A Method of Recovering HBase Records from HDFS Based on Checksum File

Lin Zeng[1], Ming Xu[1], Jian Xu[1], Ning Zheng[1], and Tao Yang[2(✉)]

[1] Internet and Network Security Laboratory,
School of Computer Science and Technology,
Hangzhou Dianzi University, Hangzhou, China
{l4l050046, mxu, jian.xu, nzheng}@hdu.edu.cn
[2] Key Lab of Information Network Security of Ministry of Public Security,
Hangzhou, China
yangtao@stars.org.cn

Abstract. Data recovery is a key problem in disaster recovery and digital forensics fields. The HDFS (Hadoop Distributed File System) is widely used for storing high-volume, velocity and variety dataset. However, previous work about data recovery mainly focuses on personal computers or mobile phones, and few attentions have been taken to HFDS. This paper analyzes the feature of HDFS and proposes a recovery method based on checksum file in order to address the records recovery problem of HBase, which is a common application on HDFS. We first carve out the Data blocks of HFile (HBase data file) using the corresponding checksum file, then analyze the format of HBase table records to extract them from the carved Data blocks. The experiments demonstrate that our method can restore HBase records effectively. The recovery rate is nearly 100% when the cluster size is 4 KB and 2 KB.

Keywords: HBase · HDFS · Records recovery · File carving · HFile

1 Introduction

HDFS is the file system primarily used by Hadoop which is currently the commonly used open-source software framework for big data. It is an abstracted file system layer that stores data in its own format to enable cross-platform functionality. Data in HDFS may be lost due to human error, device malfunction, or deliberate deletion, which highlights the need for recovering it.

File carving is an effective technology to solve the recovery problem. It does not rely on the file system meta-data and just recovers data from unstructured binary data stream (i.e. the original copy of a digital device) by analyzing the raw data contents or structures. It overcomes the disadvantages existing in traditional data recovery methods when the file system meta-data information is corrupted or lost [17].

When a file is deleted, its contents are often not actually erased but only the file system meta-data that points to file contents is altered or disabled. The actual storage of HDFS resides in the host operating system's file system, such as NTFS and ExtX.

S. Wang and A. Zhou (Eds.): CollaborateCom 2016, LNICST 201, pp. 127–139, 2017.
DOI: 10.1007/978-3-319-59288-6_12

Therefore, even if a file in HDFS is deleted, there is the possibility to recover it from the storage devices.

This work mainly aims at HBase which is a commonly used application of HDFS. We use file carving technology for recovering the Data blocks of HFile (HBase data file). Because HFile in HDFS may split into a number of blocks stored across multiple machines. Piecing all blocks together into a complete file is difficult. But we can carve out the Data blocks of HFile from a single node and recover records from them. The main challenge of file carving is the fact that file data stored within a file system may not in a linear or continuous way [11]. So the most important issue is that identifying the clusters that belong to the same file and reassembling them into a correct order.

Existing file carving technology is aimed at the typical file type (i.e. JPEG, ZIP, PNG, WORD, PDF, MPEG etc.) on personal computers and mobile phones. Current work often identifies the clusters by specific file structure or the feature of file content. But the only feature can be used in HFile Data block is the header signature and the length embedded in the header. In addition, the content is just key-value pair which has no specific format. And using file structure is very suited for the situation that a file is unfragmented. Existing methods are not suited for the recovery problem of HFile Data block. Given this, we analyze the HDFS feature and propose a carving method using checksum file which can identify and reassemble the file clusters easily. Our checksum file based method works well for HDFS files even under the highly fragmented situation.

The remainder of the paper is organized as follows. First, we position the related work in Sect. 2, and analyzing the checksum file in HDFS and the format of HFile in Sect. 3, then we outline our methodology and evaluate our empirical results in Sects. 4 and 5 respectively. In Sect. 6, we draw some conclusion and discuss the limitations and the potential options for future work.

2 Related Work

The earliest research of file carving began from the signature-based restoration technique [1]. This method simply searches for a known header signature and the corresponding footer signature, then try to merge all clusters in-between them. Some forensic tools provide this type of recovery, e.g. EnCase [4], X-Ways Forensics [8], Autopsy [7], etc. Unfortunately, this simple header-footer carving method is only applied to the files which are unfragmented.

Garfinkel [2] extended the signature-based method by using additional information stored in the file header. For some files, file header may contain file size or length. If the file footer can't be found, the information can be utilized to extract a file. The author also utilized the fast object validation for recovery of bifragmented files (the file that is broken into two fragments). The method places a gap between the header and footer, and repeats the process by growing the gap until a validation is found. But the approach is limited to the file which has two fragments and the gap is not large.

In [3], the authors proposed a method to carve image files based on graph theory. They redefined the reassembly of files into a path optimizing problem. Every cluster is considered as a vertex, and the edge between vertexes is weighted based on the

weighting function which is determined by the likelihood that one cluster follows another. The goal of reassembly problem is to find k-vertex disjoint paths but that is too costly and complex.

Cohen [13] considered the problem of carving fragmented files as being the procedure of estimating a mapping function between the file bytes and the image bytes. The core idea is based on the reason that the clusters of storage media can only belong to one file. File recovery procedure consists of generating all possible mapping functions and they are evaluated by the corresponding validator. Cohen used this approach to carve the PDF and ZIP files.

Gi-Hyun Na et al. [10] proposed the frame-based recovery of corrupted video files using the specifications of video codec. It addressed the recovery problem of a video file which was severely fragmented or even partly overwritten. Johan et al. [11] presented an advanced tool JPGcarve for automated recovery of fragmented JPEG files. JPGcarve contains four main methods: cluster size estimation, single-fragment carving, space reduction and multi-fragmented carving.

The literatures on file recovery only reported the algorithms for the typical file type on personal computers and mobile phones. These approaches cannot extend to HDFS common files directly. In this paper, we analyze the feature of HDFS and try to recover Data blocks of HFile based on the checksum file and extract records from them.

3 Background

3.1 The Checksum File in HDFS

HDFS has its own architecture to store data in a distributed way. It is so called master/slave architecture. A HDFS cluster primarily consists of two types of nodes: NameNode and DataNode. The NameNode as a master server manages the file system metadata and the DataNode stores the actual data. A file stored in HDFS is split into one or more blocks and stored across multiple DataNodes. By default, the block size is 64 M or 128 M.

In order to ensure the data integrity, the HDFS implements checksum checking on the contents of HDFS files. When a client requests to create a HDFS file, the file data being written to a DataNode is split into a number of packets. Each packet is 64 bytes and consists of multiple chunks with corresponding checksums. Each chunk is 512 bytes by default. The checksums are stored in a separate file with the meta extension in the same HDFS namespace. Figure 1 shows an example of the block files and

blk_1857886797832904623	blk_-575142751576769158
blk_1857886797832904623_1631.meta	blk_-575142751576769158_1630.meta
blk_-2001094327894351771	blk_-6899707740992334183
blk_-2001094327894351771_1859.meta	blk_-6899707740992334183_1637.meta
blk_2864080511916640498	blk_8026835587436878571
blk_2864080511916640498_1633.meta	blk_8026835587436878571_1638.meta

Fig. 1. Block files and checksum files on DataNode.

corresponding checksum files on a DataNode. The files named after "blk_" plus a block id are HDFS block files and the files with a meta extension are checksum files. When a deletion operation is executed, both two are deleted at the same time. The checksum algorithm used by HDFS is CRC32, so each checksum is stored with 4 bytes. It means every 4 bytes in the checksum file checks 512 bytes section in the block file. Part of the content of a checksum file is shown in Fig. 2. The first 7 bytes is file header that contains the file version, checksum type and the number of bytes for which a checksum is computed. From the offset at 0x07, it stores each checksum in sequence. The "checksum1" and "checksum2" in Fig. 2 denote the checksum of the first and second 512 bytes section in the corresponding block file.

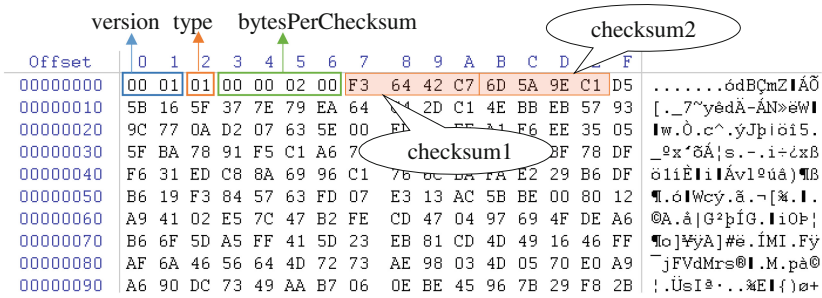


Fig. 2. The format of checksum file.

3.2 Format of HFile

HBase is an open source, non-relational, column-oriented, distributed database and runs on the top of HDFS. HBase table records are stored as "HFile" on HDFS and they are column-oriented. One HFile stores the data for one column family. In this section, we analyzed the structure of its data file in order to recover the deleted files and extract records.

The overview structure of HFile is shown as Fig. 3. HFile consists of four sections: Scanned block section, Non-scanned block section, Load-on-open section and Trailer. Every section contains one or more blocks. The Trailer block is a fixed file trailer, which contains some basic information of HFile and the offsets of other blocks. Data in Load-on-open section will be loaded into memory when HBase region server starts. It contains FileInfo, Bloom filter block, Root index block and Meta index block. Scanned block section means that it will be read when the HFile be sequential scanned. It contains Data block and Leaf index block. Root index block, Intermediate Data index block and Leaf index block are organized in a tree structure and they store the index point to the blocks in the next lever. The table records are stored on the Data block. In this paper, we aim at carving out the Data block from a raw image and extracting corresponding records. The functions of other blocks is not introduced because it is out of the main work.

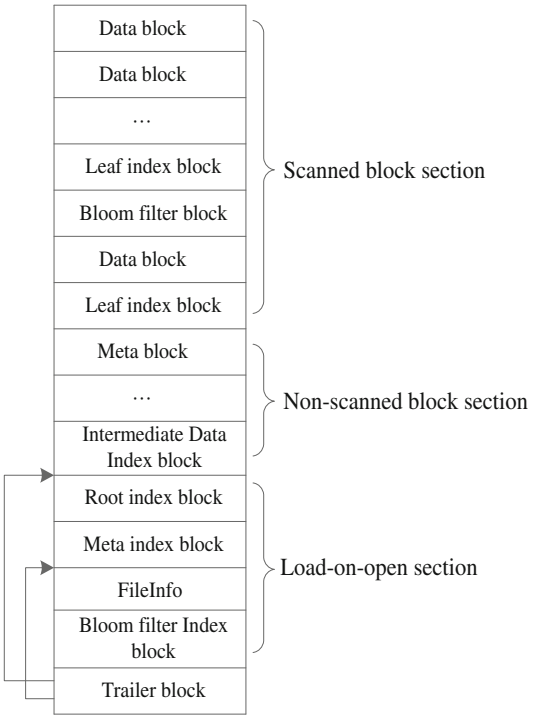


Fig. 3. The overview structure of HFile.

The physical format of the Data block is shown in Fig. 4. The first 8 bytes denotes the header signature whose value is "0x44415441424C4B2A". It can be used to identify the start position of Data block. The 4 bytes at offset 0x08 is the size of the data in the block. The following 8 bytes at offset 0x10 is the offset of the previous block on disk. Then the rest part stores records in key-value pair format.

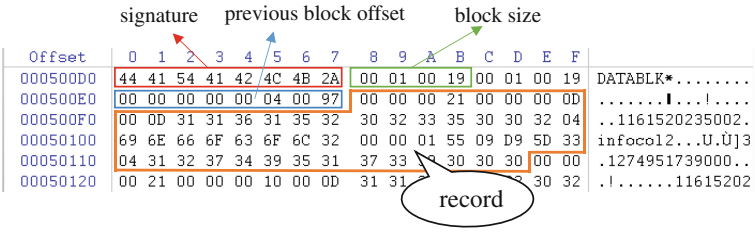


Fig. 4. The format of HFile Data block.

4 The Proposed Method

4.1 Collecting Disk Image

In order to recover data, a raw image of storage media is first required. The raw image means a bit-by-bit copy of the target media. The technique includes software based copy and hardware based copy. Some tools like Winhex, Autopsy can be used to get the disk image under Windows operate system. Under Linux operate system, DD command is a simple way which often used for generating a back-up file for the target media. When the file system is corrupted or the disk is damaged, the hardware based method will be considered.

In this paper, we use DD command to acquire a raw image of the certain disk partition. Firstly, the disk partition in which HDFS files are stored can be identified by the HDFS configuration file. If we know which partition contains the data we should recover in advance, we can just copy the certain partition instead of the whole disk. The following DD command "dd if = /dev/sda1 of = /data.img" is an example which makes a raw image of the partition "sda1" into a DD format file "data.img" in the root directory.

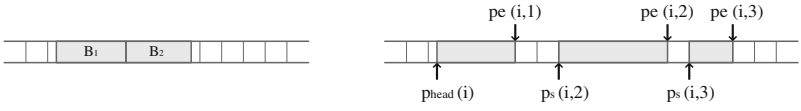
4.2 Recovering Data Blocks of HFile

HDFS is an abstracted file system layer and the actual storage of the file block is based on the host operating system's file system (e.g. NTFS/ExtX). This is so called the local file system. An important property of the most common file systems is that a file is allocated in equally sized unit, which is called "cluster" in NTFS and is called "block" in ExtX. In order to distinguish it from HDFS block, we only call the unit as "cluster".

Fragmentation in file system can only occur on cluster boundaries. It means data in one cluster only belongs to one file. Based on this fragmentation rule, the recovery problem of HFile Data blocks can be split into two subproblems, i.e.: 1. recovering of single-fragmented Data block; 2. recovering of multi-fragmented Data block.

Assuming that there are in total N blocks in a HFile, let B_i denotes the i'th block in it and the first fragment of B_i is denoted as $b_f(i)$. As described in Fig. 5(a), the Data block is stored continuously, that is there is only one fragment. To check whether it is a single fragment or not, we first find the start position $p_{head}(i)$ of $b_f(i)$ by the magic bytes, followed by acquiring the block size d_l and finding the start position $p_{head}(i + 1)$ of $b_f(i + 1)$.

If $p_{head}(i + 1)$-$p_{head}(i)$-24 equals to d_l, then B_i is single-fragmented. It means we can extract the bytes between $p_{head}(i)$ and $p_{head}(i + 1)$ directly. If the Data block is split into multiple fragments, as shown in Fig. 5(b), the recovery problem can be divided into three subproblems, i.e.:



(a) Single-framented Data block. **(b)** Multi-framented Data block.

Fig. 5. Simplified example of the Data block fragmentation.

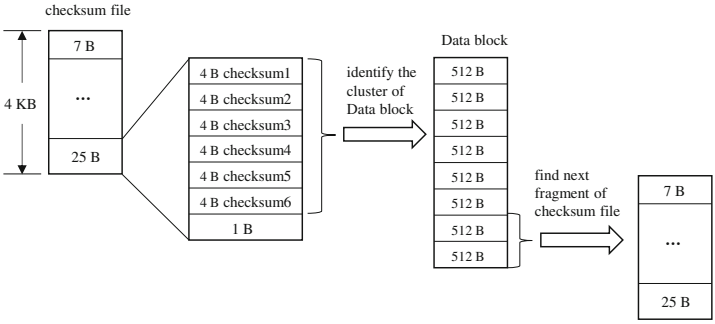


Fig. 6. Example of cross-find method under 4096-byte sized cluster.

1. Finding the start position $p_{head}(i)$ of the first fragment of B_i;
2. Determining the ending position $p_e(i, j)$ of the each fragment j of B_i;
3. Finding the start position $p_s(i, j)$ of each fragment($j > 1$);

In order to find each fragment of both block file and checksum file, we propose a "cross-find" method. It means we use the checksum file to identify the corresponding cluster of the Data block and use the identified cluster to find the next fragment of the checksum file at the same time. The method is based on the fragmentation rule. A S-byte sized cluster stores (S−7)/(4*S/512) checksums of the block file, so the last b_l bytes can be used to find the next cluster of the Data block. Once found, we can use the rest non-matching section in the identified cluster to find the next fragment of the checksum file. The b_l can be calculated through Eq. (1), where S is the cluster size and b_l is the last non-matched bytes.

$$b_l = (S - 7) mod (4 * S/512). \tag{1}$$

To detail this, an example using the proposed method under 4096-byte sized disk cluster is described in Fig. 6. The first 7 bytes is the header bytes and (4096-7) mod (4*4096/512) is 25, so the last 25 bytes can be used to determine the position of the Data block cluster. When the block file cluster is determined, we can use the last two 512-byte section to find the next fragment of the checksum file. According to this method, the Data block and the checksum file can be reconstructed easily.

The algorithm of recovering HFile Data block in pseudo-code is presented in Algorithms 1 and 2. In which DATABLKsig means the Data block header signature (0x44415441424C4B2A) and BLKsig means other block header signature. Algorithm 1 describes the single-fragmented recovery method. It is described at first in this section. The process of mutil-fragmented recovery is: from the cluster that DATABLKsig belongs to, match the checksum of each 512-byte section in the cluster with the bytes in data dump D. Once a mismatch appears, match the b_l bytes of the checksum hexadecimal string. If the match is failed, it means the Data blocks is fragmented, else the checksum file is fragmented. Then we use the above method (cross-find) to identify the next fragment of each other. Do these repeatedly until the Data block being totally reconstructed or can't find any fragments.

Algorithm 1. single-fragmented recovery

```
input  : data dump D, cluster size S
output : the recovered file R,offset list(p)
for all data in D do
    p ← find next DATABLKsig's offset;
    l ← read block size;
    locate at p+l in D;
    m ← read 8 bytes;
    if m == DATABLKsig or m == BLKsig then
        add the data between p and p+l into file R;
    else
        lsit(p) ← add p to a list;
    end if
end for
return list(p);
```

Algorithm 2. multi-fragmented recovery

```
input  : data dump D, cluster size S, offset list(p)
output : the recovered file R
for each item p in list(p) do
    C ← calculate the cluster that p belongs to;
    calculate checksums of each 512-byte section in C;
    cksum ← combines checksums into a hexadecimal string;
    for all data in D do
        match cksum with data in D;
        if not match then
            break;
        else
            ckpos ← record the matched offset;
    end for
    for all data in D do
        match the cluster from C continuously with the data from ckpos;
        add the matched data into file R;
        if not match then
        match the first b_l (see Eq. (1)) bytes;
            if not match then
                find the next fragment of Data block;
                if not found then return;
            else
                find the next fragment of checksum file;
                if not found then return;
            end if
        end if
    end for
end for
```

4.3 Extracting Records from the Carved Data Block

Unlike traditional relational database (RDBMS), each record in HBase table is a key-value pair. The record header contains key length and value length and the rest portion stores the content of key and value. The value format is just byte array which has no specific type. Key format is more complex, which comprises rowKey, column family, column, timestamp and keyType. The record structure is illustrated in Fig. 7.

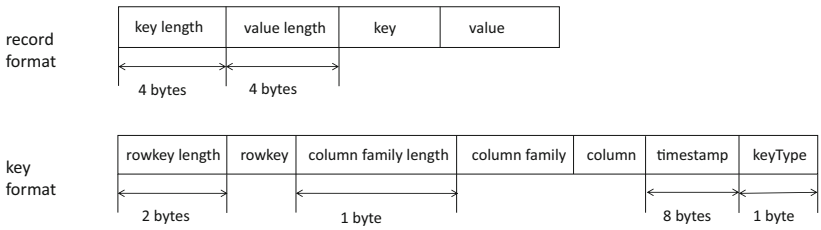


Fig. 7. The format of HBase table record.

Each column belongs to a column family, and a column name is prefixed by a column family (e.g. info: title, info: name). A record may have many versions, using timestamp to distinguish. The timestamp is a 64-bit Integer. RowKey, column and timestamp identify a record uniquely. The keyType represents the type of a record (like "put", "delete", "deleteColumn", "deleteFamily"). For example, if the keyType is 0x00, it means this is a deleted record and if it is 0x04, the type is put. According to the record structure we can extract it easily. Firstly read Data block header and then locate to the first record start position, and then read key, value content based on the key length and value length. Note that, the column size can by calculated by key length minus other fields length.

5 Experiment and Evaluation

To evaluate the performance of our method, we conducted a series of experiments, which mainly consist of two scenarios. Scenario one verifies the effectiveness of the proposed method under 4096-byte clustered disk because it is the default cluster size in Ext4 file system which is commonly used in Linux. Scenario two compares the performance of the method under different cluster sized disk.

5.1 Scenario One

In this scenario, we set up four virtual machines. One is NameNode and the other three are DataNodes. Each machine is under Linux operate system (CentOS 6.7). The Hadoop version we used is hadoop1.1.2 and the HBase version is 0.94.27. We download data in CSV format from the internet and then create corresponding table in HBase using command "create 'testtable' , 'info'", where "testtable" denotes table

name and "info" denotes column family. After this, the data is loaded into HBase database using the following command:

```
[root@NameNode /]# hadoop jar /usr/local/hbase/hbase-0.94.27.jar importtsv -importtsv.columns=
HBASE_ROW_KEY,info:col1,info:col2 -Dimporttsv.separator=, testtable /data.csv
```

Then the corresponding HFile is generated in disk. The structure of this file follows the file structure described in Sect. 4. Finally we delete this file using the HDFS shell command and using DD command to get the disk image.

Table 1. Experiment results of recovering HBase records on three DataNodes.

	Total records	Recovered records	Recovery rate (%)
DataNode1	141968	141968	100
DataNode2	141977	141857	99.91
DataNode3	55590	55500	99.83
Total	339535	339325	99.93

The experiment result of recovering HBase table records from HDFS is shown in Table 1. The HFile we deleted is split into three blocks stored across three DataNodes respectively. Each of the DataNodes contains 141968, 141977 and 55590 records respectively. As is shown in Table 1, we recovered most of the HBase table records.

5.2 Scenario Two

In order to evaluate our method under different cluster sized disk, we set up three Hadoop clusters. The cluster size of the local file system of each Hadoop cluster is 1024 bytes, 2048 bytes and 4096 bytes respectively. The experiments result is shown in Table 2. Under 2048-byte and 4096-byte cluster sized disk, the recovery rate is nearly 100%. It means the most records can be recovered effectively. Note that, the effectiveness under 1024-byte cluster sized disk is a little poorer than the other two situations. Because the information in one cluster of the checksum file can verify total 127 clusters of the block file, and the rest one byte is too small to determine the next cluster of the block file. Once the checksum file is fragmented, we should find the header signature of next Data block again.

Table 2. Experiment results under different cluster size

	Total records	1024-byte		2048-byte		4096-byte	
		Recovered records	Recovery rate (%)	Recovered records	Recovery rate (%)	Recovered records	Recovery rate (%)
DataNode1	141968	141386	99.59	141968	100	141968	100
DataNode2	141977	140415	98.89	141857	99.91	141857	99.91
DataNode3	55590	46481	83.61	55500	99.83	55500	99.83
Total	339535	328282	96.68	339325	99.93	339325	99.93

5.3 Comparing Research

In order to verify the effectiveness of the proposed approach, it is compared with the traditional head-length carving method [2] which just search the header signature and extract data continuously until the size equals to the length recorded in the header.

Two criterions that precision rate (define as Eq. (2)) and recall rate (defined as Eq. (3)) have been adopted to evaluate the proposed method, and F-value (defined as Eq. (4)) is used to evaluate the quality of the recovery method. In these equations, A is used to denote the number of recovered records belong to HFile; B is used to denote the number of recovered records do not belong to HFile while C denotes the number of records which belong to HFile but do not be recovered from the data dump D.

$$Precision(P) = \frac{A}{A+B}. \tag{2}$$

$$Recall(R) = \frac{A}{A+C}. \tag{3}$$

$$F-value = \frac{2 * P * R}{P+R}. \tag{4}$$

The comparing experiment is based on 1024-byte cluster sized disk. The comparing result is shown as Fig. 8. There is no false positive in our test because we extract the records based on their structure described in Sect. 4. It can be seen that the proposed method gets higher scores of precision, recall, and F-value than the traditional approaches. The traditional head-length carving method is suited for recovering continuous files. The recovery effectiveness depends on the data set. The less fragments the files contain, the better the recovery result is. Our method which is based on checksum file not only applies to continuous files but works very effective when the files are split into many fragments.

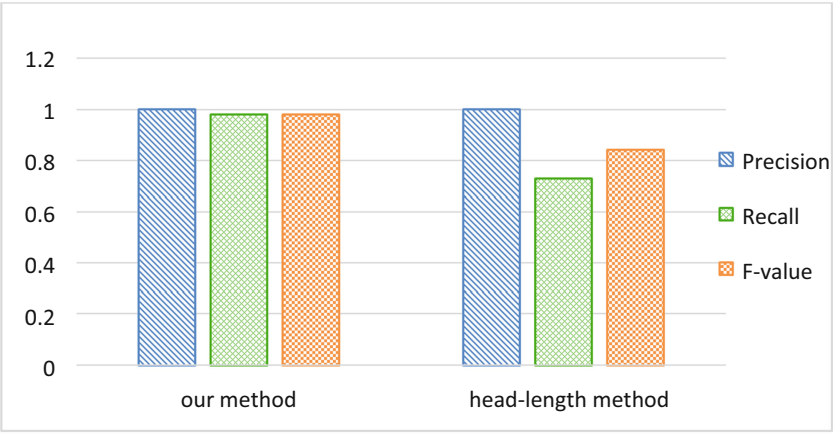


Fig. 8. The comparing result of the two methods.

6 Discussion and Conclusion

In this work, we talk about the data recovery problem in HDFS and provide an effective method of recovery HBase records. The HDFS is a distributed file system and recovery of a complete file is more difficult. However, partly recovery of data is valuable as well. We analyze the HDFS structure and HFile format in detail, and recover the Data blocks of HFile using the corresponding checksum file and extract records from them.

Our carving technique utilizes the information in checksum files which can piece the file fragments together in a correct order conveniently. So it can effectively address the problem of recovering highly fragmented files. We test the method on the disk whose cluster size is 1 KB, 2 KB and 4 KB respectively. It behaves well in 4 KB and 2 KB cluster sized disk, where the effectiveness is a little poorer on 1 KB cluster sized disk. But all the recovery rates are more than 83.61%.

Another important aspect is the computational efficiency of the proposed recovery method. To determine this, the average computation time for recovering the data from different cluster sized disk image has been measured. Our methods are implemented by JAVA and performed on a PC with Intel 3.2 GHz i5-3407 CPU and 8 GB RAM and the size of the each disk image in our experiments is 20 GB. The average computation times under 1 KB, 2 KB and 4 KB cluster sized disk image is, respectively 75, 35 and 27 min. Note that, the computational time depends on the size of dataset. We will compare the computational time of different sized disk image in our future work. In addition, data in HDFS may be stored across several machines. How to choose the most correlated machine for recovery is of great importance. This will also be considered in our future work.

Acknowledgments. This work is supported by the Natural Science Foundation of China under Grant Nos. 61070212 and 61572165, the State Key Program of Zhejiang Province Natural Science Foundation of China under Grant No. LZ15F020003 and Key Lab of Information Network Security, Ministry of Public Security.

References

1. Richard III, G.G., Roussev, V.: Scalpel: a frugal, high performance file carver. In: DFRWS (2005)
2. Garfinkel, S.L.: Carving contiguous and fragmented files with fast object validation. Digital Invest. **4**, 2–12 (2007)
3. Memon, N., Pal, A.: Automated reassembly of file fragmented images using greedy algorithms. IEEE Trans. Image Process. **15**(2), 385–393 (2006)
4. EnCase Forensic. http://guidancesoftware.com/encase-forensic.htm
5. Adroit Photo Forensics. http://digital-assembly.com/products/adroit-photo-forensics/
6. Pal, A., Sencar, H.T., Memon, N.: Detecting file fragmentation point using sequential hypothesis testing. Digital Investigation. **5**, 2–13 (2008)
7. Autopsy/The Sleuth Kit. http://sleuthkit.org
8. X-Ways Forensics. http://x-ways.net/forensics

9. Cohen, M.: Advanced jpeg carving. In: Proceedings of the 1st International Conference on Forensic Applications and Techniques in Telecommunications. Information, and Multimedia and Workshop, pp. 16:1–16:6 (2008)
10. Na, G., Shim, K., Moon, K., Kong, S., Kim, E.: Lee, J: Frame-based recovery of corrupted video files using codec specifications. IEEE Trans. Image Process. **23**(2), 517–526 (2014)
11. Bock, J., Smet, P.: JPGarve: An advanced tool for automated recovery of fragmented JPEG files. IEEE Trans. Inf. Forensics Secur. **11**(1), 19–24 (2016)
12. Uzun, E., Sencar, H.T.: Carving orphaned JPEG file fragments. IEEE Trans. Inf. Forensics Secur. **10**(8), 1549–1563 (2015)
13. Cohen, M.: Advanced carving techniques. Digital Invest. **4**, 119–128 (2007)
14. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In IEEE Symposium on Mass Storage Systems (2010)
15. Martini, B., Choo, K.R.: Distributed filesystem forensics: XtreemFS as a case study. Digital Invest. **11**, 295–313 (2014)
16. Yoon, J., Jeong, D., Kang, C., Lee, S.: Forensic investigation framework for the document store NoSQL DBMS: MongoDB as a case study. Digital Invest. **17**, 53–65 (2016)
17. Pal, A., Memon, N.: The Evolution of File Carving. IEEE Signal Process. Mag. **3**, 59–71 (2009)
18. Karresand, M., Shahmehri, N.: Fileprints: identifying file type by n-gram analysis. In: Proceedings of 7th IEEE Systems, Man and Cybernetics Information Assurance Workshop, pp. 64–71. IEEE (2006)
19. Veenman, C.J.: Statistical disk cluster classification for file carving. In: Proceedings of 3rd International Symposium on Information Assurance and Security, pp. 393–398. IEEE Computer Society (2007)
20. Jeon, S., Bang, J., Byun, K., Lee, S.: A recovery method of deleted record for SQLite database. Pers. Ubiquit. Comput. **16**(6), 707–715 (2012)
21. Xu, M., Yang, X., Wu, B., Yao, J., Zhang, H.P., Xu, J., Zheng, N.: A metadata-based method for recovering files and file traces from YAFFS2. Digital Invest. **10**, 62–72 (2013)
22. Sencar, H.T., Memon, N.: Identification and recovery of JPEG files with missing fragments. Digital Invest. **6**, 88–98 (2009)
23. Yoo, B., Park, J., Lim, S., Bang, J., Lee, S.: A study on multimedia file carving method. Multimedia Tools Appl. **61**(1), 243–261 (2012)