Implementing WebGIS on Hadoop: A Case Study of Improving Small File I/O Performance on HDFS

Xuhui Liu[1,2], Jizhong Han[1], Yunqin Zhong[1,2], Chengde Han[1]
1. Institute of Computing Technology
2. Graduate University
Chinese Academy of Sciences
Beijing, China
{mafish, hjz, zhongyunqin, han}@ict.ac.cn

Xubin He
Electrical and Computer Engineering Department
Tennessee Technological University
Cookeville, TN 38505, U.S.A
hexb@tntech.edu

***Abstract*—Hadoop framework has been widely used in various clusters to build large scale, high performance systems. However, Hadoop distributed file system (HDFS) is designed to manage large files and suffers performance penalty while managing a large amount of small files. As a consequence, many web applications, like WebGIS, may not take benefits from Hadoop. In this paper, we propose an approach to optimize I/O performance of small files on HDFS. The basic idea is to combine small files into large ones to reduce the file number and build index for each file. Furthermore, some novel features such as grouping neighboring files and reserving several latest version of data are considered to meet the characteristics of WebGIS access patterns. Preliminary experiment results show that our approach achieves better performance.**

***Keywords*-Hadoop; HDFS; WebGIS; Small File I/O Performance**

I. Introduction

With the rapid growth of Internet services, many large server infrastructures have been set up as data centers and cloud computing platforms, such as those in Google, Amazon and Yahoo!. Compared with traditional large scale systems built for HPC (High Performance Computing), they focus on providing and publishing specialized services on Internet which are sensitive to application workloads and user behaviors. Moreover, they provide both scalability, manageability, reliability for large scale data management and an efficient MapReduce [1] programming model for data-intensive computing.

The key components of Internet server infrastructures are distributed file systems. Three famous examples are Google file system [2] (GoogleFS), Hadoop [3] distributed file system (HDFS) and Amazon Simple Storage Service (S3) [4]. Among them, HDFS is an open-source implementation influenced by GoogleFS, so more details can be discovered. HDFS provides optimized methods to handle huge amount of data. Computation are deployed on the nodes storing input data to minimize transfer overhead. Moreover, data replication can guarantee high availability during software and hardware failures. Prefetching policies can also pipeline the data transfer in small units. However, varieties of workloads of Internet applications cannot be satisfied by one layer. Therefore, middlewares, such as HBase [5], Hive [6], etc., are built on HDFS for specialized requirements on different levels of Internet services.

WebGIS [7] is one of the popular Internet services born with rapid development of web technology and Geographic Information System (GIS). GIS is built for acquisition, storage, management and release of spatial data and attributes [8], and WebGIS is provided for users to access, manage and share global geographic information concurrently no matter where they are from or what platforms they are using. Early efficiently running on PC, modern WebGIS has become computing-intensive and data-intensive with the expansion of spatial data sets and concurrent users. In traditional WebGIS, single-node system could meet the requirements for application when spatial data sets are small, but it has obvious limitations in performance, scalability and reliability while processing large data sets.

In this paper, we use Hadoop to support WebGIS applications. Hadoop is a decent candidate platform for building WebGIS on clusters for its superior storage and computing capacity. However, HDFS is designed to manage large files. Mismatch of accessing patterns will emerge if HDFS is used to manage data of WebGIS directly. In WebGIS system, to reduce the total amount of data transferred between browser and server, data are often divided into small tiles with size of tens of KB around. Therefore, the total spatial data sets has large scale with huge number of files while sizes of single file are very small.

In this paper, we propose an approach to optimize I/O performance of small files on HDFS, which is used to build a middleware at application level. The basic idea of this approach is to combine small files into large ones to reduce the file number and build index for each file. The basic file operations such as read, write, update and delete are supported. Taking the accessing patterns of WebGIS into consideration, some novel features will be added such as grouping neighboring files and reserving several latest versions of data since spatial data in WebGIS will not be out of date. Experiment results show that our design can achieve remarkable improvements on I/O performance. Execution time of reading and storing files are improved significantly while memory usage rate decreases several times. Furthermore, we also present the design and implementation of a prototype WebGIS system, HDWebGIS, and show the performance improvement on a real application based on our small-file-tuning methods.

It also should be noticed that those distributed file systems for Internet services are not tightly-coupled with upper middlewares and applications. Traditional distributed file systems for HPC, such as PVFS [9], pNFS [10], GlusterFS [11], can also support storage of vast mount of data and MapReduce computing framework [1]. Besides HDFS, our approach can also be applied to other distributed file systems.

The rest of this paper is organized as following: section II discusses background on distributed file systems; section III describes the design details; performance evaluations are conducted in section IV; section V introduces related work and conclusion is drawn in section VI.

II. Background

There are two types of file systems that are used in data-intensive applications, parallel file system [12] and distributed file system.

A parallel file system is designed for HPC applications which run on large clusters and are in need of highly scalable and concurrent storage I/O. Examples of parallel file systems include IBM's GPFS [13], Sun's LustreFS [14], and the open source Parallel Virtual file system (PVFS) [9].

Distributed file system is widely used in Internet services, and the leading examples include the Google file system, Amazon Simple Storage Service and the open-source Hadoop distribute file system.

In recent years, Google has released series of spatial information services, such as Google Earth, Google Map, Google Sky ,etc. These services are built on Google infrastructure which consists of two main components: Google File System (GFS) [2] and Map/Reduce framework [1]. Map/Reduce is beyond discussion of this paper and GFS is briefly discussed below.

Google File System (GFS), aims to manage large scale data sets in clusters. Applications on GFS are usually to access large files. The GFS typically consists of a single master, multiple chunk servers and several clients. Large files that exceed a threshold are partitioned into fixed-size chunks (64 MB by default), each chunk is replicated on multiple chunk servers for reliability. The master manages all metadata of files which include not only the basic information of files, such as file size, creating time, accessing permissions, etc, but also some features for distribution, for example, maps of file name to chunks, locations of each chunks etc. The master stores information of maps of file name to chunks in local disk, but does not store chunks' locations permanently. Locations of chunks will be reported to the master by chunk servers through heartbeat messages periodically at run time. Chunk servers are in charge of chunks and reported chunks' information to master. As a consequence, the master can obtain the latest chunks' location information at any time. Data in GFS are accessed through clients. At first clients get file metadata from the master and then communicates with chunk servers to read or write data.

Hadoop platform is inspired from GFS and MapReduce. It provides GFS-like distributed file system called HDFS and MapReduce framework. Working mechanism of HDFS is similar to that of GFS but it's light-weighted. In GFS, three components are client, master and chunk server, while in HDFS they are client, name node and data node. HDFS does not support concurrent writes, only one writer is allowed at time. There is no optimization for small files, and the data node buffers data until the amount of data reaches the chunk size (64MB).

III. Small File Tuning Approach for HDFS

A. Motivation

HDFS is originally designed to manage large files, and not optimized for small files. Therefore, it suffers performance penalty when managing a large amount of small files. For example, when 550,000 small files which sizes range from 1KB to 10KB were stored into HDFS, following phenomena were observed:

- Unacceptable execution time. About 7.7 hours are spent in storing these small files into HDFS while in a local file system, such as ext3, the time is about 660 seconds.
- High memory Usage rate. Almost 63.53% are occupied during storing operations in an idle system.

In HDFS system, each file has its metadata and is stored with several replicas, for instance, 3 copies by default. Metadata management in HDFS is a time-consuming task because it need cooperation between at least three nodes. For small file I/O, most time is spent in managing metadata and little time is spent on data-transferring. The large amount of small files increases the overhead of metadata operations in HDFS. This is the reason why HDFS spent extremely long time in storing these files. On the other hand, metadata are kept in name node and chunks' information are kept in data nodes. Furthermore, all of these information are loaded into physical memory. As a consequence, memory usage rate increases rapidly with the number of small files increasing sharply. In order to satisfy the application requirements of small files IO performance, an optimized approach should be adopted to build a middleware on HDFS at the application level, and the file access patterns of special applications should be considered.

B. File Access Patterns in WebGIS

Before further discussions, file access patterns in WebGIS are concluded as following:

- Some attributes of files' metadata are same, such as file owner, access permissions and so on. Other attributes such as created date and modified date may be different, but these differences can be ignored because they are not concerned by WebGIS applications. These attributes can be abstracted during optimization.
- WebGIS provides a map-browsing interface for clients. When an image file is accessed, its geographic proximal images have high probability to be accessed later. This feature is taken into consideration while designing our

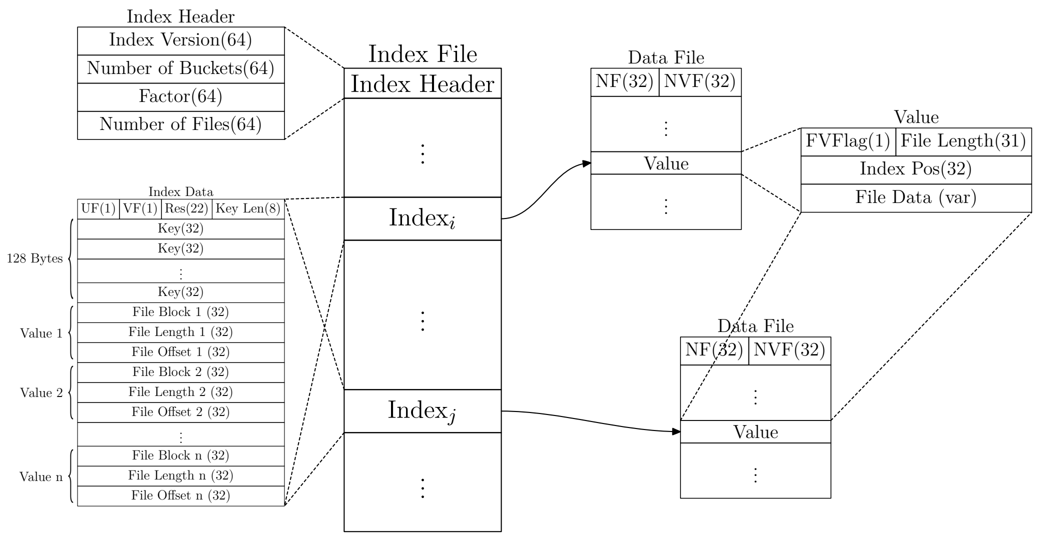

Fig. 1. Description of Our Approach

prototype WebGIS system. All image files are grouped by their geographic locations and files in a group will be stored consecutively in memory.

- Data in GIS will not run out of date forever. The history versions of data of certain region can record its unique development history. Therefore, several latest versions of data need to be reserved during updates.

C. Principle of Our Approach

The basic idea of our approach is to merge small files into big ones and to build index for each small file where hash index is used. Moreover, disk swapping of index is supported. During the process of designing this approach, access patterns of WebGIS applications are well considered and novel features are added to meet these patterns. First, files are grouped and stored consecutively in physical memory in terms of their geographic locations. Second, the history spatial data in WebGIS are stored in several versions to keep their values. Several latest versions of data are reserved during updating. Here, we treat files smaller than 16MB as *small files*, and those files which exceed 16MB are beyond the scope of management of this approach. The structure is shown in figure 1.

The design is based on two types of files: index file and data file. Index information is fixed-length, which is formatted into fixed-length blocks, called index blocks. The length of file data are variable, so blocks in data files are variable-length.

D. File Grouping

We make all files organized as small tiles of 256×256 pixels, while location information of files represented by directories and file name. A tile's position is decided by three variables: $(L, (x, y))$. Among them, L is scale level, valued from 0 to 17, and x and y are the coordinates of X axis and Y axis respectively.

Grouping schema combines $n \times n$ files which are adjacent in locations to one group. n can be chosen according to specific application, but must be power of 2. The file at the position of up-left of a file group is defined as *first file*. Given a tile of $(L, (x, y))$, the *first file* can be computed as:

$$\begin{aligned} x' &= \lfloor \tfrac{x}{n} \rfloor \times n \\ y' &= \lfloor \tfrac{y}{n} \rfloor \times n \end{aligned} \tag{1}$$

After decision of the *first file* of a group, other files can be computed easily.

E. Index File

An index file contains two parts: index header and index data block.

Index Version(64)	Number of Buckets(64)	Factor(64)	Number of Files(64)

Fig. 2. Index Header

An index header is shown in figure 2. The maximum number of indexed files can be decided by value *Number of Buckets* $\times$ *Factor*. If a file number exceeds this value, the *Number of Buckets* field should be increased and all the hash values in system have to be recalculated. This is a consuming operation which could be avoided by choosing a suitable value that is big enough for *Number of Buckets* field from start in practical.

UFlag(1)	VFlag(1)	Reserve(22)	Key Length(8)
Key(64)			
⋮			
Key(64)			
File Block 1 (32)	File Length 1 (32)	File Offset 1 (32)	
⋮			
File Block n (32)	File Length n (32)	File Offset n (32)	

Fig. 3. Index block

An index data block is shown in figure 3. *UFlag*(Used Flag) is a tag to identify whether this block has been used. *VFlag* (Valid Flag) tag indicates the validity of this block, which is always zero when index data is deleted. *Key Length* field shows the valid length of key of this index block while actual key is stored in *key* field with length limitation of 128 Bytes. *File Block, File Length, File Offset* consists the value of this index block and represents the serial number of data file in which this small file is stored, length of small file, offset in data file, respectively. Serial number of data file is a positive integer value and starts from 1. If *File Block* value is zero, it is invalid. An index data block may contain more than one value to store multiple latest versions sorted by their create time in the version array.

F. Data Files

A data file also consists of two parts: data file header and data file block.

Number of Files(32)	Number of Valid Files(32)

Fig. 4. Data header

A data file header is shown in figure 4. *Number of Files* field records the total number in this data file and *Number of Valid Files* represents the valid file in this data file. When files being added to data file, both of these two fields increase. While deleting files, *Number of Valid Files* field decreases and *Number of Files* field is unchanged.

FVFlag(1)	File Length(31)	Index Position(64)	File Data(var)

Fig. 5. Data Block

A data file block is shown in figure 5. *FVFlag* (File Valid Flag) tag indicates the validity of this block. *File Length* field records file length. *Index Position* field records the index position of this file in index file. The actual data are stored in *File Data* field.

Here, length of Data files is limited to 64MB, which is chunk size in HDFS.

G. File Operations

In this section, basic file operations will be discussed, including storing, reading, deleting and updating.

1) File Storing: File storing is a process of merging small files to large ones. During this process, two files should be opened in memory, a data file for storing file data and an index file for storing file index. The details are described as following steps:

STEP 1: Collecting files. For those files to be stored, value in term of (L, (x,y)) is computed from file name. Then the value of *first file* in group is calculated according to formula 1, and then all values of files in group can be achieved. For every file in group, goto step 2.

STEP 2: Creating data block. Taking file name as key, hash value is calculated. Look up index file for block to place this hash value. And then, judge if this file exceed 16MB. If it does, set *File Length* field in index block -1 to indicate this file is managed by file system directly. And if not, data block is created, in which *FVFlag* tag is set valid.

STEP 3: Appending data block. Length of the sum of current data file and small file is compared to 64MB. If it exceeds 64MB, the current data file is closed and is replaced by a new one. Then, the data block in step 2 is appended to the end of data file. After that, *Number of Files* and *Number of Valid Files* fields are increased.

STEP 4: Updating index. Index block is generated. Before index block written to index file, index file should be compared to see whether it is full or not. If yes, re-hash operation need to be conducted: all indexes in index file have to be recalculated and all index information in data file must be modified. And then, index block should be written into index file.

STEP 5: Looping from step 1 until all files are stored.

2) Read Request: Because several latest versions of files are reserved, read request consists of not only the requested file name, but also a value n to indicate that which version is concerned by this request. The value of n is 1 by default. Read process is described as following steps:

STEP 1: Accepting read request, which contains a file name with full path and a version number n.

STEP 2: Calculating and examining the hash values in the index file. If an index value is found, go to next step. Otherwise, return with error since the requested file does not exist.

STEP 3: Collecting the number of valid versions m and comparing it with n. If $m >= n$, go to next step. Otherwise, return with error since the requested version number exceeds the total version number.

STEP 4: Fetching index value of the n-th version, so data can be read from data file according to index value.

3) File Deleting and Updating: Update-out-of-place mode is used while updating files. That is, when a file being deleted from system, it cannot be removed immediately instead of being marked as deleted. It remains until certain condition being satisfied. The *certain condition* refers to that value of *Number of Valid Files* field is less than half value of *Number of Files* field in data file. If this condition is satisfied, space reclamation operation, which re-allocate valid files in one data file to current data file, is performed. The deleting steps are described as following steps:

STEP 1: Looking up index by name. Given a file name and version number n to be deleted, the file name is looked up in the index file. If the file does not exist in index, return error. Otherwise, go to step 2.

STEP 2: Getting versions by n. Index information of a file is read from the index file. And m, number of valid versions, is retrieved. If $m < n$, that means the version to be deleted is invalid, return error. Otherwise, go to step 3.

STEP 3: Modifying data file. A data file block in a data file is positioned according to the index information. *FVFlag* tag in data file block is set invalid and *Number of Valid File* field in data file header is decreased by 1.

STEP 4: Modifying index file. If $m = 1$, that means the version to be deleted is unique and current index block is useless after this version being removed. Therefore, *VFlag* tag in index block is set to zero. Otherwise, if $m \geq 1$, the n-th version is removed from version array and following versions are brought forward in version array.

STEP 5: Performing space reclamation. Data file should be shrinked if *Number of Valid Files* field is less than half of *Number of Files* field. After scanning, we can find valid data files and perform space reclamation operations, which includes:

1) If the total size of current data file and the valid file is greater than 64MB, current data file is closed and replaced by a newly created one.
2) Data file block of valid file is appended to the end of current data file.
3) The index information in the index file is updated accordingly.

In this approach, small files are merged into large ones. Once a small file is merged into data file, system overhead of update-in-place is unacceptable. As a consequence, out-of-place method is selected as updating policy.

Different from deleting, the updating request do not need to specify version number n. Updated data will be treated as latest version automatically. The updating operation is a combination of storing and deleting operation which is described as following:

STEP 1: A request to update a certain file is sent with a file name. All files in the same group are determined. For all files in the group, go to step 2.

STEP 2: If the file to be updated does not exist in the system, it will be stored by a regular storing operation. Otherwise, go to step 3.

STEP 3: The new data is appended to the end of current data file.

STEP 4: The old file is marked as deleted.

STEP 5: Index information of this file is updated.

STEP 6: The old data file is checked and garbage collecting operation is performed if necessary.

IV. Performance Evaluation

A. Experiment Environment

Our test platform is built on a cluster with five nodes of Dell Power Edge SC430. Each node has an Intel Pentium 4 CPU of 2.8GHz,1GB or 2GB memory, 80GB or 160GB SATA disk. The operating system is Red Hat AS4.4 with kernel 2.6.20. Hadoop version is 0.16.1 and java version is 1.6.0.

In these five nodes, one node acts as master and others are data nodes. The number of replications is set to 2 during the tests.

Testing items include two categories: file operations and system evaluation. File operations include read and write. Update and delete operations are not tested in this paper because they are not as common as read/write in practical WebGIS systems.

Original HDFS	Tuned HDFS
22,719	431

TABLE I
Comparison of storing time(seconds)

B. Data Set

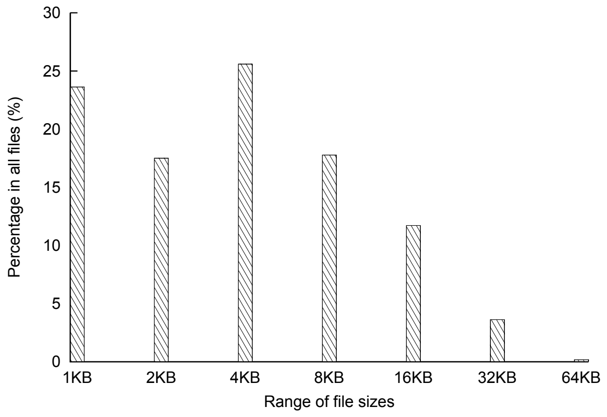

Fig. 6. Distribution of small files' sizes

The data set consists of 558,726 files and the total size of these files is 3.6GB. File sizes range from several KB to around one hundred KB. Figure 6 shows the distribution of file sizes and files with size smaller than 16KB account for 96.19% of total files.

C. Write Operations

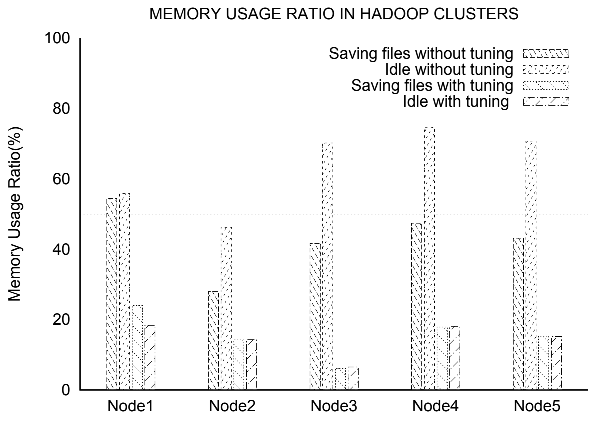

Fig. 7. Memory usage ratio in cluster

Write operations are performed on both original HDFS and tuned HDFS. During these periods, memory usage ratio is monitored for all nodes. Table I shows the total time for storing these files. It costs 27,719 seconds for original HDFS and 431 seconds for tuned HDFS. Figure 7 shows the memory usage ratio of each node during and after the storing period. For original HDFS, average memory usage ratio is 42.92% during storing period and is 63.53% when system is idle. After tuning, values are reduced by 15.51% and 12.05% respectively.

D. Read Operations

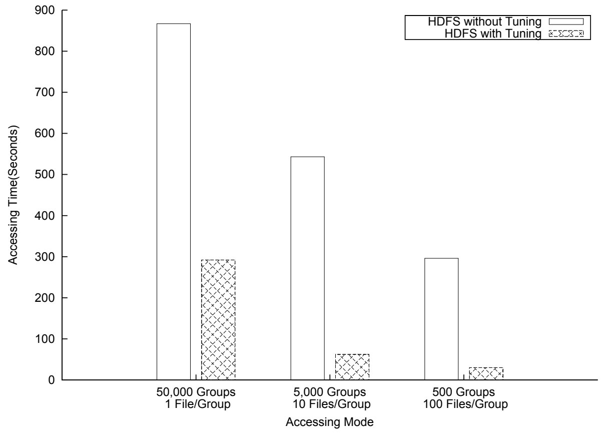

Fig. 8. Comparison of reading time

Three file access lists are generated in order to simulate different access modes.

- 50,000 distinct random files.
- 5,000 groups of random files, and each group contains 10 sequence files.
- 500 groups of random files, and each group contains 100 sequence files.

Test result is shown in figure 8 . Access times of three lists are 867 seconds, 543 seconds, 296 seconds for original HDFS. While after tuning, the times are 292 seconds, 62 seconds, 30 seconds, respectively.

E. Prototype System Verification

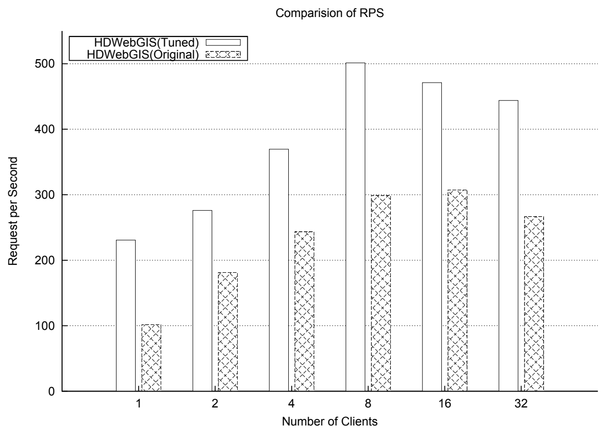

Fig. 9. Comparison of RPS

To further verify our approach, we designed and implemented a prototype WebGIS system, HDWebGIS, or Hadoop Distributed WebGIS. A small file tuning approach is developed and implemented in this system. In this section, effectiveness of this approach will be evaluated from system's perspective.

TABLE II
COMPARISON OF TIME (SECONDS) AND RPS TO ACCESS 10 CITIES

Number of Clients	Original HDFS		Tuned HDFS	
	Time	RPS	Time	RPS
1	34.43	101.66	14.91	230.72
2	38.63	181.21	24.92	276.08
4	57.52	243.39	37.24	369.50
8	93.73	298.73	54.90	501.28
16	182.28	307.22	116.81	471.19
32	420.09	266.61	247.99	443.89
64	N/A	N/A	488.08	451.07
Average		233.14		382.11

HDWebGIS provides a map-browsing interface to users. To test system's performance, a benchmark program is developed. This program simply sends URL requests to the server and records the response time. A benchmark maintains a city list, in which 100 cities are recorded. During the test, program can simulate the behaviors of real users. It first chooses one or more cities to access, sends the requests to the server and then zooms in to up to 17 levels. At last, the map is dragged around to access neighboring images. A city access need retrieve about 350 image files. The benchmark program uses multi-threads to simulate concurrent users.

During the testing, the benchmark simulates 1, 2, 4, 8, 16, 32, 64 clients to send requests. Table II shows the access time and throughput in terms of requests per second (RPS) of different systems. It should be noticed that one of the tests failed when sending requests to original HDFS with 64 clients after several retries, so the corresponding result is absent. Comparison of throughput is shown in figure 9. The average throughput of original HDFS is 233.14 requests per second, comparing to 382.11 after tuning using our approach.

V. RELATED WORK

Metadata access is a key factor to influence I/O performance in file system, and hash index [15] is widely used for indexing metadata. I/O pre-fetching is applied to reduce frequency of disk access. Moreover, grouping-schema is adopted to improve the hit rate of pre-fetching. Files accessed sequentially are grouped; metadata and data of files in the same group are stored consecutively on disks [16], [17], [18].

Two different modes are used to update file data: update-in-place and update-out-of-place. In update-in-place mode, a file is updated by overwriting its old version. This method could utilize disk space efficiently, but the read performance may become lower because file may not be stored sequentially on disk if it's updated several times. This method was applied in file systems such as FFS [19] and ReiserFS [20]. In update-out-of-place mode, a file is updated by appending new data to new blocks instead of overwriting old blocks. This mode has several advantages: eliminating the time of searching for original file blocks, storing files consecutively on disk, and keeping history file data easy to access. However, disk space utilization rate may become lower and the dynamic space reclamation policy should be executed periodically. File system LFS[21], [22] uses this mode. Some file systems such as HFS [23] use a hybrid mode: update-in-place mode for large files and update-out-of-place mode for small files. According to characteristics of WebGIS application, update-out-of-place was adopted in HDWebGIS because it accesses large amount of small files and data of old files also have value. When a file is updated by a user or an application, its history data won't be immediately removed from HDFS. Instead, the latest three versions of history data will be kept.

In addition, a lot research focus on reducing the latency of small-file access in distributed or parallel file systems. Shaikh proposed three optimizations for small file IO access patterns on PVFS2, which are *small file packing*, *eager IO* and *delayed data handle creation for files* [24]. Carns et al [25] proposed five optimizations for PVFS2: *Precreating Objects, Stuffing in Parallel File Systems, Coalescing Metadata Commits, Eager I/O* and *POSIX Extensions for Directory Access*. To "support efficient access to large numbers of small files while still preserving good performance for large files and avoiding any change in user behavior?", Thain and Moretti [26] proposed a new protocol called "Chirp", which is a combination of streaming and RPC. Hendricks et al [27] proposed architectural refinements, server-driven metadata pre-fetching and namespace flattening, for improving the efficiency of small file workloads in object-based storage systems. Kuhn et al [28] designed a schema without metadata for a special accessing pattern that many small files are deleted shortly after being created to eliminate the cost of metadata operation.

Compared to above research, optimizations proposed in this paper differ in two aspects:

- The level of optimization. Optimization proposed in this paper is above the file system while most related researches were at the disk level.
- Grouping schema presented in this paper is specific to the WebGIS application.

VI. Conclusion

HDFS originally was designed for storing large files. When it is used to support web applications, small file I/O performance becomes the bottleneck. We present a small file tuning approach on HDFS and apply it to the WebGIS applications. The experiment results show that our approach can achieve remarkable improvements on small I/O performance. Execution time is reduced from 27,719 to 431 seconds to write 550,000 files into HDFS, and from 867 to 292 seconds to read 50,000 files from HDFS. At the same time, memory usage rate decreases from 55.78% to 18.36%.

While HDFS is designed to support next-generation applications on cloud computing platform, currently vast majority of web applications are based on small scale clusters. Using WebGIS as our case study and adopting real data sets, we demonstrate the feasibility and efficiency of using tuning method for HDFS on small scale clusters to support popular web applications which are sensitive to small file I/O performance. Therefore, we design and implement a prototype WebGIS system, HDWebGIS,to integrate our small file tuning method and show the performance improvement on a real WebGIS application.

Acknowledgement

This research is partially supported by the National Basic Research Program of China (973 Program, 2004CB318202), the Project of Research & Development Plan of High Technology in China (863 Project, 2009AA12Z226). The work of X. He is supported in part by the US National Science Foundation (NSF) grant CNS-0720617. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150, December 2004.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[3] The Apache Software Foundation. Hadoop. http://hadoop.apache.org/core/, 2009.

[4] Amazon-S3. Amazon simple storage service (amazon s3). http://www.amazon.com/s, 2009.

[5] The Apache Software Foundation. HBase. http://hadoop.apache.org/hbase/, 2009.

[6] The Apache Software Foundation. Hive. http://hadoop.apache.org/hive/, 2009.

[7] Kang-Tsung Chang. *Introduction to Geographic Information Systems*. McGraw-Hill, 2002.

[8] Paul Bolstad. *GIS Fundamentals: A First Textbook on Geographic Information Systems, 3rd Edition*. Bookmasters Dist, June 2008.

[9] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *In Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association.

[10] Peter Honeyman, Dean Hildebrand, Dean Hildebrand, Lee Ward, and Lee Ward. Large files, small writes, and pnfs. In *in Proceedings of the 20th ACM International Conference on Supercomputing*, pages 116–124, 2006.

[11] Gluster. Glusterfs. http://www.gluster.org/, 2009.

[12] Swapnil.Patil Wittawat.Tantisiriroj and Garth.Gibson. Data-intensive file systems for internet services: A rose by any other name ..., 2008.

[13] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, 2002.

[14] Lustre. Lustre file system. http://www.lustre.org, 2009.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[16] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[17] Ahmed Amer, Darrell D. E. Long, and Randal C. Burns. Group-based management of distributed file caches. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 525, Washington, DC, USA, 2002. IEEE Computer Society.

[18] Shyamala Doraimani and Adriana Iamnitchi. File grouping for scientific data management: lessons from experimenting with real traces. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 153–164, New York, NY, USA, 2008. ACM.

[19] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[20] Hans Reiser. Reiserfs. http://ftp.kernel.org/pub/linux/utils/fs/reiserfs/, 2009.

[21] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[22] Margo Seltzer, Keith Bostic, Marshall Kirk Mckusick, and Carl Staelin. An implementation of a log-structured file system for unix. In *USENIX'93: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[23] Zhihui Zhang and Kanad Ghose. hfs: a hybrid file system prototype for improving small file and metadata performance. *SIGOPS Oper. Syst. Rev.*, 41(3):175–187, 2007.

[24] Faraz Shaikh and Mikhail Chainani. A case for small file packing in parallel virtual file system (pvfs2). In *Advanced and Distributed Operating Sytems Fall 07*, 2007.

[25] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, April 2009.

[26] Douglas Thain and Christopher Moretti. Efficient access to many small files in a filesystem for grid computing. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 243–250, Washington, DC, USA, 2007. IEEE Computer Society.

[27] James Hendricks, Raja R. Sambasivan, and Shafeeq Sinnamohideenand Gregory R. Ganger. Improving small file performance in object-based storage. Technical report, Carnegie Mellon University Parallel Data Lab, 2006.

[28] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Directory-based metadata optimizations for small files in pvfs. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 90–99, Berlin, Heidelberg, 2008. Springer-Verlag.