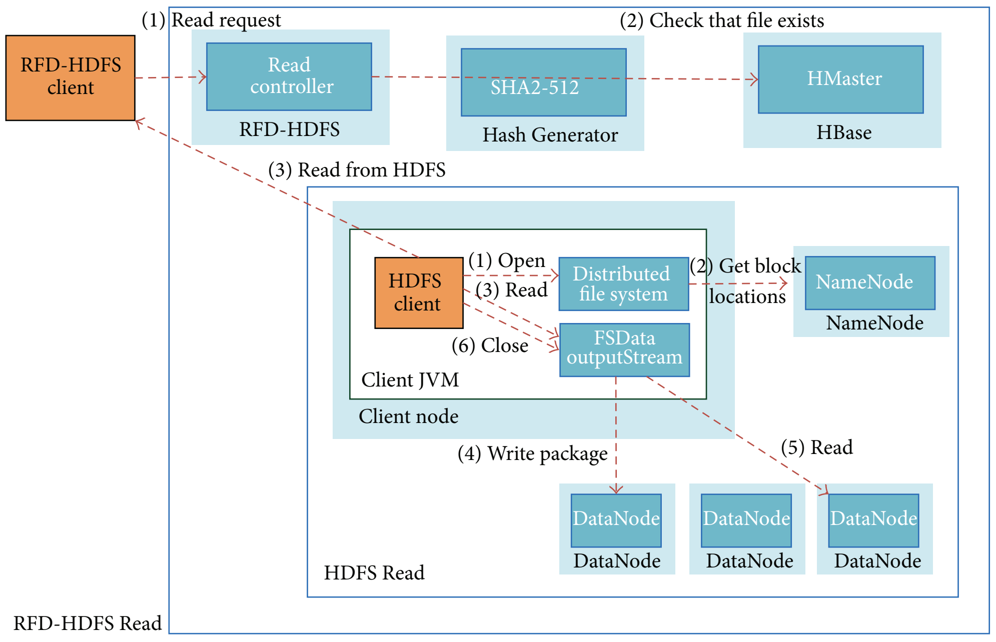

Figure 7: Instruction invoking path of RFD-HDFS Read.

Figure 9 illustrates the detail of FD-HDFS Read algorithm. While the FD-HDFS Client calls the Read instruction, it will firstly calculate the hash value for the given file locally. Then, two parallel tasks are issued for further data processing. The first one is to check whether the file is already deduplicated or not by HMaster to check the existence of hash value in the HBase. If the hash value of the given file exists, and it is already read by the client, a local cache of previously loaded file content with file path will be returned. If the file is not previously loaded, then a normal HDFS read invoking

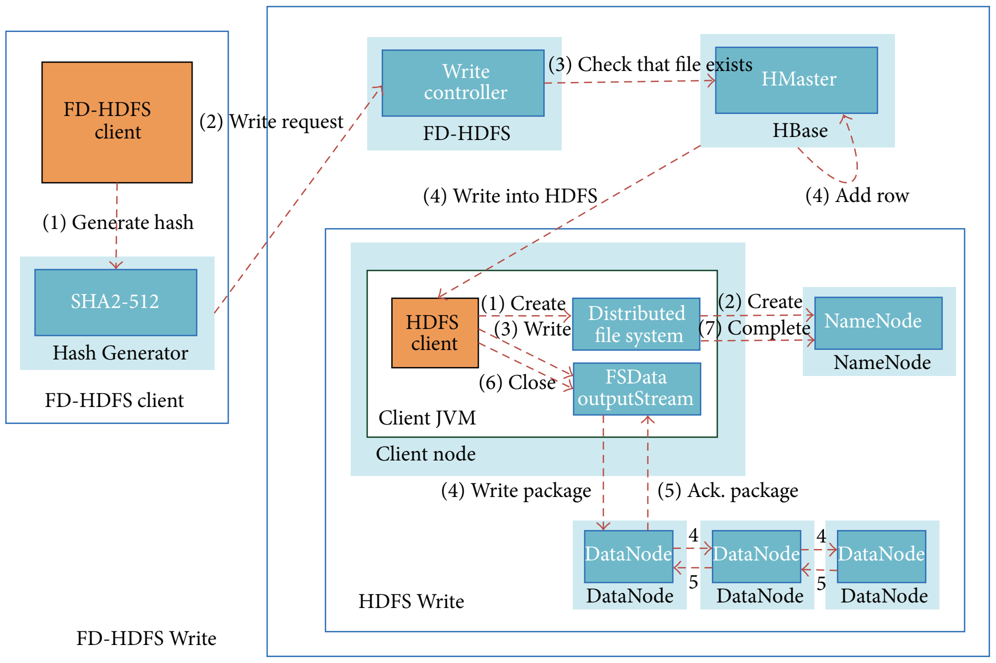

Figure 8: Instruction invoking path of FD-HDFS Write.

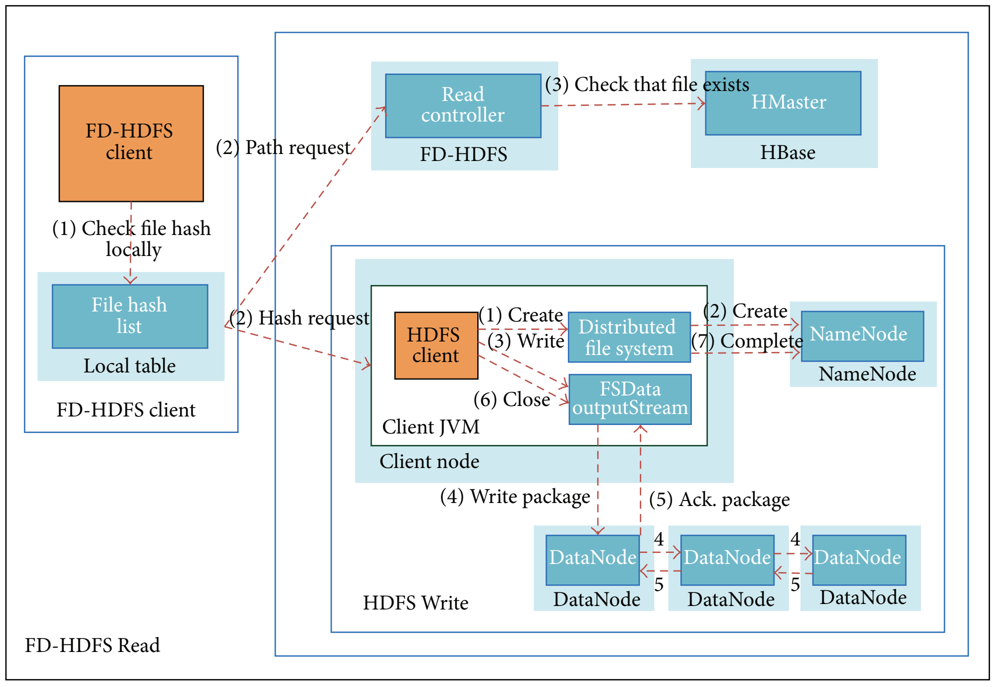

Figure 9: Instruction invoking path of FD-HDFS Read.

Table 1: Hadoop node settings.

Host name	Items			
	OS	IP	HBase role	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	

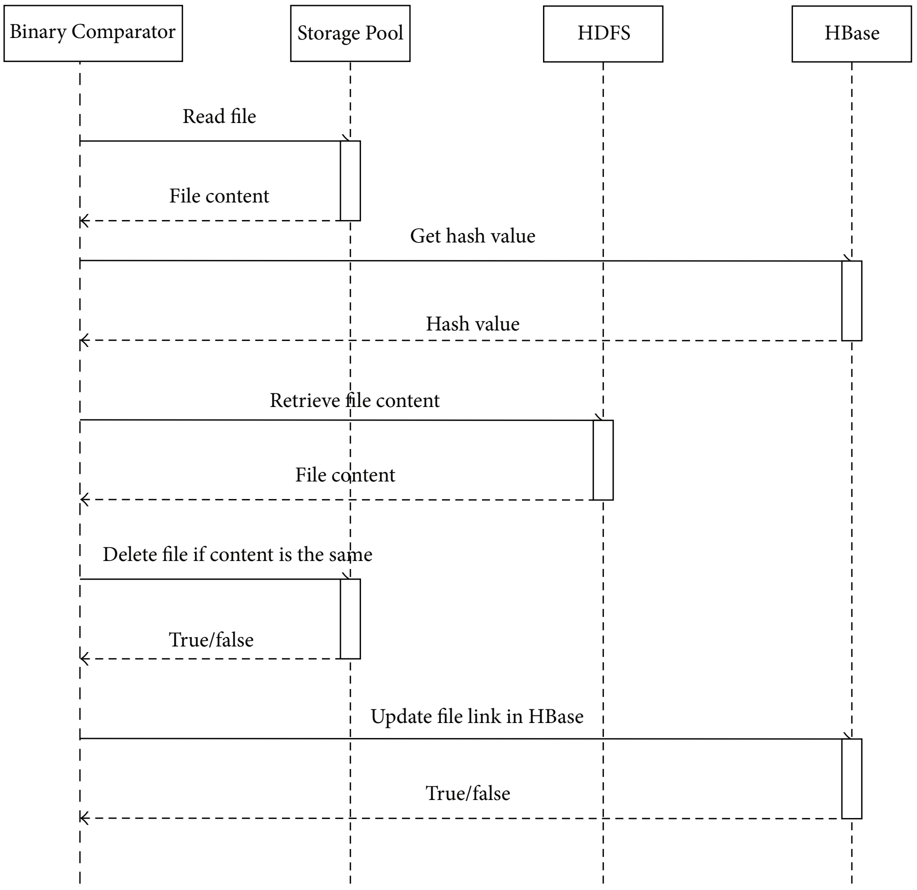

FIGURE 10: The sequence diagram of data similarity detector.

TABLE 2: The schema and samples on HBase table.

Row key	Time stamp	Full path	File attributes		
			Permissions	Size	Time
File HASH	T_1	Test/apache-solr-4.0.0.tgz	-rwxr-xr-x	200 M	T_{11}
File HASH	T_2	Test/apache-solr-4.0.1.tgz	-rwxr-xr-x	200 M	T_{12}
File HASH	T_3	Test/apache-solr-4.0.2.tgz	-rwxr-xr-x	200 M	T_{13}

process will be issued in the other task. The key of FD-HDFS Read is that the file deduplication is transparent to users. That is, once files of the same hash value are already loaded into the HDFS environment, the file contents will be returned right away from local cache to reduce the file transmission. On the other hand, if the file is not previously loaded, the original HDFS file read request will be issued as normal HDFS file read instruction.

4.3. Implementation of Data Similarity Detection. Data similarity detection is a postprocessing procedure and responsible for checking the same or similar data after the data files is received by the proposed deduplication framework. For both Write instructions of RFD-HDFS and FD-HDFS, if a similar record exists in HBase, it means that there is at least one file having the same content that is stored in HDFS. If the existing file path is the same as the given file, there is no need to write the file into HDFS. That is, no more file transmission is needed from client to HDFS server for FD-HDFS because the HDFS can get the file content from itself. If the existing file path is not the same as the given file, the RFD-HDFS will need to create a new record in HBase and store the file into the temporary file pool to prevent hash collision and guarantee the reliability of further file content retrieve. Again, for file Read instructions, the data similarity detector can help to enhance file downloading performance for FD-HDFS, if the hash value exists in the HBase table.

Figure 10 shows the detailed implementation of data similarity detector for the Background Worker component of the RFD-HDFS framework. To simplify the implementation of RFD-HDFS, when a file is written to the RFD-HDFS, as shown in Figure 6, it writes the file to HDFS and creates a new record in HBase. Then the Background Worker will deduplicate the redundant files in the background by excusing

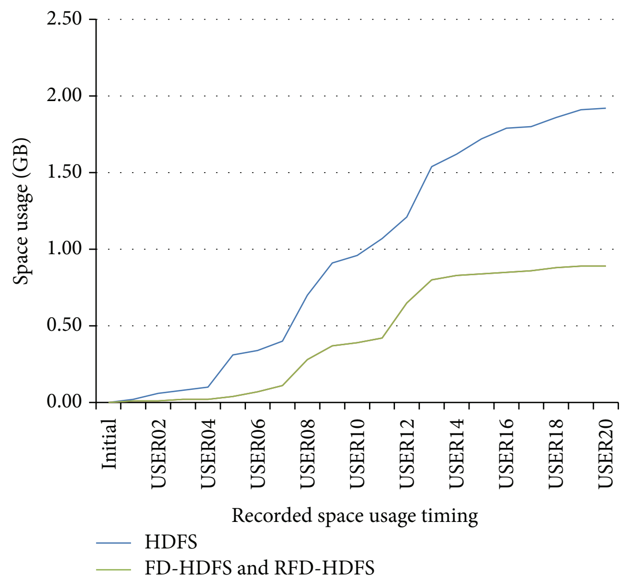

Figure 11: Comparison of HDFS and our proposed deduplication framework.

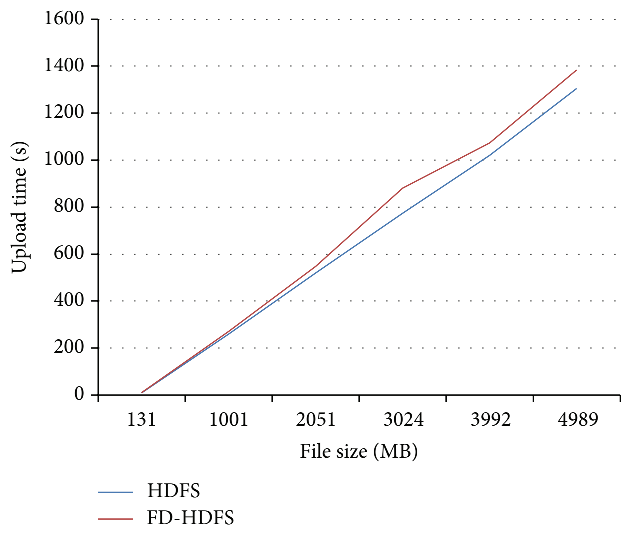

Figure 12: Performance comparison between HDFS and FD-HDFS.

the data similarity detection process. A Binary Comparator is designed for this purpose, and it firstly retrieves files of the same hash value from Storage Pool and HDFS. Then, it compares the content of these files. Once the contents are the same, The Binary Comparator deletes the files in Storage Pool for file deduplication purpose. If the contents are not the same, it will keep the files in the Storage Pool for further file retrieve requests and replacing the HBase records with correct physical links of files.

Table 3: Test data.

Host name	Items			
	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

5. Experimental Analysis

In this paper, we assume that the storage serves as an EndNote [22] storage over cloud and we import the collection of papers from 20 students. Also, we simulate users uploading all the files to the HDFS.

5.1. Experimental Environment. The settings for each Hadoop node are listed in Table 1. Table 2 gives the schema and samples of HBase. Table 3 gives the file count, total file size, and the percentage of duplicated files. The experiments were running on VMware WorkStation 9, the hard disk drive size is 1TB, and the rotation speed is 5400 rpm. All of nodes are deployed on a same host.

5.2. Disk Space Consumption Evaluation. In this experiment, around 2,000 files are uploaded into HDFS before the experiment starts. The test scenario repeats uploading files into HDFS and records the amount size of used disk space.

The results are listed in Table 4 and shown in Figure 11 which illustrates that, after 20 users uploading their files, the space usages of RFD-HDFS and FD-HDFS are much lower than normal HDFS. As shown in Table 3, 40.9% of uploaded test files are duplicated ones. The result shows that only 46% of HDFS disk space is used for both FD-HDFS and RFD-HDFS. It appears that the proposed systems can effectively reduce the disk usage of duplicated files.

Table 4: Test results.

Time	HDFS type		
	HDFS (GB)	FD-HDFS (GB)	RFD-HDFS (GB)
Initial	0.00	0.00	0.00
USER01	0.02	0.01	0.01
USER02	0.06	0.01	0.01
USER03	0.08	0.02	0.02
USER04	0.10	0.02	0.02
USER05	0.31	0.04	0.04
USER06	0.34	0.07	0.07
USER07	0.40	0.11	0.11
USER08	0.70	0.28	0.28
USER09	0.91	0.37	0.37
USER10	0.96	0.39	0.39
USER11	1.07	0.42	0.42
USER12	1.21	0.65	0.65
USER13	1.54	0.80	0.80
USER14	1.62	0.83	0.83
USER15	1.72	0.84	0.84
USER16	1.79	0.85	0.85
USER17	1.80	0.86	0.86
USER18	1.86	0.88	0.88
USER19	1.91	0.89	0.89
USER20	1.92	0.89	0.89

Table 5: Test data of performance evaluation.

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

Table 6: Test result of performance evaluation.

File size	DFS file upload time (second)	
	HDFS	FD-HDFS
131	9	10.4
1001	260	270.6
2051	520	548.3
3024	774	881
3992	1020	1073
4989	1305	1383

5.3. File Upload Performance Evaluation. In this experiment, only the performance of FD-HDFS is evaluated. The performance of RFD-HDFS is very similar to FD-HDFS, since RFD-HDFS adopts postprocessing for file deduplication. Only when data similarity detector works, the performance may be affected. Totally, six files are used in this experiment. As shown in Table 5, the file sizes are ranged from 131 MB to 4989 MB. The test scenario is to upload files one by one and record the upload time for each.

The results of performance evaluation are shown in Table 6. It shows that the upload performance of FD-HDFS is comparable to HDFS. Figure 12 illustrates the trend of upload time. It appears that HDFS upload time is linearly increasing and the trend of FD-HDFS upload time is very similar to HDFS upload time. The deduplication processing did not affect the upload performance significantly, and the overhead is almost neglectable.

6. Conclusion

In this paper, the deduplication framework based on HDFS is proposed. The details of design and implementation of RFD-HDFS and FD-HDFS are shared and evaluated. RFD-HDFS is suitable for applications which cannot have any errors, such as financial or nuclear related applications. On the other hand, FD-HDFS could be used for most applications with acceptable errors. The experiments results show that the duplicated data space can be saved and the upload performance is not affected by the integrated schemes significantly. That is, the proposed framework indeed reduces the storage space consumption by removing redundant files for HDFS.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by the National Science Council of the Taiwan, Taiwan, under Grant no. 101-2221-E-009-034-MY2. The authors also would like to express their deep appreciation to all anonymous reviewers for their kind comments.

References

[1] Facebook, http://www.facebook.com/.

[2] Youtube, http://www.youtube.com/.

[3] DropBox, http://www.dropbox.com/.

[4] L. DuBois, M. Amaldas, and E. Sheppard, “Key considerations as deduplication evolves into primary storage,” White Paper 223310, March 2011.

[5] H. Köpcke and E. Rahm, “Frameworks for entity matching: a comparison,” *Data and Knowledge Engineering*, vol. 69, no. 2, pp. 197–210, 2010.

[6] G. Sanjay, G. Howard, and L. Shun-Tak, “The google file system,” in *Proceedings of Symposium of Operating System Principle*, pp. 19–22, New York, NY, USA, October 2003.

[7] S. Maddodi, G. V. Attigeri, and A. K. Karunakar, “Data deduplication techniques and analysis,” in *Proceedings of the 3rd International Conference on Emerging Trends in Engineering and Technology (ICETET ’10)*, pp. 664–668, November 2010.

[8] C. Dubnicki, L. Gryz, L. Heldt et al., "Hydrastor: a scalaable secondary storage," in *Proceedings of 7th USENIX Conference on File and Storage Technologies*, 2009.

[9] C. Ungureanu, B. Atkin, A. Aranya et al., "Hydrafs: a high-throughput file system for the hydrastor content-addressable storage system," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, 2010.

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

[11] C. H. Ng, M. Ma, T. Y. Wong et al., "Live deduplication storage of virtual machine images in an open-source cloud," in *Proceedings of the 12th International Middleware Conference*, pp. 80–99, 2011.

[12] Y. Tan, H. Jiang, D. Feng, L. Tian, Z. Yan, and G. Zhou, "SAM: a semantic-awaremulti-tiered source de-duplication framework for cloud backup," in *Proceedings of the 39th International Conference on Parallel Processing (ICPP '10)*, pp. 614–623, September 2010.

[13] Y. Shang and H. Li, "Data deduplication in cloud computing systems," in *Proceedings of International Workshop on Cloud Computing and Information Security*, pp. 483–486, 2013.

[14] Hadoop, http://hadoop.apache.org/.

[15] D. Eastlake and P. Jones, "US secure hash algorithm 1," IETF RFC 3174.

[16] J. G. Conrad and E. L. Raymond Jr., "Essential deduplication functions for transactional databases in law firms," in *Proceedings of the 11th International Conference on Artificial Intelligence and Law*, pp. 261–270, June 2007.

[17] J. Zhang, S. Zhang, Y. Lu, X. Zhang, and S. Wu, "Hierarchical data deduplication technology based on bloom filter array," in *Proceedings of the International Conference on Information Engineering and Applications (IEA '13)*, vol. 216 of *Lecture Notes in Electronic Engineering*, pp. 725–732, Springer, 2013.

[18] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pp. 1–13, 2011.

[19] J. J. Rao and K. V. Cornelio, "An optimized resource allocation approach for data-intensive workload using topology-aware resource allocation," in *Proceedings of IEEE International Conference on Cloud Computing and Emerging Markets*, pp. 1–4, 2012.

[20] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[21] B. Nutch, "Open source search," *Queue*, vol. 2, pp. 54–61, 2004.

[22] M. Reiss and G. Resiss, "EndNote 5 reference manager—functions—improvements—personal experiences," *Schweizerische Rundschau für Medizin Praxis*, vol. 91, no. 40, pp. 1645–1650, 200.