Journal of Parallel and Distributed Computing 156 (2021) 119–130

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

# Hadoop Perfect File: A fast and memory-efficient metadata access archive file to face small files problem in HDFS

Yanlong Zhai [a,*], Jude Tchaye-Kondi [b,*], Kwei-Jay Lin [c], Liehuang Zhu [a], Wenjun Tao [b], Xiaojiang Du [d], Mohsen Guizani [e]

[a] School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China
[b] School of Computer Science, Beijing Institute of Technology, Beijing 100081, China
[c] Department of Electrical Engineering and Computer Science, University of California, Irvine 92697, CA, USA
[d] Department of Computer and Information Sciences, Temple University, Philadelphia, USA
[e] Department of Computer Science and Engineering, Qatar University, Qatar

ARTICLE INFO

*Article history:*
Received 7 September 2020
Received in revised form 7 April 2021
Accepted 26 May 2021
Available online 6 June 2021

*Keywords:*
Distributed file system
Massive small files
Fast access
HDFS

ABSTRACT

HDFS faces several issues when it comes to handling a large number of small files. These issues are well addressed by archive systems, which combine small files into larger ones. They use index files to hold relevant information for retrieving a small file content from the big archive file. However, existing archive-based solutions require significant overheads when retrieving a file content since additional processing and I/Os are needed to acquire the retrieval information before accessing the actual file content, therefore, deteriorating the access efficiency. This paper presents a new archive file named Hadoop Perfect File (HPF). HPF minimizes access overheads by directly accessing metadata from the part of the index file containing the information. It consequently reduces the additional processing and I/Os needed and improves the access efficiency from archive files. Our index system uses two hash functions. Metadata records are distributed across index files using a dynamic hash function. We further build an order-preserving perfect hash function that memorizes the position of a small file's metadata record within the index file.



## 1. Introduction

The main purpose of Hadoop [32] is for efficient and swift storage and processing of big data. Hadoop's File System (HDFS) [32] uses a master-slave architecture (see Fig. 1) based on GFS [17] to store and access data. The entire HDFS is managed by a single server called the NameNode (NN) as the master and files content stored on DataNodes (DNs) as slaves. By default, each HDFS data block size is 128 MB but configurable according to the desired I/O performance. To store a big file, Hadoop splits it into many data blocks and stores them in different DNs. With Hadoop's built-in replication system, each data block is replicated on several DNs (3 by default) to avoid data loss in case of a DN failure. HDFS is very efficient when storing and processing large data files. But for a large number of small files, HDFS faces the **small file problem**. Social networks, e-commerce, digital libraries, healthcare, meteorology, and satellite imagery are only a few examples of applications that produce large amount of data but in the form of small files. Typically, applications deployed on a server generate many log files. Depending on its configuration, an application can generate a log file per hour or daily. Regardless of the size of a website, log analysis can give direct answers to problems encountered on websites. Log analysis is useful for performing SEO audits, debugging optimization issues, monitor the health of a website and its natural referencing. Such data files are often small in size, ranging from some KB to several MB, but very important for data analysis.

There is no effective DFS (Distributed File System) that works well for massive small files. Massive small files generate a lot of metadata in HDFS, and since the NN holds all of its metadata in memory, this may cause it to run out of memory and hurt its performance. Other issues caused by massive small files, in addition to NN's memory overload, include:

1. **Long storage time:** In our experiment, uploading 400,000 files with sizes ranging from 1 KB to 10 MB to HDFS took up to 11 hours.
2. **Bad processing performance:** Processing a large number of small files requires MapReduce [11] (HDFS processing frame-

* Corresponding authors.
*E-mail addresses:* ylzhai@bit.edu.cn (Y. Zhai), tchaye59@gmail.con (J. Tchaye-Kondi).

https://doi.org/10.1016/j.jpdc.2021.05.011
0743-7315/

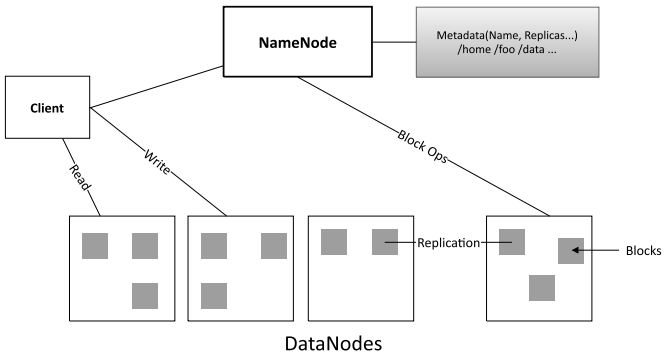



**Fig. 1.** NameNode stores metadata in memory while DataNode stores the actual content on hard disks. Before performing any file reading or writing operations, the client requests metadata from NameNode and obtains DataNodes locations.

work) to perform several reads and writes across different nodes in a cluster, which takes much longer than reading or writing a single large file.
3. **NameNode performance:** When several clients attempt to access files at the same time, it hurts the NN resources.

To overcome the small file problem, Hadoop provides archiving systems such as HAR file [18], SequenceFile [36], and MapFile (see Fig. 2). These solutions work by combining multiple small files into large files, in the same way, reducing the amount of metadata required by the NN to represent them. The problem with such archive files is that as a side effect, the access performance of small files located inside the large ones can become very slow considering that their metadata previously maintained in NN memory to ensure fast access is now dump in some index files stored as regular HDFS files. We identified three access mechanisms after analyzing file access in HAR, SequenceFile, and MapFile:

- The simplest case is when the archive uses a single index file. First, all files' metadata from the index file are loaded back in memory (usually at the client-side). We then search for the metadata of the file we are looking for (position in the big file, file size, etc.). After that, we will have all the required information to recover the file's content from the merged file (e.g. MapFile).
- When the archive uses several index files, it may be necessary to load them back into memory before retrieving metadata and recovering a file's content. This is the case of the HAR file that uses two index files.
- The worst-case situation is when the archive file does not use any index file. This archive type generally requires reading the large file from the beginning to the end until finding the searched file (e.g. SequenceFile).

As observable, provided solutions have extra I/Os and processing during access which deteriorates their access efficiency. Caching and prefetching strategies, which consist of keeping all or part of the metadata in client memory, are used to mitigate this situation; however, unlike NN and DN servers, client memory size can be limited and less secure.

In this paper, we present Hadoop Perfect File (HPF), a new design of index-based archive file. HPF, like other archives, merges several small files into large ones; however, unlike other solutions, HPF has less overhead during access and is designed for clusters with high frequent access. Our proposed solution can directly access file metadata within the index file, which mean rather than loading the whole index files in memory and search for metadata as done so far, HPF only loads useful information to recover the file's content by only looking at the part of the index file containing it. As a result, HPF uses fewer I/Os and processing overheads during access. With the help of a perfect hash function in its index system, HPF can compute for any file present in its merged files the exact offset and limit of where to read the metadata within its index file. When the offset and limit are known, a simple seek operation in the index file is used to move the reader to the offset position before reading the metadata information. In HDFS, seeking to random positions of a file that have its blocks across different DNs can have some additional network cost. To avoid this cost, we make sure that each of our index files is stored on a single DN and occupies at maximum a single HDFS block. This is why we use the extendible hashing and split the small files' metadata into several limited-size index files. Finally, unlike HAR files, HPF allows the client to add more small files after the archive file has been created, and unlike MapFile, HPF does not require the client to order the files before creating the archive file or adding new files.

The rest of the paper is structured as follows. Section 2 reviews related work. Section 3 presents the design of HPF. In Section 4 we investigate some issues of our implementation. Section 5 evaluates the performance of our HPF implementation against the native HDFS, HAR file, MapFile, LHF, and analyzes experimental results. Finally, the paper is concluded in Section 6.

## 2. Related work

In this section, we discuss existing threats against small files' problems in HDFS. We present a brief summary of research efforts, their methodology, and weaknesses.

Big Data is not only about the Hadoop ecosystem. Massive Parallel Processing (MPP) [5] was also widely used before Hadoop and is becoming less popular. Hadoop is young but widely preferred over MPP by companies due to its maturity, commodity, and free nature. The small file problem in HDFS is because the NN keeps metadata in the memory. Each folder, file, and block generate metadata. In general, a file's metadata takes up about 250 bytes of memory. For each block with 3 default replicas, its metadata can consume about 368 bytes. Suppose 24 million files are in HDFS. The NN will require 16 GB of memory for storing metadata [31]. Early works on handling small files in HDFS can be classified into three main classes.

### 2.1. Combining small files into large files

The first class of solutions consists of combining several small files into one large file. By doing so, only the metadata of the large file is needed in NN's memory. Combining small files efficiently reduces the NN's memory overheads and is the main concept behind HDFS's default archive files: HAR file, SequenceFile, MapFile. HAR [18] file is an archive file that holds metadata through two index files: _index and _masterindex. As shown by Fig. 2 the small files' content is stored in large part-* files. The weakness of HAR files is that they are immutable. Once created, it is not possible to modify their content and add more files. This functionality is necessary when files are generated continuously. Furthermore, HAR offers relatively poor access performance.

Hadoop first proposed the SequenceFile to solve the binary log problem [36]. In this format, the data is recorded as a sequence of key-value pairs (see Fig. 2). The well-known limitation of SequenceFile is that when searching for a file, it needs to traverse the pairs one by one. That has the worst-case complexity of O(n). To overcome SequenceFile limitation, MapFile was proposed. MapFile is a sorted SequenceFile with an index to permit lookups by key (see Fig. 2). The pairs in MapFile are sorted by the key (name) and MapFile used binary search during lookup to bring down the complexity to $O(logn)$. The fact that MapFile sorts keys to allows

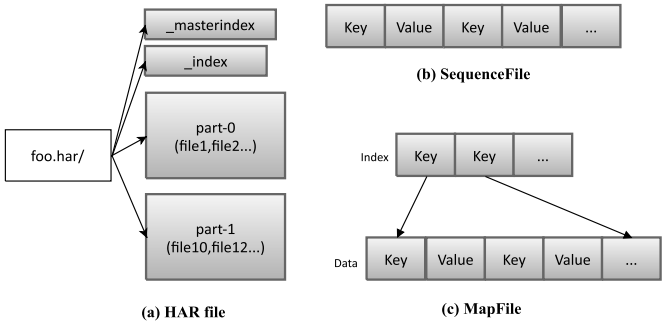



**Fig. 2.** The HAR file is actually a folder in which files' content is located into large part-* files, and the metadata in the _masterindex and _index files. The Sequence file stores data as a sequence of key-value pairs which doesn't support arbitrary file access. The MapFile is a folder with two sequence files storing respectively files' content and their metadata.

fast lookup is also its weakness, because, once the archive is created, it is not possible to add files with arbitrary names. Hadoop's default solutions are limited. Some offer poor access efficiency, while others make adding new files difficult or impossible. Many researches addressed the small files issue in HDFS by proposing new designs and solutions. Tong Zheng et al. introduced a method for storing file metadata in the HBase database [39]. They further used prefetching based on access log analysis and cached metadata of regularly accessed files in the client's memory. Unlike HAR, NHAR [35] (New HAR) merges small files into large ones and distributes their metadata in a fixed number of index files using hashing. NHAR and the HAR still suffer from the slowness during the archive creations since they require a prior upload of the small files to HDFS. Kyoungsoo Bok et al. proposed a distributed caching scheme to efficiently access small files in HDFS [8]. Bo Dong et al. developed a novel method for improving the efficiency of storing and accessing small files on HDFS in the BlueSky system [12] (one of the most widely used eLearning resources sharing systems in China). In their solution, all correlated PPT files are merged into a larger file. They also introduced a two-level prefetching mechanism to improve access efficiency. OMSS (Optimized MapFile based Storage of Small files) [30] was proposed to merge files based on the worst fit strategy. The strategy helps in reducing internal fragmentation in data blocks hence leads to fewer data block consumption. TLB-MapFile [25] is designed base on MapFile to provide more access efficiency. TLB-MapFile adds a fast table structure (TLB) in DataNode to improve retrieval efficiency by mapping information between data blocks and small files. Since OMSS and TLB-MapFile are MapFile-based, they require sorted keys, therefore, not optimize for random file add and access. Some suggested solutions include altering HDFS by adding hardware to speed up small file processing or letting HDFS automatically combine small files before storage. Peng et al. proposed the Small Hadoop Distributed File System (SHDFS) [28], which is based on the original HDFS but includes a merging and caching module. The merging module employs a correlated files model to identify and merge correlated files using user-based collaborative filtering. A Log-linear model is used in the caching module to find out frequently accessed hot-spot data. They then create a special memory subsystem to cache these hot-spot data and improve access. To solve the small file problem, Hou et al. suggest using additional hardware called SFS [20] (Small File Server) between clients and HDFS. Their solution includes a file merging algorithm based on temporal continuity, an index structure to retrieve small files, and a prefetching mechanism to improve file reading and writing. Some of these archiving systems, such as LHF [34], DQSF [21], [19], [9], before merging the files classify them or rely on some distributions criteria that help to optimize the storage or access efficiency. In case the proposed solution is built on top HDFS, it is easy to migrate to the latest version of HDFS. It is not desirable when a solution modifies HDFS, because it makes maintenance and upgrades challenging and costly for companies.

### 2.2. Special DFS for small files

The second class consists of building DFS specialized only in the processing of small files. As example we can cite Taobao's TFS [1], Facebook's Haystack [6], Twitter's Cassandra [24]. Facebook must process over a million images per second. To ensure a good user experience, Facebook sets up its Haystack architecture. In this architecture, users' pictures are combined in big files, and their metadata for retrieval is stored in index files. The Haystack maintains all the images' metadata main memory in order to minimize the number of disk operations to the only one necessary for reading the file content. Taobao, one of China's largest online marketplaces, also has to deal with small file issues. Taobao generates about 28.6 billion photos with average size of 17.45 KB [15]. To provide high availability, reliability, and performance, Taobao creates TFS (Taobao File System) [1], a distributed file system designed for small files less than 1 MB in size. TFS is based on IFLATLFS [15] a Flat Lightweight File System and is similar to GFS. Unlike other file systems IFLATLFS aims to reduce the metadata size needed to manage files to a very small size to maintain them all in memory.

### 2.3. Better processing framework

The third class of solutions concentrates only on building a better accessing and processing framework for small files. Priyanka et al. have designed a CombineFileInputFormat to improve small file processing using MapReduce framework [29]. A map task takes as input a split that is a block of data. For small files, as the file size is smaller than the block size, the map task receives a small amount of input data. To solve this issue, the CombineFileInputFormat combines several small files into big splits before providing them as input to the map task. This approach has then been improved by Chang Choi et al. in [10]. They integrate the CombineFileInputFormat and the reuse feature of the Java Virtual Machine (JVM). This integration allows reusing a JVM to run multiple mappers. In [27][22] researchers have attempted to modify the OS file system to improve access efficiency. [27] designed the stuffed inode for small files that embeds the content of small files in the inodes' metadata in a variant of HDFS with distributed metadata called HopsFS [26]. [22] modified both the in-memory and on-disk inode structure of the existing filesystem and were able to dramatically reduce the amount of write and access I/Os.

### 2.4. Comparison

Together these earlier studies provide valuable insights to the small file problem. However, none of them seems to satisfy all the requirements to be used as the only solution for big data storage systems including both small files and big files. The perfect solution should comply with the following characteristics: a short build time, generates less metadata for the NN, and provides fast file access. Combining small files into large files effectively reduces the NN memory consumption, but at the same time, it badly deteriorates the file access performance since some extra IO operations are required to retrieve metadata. Specialized DFS for small files does not work very well or does not support large files. Only improving the small file processing framework or underlining file system brings no advance to the NN memory overload. As summarized in Table 1, the detailed comparison of HPF with some existing solutions shows that HPF gives the best access performance. Our work focused on designing a storage solution that reduces the

**Table 1**
Comparison of solutions to small files problem.

| Paper Name/Feature | Type | NameNode Memory usage | Support Append | Use extra System | HDFS pre-upload required | Creation Overhead | Reading Efficiency |
|---|---|---|---|---|---|---|---|
| HDFS | DFS | Very High | Yes | – | Yes | Very High | High |
| HAR | Archive&Index Based | Low | No | No | Yes | Very High | Low |
| MapFile | Archive&Index Based | Very Low | For special keys | No | No | Moderate | High(O(logn)) |
| SequenceFile | Archive Based | Very Low | Yes | No | No | Low | Low(O(n)) |
| BlueSky [12] | Archive&Index Based | Low | Yes | No | No | High | High |
| T. Zheng et al. [39] | Archive&HBase Based | Low | Yes | Yes | No | High | High |
| NHAR [35] | Archive&Index Based | Low | Yes | No | Yes | High | High |
| OMSS [30], TLB-MapFile [25] | MapFile Based | Very Low | For special keys | No | No | Moderate | High |
| SHDFS [28] | Archive&Index Based | Low | Yes | Yes | No | High | High |
| SFS [20] | Archive&Index Based | Low | Yes | Yes | No | High | High |
| LHF [34] | Archive&Index Based | Low | Yes | No | No | Moderate | High |
| DQSF [21], He [19], Cai [9], Kyoungsoo [8] | Archive&Index Based | Low | Yes | No | No | High | High |
| HPF | Archive&Index Based | Low | Yes | No | No | Moderate | Very High(O(1)) |

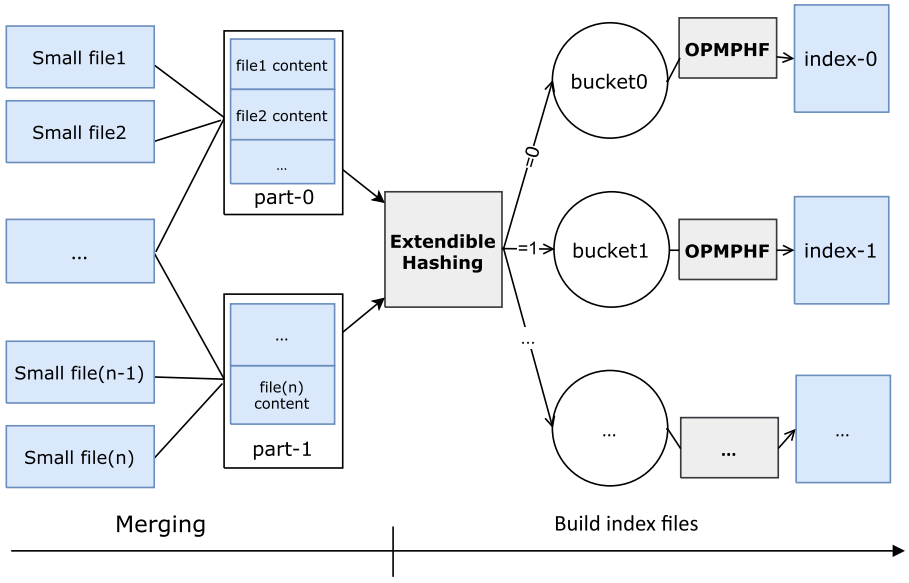



**Fig. 3.** During merging, the client's device concurrently appends small files to large part-* files located on HDFS. To build the index files, extendible hashing splits metadata into buckets, and the MMPH memorizes the records writing order within the index file.

NN memory load and still allows good access performance. HPF is built on top of HDFS and doesn't require any modification of this one. In the meantime, HPF supports file append functionality with little cost and allows HDFS to be very efficient with small files and large files.

## 3. HPF design

HPF is a new index-based archive file proposed as a solution to Hadoop's small file problem. Our goal is to make metadata queries faster with fewer I/Os, memory, and processing overheads. Besides, HPF supports HDFS's new file appending functionality as well as file-level compression. The most important feature of HPF is its direct metadata access ability within index files. This feature prevents HPF from loading the whole index files in memory for just retrieving a single file's metadata information, which is the main cause of slowness during accesses. This feature also enables constant-time metadata lookup, and improves the processing and accessing efficiency of small files. HPF has been redesigned from scratch and is different from LHF that we suggested in [34]. Unlike HPF, LHF does not support compression, direct access to metadata, and parallel files merging during creation.

The HPF file creation, as displayed in Fig. 3, consists of four steps:

1. **The merging step:** Concurrently, the client merges small files to make larger part-* files and collects the needed information to retrieve each file's content, which we refer to as metadata record. It is implemented using multiple threads, each of which operates on a single part-* file.
2. **Buckets creation:** When the merging process is completed, the metadata records are arranged into buckets using extendible hashing.
3. **Order memorization:** An order-preserving minimal perfect hash function (OPMPHF) memorizes for each bucket the order in which metadata records will be written in the corresponding index file.
4. **Writing to index file:** Finally, the metadata records and the OPMPHF are written into the corresponding index file with the OPMPHF first, followed by metadata records.

The final result of the HPF file creation is a folder (see Fig. 4) containing index-* files (index-0, index-1, etc.), part-* files (part-0, part-1, etc.), and one _names file. The index-* files contain all necessary information to retrieve a file's content except its full name. For the sake of making each metadata record fixed size within the index file, we move file names that cannot have fixed size to the _names files. The _names file is there only for listing purposes and

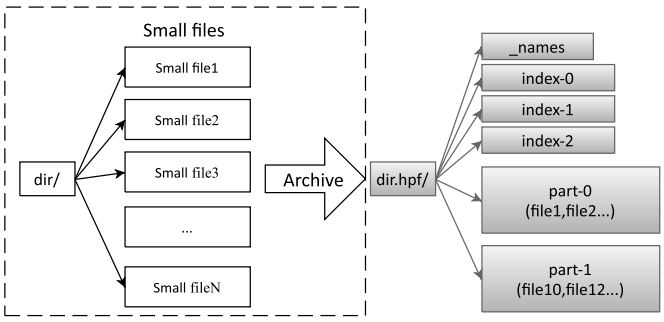



**Fig. 4.** Transformation to HPF folder.

only stores files' names, while the part-* files hold the actual file content.

### 3.1. File merging process

Fig. 3's first block illustrates HPF's merging step At this stage, small files are combined into large **part-*** files that have the effect of reducing the metadata size needed by the NN to represent them. Merging is performed in parallel by several threads, with each thread appending data to just one part file. HPF uses two threads by default, and starts by creating two empty files (**part-0** and **part-1**). The first thread appends files to the **part-0**, while the second appends them to the **part-1**. To append a file, we first load its content into memory, compress it if compression is enabled, then transfer the content from the client device to HDFS and append it to a *part* file. After appending, we retrieve all the needed information to recover the appended content referred to as metadata record. Our experimental HPF prototype uses the LZ4 [16] compression algorithm, which is incredibly fast and able to achieve a compression rate of more than 500 MB/s per core and a decompression rate of multiple GB/s per core.

Before building the index files, our system needs the merging process to fully complete in order to be in possession of all metadata records. Furthermore, just keeping these records in memory while waiting for the merging to complete is not secure because if the merging fails due to possible network errors, we will have to restart everything. To avoid that, metadata records are temporarily stored in a temporary index file (**_temporaryIndex**). When a file is appended to a part file, its metadata is also appended to the temporary index file as well. Therefore, the temporary index file holds metadata records during the entire merging process and is also used to ensure recovery in case of failures. When all small files are merged, the merging stops, and the index file creation step discussed in the section below, starts.

### 3.2. Index files building process

The metadata of each file has a fixed size and contains the following information:

- **File name hash:** A unique integer derived by submitting the file name to the hash function. This hash value uniquely identifies a small file within the archive file.
- **Part file position:** This field identifies the part file storing the small file. For example, a value of 0 indicates that the small file is saved in part-0.
- **Offset:** The exact offset to read the small file's content from the part file.
- **Size:** The file's size.

Each of these fields has a fixed size. The file name is hashed to a 128-byte integer, which is large enough to avoid collisions. A single metadata record occupies exactly 176 bytes from the index file (see Table 2). Let recall that the maximum size of our index file is restricted to the HDFS block. So, if the index file's block size is 128 MB, the maximum number of records that can be stored in it would be $128 * 1024 * 1024/176 = 762,600$, which is not large enough for practical purposes since small files in Hadoop can grow in quantity of millions. For the HPF archive to support more than 762,600 small files without the index file exceeding the block size, our workaround is to split metadata records into multiple index files using an extendible hashing mechanism. Limiting the size of the index file to the HDFS block has the benefit of preventing the deterioration of seek operations during random metadata lookups. The seek operation is used to move the reader to a specified offset in a file. When the client accesses multiple files, HPF seeks different offsets of the index files to read their metadata. Let's assume that the index file is too large and occupies several HDFS blocks located on distinct DNs. Every time the seek operation is performed on a different data block, the client needs to establish a new connection to the DN. This operation becomes expensive for random seeks between different blocks of the same file, as it takes time to establish a new connection with the DN.

**Table 2**
Metadata fields information.

| Field | Size (bytes) |
|---|---|
| File Name Hash | 128 |
| Data Part File Position | 16 |
| offset | 16 |
| Size | 16 |
| Total Size | 176 |

The extendible hashing technique has been specially chosen in our design because of its ability to easily split an overflowed bucket by creating a new one and provides constant access time to them. The bucket represents an index file when stored on HDFS. A second hash function called the order-preserving minimal perfect hash function (OPMPHF) is built for each bucket and is responsible for memorizing the records writing order before they are written into the corresponding index file. Since a metadata record is of fixed size and the OPMPHF is there to give us its position within the index file, it is very easy to estimate the exact part of the index files to read a file's metadata information (offset and limit). In Section 3.2.1 and Section 3.2.2, we discuss how we use the extendible hash function and the OPMPHF.

One known problem with HAR is that it employs a two-level index, which degrades access efficiency. HPF has a single-level index, but it splits the information into several index files. As shown earlier, a small file's metadata contains the minimal information required to restore the file's content from one of the part files. The HPF index system uses two hash functions to identify a metadata location. Every small file's metadata is stored in one of the index files, and the extendible hashing [13][38] helps to determine that index-* file. Moreover, the OPMPHF [14] helps to quickly find the exact location of the metadata information within that index file.

#### 3.2.1. Index file access

An Extendible Hash Table (EHT) [13][38] is a dynamic hashing technique. As defined in [37]:

**Definition 1.** An EHT is a hash table in which the hash function is the last few bits of the key and the table refers to buckets. Table entries with the same final bits may use the same bucket. If a bucket overflows, it splits, and if only one entry referred to it, the table doubles in size. If a bucket is emptied by deletion, entries using it are changed to refer to an adjoining bucket, and the table may be halved.

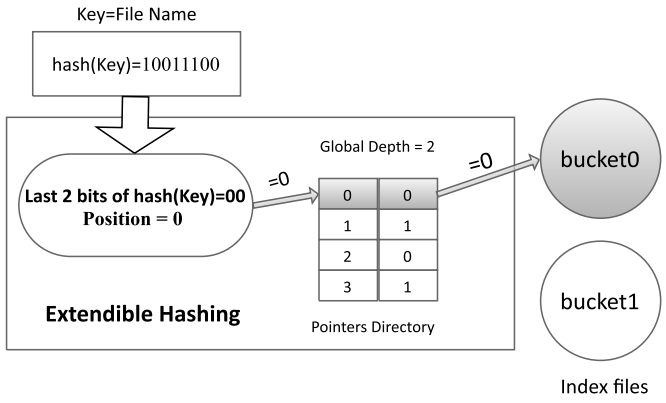



**Fig. 5.** To insert a record, extendible hashing selects the last bits of hash(key) according to the global depth and looks for the corresponding bucket from its pointers directory.

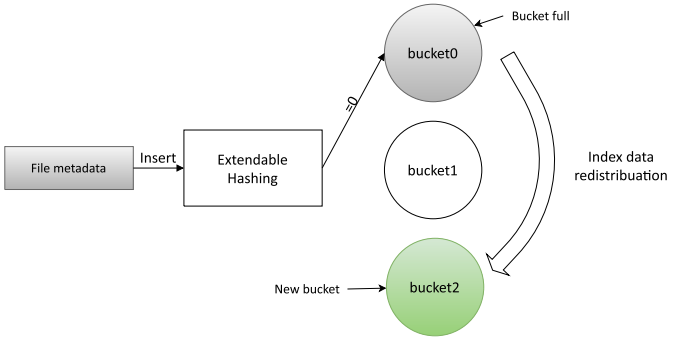



**Fig. 6.** Bucket split.

The EHT, first, helps to know which index files hold information about a file in constant time and saves from searching through all index files. And secondly, when an index file's size becomes larger and exceeds its maximum capacity, the EHT provides a mechanism to split it and move some of its contents in a new index file. EHT has three key components: The global depth, a directory containing pointers to buckets, and buckets' local depth. The global depth and pointers directory are used to select the bucket where to insert a key and the buckets' local depth is used to split a full bucket by creating a new one. Assume that the $hash(key)$ function returns a string of bits. The first $k$ bits of each string will be used as indices to decide which bucket the key will be put in using the pointers' directory (see Fig. 5). $k$ is the global depth and is chosen small such that the index of every item in the table is unique. More detail about extendible hashing can be found in [13]. So, a metadata record is inserted in the bucket by using the file name hash value as a key and the described process. Only one bucket is available when the HPF archive file is newly created. When the bucket is full, a new bucket is created by calling the EHT split operation as illustrated by Fig. 6. When a bucket reaches its maximum capacity, EHT's split mechanism is used to dynamically increase the number of buckets (index files) while maintaining direct access during lookups. The first step in bucket splitting is to create a new bucket, and the second is to redistribute data from the old bucket to the new one. Bucket data relocation is performed as follows: we recalculate all records' positions from the old bucket and move those whose locations have changed to the new one. Finally, the EHT is serialized and stored as an extended attribute [2] of the final archive file. At this stage, the metadata records are in memory but arranged in buckets. The next step will be to determine the order in which the records should be written into the index file.

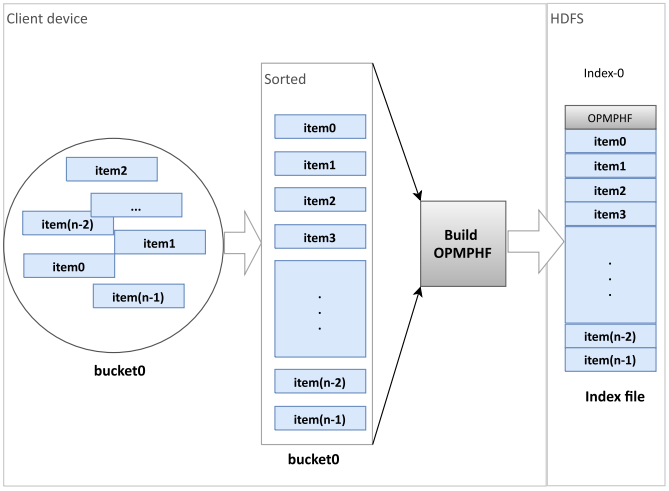



**Fig. 7.** We sort the records, build the OPMPHF, and write data to the index file.

### 3.2.2. Index file structure

HPF does not directly save bucket records in the index file, if we do so, we will have to load the entire index file during access before obtaining a metadata record, something we absolutely want to avoid. Instead, we'll employ a hash function to tell us where to save and search for metadata inside the index file. Notice that the metadata of two different files cannot be at the same position in the index file, therefore, the hash function we need must not present a collision risk. This restriction makes it impossible to use standard hashing techniques such as linear hashing, extendible hashing, etc. Fortunately, there are so-called perfect hash functions that are collision-free and meet our requirements. A perfect hash function maps a static set of $n$ keys to a set of $m$ integer numbers without collisions, where $m$ is greater than or equal to $n$. If $m$ equals $n$, the function is known as the minimal perfect hash function (MPHF) and is a bijection function. One special type of the MPHF is the order-preserving MPHF (OPMPHF) [14]. In [33], the perfect hash function is defined as order-preserving if the keys are arranged in a given order and the function preserves this order in the hash table, as illustrated in Fig. 7. If the order is lexicographic, the OPMPHF is called Monotone Minimal Perfect Hash Function [7] (MMPHF).

As displayed in Fig. 7, we firstly sort each bucket keys lexicographically, and then we use these keys to build the OPMPHF. The OPMPHF takes a key and returns its order among all keys, so OPMPHF($key_k$) will return $k$ if a $key_k$ is at position $k$. Finally, we persist the OPMPHF and the bucket's records within the index file with the OPMPHF on top followed with metadata records by preserving the ordering. The creation of the HPF file ends when all index files are built, and the temporary index file is deleted. Algorithm 1 condenses the entire creation process. The OPMPHF doesn't take much space. Its size depends on the total number of keys in the bucket. HPF uses the file name as the key, and it is interesting to note that OPMPHF algorithms don't need the entire key. Generally, perfect hash functions require less than 3 bits per key during their building. According to [7], for a set $S$ of $n$ elements out of a universe of $2^w$ elements, $O(n \log \log w)$ bits are sufficient to hash monotonically with evaluation time $O(\log w)$. We can get $O(n \log w)$ bits for space and $O(1)$ query time. This implies that a sorted table can be searched using just $O(1)$ table accesses.

### 3.3. File access & append

The example in Fig. 8 shows the 4 steps of the file's access in HPF:

**Algorithm 1:** The first loop sequentially shows instructions used by each thread for merging. The second loop process each bucket to get the final index file.

```
1  files = A set of small files;
2  buckets = EHF buckets;
3  Create the data part and the temporaryIndex file;
4  Create one bucket and add it to buckets;
   /* Files merging                                    */
5  for f in files do
6      Merge f with part file;
7      Get f metadata;
8      Append the metadata to the temporaryIndex file;
9      Append the f name to the names file;
10     bucket = Get from buckets using EHF;
11     Add f metadata bucket;
12     if bucket is full then
13         newBucket = Create new bucket and it index file;
14         Redistribute data to newBucket using EHF;
15         Add the newBucket to buckets;
16     end
17 end
   /* Building index files                             */
18 for b in buckets do
19     Sort b's metadata records;
20     Build the OPMPHF;
21     Create an empty index file to HDFS;
22     Write the OPMPHF to the index file;
23     Write the metadata records to the index file;
24 end
```

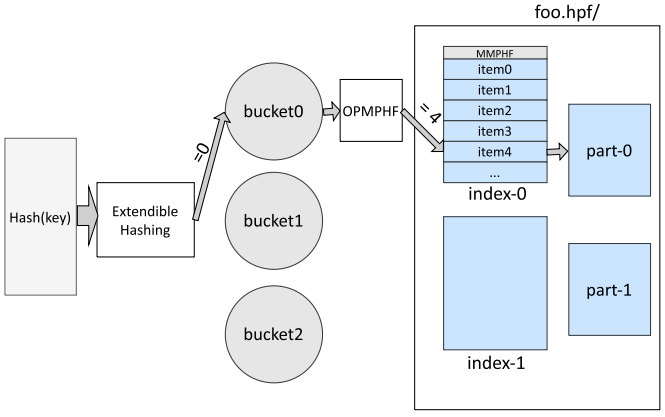



**Fig. 8.** The extendible hashing determines the index file containing a metadata record. After the OPMPHF determines the exact offset of the metadata information within the index file.

(1) Firstly, we derive from the file name provided by the client the corresponding hash value.

(2) Secondly, from the hash value, the EHF gives the bucket position, which also corresponds to the index file containing the metadata. If EHF returns **i**, the metadata can be found in the index file named **index-i**.

(3) Thirdly, from the hash value, OPMPHF retrieves the metadata record position within the index file. Since each file's metadata occupies 176 bytes, we get the exact metadata $offset$ from Equation (1) and then read the metadata information from $offset$ to $offset + 176bytes$ of the index file.

$$offset = \Upsilon + OPMPHF(file\_key) * 24Bytes, \quad (1)$$

Where:

$\Upsilon$ = OPMPHF size in index file,

$file\_key$ = file name hash code value.

(4) Finally, having the small file metadata (filename hash, part file, offset, file size), we compare the derived file name hash value with the one present in the metadata information. If they are not equal, this means the file we are looking for doesn't exist in the archive. In case they are equal, we access the part file and read the file's content. As a result, metadata lookups in the HPF are guaranteed to take $O(1)$ time.

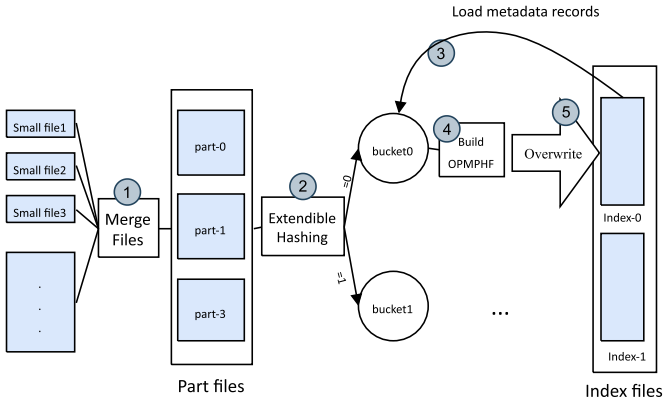



**Fig. 9.** After merging the files, the index files that receive new metadata records are rebuilt.

Adding new files to the HPF is almost identical to the creation process. Whenever the client wants to add more files, we provide as in Fig. 9:

(1) The client uses multiple threads for merging small files and, temporarily saves their metadata records to a temporary index file.

(2) After the merging, the newly added files metadata records are distributed into existing buckets using the EHF.

(3, 4, 5) The only difference from the creation process is that before rebuilding the OPMPHF, we have to reload into the buckets that have new records the content of their associated index file. For each of these buckets, build again their OPMPHF and overwrite the contents of their index file.

Adding new files is the weakest point of HPF as it requires rebuilding certain index files. This is because an ordering must be maintained to optimize access. Since accesses are more frequent than adding new files, HPF's current design prioritizes access time over new files adding efficiency. Unlike the MapFile, HPF does not require the client to sort the files before the archive creation or when adding more files. For this reason, if the client adds new files, we must rebuild the concerned index files. The goal of HPF is to optimize for file accesses with fewer I/Os, memory, and processing in counterpart adding more files to the archive files is possible but requires more overhead since it can involve the reconstruction of some of the index files.

## 4. Additional implementation issues

### 4.1. Recovery from failures

Waiting for the merging to complete before building the index files is risky, there are benefits and disadvantages to doing so. As benefit, the creation of the HPF file becomes faster since it avoids the multiple network communications that might occur between the client and nodes of the cluster. As a disadvantage, the client could be unreliable and crash at any moment. A crash can interrupt the HPF file creation or files appending operation. When building index files, buckets containing metadata records, and all information concerning our hash functions are built at the client-side. If the client crashes during the process, those information will be lost and, it will be impossible to restore files appended to big part files. Because of this problem, we have implemented a recovery mechanism that enables us to prevent losing these important

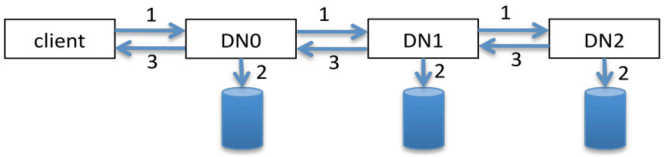



**Fig. 10.** HDFS block Writing&Replication process [23].

information in case of client failure. Our recovery mechanism uses a temporary index file. During the merging process, every time a file is appended file to a part file, its metadata record is directly added to the temporary index file. So even if the client crashes during the merging, we can easily restore the merging state before the crash as the already processed files' metadata will be in the temporary index file, there is no information loss. HPF offers a second option that moves the index file construction process to the NN side in order to benefit from its High availability. Once this option is specified, after the merging, a signal is sent to the NN that uses the temporary index file to build the index files. This high availability of the NN can also help to face the problems that could happen in the case of client failure, but since the temporary index file is this time on HDFS, the merging step requires additional network calls from the client to HDFS to add the metadata to the temporary index file.

### 4.2. Improving IO performance

#### 4.2.1. Write performance

For creating a file or appending data to an existing one on HDFS, the client firstly interacts with the NN. The NN provides the addresses of the DNs on which the client starts writing data. By default, HDFS performs three replicas for every data block on three different nodes. As shown in Fig. 10, the client only writes data on the first DN, this one performs the first replication by copying on the second DN and the second DN on the third. Once the replicas are created an acknowledgment is sent to the client before the client continues writing more data. Replication is done serially [23] from one DN to another, not in parallel. During the data blocks writing to the DN storage space, blocks are written as regular files on the disk. Transferring data through the network and writing data blocks to disk are the most time-consuming operations of HDFS. For the data transfer, the problem can be the network state between the client and the first DN because it is often an external network to the cluster and there is no guarantee of its reliability. This external network can be slower than the internal one (between the DNs and the NN) which often is more stable, reliable, and high throughput. For slow disk writing, this is because the majority of Hard Drives are mechanical. Fortunately, Hadoop comes with another data writing mode or **Storage Policy** named the Lazy Persist write [3]. In the Lazy Persist mode, data is written in each DN in an off-heap memory (Fig. 11) located in the RAM. Writing in the off-heap memory is faster than writing on the hard drive, it saves the client from waiting for the data to be written to the disk. According to [3] the DataNodes will flush in-memory data to disk asynchronously, thus removing expensive disk IO and checksum computations from the performance-sensitive IO path. HDFS provides best-effort persistence guarantees for Lazy Persist Writes. Rare data loss is possible in the event of a node restart before replicas get persisted. Unlike HAR and MapFile, HPF support by default this lazy persists policy. We used the LazyPersist storage policy in our approach to append files to part-* files and speed up the HPF file creation. The weakness of LazyPersist is that, in version 2.9.1 of Hadoop that we used to perform our experiment, files created with the LazyPersist storage policy do not support the data-append functionality. To maintain the HPF files appending functionality after the creation, we reset the storage policy of

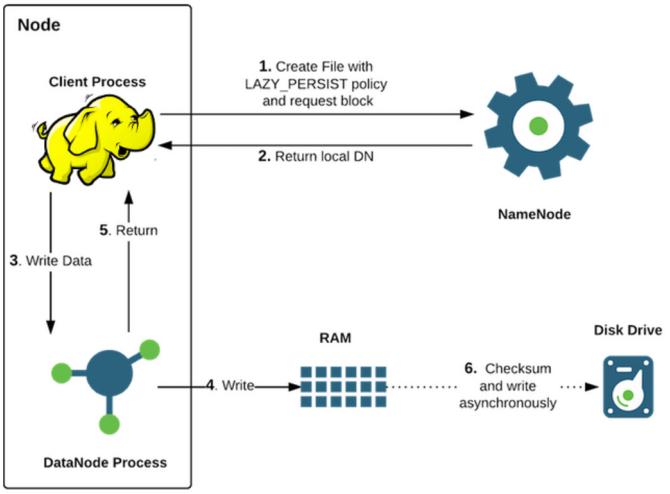



**Fig. 11.** Lazy Persist Writes [3].

all HPF part-* files to the default mode. Our experiments confirm that the Lazy Persist Writing boosts HPF file creation compared to other solutions that don't support this functionality.

#### 4.2.2. Read performances

Even if the HPF file can directly access the part of the index file that holds a metadata record, we still need some little I/Os operations to read this information. The primary purpose of HPF is to improve small file access in its archive as if the files were saved as normal files on HDFS. It is then important to totally eliminate any disk operation related to metadata access. One option is to cache the metadata in the client or the NN memory but this will look like bringing back the small file problem we are fighting for. The only option we have left is to find a way and tell the DN at access time to keep the index files block in memory and not in disc. This is done by using the Centralized Cache Management system of HDFS. According to [4], the Centralized cache management in HDFS is an explicit caching mechanism that allows users to specify paths of files to be cached by HDFS. The NameNode will communicate with DataNodes that have the desired blocks on disk, and instruct them to cache the blocks in the off-heap caches. This caching system allows us to tell the DN to maintain our index files' blocks in the off-heap memory, not on the disk. By doing so, we avoid additional I/Os operation during metadata lookups, furthermore reducing the disc I/Os to the only one required to read the file's content from part files.

### 4.3. File access performance analysis

Let $T_M$ be the time an archive file needed to retrieve a file's metadata from its index file(s) and $T_C$, the time needed to restore file's content from the merged file. The total time it takes to access a file ($T_{Access}$) from the archive file is defined by Equation (2).

$$T_{Access} = T_M + T_C \tag{2}$$

If $T_{Access/HAR}$, $T_{Access/MapFile}$ and $T_{Access/HPF}$ are respectively the access times in the HAR file, MapFile, and HPF files, we will have:

$$T_{Access/HAR} = T_{M/HAR} + T_{C/HAR}$$
$$T_{Access/MapFile} = T_{M/MapFile} + T_{C/MapFile}$$
$$T_{Access/HPF} = T_{M/HPF} + T_{C/HPF}$$

Metadata access from the HAR file and the Map file require to read and process entirely all the index file(s). Since the HAR

file's index files are bigger and hold much more information than MapFile's index file, the metadata access from MapFile is faster than the metadata access from HAR file: $T_{Acess\_metadata/MapFile} < T_{Acess\_metadata/HAR}$. Metadata access from HPF index file(s) is direct and does not require reading and processing the entire index file. That's why the HPF file metadata access time is almost negligible compared to HAR file and MapFile:

$$T_{M/HPF} < T_{M/MapFile} < T_{M/HAR} \quad (3)$$

$T_C$, the time needed to restore a file's content from the merged file can be different when the file is accessed from HAR file, MapFile or HPF files. This time is influenced by several factors like the cost of the seek operation, the use or not of compression algorithm to reduce the file size. If we assume that for the same file: $T_{C/HPF} = T_{C/HAR} = T_{C/MapFile}$. According to equation (3),

$$T_{Acess/HPF} < T_{Acess/MapFile} < T_{Acess/HAR} \quad (4)$$

Equation (4) suggests that file's access in HPF file is faster than access to files in HAR and MapFile. This is also confirmed by our experiments presented in Subsection 5.2.1.

## 5. Experimental evaluation

We have implemented an open-source HPF prototype[1] and compared its performance against LHF, HAR file, MapFile, including the native HDFS. Our tests considered metrics such as access time, creation time, and the NameNode memory utilization.

The experimental environment is built on a cluster of 6 nodes. One serves as the NameNode, while the other five serve as DataNodes. Each node is a server with two CPU cores of 2.13 GHz each, 8 GB for RAM, and 500 GB Hard Disk. The nodes operating system is Ubuntu. The client on which our datasets are located is a Laptop running Microsoft Windows 10 Pro and has 16 GB of RAM, 1 TB of disk, Processor Intel® Core™ i7-6500U. The Hadoop version is 2.9.1 and the JDK version is jdk1.8.0_102. The number of replicas is set to 3 and the HDFS block size is set to 512 MB. For the test purpose, we process small log text files collected from applications running on different servers.

We use 5 datasets containing respectively 100000, 200000, 300000, and 400000 files. Their total size is respectively 1.44 GB, 2.37 GB, 3.30 GB, and 4.23 GB. Files sizes range from 1 KB to 10 MB. In Hadoop, a file is considered small if its size is less than the block size. For a file larger than the block size, it is recommended to directly save it as a normal HDFS file. Archive systems intend to combine small files into big ones that can take at least one HDFS block and reduce the amount of metadata needed to represent them. So, adding large files that already have one or more HDFS blocks to an archive is not that interesting. The first category of our tests is intended to evaluate the archives creation and the datasets upload time to HDFS. The second category evaluates access performance in HPF, LHF, HAR, MapFile, as well as their performance when accessed directly from HDFS. We also look at other aspects such as disk space usage on DNs, the NN memory usage. For the experience, the maximum capacity of the HPF index file is set to 200000 records. We run all our tests several times in order to reduce potential errors that may be due to network congestion or other factors.

### 5.1. Archives creation

For each method, we built four archive files, each containing 100000, 200000, 300000, and 400000 small files. We then measure the time taken by each solution to create the archives and the time it takes to upload each dataset to HDFS. The collected results are displayed in Fig. 12. Being the only one to support LazyPersist during creation, HPF outperforms other solutions as we were expecting. LHF, MapFile, and HAR follow. As compared to other solutions, the construction of HAR archives is extremely slow and cumbersome. The HAR file creation requires a prior upload of the datasets on HDFS, which is its biggest bottleneck since uploading massive small files on HDFS is a slow process as observable in Fig. 12. Furthermore, due to the prior files upload to HDFS, HAR can easily trigger small file problems before the archive construction process begins. Therefore, we can easily derive that for a massive quantity of files that the NameNode's memory can't handle, constructing the HAR file would be impossible. Once the files are on HDFS, HAR creates the archive file using a map-reduce job. We considered the file upload time as part of the HAR file construction time. The fact that HAR does not use compression also means that it takes more time to write data on the disc and using MapReduce to create its archive file means a lot of network calls and data transfer within the cluster's nodes.

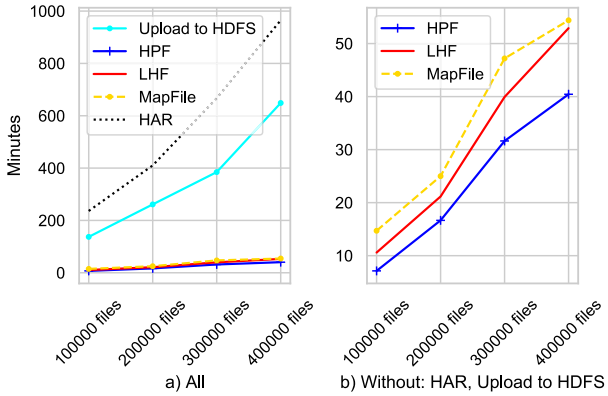



**Fig. 12.** The performance of creating a new archive file.

Unlike HAR, HPF, LHF, and MapFile do not require the datasets to be uploaded to HDFS prior to building their archives. Rather, the small file's content is directly appended from the client device to the archive files on HDFS. This is the reason why they are faster. MapFile and HPF files have better use of the bandwidth since the data are compressed before being transmitted over the network. HPF applies the compression at file level, while MapFile compresses both files and blocks. HDFS encourages archives creation to be a one-time operation. The majority of the time, once the archive is created, only accesses are performed. The reason why access efficiency presented in the following section, is the key metric used to measure the performance of each solution.

### 5.2. Access efficiency

To evaluate the file access efficiency within the archives, for each dataset, we randomly choose 100 files and measure the time each solution takes to recover their content. We also measure the time it takes to retrieve these files when they are stored on HDFS as regular files. As stated in Section 2, LHF, MapFile, and HAR use some caching techniques to improve their access efficiency. To get a sense of their true performance when compared to the HPF file, we first evaluated the access performance without considering the caching effect, then by considering the caching effect.

#### 5.2.1. Without the caching effect

The comparison results are displayed in Fig. 13. Without the caching effect, HAR gives the worst performance which degrades linearly with the size of the dataset. HPF is the fastest, even faster

[1] The source code of Hadoop Perfect File is available at https://github.com/tchaye59/Hadoop-Perfect-File.

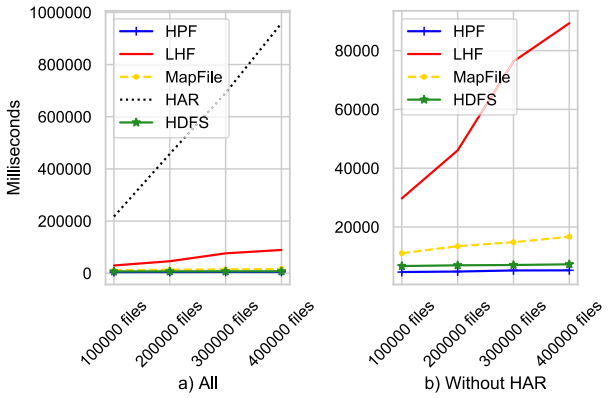



**Fig. 13.** The access performance without caching.

than the original HDFS, with HDFS, MapFile, LHF, and HAR following closely behind. Small file access in HPF is estimated to be about 40% faster than in the initial HDFS, 535% faster than LHF, 179% faster than MapFile, and 11294% faster than HAR. Before the tests, we expected HPF to be faster than MapFile, HAR, and LHF, but not HDFS. How is this possible, one may wonder? These are some of the reasons we noticed:

- Since we manage to disable their caching effect, each file access requires MapFile, LHF, and HAR to read and process the entire index file(s) for metadata lookup. They then take much more time to recover the actual file's content.
- MapFile compresses each small file and block of its data file. Therefore, during access, it needs two levels of decompression. It means during access MapFile needs to decompress a whole HDFS block and after the accessed small file. This strategy is not optimal for random file accesses since accessing randomly can require jumping from one block of the data file to another. HPF applies compression only at the file level, therefore, requires one level of decompression, and is well suited to random accesses.
- HDFS keeps its metadata in memory of NN, while HPF keeps its index files in memory of DNs using the Hadoop centralized cache management system. So, why is HPF access slightly faster than HDFS? We noticed that this is due to the communication protocols. With HDFS, the communication between the client and the NN to get the file metadata is done by using the RPC (Remote Procedure Call) protocol. But to get the file metadata with HPF, the communication is done between the client and the DN by using sockets that are faster than RPC calls. HDFS read and write files on DNs discs using sockets while HPF just performs a file read operation on a small part of the index file to get metadata.

*5.2.2. With the caching effect*

The previous tests are repeated, but this time the caching effect is taken into account. The results are displays by Fig. 14. We observe a major improvement with LHF, HAR, and MapFile but, there is no much difference with HPF and HAR. Despite these improvements, HPF continues to perform better, followed by LHF, MapFile, HDFS, and HAR. One important thing to notice is that, except HAR, all other solutions outperform HDFS. File access in HPF is about 48%, 17%, 35%, 105% faster than the original HDFS, LHF, MapFile and HAR if we consider the caching effect.

*5.3. NameNode's Memory usage*

To have a visual perception of the small file problem, it is important to plot the quantity of memory it costs to the NN when the

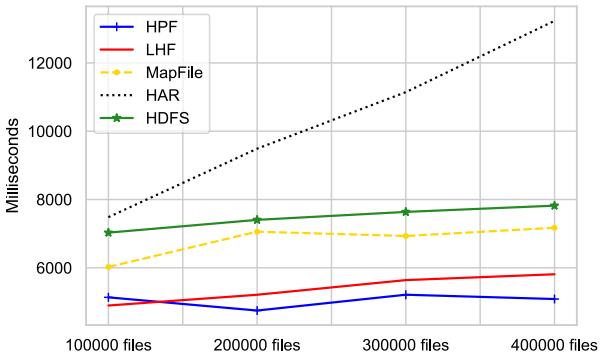



**Fig. 14.** The access performance with caching.

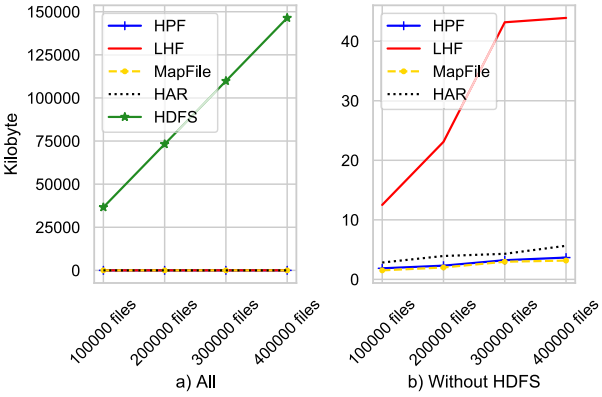



**Fig. 15.** NameNode Memory usage.

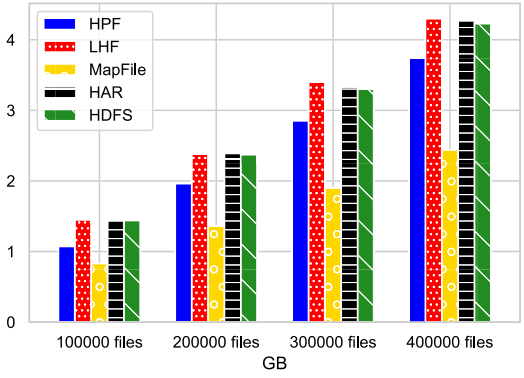



**Fig. 16.** Sizes comparison.

datasets are directly stored on HDFS or stored using an archive solution as done in Fig. 15. This Figure shows the importance of the archive files systems in HDFS. They are more efficient than HDFS at storing small files. They occupy less memory on the NameNode than native HDFS. That is why they are the primary solution to HDFS's small file problem. From Fig. 15b we can observe that MapFile uses less metadata than all other files. Compared to other approaches, MapFiles is only composed of two files (data file, index file), and its data file is highly compressed. With two-level compression, MapFile consumes fewer HDFS blocks for storage than other solutions. After the MapFile file, HPF, HAR, LHF follow respectively.

*5.4. Archive files sizes after creation*

Finally, we measured archives sizes which gives us an idea of the disk space used from the DNs (see Fig. 16). Since the HAR and

LHF do not use compression, their sizes are almost equal to the size of the datasets when stored directly on HDFS. The size of the HPF and MapFile is reduced due to compression. According to our analysis, MapFile saved about 42% of disk space and HPF about 11%.

## 6. Conclusion

HDFS was designed to provide the best performance with large files rather than small files. To deal with the small file issue in HDFS, the best archiving systems must consider the NN's memory overflow while still ensuring fast access to small files. The previous works mainly focus on reducing NN's memory overflow. They effectively reduce the metadata load in NN's memory, but at the cost of poor access efficiency. This paper represented HPF, a new type of index-based archiving system. HPF is specially designed to greatly reduce the extra I/Os and computations caused by index file processing during access. Our experiment confirms that our solution outperforms other file systems such as LHF, HAR, MapFile, and the original HDFS when it comes to file access. We have made our design and implementation open source so that other researchers can use it to improve their systems of small files in HDFS.

## CRediT authorship contribution statement

**Yanlong Zhai:** Conceptualization, Resources, Writing – review & editing. **Jude Tchaye-Kondi:** Software, Writing – original draft, Writing – review & editing. **Kwei-Jay Lin:** Validation, Writing – review & editing. **Liehuang Zhu:** Resources, Supervision. **Wenjun Tao:** Data curation. **Xiaojiang Du:** Supervision. **Mohsen Guizani:** Supervision.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

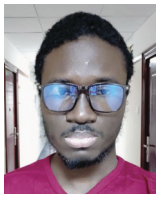
## Acknowledgments

The authors thank the anonymous reviewers for their insightful suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61602037).

## References

[1] Alibaba, alibaba/tfs, https://github.com/alibaba/tfs, 2021.
[2] Apache Hadoop, Extended attributes in hdfs, https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-hdfs/ExtendedAttributes.html, 2021.
[3] Apache Hadoop, Memory storage support in hdfs, http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/MemoryStorage.html, 2021.
[4] Apache Hadoop, Centralized cache management in hdfs, http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html, 2021.
[5] K. Batcher, Design of a massively parallel processor, IEEE Trans. Comput. 29 (1980) 836–840, https://doi.org/10.1109/TC.1980.1675684.
[6] D. Beaver, S. Kumar, H.C. Li, J. Sobel, P. Vajgel, et al., Finding a needle in haystack: Facebook's photo storage, in: OSDI, vol. 10, 2010, pp. 1–8.
[7] D. Belazzougui, P. Boldi, R. Pagh, S. Vigna, Monotone minimal perfect hashing: searching a sorted table with o (1) accesses, in: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2009, pp. 785–794.
[8] K. Bok, H. Oh, J. Lim, Y. Pae, H. Choi, B. Lee, J. Yoo, An efficient distributed caching for accessing small files in hdfs, Clust. Comput. 20 (2017) 3579–3592.
[9] X. Cai, C. Chen, Y. Liang, An optimization strategy of massive small files storage based on hdfs, in: 2018 Joint International Advanced Engineering and Technology Research Conference (JIAET 2018), Atlantis Press, 2018.
[10] C. Choi, C. Choi, J. Choi, P. Kim, Improved performance optimization for massive small files in cloud computing environment, Ann. Oper. Res. 265 (2018) 305–317.
[11] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113.
[12] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, Y. Li, A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by powerpoint files, in: 2010 IEEE International Conference on Services Computing (SCC), IEEE, 2010, pp. 65–72.
[13] R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong, Extendible hashing—a fast access method for dynamic files, ACM Trans. Database Syst. 4 (1979) 315–344, https://doi.org/10.1145/320083.320092.
[14] E.A. Fox, Q.F. Chen, A.M. Daoud, L.S. Heath, Order preserving minimal perfect hash functions and information retrieval, in: Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 1989, pp. 279–311.
[15] S. Fu, L. He, C. Huang, X. Liao, K. Li, Performance optimization for managing massive numbers of small files in distributed file systems, IEEE Trans. Parallel Distrib. Syst. 26 (2015) 3433–3448.
[16] G. lz4, lz4/lz4, https://github.com/lz4/lz4, 2021.
[17] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google File System, vol. 37, ACM, 2003.
[18] Hadoop, Hadoop archives guide, https://hadoop.apache.org/docs/r2.7.5/hadoop-archives/HadoopArchives.html, 2021.
[19] H. He, Z. Du, W. Zhang, A. Chen, Optimization strategy of hadoop small file storage for big data in healthcare, J. Supercomput. 72 (2016) 3696–3707.
[20] Y. Huo, Z. Wang, X. Zeng, Y. Yang, W. Li, C. Zhong, Sfs: a massive small file processing middleware in hadoop, in: Network Operations and Management Symposium (APNOMS), 2016 18th Asia-Pacific, IEEE, 2016, pp. 1–4.
[21] W. Jing, D. Tong, G. Chen, C. Zhao, L. Zhu, An optimized method of hdfs for massive small files storage, Comput. Sci. Inf. Syst. 15 (2018) 533–548.
[22] H. Kim, H. Yeom, Improving small file i/o performance for massive digital archives, in: 2017 IEEE 13th International Conference on e-Science (e-Science), IEEE, 2017, pp. 256–265.
[23] H. Kuang, K. Shvachko, N. Sze, S. Radia, R. Chansler, Append/hflush/read design, Yahoo! HDFS Team, 2009.
[24] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, Oper. Syst. Rev. 44 (2010) 35–40.
[25] B. Meng, W.-b. Guo, G.-s. Fan, N.-w. Qian, A novel approach for efficient accessing of small files in hdfs: Tlb-mapfile, in: 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), IEEE, 2016, pp. 681–686.
[26] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, M. Ronström, Hopsfs: scaling hierarchical file system metadata using newsql databases, in: FAST, 2017, pp. 89–104.
[27] S. Niazi, M. Ronström, S. Haridi, J. Dowling, Size matters: improving the performance of small files in hadoop, in: Proceedings of the 19th International Middleware Conference, ACM, 2018, pp. 26–39.
[28] J.-f. Peng, W.-g. Wei, H.-m. Zhao, Q.-y. Dai, G.-y. Xie, J. Cai, K.-j. He, Hadoop massive small file merging technology based on visiting hot-spot and associated file optimization, in: International Conference on Brain Inspired Cognitive Systems, Springer, 2018, pp. 517–524.
[29] P. Phakade, S. Raut, An innovative strategy for improved processing of small files in hadoop, Int. J. Appl. Innov. Eng. Manag. (2014) 278–280.
[30] S. Sheoran, D. Sethia, H. Saran, Optimized mapfile based storage of small files in hadoop, in: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE, 2017, pp. 906–912.
[31] K. Shvachko, Name-node memory size estimates and optimization proposal, Apache Hadoop Common Issues, HADOOP-1687, 2007.
[32] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2010, pp. 1–10.
[33] sourceforge, Minimal perfect hash functions - introduction, http://cmph.sourceforge.net/concepts.html, 2021.
[34] W. Tao, Y. Zhai, J. Tchaye-Kondi, Lhf: a new archive based approach to accelerate massive small files access performance in hdfs, in: Proceedings of the Fifth IEEE International Conference on Big Data Service and Applications, 2019.
[35] C. Vorapongkitipun, N. Nupairoj, Improving performance of small-file accessing in hadoop, in: 2014 11th International Joint Conference on Computer Science and Software Engineering (JCSSE), IEEE, 2014, pp. 200–205.
[36] T. White, Hadoop: The Definitive Guide, 4th ed., O'Reilly, Beijing, 2015, https://www.safaribooksonline.com/library/view/hadoop-the-definitive/9781491901687/.
[37] xlinux, https://xlinux.nist.gov/dads/HTML/extendibleHashing.html, 2021.
[38] D. Zhang, Y. Manolopoulos, Y. Theodoridis, V.J. Tsotras, Extendible hashing, in: Encyclopedia of Database Systems, Springer, 2009, pp. 1093–1095.
[39] T. Zheng, W. Guo, G. Fan, A method to improve the performance for storing massive small files in hadoop, in: 7th International Conference on Computer Engineering and Networks, 2017.

**Yanlong Zhai** received the B.Eng. degree and Ph.D. degree in computer science from Beijing Institute of Technology, Beijing, China, in 2004 and 2010. He is an Assistant Professor in the School of Computer Science, Beijing Institute of Technology. He was a Visiting Scholar in the Department of Electrical Engineering and Computer Science, University of California, Irvine. His research interests include cloud computing and big data.



**Jude Tchaye-Kondi** received the BS degree from the Department of Computer Science, Catholic University of West Africa. He joins in 2017 Beijing Institute of Technology, China as a graduate student and is currently pursuing his Ph.D. His research interest includes parallel and distributed computing, edge intelligence, machine learning.

**Kwei-Jay Lin** is a Professor in the University of California, Irvine. He is a Chief Scientist at the NTU IoX Research Center at the National Taiwan University, Taipei. He is an IEEE Fellow, and Editor-In-Chief of the Springer Journal on Service-Oriented Computing and Applications (SOCA). He was the Co-Chair of the IEEE Technical Committee on Business Informatics and Systems (TCBIS) until 2012. He has served on many international conferences. His research interest includes service-oriented systems, IoT systems, middleware, real-time computing, and distributed computing.

**Liehuang Zhu** received the B.Eng. and Master Degrees in computer application from Wuhan University, Wuhan, Hubei, China, in 1998 and 2001 respectively. He received the Ph.D. degree in computer application from Beijing Institute of Technology, Beijing, China, in 2004. He is currently a Professor in the Department of Computer Science, Beijing Institute of Technology, Beijing, China. He is selected into the Program for New Century Excellent Talents in University from Ministry of Education, China. His research interests include internet of things, cloud computing security, internet, and mobile security.

**Wenjun Tao** joins in 2016 Beijing Institute of Technology, China as a graduate student. His research interest includes big data and cloud computing.

**Xiaojiang Du** received the B.S. and M.S. degree in electrical engineering from Tsinghua University, Beijing, China, in 1996 and 1998, respectively, and the M.S. and Ph.D. degrees in electrical engineering from the University of Maryland, College Park, MD, USA, in 2002 and 2003, respectively. He was an Assistant Professor in the Department of Computer Science, North Dakota State University, between August 2004 and July 2009. He is currently a Professor in the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, USA. He is a Life Member of the ACM. He received the Excellence in Research Award at North Dakota State University in May 2009. His research interests include security, wireless networks, computer networks, and systems. He has published more than 200 journal and conference papers in these areas.

**Mohsen Guizani** received the B.S. (with distinction) and M.S. degrees in electrical engineering and the M.S. and Ph.D. degrees in computer engineering from Syracuse University, Syracuse, NY, USA, in 1984, 1986, 1987, and 1990, respectively. He was an Associate Vice President with Qatar University, the Chair in the Computer Science Department, Western Michigan University, the Chair in the Computer Science Department, University of West Florida, and the Director of graduate studies with the University of Missouri-Columbia. He is currently a Professor in the Department of Computer Science and Engineering, Qatar University, Qatar. He is the author/coauthor of nine books and more than 450 publications in refereed journals and conferences. His research interests include wireless communications and mobile computing, smart grid, cloud computing, and security.