Addressing Hadoop's Small File Problem With an Appendable Archive File Format

Thomas Renner, Johannes Müller, Lauritz Thamsen, Odej Kao
Technische Universität Berlin, Germany
{firstname.lastname}@tu-berlin.de

ABSTRACT

Hadoop has been used widely for data analytic tasks in various domains. At the same time, data volume is expected to grow even further in the next years. Hadoop recently introduced the concept Archival Storage, an automated tiered storage technique for increasing storage capacity for long-term storage. However, Hadoop Distributed File System's scalability is limited by the total number of files that can be stored, and it is likely that the number of files increases fast when using it for archival purposes.

This paper presents an approach for improving HDFS' scalability when using it as an archival storage. We present a tool that extends Hadoop Archive to an appendable file format. New files are appended to one of the existing archive data files efficiently without rewriting the whole archive. Therefore, a first fit algorithm is used to fill up the often not fully utilized fixed-sized data blocks of the archive data files. Index files are updated using a red-black tree providing guaranteed fast lookup and insert performance. We show that the tool performs well for different sizes of archives and number of files to add. By distributing new files efficiently, we also reduce the number of data blocks needed for archiving and, thus, reduce the memory footprint on the NameNode.

KEYWORDS

File Systems, Hadoop Distributed File System, HDFS, Metadata Management, Archival Storage

1 INTRODUCTION

As Data volume is growing, gaining insights into this data is becoming increasingly relevant for more and more application domains. One challenge is to evolve storage and processing infrastructure so it can handle these increased quantities in an economically viable way. In recent years, Hadoop Distributed File System (HDFS) [10] has emerged as a commonly used system for storing and accessing large amounts of data for analytical tasks. Key reasons for HDFS' success are its scalability and fault tolerance while using commodity hardware and its tight integration with modern distributed data flow engines, such as MapReduce [3], Spark [14], and Flink [1].

One technique to deal with the growing amount of data is the use of a automated tiered storage system [6, 12]. Thereby, data can be characterized, for instance as cold or hot data, and moved across tiers automatically, like a processing and an archival tier, composed of different storage types. Recently, HDFS also started to support an automated tiered storage based on user-defined policies known as Archival Storage. Unlike the well-known approach of "bringing compute to data" (i.e. data locality), the storage capacity of an archival storage is separated from the compute capacity. The archival tier consists of nodes with low compute power and disks designed for long-term storage. These nodes are used only for storing cold data, which should be data that needs to be processed less frequently, but nonetheless needs to be stored for archival purposes. In comparison, the processing tier consists of nodes with more compute capabilities and faster disks. Nodes of the processing tier are used for storing and processing hot data, which needs to be processed fast and more frequently. One advantage of the storage separation is that the archival tier can grow independent of the processing tier by simply adding new nodes. Another advantage is that all data is stored in a single HDFS instance and archival data does not need to be ingested from a separate storage system outside of HDFS. Thus, data analytic frameworks can transparently access archival data without taking care of data ingestion.

However, when using HDFS as an archival storage, it is likely that more and more data is stored and the total number of files increases, and thus HDFS can run into scalability and performance limitations. Currently, the total memory available on the NameNode, the central HDFS master node responsible for metadata management, is the primary scalability limitation of HDFS. This is because, every file, directory, and block is represented as an object in the NameNode memory. Each of these objects requires approximately 150 bytes. Various authors have investigated this limitation in HDFS, which is often referred to as the "HDFS small file problem". Some increase the available memory with federated NameNodes, thereby allowing to have multiple NameNodes, each storing a subset of metadata [9]. Others merge many files into one archive file, similar to a tarball and known as Hadoop Archives (HAR). A similar approach is called SequenceFile, which is a merged file consisting of binary key-value pairs that store the filename as key and the file contents as value. This is done to decrease the total number of files and, thus, memory footprint on the NameNode [4, 5, 7, 8]. Most of this work focuses on increasing the performance of small file access through new indexing strategies or caching methods and treat an archive as immutable. However, for an archival storage, time is not that critical, yet we need an archiving file format that allows to append data to an existing HAR. This would enable us to design an automated archival storage, in which cold data is automatically stored and appended to existing archives for long term storage with a low memory footprint at the NameNode allowing us to store more files.

In this paper we present Appendable Hadoop Archives (AHAR) [1], which extends the HAR file format with the functionality of efficiently appending new files to an already existing HAR. HAR is a widely used archive file format for HDFS and files can be accessed by many modern data analytic engines with the default HDFS connector. In addition, the file format allows us to merge files of different types and sizes. New files are appended to one

[1] AHAR in version 1.0, https://github.com/apache/trenner/ahar, accesPsed 2017-01-03

of the existing archive data files efficiently without rewriting the whole archive. Furthermore, a first fit algorithm is used to fill up the often not fully utilized existing fixed-sized data blocks of archive files, reducing unused space between the data blocks. HAR's index files for locating the data within the archive are updated using a red-black tree providing sufficient lookup and insert performance.

We evaluated AHAR's implementation on a 40 node cluster by adding different amounts of files to an already existing archive with various sizes. We compare AHAR with the currently only alternative strategy for appending new files to existing HAR archives: unarchiving and re-archiving the archives to include new files. Additionally, by distributing new data efficiently, we also reduce the number of data blocks needed for archiving, and thus can increase scalability by reducing the memory footprint on the NameNode.

Outline. The remainder of the paper is structured as follows. Section 2 presents the background. Section 3 presents the approach and implementation of AHAR. Section 4 presents our evaluation. Section 5 presents related work. Section 6 concludes this paper.

2 BACKGROUND

This section describes how files in HDFS are stored and accessed. In addition, it explains how HAR files can be used to increase HDFS scalability by reducing the namespace usage, and thus enabling to store more files.

2.1 Hadoop Distributed File System

HDFS is a distributed file system for reliably storing large files across nodes in a large cluster built from commodity hardware. As shown in Figure 1, HDFS follows a master-slave paradigm. The master node (i.e. NameNode) is responsible for metadata management that includes (a) maintaining the directory tree of all files in the file system and (b) tracking where across the available nodes the file is kept. The file itself is stored on the slave nodes (i.e. DataNode), which serve as a pure data storage.

Files are split into a series of blocks with a fixed size (i.e. block size) and multiple replicas, where each replica is stored on a different DataNode. Thus, if a node goes down, other replicas are available on different DataNodes. In Figure 1, for instance, the file *"/usr/marc/file.log"* consists of three blocks with a replication factor of two, labeled as 1, 2 and 4. All of these blocks are distributed on all available DataNodes. The block replication mechanism provides high fault tolerance, but also good data access performance for distributed dataflow engines. This is because data transfer bandwidth can be increased by concurrently accessing multiple disks and by more opportunities for data locality by placing computational tasks on the same node they have direct access to.

HDFS keeps the entire metadata management in the memory of the NameNode. For every file, directory and block, an object is stored in memory that occupies approximately 150 bytes. Therefore, it can be inefficient to store many small files in HDFS. Small files are files that are significantly smaller than the default block size. For example, if the default block size is 64 MB, a 64 MB file consumes the same amount of memory for metadata management at the NamenNode as a 1 MB file. Therefore, a recommendation to increase scalability is to keep the average file size large and if possible, merge smaller files into larger ones.

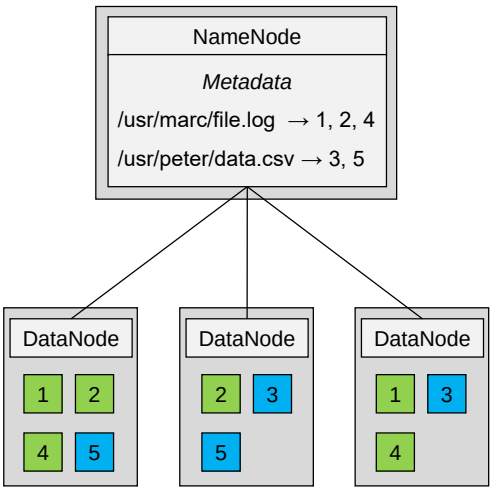

Figure 1: Exemplary HDFS setup storing two different files.

2.2 Hadoop Archive

Hadoop Archive (HAR) is a Hadoop file format that merges a collection of files into one or more large files, similar to tarballing. With HAR it is possible to increase the average file size and thus, increase scalability of HDFS by reducing namespace usage and memory footprint at the NameNode. Any HAR file contains two metadata files, called index and masterindex, and multiple data files, called $part_n$.

- $part_n$ are multiple files that together contain all data of the files that are collected in the archive. An HAR consists of multiple $part_n$ files, because the archiving process is done in parallel using a MapReduce job that produces the partitioned $part_n$ files.
- *index* stores location information on files and directories that are collected in the archive. The location is described through the part file containing that file and its specific offset and length within part file. For instance, an entry *"/input/sub0/fileA.txt file part-0 0 128 [...]"* specifies a "fileA" stored in "$part_0$", starting at offset 0 with a length of 128 bytes.
- *masterindex* is a level of indirection into the index file to make look ups faster. The index file is sorted by hash codes of the paths that it contains and the master index contains pointers to the positions in index for ranges of hash codes.

However, currently, HAR archives are immutable. Thus, adding or removing files to an already existing HAR file can only be done by unarchiving and re-archiving the archive including some new files. This can cost a lot of processing time and occupy extra cluster resources, because both functionalities are implemented as a MapReduce Job.

3 APPENDABLE HADOOP ARCHIVE

This section presents the design and implementation details of Appendable Hadoop Archive (AHAR). First, an overview about AHAR and its components is given. Afterwards, the functionality of adding new files to an existing HAR is discussed in more detail.

3.1 Overview

AHAR changes Hadoop Archive's characteristic from being immutable to be appendable. In particular, users can add a set of new files that is stored in HDFS to an already existing archive by using our java-based tool. For this, AHAR automatically a) appends the data of the new files to the existing $part_n$ files of the archive and b) effectively updates the metadata of an archive by adding new index entries for the new files.

Figure 2 shows how AHAR is integrated with the HAR file format. New files that should be added to the archive can either be stored on HDFS or on the users local file system. First, the binary data of new files is appended to one or multiple $part_n$ files stored in HDFS. Therefore, a first fit algorithm is used, which appends the binary data of a new file to a part file that has sufficient space to append the additional data without the need of creating a new data block. The goal is to avoid the creation of new blocks to reduce the memory footprint on the NameNode. Afterwards, the index and masterindex files for the new entries are updated and rewritten to HDFS. Therefore, the index file is parsed into a red-black binary tree, which allows us to quickly find, update and insert new index entries.

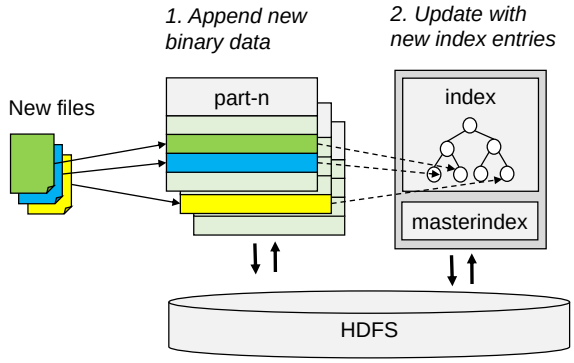

Figure 2: AHAR main components. First, the new data is appended to one $part_n$ file. Afterwards, the index files are updated with the new entries

We implemented AHAR as an extension of HAR without changing the two-way indexing strategy. Therefore, HDFS compatible data-analytics frameworks like MapReduce, Spark, or Flink can access the files of a HAR without the need of a new data format connector. Files in a HAR can be accessed simply by using the har prefix, e.g. *har://filepath* instead of *hdfs://filepath*. Furthermore, this design allows AHAR to be used with HDFS setups without any source code modification or additional configuration. In addition, the tool can be simply used by executing its jar with the arguments *addFiles harPath addingFilesPaths*, where *addingFilePaths* is a comma-separated list of folders and files that should be appended to the existing archive stored under the path *harPath*.

3.2 Appending New Data

The process of adding new files to an existing HAR consists of a data and index update step. First, the data of the new files are appended to one of the multiple part files, which contain the data of the already archived files. Afterwards, the indexing files are updated and rewritten with new entries in order to allow access to the new files within the archive.

3.2.1 Part File Data Update. First, the binary data of the new files that should be added to an existing archive is appended to one of its archives $part_n$ files. A $part_n$ file, like every file in HDFS, is stored in series of data blocks db_n with a fixed data block size, which are all replicated with a factor rep_n across all datanodes. Thus, formally a part file can be defined as $part_n : \{db_{n_1}, db_{n_2}, ..., db_{n_m}\} * rep_i$. In addition, there is no free space between data blocks and thus, only the last block db_{n_m} may not fully utilized. Because HDFS is appendable only file system, new data of the new archiving files is always appended at the end of the data blocks chain of one of the $part_n$ file. The block size of a $part_n$ file is set to 512 MB, instead of HDFS' default block size of 128 MB. By choosing a larger block size, it is possible to store more files in an allocated data block and thus, reduce memory footprint at the NameNode.

In AHAR, data blocks of the $part_n$ files are appended with the binary data of new files until the available space of the last $part_n$ data block db_{n_m} is fully utilized. For example, having a $part_1$ file with a block size of 512 MB and current block length of 128 MB, a new file with a size of 128 MB would fit into this part file. For deciding at which $part_n$ file the adding files should be appended, a first fit algorithm is used. For each adding file $addFile_i$, the algorithm finds the lowest numbered $part_n$ file, in which a $addFile_i$ file fits first, or if no $part_n$ fits, a new $part_{n+1}$ is created. If an $addFile_i$ file fits into a specific $part_n$ file is determined by $freeBytes_n$, the remaining bytes left until the last data block db_{n_m} of a $part_n$ file is fully utilized. With this, we avoid that all data is appended to just one very large $part_n$ file with many data blocks and also fill up other already existing part files. Formally, if an $addF_i$ fits into a $part_n$ is determined by $addFileBlockSize_i \leq freeBytes_n$, where $freeBytes_n :=$ $partBlockSize_n$ - ($partLength_n$ mod $partBlockSize_n$).

For example, as shown in Figure 3, the current block length (i.e. $partLength$) of an existing $part_0$ file is 996 MB and of $part_1$ file is 448 MB. Thus, $part_0$ consists of 2 blocks and $part_1$ of 1 block. The remaining space left until all existing blocks of $part_0$ would be 28 MB and of $part_1$ would be 64 MB. Assuming, the size of the file to add is 40MB, it would not fit into $part_0$, but in $part_1$, because its size is lower or equal then the left free space in the last data block of $part_1$.

$aFileLen \leq leftSpaceInBlock$.

Instead of appending the data of a file immediately to its located $part_n$ file, AHAR first determines all $aFile_i$ to $part_n$ pairs. Thus, we can append the data of multiple files to one specific $part_n$ in a batch. By appending every single new file to one $part_n$, the data of the existing $part_n$ file would be rewritten and replicated multiple times on the cluster, which causes a lot of unnecessary network traffic and thus, a higher execution time.

3.2.2 Index Update. After the new binary data was appended to the existing archive, the index and masterindex files of the HAR archive must be updated and rewritten. The index file stores the directory structure of all files and folders within that archive. An index entry can either be a directory or file entry. A directory index entry stores the mapping between directories and subdirectories and files within that directory. For example, an entry */input dir sub0*

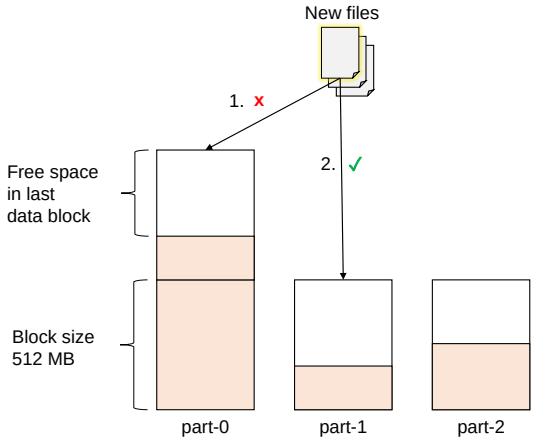

Figure 3: Finding a $part_n$ file for storing the binary data of a new file is done using a first fit algorithm.

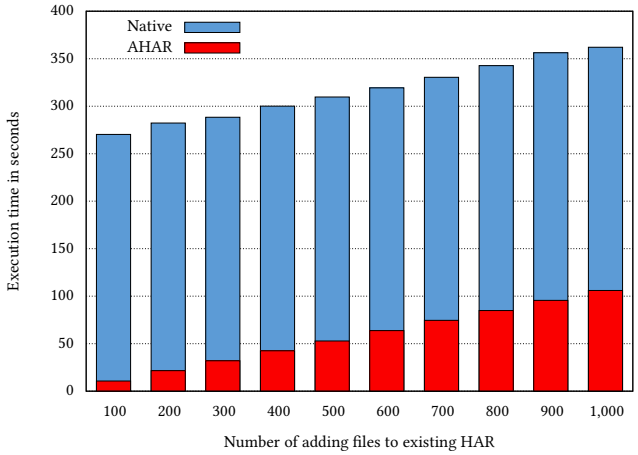

Figure 4: Total execution time for adding a varying number of new files between 100 and 1,000 to an existing HAR consisting of 10,000 files, each file was 10 MB.

sub1 [...], specifies that under the path "/input", two sub folders "/input/sub0" and "/input/sub1" exist. A file index entry maps a filename to a specific $part_n$ file storing its binary data including its offset and length to exactly locate that data within that part file. For instance, the entry */input/sub0/fileA.txt file part-0 0 128 [...]* specifies that "fileA" is stored in $part_0$, starting at offset 0 with a length of 129 bytes. The index file is sorted by the file or directory path hash. Since the index can be many hundred thousand lines long, depending on the number of files stored in an archive, a preliminary *masterindex* is kept, holding information about which part of the index is mapped to which portion of the hash space. Both files, the _index and _masterindex needs to be updated after appending a new file to an archive.

Before adding new index file entries or updating index directory entries, the existing index file is parsed and kept into a red-black tree [2] using a java-based TreeMap implementation. The map is, similar to the index file, sorted by the path hashes. We choose a red-black tree, because it allows to lookup entries quickly with guarantee of $O(log_n)$. A fast search is needed, because for every new file that is supposed to be added, it must be checked whether a file is already contained to prevent duplicates as well as updating directory entries including their sub files and directories. For example, a file $/foo/bar/file.txt$ that is added, effects three folder entries /, $foo/$ and $bar/$ are created, each containing a reference to their respective subdirectories. Therefore new directory entries following the given pattern have to be created or already existing entries need to be updated.

After all $part_n$files are written to HDFS both index and masterindex are discarded and rewritten in much the same way that HAR archiving method is writing them initially. For every 1000 lines of index entries one line in the master index is written, denoting the bytewise location of above lines in the index and the range they cover in the underlying hash space. By this, we can ensure that new files added with AHAR can be used like any other files in the archive.

4 EVALUATION

This section presents our experimental setup, the test workload, and benchmark results.

4.1 Experimental Setup

All experiments were done using a 40 node cluster. Each node is equipped with a quad-core Intel Xeon CPU E3-1230 V2 3.30GHz, 16 GB RAM, and three 1 TB disks with 7200RPM organized in a RAID-0. All nodes are connected through a single switch with a one Gigabit Ethernet connection. Each node runs Linux (kernel version 3.10.0), Java 1.8.0, and Hadoop 2.7.1 with default configuration.

4.2 Results for Adding Multiple Files to an Existing HAR

This section reports results of adding multiple files to an existing HAR archive. We compare the execution time of AHAR append functionality versus a native approach of unarchiving and re-archiving all files including the new files. Currently, to the best of our knowledge, this is the only possible method to add new files to an existing HAR archive. For all experiments, we report the median execution time of seven runs. In addition, for every run a new set of files was created and put into HDFS, to force a new data block placement distribution.

In the experiment as shown in Figure 4, we add a varying number of files between 100 and 1,000 to an existing HAR archive that already contains 1,000 files. For every run, we increased the number of new files to be added by 100. In addition, a fixed size of 10 MB per file was chosen. Thus, the size of the archive was 1 GB before the new files were added. In another experiment, we increased the existing HAR archive to 10,000 files, and thus the total size was 10 GB. Again, for every run we increased the number of new files to add by 100. The results for both experiment are shown in Table 1.

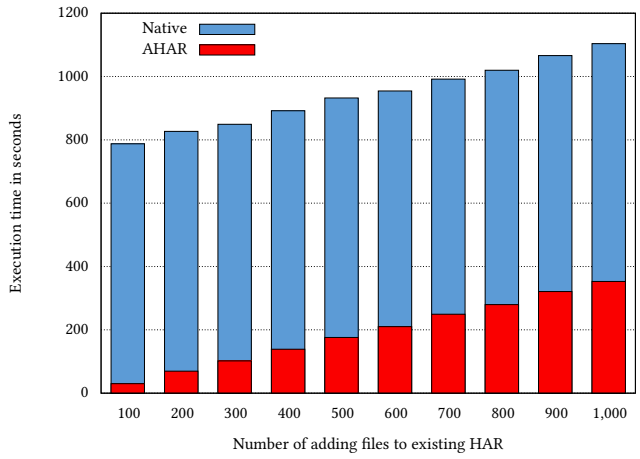

Figure 5: Total execution time for adding a varying number of new files between 100 and 1,000 to an existing HAR consisting of 10,000 files, each file has a different size between 1 and 128 MB.

In the experiment as shown in Figure 5, we changed from a fixed to a varying file size. The size of every file was choosen by $f_i := 1 + (i \mod 128)$ MB, where i is the number of files. As a result, we have a series of files, where f_0 is 1MB, f_1 is 2MB, f_2 is 3MB, and f_{128} starts again with 1 MB. Similar to the previous experiments, we added a varying number of files between 100 and 1,000 to an existing HAR archive that already consists of 1,000 files.

Table 1 summarizes the results of our benchmarks.

Adding between 100 and 1000 files to an archive with 1,000 files using AHAR, we decrease execution time varying from 96% to 71%. By increasing the existing files in a HAR file to 10,000, we decrease execution time varying from 99 to 95%. In all experiments, we observe as more files we add with AHAR, less speed-up is gained. One reason for that is because the node running the tool acts as a proxy node, which performs the part file data update process described in Section 3.2 locally. The node loads the data from the $part_n$ file, appends the new data and uploads it back to HDFS. Thus, the network of the node can quickly become a limitation, especially when adding large size of data. On the contrary, unarchiving and re-archiving a file in HDFS is implemented as a MapReduce job that executes the job on multiple nodes in parallel.

Another finding is that, the execution time approximately stays the same for adding the same number of files with the same size, e.g. 100 files each with 10 MB, independently of the size of the existing HAR. The reason for that is that the same number of $part_n$ files are appended and the index update process only takes a few seconds for an archive with 1,000 or 10,000 entries. For instance, adding 500 files with the same size to an archive with 10,000 files only takes approximately 5 seconds more than adding the files to the 1,000 files sized archive. Thus, AHAR is a great tool for continuously adding small batches of files to an existing archive.

Table 1: Benchmark overview of appending new files with AHAR.

Existing	Adding	Size	Exec. Time	Reduction
1,000	100	10 MB	10.7 sec	96 %
1,000	500	10 MB	52.83 sec	83 %
1,000	1,000	10 MB	106.02 sec	71 %
10,000	100	10 MB	12.04 sec	99 %
10,000	500	10 MB	56.45 sec	97 %
10,000	1,000	10 MB	113.59 sec	95 %
1,000	100	VAR	30.19 sec	96 %
1,000	500	VAR	176.09 sec	81 %
1,000	1,000	VAR	352.70 sec	68 %

5 RELATED WORK

There are two categories of improving scalability focusing on allowing to store more files on HDFS: NameNode federation and merging small files to larger. Federated NameNodes allow to have multiple NameNodes, each storing a subset of metadata [9]. Instead of changing the number of files directly stored in HDFS, a federation allows to store more files by increasing the available memory for metadata management. The technique of merging multiple files to a larger reduces the memory footprint at the NameNode. However, an additional indexing mechanism outside of the NameNode is needed.

AHAR relates to the merging approach, whereby Hadoop Archive and Sequence File are file formats that are officially supported by Hadoop. Sequence File is a merged file consisting of binary key-value pairs that store the filename as key and the file contents as value. In comparison to HAR, the retrieval of a list of file names within a Sequence File requires processing the entire file. One drawback of both approaches is that an increased access time, due to the additional indexing step. Thus, authors introduce specific solutions to their application scenarios using new index strategies [4, 5, 7, 8, 13]. Most of this work focus on increasing the performance of small file access through new indexing strategies or caching methods and treat an merged file as immutable. However, for an automated storage, time is not that critical, yet we need an archiving file format that allows to append data to an existing archive. We assume that archives are stored on archival disks and over time we want to append new data to existing archives.

Similar to our approach, New Hadoop Archive (NHAR) [11] extends HAR. In addition, they introduce a mechanism to overcome HAR immutability. The authors redesign the indexing mechanism. Instead of a two-way indexing, a single index mechanism without the masterindex file is used combined with a global hashing component. The approach consists of multiple fixed indexing files per archive. When a new file is appended to an archive, the hash of the files is calculated and with a modulo function assigned and merged to one of the index files. For appending new files to an archive, an temporary HAR archive is created. This temporary archive has a unique names for its part files, so that both archives including their part files can be merged. In addition, the entries of fixed indexing files are merged. A limitation is a fixed number of index files, which

can be quite low or large. In addition, when merging multiple times, you can have a lot of underutilized $part_n$ files.

6 CONCLUSION

In this paper, we proposed a tool to append files to an existing HAR archive. The tool effectively adapts the metadata and data files of an existing archive with new files without rewriting the whole archive. Thus, files can be managed more effectively for archiving purposes, reducing the memory footprint on the NameNode to increase scalability and to face the small file problem. The presented approach is based on Java and can be simply used via its command line interface. The presented experiments show good speed-ups when adding new files of various sizes to existing archives.

In the future, we want to use AHAR to implement an automated archive storage, which automatically stores and appends cold data to existing archives for long term storage with a low memory footprint at the NameNode. This allows us to store more files on HDFS. In addition, we want to implement a remove function to delete files from a HAR archive. We are also planning to to implement the approach in a more distributed fashion, e.g. as a MapReduce or Spark job, to add the binary data of new files in parallel to an existing archive data. This may increase execution time when adding large amounts of files at once.

7 ACKNOWLEDGMENTS

This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere and by the German Ministry for Education and Research (BMBF) as Berlin Big Data Center BBDC (funding mark 01IS14013A).

REFERENCES

[1] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Data Engineering* (2015), 28.

[2] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. 2001. *Introduction to algorithms*. Vol. 6. MIT press Cambridge.

[3] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI'04)*. USENIX Association, 10–10.

[4] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, and Y. Li. 2010. 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*. 65–72.

[5] Liu Jiang, Bing Li, and Meina Song. 2010. THE optimization of HDFS based on small files. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. 912–915.

[6] KR Krish, Ali Anwar, and Ali R Butt. 2014. hatS: A Heterogeneity-Aware Tiered Storage for Hadoop. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 502–511.

[7] X. Liu, J. Han, Y. Zhong, C. Han, and X. He. 2009. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *2009 IEEE International Conference on Cluster Computing and Workshops*. 1–8.

[8] G. Mackey, S. Sehrish, and J. Wang. 2009. Improving metadata management for small files in HDFS. In *2009 IEEE International Conference on Cluster Computing and Workshops*. 1–4.

[9] S. Radia and S. Srinivas. 2010. Scaling HDFS Cluster Using Namenode Federation, HDFS-1052. (2010).

[10] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. 1–10.

[11] C. Vorapongkitipun and N. Nupairoj. 2014. Improving performance of small-file accessing in Hadoop. In *Computer Science and Software Engineering (JCSSE), 2014 11th International Joint Conference on*. 200–205.

[12] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. 2010. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 1–9.

[13] Cairong Yan, Tie Li, Yongfeng Huang, and Yanglan Gan. 2014. Hmfs: efficient support of small files processing over HDFS. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 54–67.

[14] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 423–438.