Research Article

Optimized Data Replication for Small Files in Cloud Storage Systems

Xiong Fu,^{1,2} Wenjie Liu,¹ Yeliang Cang,¹ Xiaojie Gong,¹ and Song Deng³

¹School of Computer and Technology, Nanjing University of Posts and Telecommunications, Nanjing 210023, China

²Jiangsu High Technology Research Key Laboratory for Wireless Sensor Networks, Nanjing 210023, China

³Institute of Advanced Technology, Nanjing University of Posts and Telecommunications, Nanjing 210023, China

Correspondence should be addressed to Xiong Fu; fux@njupt.edu.cn

Received 1 December 2015; Accepted 28 December 2015

Academic Editor: Chunlin Chen

Copyright © 2016 Xiong Fu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Cloud storage has become an important part of a cloud system nowadays. Most current cloud storage systems perform well for large files but they cannot manage small file storage appropriately. With the development of cloud services, more and more small files are emerging. Therefore, we propose an optimized data replication approach for small files in cloud storage systems. A small file merging algorithm and a block replica placement algorithm are involved in this approach. Small files are classified into four types according to their access frequencies. A number of small files will be merged into the same block based on which type they belong to. And the replica placement algorithm helps to improve the access efficiencies of small files in a cloud system. Related experiment results demonstrate that our proposed approach can effectively shorten the time spent reading and writing small files, and it performs better than the other two already known data replication algorithms: HAR and SequenceFile.

1. Introduction

In recent years, cloud computing has become an important web computation pattern and attracted more and more attention around the world. In a cloud system, a kind share of heterogeneous computation resources is achieved based on visualization technology. Hence, it is possible to deliver elastic and on-demand services to cloud users. Cloud storage is an important part of cloud computing [1]. It can provide data storage and access services for users whenever and wherever possible by constructing information sharing environment based on distributed file system [2]. Data replication is widely used in a cloud storage system to improve the data availability. And it can also cut down the data access latency and keep load balance of servers at the same time. Therefore, system performance is improved.

In a large scale distributed cloud storage system, the access time of a data file can vary a lot because of the arbitrarily changing distances. A huge amount of data can be generated in distributed systems. Reasonably dispersing these data in a system of large scale to improve the system performance

is of great importance. Therefore, data replication is widely used to ensure the data availability and improve the system reliability [3]. The data accessed by a computation node can be stored in a storage node that has some geological distances away from this computation node. Therefore, we can create a replica of this data in another storage node close to this computation node and the data access time will be reduced [4]. In a large scale cloud system, appropriate data replica schemes can reduce the energy cost and therefore cut down the expenses of operating the system [5]. Moreover, multireplica helps to handle difficult circumstances where some data are lost. As a result, it makes the system more resistant to failure.

Currently, well-known distributed file systems like GFS, HDFS, and so forth are designed for big file storage management. However, most of them do not perform well for storing large numbers of small files. Hence, how to manage small file storage and replication well is becoming a critical issue in cloud storage systems. For instance, Facebook, the biggest photo sharing website in the world, has stored more than 260 billion images, which translates to over 20 petabytes of

data [6]. As a matter of fact, there are huge amounts of small files in the current systems in the area of astronomy, climatology, biology, energy, e-Learning, e-Business, and e-Library [7, 8].

Lots of researches have been done to resolve the problem of small file storing. Small files are usually in large quantities and have to be accessed concurrently. Therefore, huge amounts of small files are usually stored in distributed file systems such as Google file system (GFS) [9], Hadoop distributed file system (HDFS) [10], and Amazon Simple Storage Service (S3) [11]. However, most of the current distributed systems are designed for storing large files. As an open-source software framework influenced by GFS, HDFS is well suited for the analysis of large datasets. However, storing huge amounts of small files will bring serious problems to Hadoop's performance. There is only one NameNode in HDFS, and all the metadata of files are stored in it. Therefore huge amounts of small files will greatly reduce the RAM utilization of NameNode [12]. Solutions to small files problem can be divided into two categories: general solutions like Hadoop archives (HAR) [13] and SequenceFile [14]; special solutions which are used in specific scenarios.

As a file archiving facility, HAR packs files into HDFS blocks. An HAR contains data files and metadata files. It can improve the memory utilization of NameNode. But a copy of original files will be generated when creating an archive, which brings extra burden on disk space. In a SequenceFile, a persistent data structure is used for storing binary key-value pairs. The key is file name and value is file contents. The evident defect of SequenceFile is that the whole SequenceFile needs to be read in order to look up a particular key. Also a specified key cannot be updated or deleted. Therefore, the access efficiency is greatly affected.

There are special solutions to deal with the small file storing problem in specific scenarios. Dong et al. [15] proposes a PowerPoint (PPT) file merging approach. Correlations between these files are considered. It introduces a two-level prefetching mechanism to improve the access efficiency of small files. The work in [16] introduces an approach to optimizing the I/O performance of small files and implements WebGIS on HDFS. Its main objective is to merge small files into large ones and then build index for each file. But no specific measures are provided to improve the access efficiency of small files.

Huge amounts of small files will cause heavy memory consumption of NameNode. However, multiple small files can be merged into large files to save the memory space of NameNode. Therefore, we propose a data replication optimization approach for small files in cloud storage systems. All the small files can be classified into four categories based on their read and write frequencies: the read and write intensive, read intensive, write intensive, and read and write sparse. Then they are merged into large files according to the file merging algorithm. These merged files are stored in blocks which are the basic units of storing data. A block will have several replicas. To manage the small file storage well, an Innode model is proposed to store small files and optimize the access frequencies of small files. The Innode consists of two parts and it is read and write separating. Also, a block

replica placement algorithm is introduced to migrate small files within Innodes. Related experiments demonstrate that our approach performs better than HAR and SequenceFile.

The rest of the paper is organized as follows. Section 2 presents the system model. The operations on block replicas are shown in Section 3. The file merge algorithm and block replica placement algorithm are introduced in Section 4. Section 5 shows experiment results and the performance discussions about the proposed algorithm. At last, conclusions and future work are presented in Section 6.

2. System Model

If the block size of a cloud system is smaller than 64 MB, then any file whose size is smaller than 2 MB will be called small files. Otherwise, files whose sizes are smaller than 10 MB will be called small files. The definition of a small file is shown as follows:

SmallFile

$$= \begin{cases} \{\text{file} \mid \text{file.size} < 2 \text{ MB}\}, & \text{block.size} \leq 64 \text{ M} \\ \{\text{file} \mid \text{file.size} < 10 \text{ MB}\}, & \text{block.size} > 64 \text{ M}. \end{cases} \quad (1)$$

In a cloud system, new challenges will be confronted when storing lots of small files (LOSF). It is a traditional practice to store data and its index on different storage nodes. And this will make it easier to manage data storage. Except for the cost of traversing among physical nodes, it will have I/O operations at least twice to get the target data. For accessing a large file, these I/O operations are only minor overhead amortized over many data accesses. However, for small files, these I/O operation costs are overwhelming compared to the data transmission time. Therefore, we try to store small files and their indexes on the same storage node (Innode) to reduce I/O cost. The model of the system that store small files is shown in Figure 1.

The round chain structure is designed in order to keep load balance of each Innode and make it easier to recover from failure. The structure of an Innode in cloud systems is shown in Figure 2.

The Innode consists of two parts: one part is the memory area that stores the indexes of files (IOF) and another part stores the cache of files (COF). The data stream in the proposed system model is unidirectional; therefore the storage pattern of COF in an Innode is read and write separating. The direction of data stream in read area of COF (RCOF) is from a DataNode to an Innode, and that of write area (WCOF) is the other way around. COF is system log based and is also layered (which is three layers in this paper). The block sizes of level 0, level 1, and level 2 in COF are 16 MB, 32 MB, and 64 MB, respectively. The block sizes in level 2 of COF are the same as that in DataNodes.

Blocks are the basic units of storing files in cloud systems. Each block has its key which is stored in the system log. The key of a block has a value range that indicates which level the block belongs to. The value range of a block key in each level is stored in the system log. The blocks in each level are sorted by their keys. A block in each level will have its maximum

FIGURE 1: Replication for small files in cloud storage systems.

FIGURE 2: An Innnode in a cloud system.

number of connections. Blocks in the highest level (level 0) in Figure 2 will have the maximum number of connections because they are the nearest to IOF and therefore they have the quickest response to data access. Note that blocks in WCOF also can be read. If there are any update operations on blocks, we must add the updated information to the end of WCOF because of the unidirectionality of data stream. Blocks that have the frequent accesses will be migrated to higher level of the memory area in RCOF as time goes on. And WCOF plays a role of writing the update information to DataNodes.

3. Operations on Small Files

3.1. Read Small Files. When querying a small file in the system, we should firstly determine whether the block that stores the file or its replica exists in the current level of COF by the block's key. Assume that the key of the block we are looking for is k . We can know the value range $[L(N), R(N)]$ of the block key of the current level by looking up the system log. $L(N)$ and $R(N)$ are the minimum and maximum block keys

in level N , respectively, and $0 \leq N \leq 3$. If $L(N) \leq k \leq R(N)$, then the block is in this level. Otherwise, go to the next level and continue to find the block. The querying process is shown as follows.

Step 1. Let $N = 0$ denote the first level we are about to look up for the block whose key is k .

Step 2. Initialize the variables $L(N)$ and $R(N)$ of level N based on the system log.

Step 3. If $L(N) \leq k \leq R(N)$, then locate the block in this level using the binary search algorithm and go to Step 4. Otherwise go to Step 5.

Step 4. If the block is found then go to Step 6. Otherwise go to Step 5.

Step 5. If $0 \leq N < 2$ then $N = N + 1$ and go to Step 2. Otherwise, go to Step 6.

Step 6. This is the end of the algorithm.

After the target block is found, the small file stored in this block will be located based on its index in IOF of the Innnode.

3.2. Write Small Files. When we update a small file in the system, we firstly check whether the block that stores this file or its replicas exists in level 0 based on the process of read small files depicted in Section 3.1. If it does, updated operations will be directly carried out on this block. If it does not, a new block will be added into level 0 and write the updated data into this new block. Because of the existence of the system log, when the blocks in this level are merged downward, the new blocks will cover or be merged with the old ones. Write operations can bring the problem of replica inconsistency. A write operation should finish the updates of multiple replicas of a block. These replicas may exist in

different data nodes which have some geological distances from each other. If all of these replicas are updated every time a write operation is carried out, the system bandwidth will be insufficient and therefore affects the real time performance of the system.

Therefore, we adopt delayed update strategy with the help of the log dairy. When accessing a block, the newest replica of this block will be located based on the log dairy. The replicas of a block will be updated if the following two conditions are satisfied. One is that the current replica is dumped downward to another level of COF. Another is that it has reached the maximum connections of the newest replica.

The processes of writing and updating data are shown as follows.

The writing data process is as follows.

Step 1. Find the target block based on the process of read small files.

Step 2. If the block is in level 0 of COF, then check whether the block has empty space. If there is empty space, write the data directly into this block, and go to Step 4.

Step 3. Add a new block in level 0 of COF. Write the data into the new block.

Step 4. This is the end of the algorithm.

The updating process is as follows.

Step 1. The new data is firstly written into blocks in level 0 of COF.

Step 2. Delay the update.

Step 3. If the replica of the block is dumped downward to the next level or it reaches the maximum connections of the replica, then write the updated data into the remaining replicas of this block immediately and go to the next step. Otherwise go to Step 2.

Step 4. This is the end of the algorithm.

The newly created replicas of blocks will be always in level 0 while cold replicas will be moved to lower levels. If a replica is in the first level and no more connections can be created to this replica, this replica will be copied into the cache of the neighbouring DataNode. However, if this replica is not in the first level, it will be moved to a higher level when no more additional connections can be created.

4. Optimization Algorithms

4.1. The Merging Algorithm of Small Files. Lots of small files can be stored in large files. This can reduce the number of files and therefore the size of file metadata can be reduced. In this way, data query efficiency and latency are improved, and much data transmission time is saved. Hence, we propose a small file merging algorithm. But firstly, we need to classify small files. Files in a cloud system can be classified into four

categories: the read and write intensive, read intensive, write intensive, and read and write sparse. There are concrete steps toward data classification.

Step 1. Assume that Pr_k and Pw_k are the read popularity and write popularity of data k , respectively. Read popularity and write popularity are the read times and write times within every minute. Then we can define the read and write mark of k : Sr_k and Sw_k :

$$\begin{aligned} Sr_k &= \begin{cases} 0 & \frac{Pr_k}{R_k} < 1 \\ 1 & \frac{Pr_k}{R_k} \geq 1, \end{cases} \\ Sw_k &= \begin{cases} 0 & \frac{Pw_k}{W_k} < 1 \\ 1 & \frac{Pw_k}{W_k} \geq 1, \end{cases} \end{aligned} \quad (2)$$

where R_k and W_k are the read times threshold and the write times threshold within a minute defined by users.

Step 2. We can classify all the data into four categories: S_1 , S_2 , S_3 , and S_4 ,

$$\begin{aligned} S_1 &= \{k \mid Sr_k = 0, Sw_k = 0\}, \\ S_2 &= \{k \mid Sr_k = 0, Sw_k = 1\}, \\ S_3 &= \{k \mid Sr_k = 1, Sw_k = 0\}, \\ S_4 &= \{k \mid Sr_k = 1, Sw_k = 1\}. \end{aligned} \quad (3)$$

S_1 is the read and write sparse; S_2 is the read intensive; S_3 is the write intensive; S_4 is the read and write intensive.

The small file merging algorithm is shown as follows.

Step 1. Classify all the small files in the system into the four types: S_1 , S_2 , S_3 , and S_4 .

Step 2. Merge all the files that belong to S_4 and allocate these files into blocks.

Step 3. If there are some blocks that are not full, then calculate the read/write ratio of these blocks and go to the next step. Otherwise go to Step 5.

Step 4. If the read/write ratio of a block is larger than 1, merge files of S_3 to this block. Otherwise merge files of S_2 to this block.

Step 5. Merge all the files that belong to S_2 and S_3 , respectively.

Step 6. If there are some blocks that are not full, then merge files of S_1 to these blocks. Otherwise go to the next step.

Step 7. Merge all the files that belong to S_1 .

Step 8. This is the end of the algorithm.

4.2. Replica Placement Algorithm. Based on the small file merging algorithm, we propose a replica placement algorithm. A set of small files $SF = \{f_1, f_2, \dots, f_m\}$ are selected based on the definition of a small file in Section 2. And this placement algorithm will be activated at intervals of t . These small files are merged into large files and allocated into blocks according to the small file merging algorithm discussed above. The merged large files are firstly placed in level 2 of COF. When some blocks in level 2 are accessed frequently, these files will be copied into upper levels of COF. The data in higher levels only comes from that of the neighbouring lower level. And data of level 2 comes from DataNodes. The main objective of the replica placement algorithm is to determine whether a block that stores small files should be placed in an Innode. The placement algorithm is depicted as follows.

Step 1. Initialize the set of small files whose sizes are not more than 1 MB and denote them as $SF = \{f_1, f_2, \dots, f_m\}$. The set $T = \{t_1, t_2, \dots, t_m\}$ refers to the latest access time of each file.

Step 2. Within every interval of t , for a small file f_i , the number of read requests is denoted as R_i ($0 < i < m$), and the number of effective read requests (which mean reading data successfully) is denoted as ER_i ($0 < i < m$). Similarly, the number of write requests is referred to as W_i ($0 < i < m$) and EW_i ($0 < i < m$) denotes the number of effective write requests. Then the weight we_i of the file f_i can be obtained based on the following formula:

$$we_i = \left(1 - \frac{ER_i + EW_i}{R_i + W_i}\right) \times \frac{t_i}{t}. \quad (4)$$

And the weights of all files can be denoted as $we = \{we_1, we_2, \dots, we_m\}$.

Step 3. Sort all the elements in WE and classify all the files into four categories based on formulas (2) and (3).

Step 4. Merge all the small files using small file merging algorithm depicted above. All the merged files can be placed in level 2 of COF since the block size of this level in COF is the same as that in DataNodes.

Step 5. The weight of a file in an Innode will suffer from attenuation if this file is not accessed during the next time interval. The weights of all the files in Innodes can be obtained based on the following formula:

$$WI_i = \begin{cases} we_i + \frac{1}{2}we'_i & \text{acFlag}_i = 0 \\ we_i + we'_i & \text{acFlag}_i = 1, \end{cases} \quad (5)$$

where we'_i is the weight of f_i in the last time interval and $\text{acFlag}_i = 0$ denotes that f_i was not accessed during last time interval.

Step 6. If the weights of files in DataNodes are larger than that of files in level 2 of COF in the Innode, then swap these files. Similarly, swap the files in level 2 to level 1 and ensure that the

FIGURE 3: Number of blocks when storing files.

weights of all files in level 2 are smaller than that in level 1. The same operation is carried out between level 1 and level 0.

Step 7. This is the end of the algorithm.

5. Performance Evaluation

The proposed approach (which is called D2I for short) is compared with native HDFS and Hadoop archives (HAR). There are seven servers in the experiment. One of them is the Innode and the others are DataNodes. When a write operation of a file is finished in a server, there is no need to wait for finishing updates of all the file's replicas. The size of a block in level 2 of COF or in DataNodes is 32 MB for the restrictions in experimental conditions.

Figure 3 shows the number of blocks when storing small files using different schemes. The performance of HAR or D2I is obviously better than that of SequenceFile. This is because data archiving approaches are adopted in HAR and D2I. The file merging algorithm in SequenceFile does not consider the relations between files and does not support block appending. The reason why D2I performs better than HAR is that D2I adopts read and write archiving and D2I can effectively avoid block recombinations.

Figure 4 shows the average disk utilization (DU) of DataNodes when there are different numbers of files. The disk utilization of a DataNode can be obtained based on the following formula:

$$DU = \frac{\sum_{i=0}^{n-1} \text{size}(f_i)}{m \times \text{size}(\text{Block})}, \quad (6)$$

where n is the number of files in the DataNode and m is the number of effective blocks storing files. When there is no data in a block, this block is not an effective block. $\text{size}(\text{Block})$ is the block size in the current system and it is 32 MB in our experiments.

Because SequenceFile does not support small file appending, we cannot append files into the same block at different

FIGURE 4: Average disk utilization of DataNodes when there are different numbers of files.

FIGURE 5: Memory utilization of DataNodes when there are different numbers of files.

time intervals. Therefore, it underutilizes disk space. In our experiment it only has 60 percent utilization. D2I and HAR have high utilizations of disk space which are over 80 percent. D2I has 90 percent utilization at some points.

Figure 5 shows the memory utilization of DataNodes when there are different numbers of files. When SequenceFile is adopted, the DataNode has the lowest utilization of memory. In D2I, when a small file is in an Innode, it will be accessed directly from the Innode. Compared with preloading data into the memory of the DataNode, the COF of the Innode has higher cache hit rate and low accessing cost.

The main objective of our proposed approach is to improve the accessing efficiency of small files. There are different numbers of randomly generated small files. The time spent reading and writing these files is shown in Figures 6, 7, 8, and 9. The results are divided into two groups: random access and sequential access. The files will be accessed one by

FIGURE 6: The file reading time in sequential access.

FIGURE 7: The file reading time in random access.

one in sequential access. But these files may be accessed more than once in random access.

Figures 6 and 7 show the randomly and sequentially read time of files. We can see that the performance of HAR and D2I is better than that of SequenceFile. This is because, in SequenceFile, sequential search is needed within the blocks to locate files. While in HAR, secondary index is adopted to locate files. And files can be directly found in Innode when D2I is put in use. If the number of files is less than 8000, D2I performs better than HAR. But it does not have a deciding advantage over HAR when there are more than 8000 files because of the limited size of COF in Innode. The D2I performs better than HAR on the whole in random access.

Figures 8 and 9 show the randomly and sequentially write time of files.

Generally, the time spent writing is longer than that spent reading. The reason is that file reorganization and block allocation can be involved in writing files. From Figures 8 and 9 we can conclude that D2I performs better than the other two schemes when randomly or sequentially writing files. This is because read and write separating pattern is adopted in

FIGURE 8: The file writing time in sequential access.

FIGURE 9: The file writing time in random access.

D2I. Small files can be directly written in Innode and then be asynchronously dumped into DataNode. Block allocation will spend too much time in SequenceFile and secondary indexes need to be built in HAR. Therefore, these two schemes are not as good as D2I when writing files into servers.

6. Conclusions

More and more scientific or commercial applications are cloud based nowadays. Cloud storage is an important part in a cloud system. However, small file storage is barely considered in the current cloud storage systems. This paper classifies small files into four types and all these files are merged into large files based on their types. Then a block replica placement algorithm is proposed to optimize file accessing in a cloud system. The experiment result shows that D2I can effectively shorten the time spent reading and writing small files. And its performance has better performance than the other two already known data replica algorithms. However, the proposed approach has not been evaluated in a large scale cloud system because of the limited experimental

conditions. In the future, we endeavor to implement the proposed approach on a cloud system of large scale and gather more realistic information to refine the approach.

Conflict of Interests

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

Acknowledgments

This work is sponsored by National Science Foundation of China (no. 61202354, no. 51507084, no. 61272422, no. 61572260, no. 61373017, and no. 61572261) and Technological Innovation Funds for Technology-Based Enterprises in Jiangsu (BC2014195).

References

- [1] D. Slamanig and C. Hanser, "On cloud storage and the cloud of clouds approach," in *Proceedings of the 7th International Conference for Internet Technology and Secured Transactions (ICITST '12)*, pp. 649–655, London, UK, December 2012.
- [2] T. Sivashakthi and N. Prabakaran, "A survey on storage techniques in cloud computing," *International Journal of Emerging Technology and Advanced Engineering*, vol. 3, no. 12, pp. 125–128, 2013.
- [3] J.-W. Lin, C.-H. Chen, and J. M. Chang, "QoS-aware data replication for data-intensive applications in cloud computing systems," *IEEE Transactions on Cloud Computing*, vol. 1, no. 1, pp. 101–115, 2013.
- [4] D. T. Nukarapu, B. Tang, L. Wang, and S. Lu, "Data replication in data intensive scientific applications with performance guarantee," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 8, pp. 1299–1306, 2011.
- [5] B. Li, S. Song, I. Bezakova, and K. W. Cameron, "Energy-aware replica selection for data-intensive services in cloud," in *Proceedings of the IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '12)*, pp. 504–506, IEEE, Washington, DC, USA, August 2012.
- [6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebook's photo storage," in *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation (OSDI '10)*, pp. 47–60, Vancouver, Canada, October 2010.
- [7] C. Shen, W. Lu, J. Wu, and B. Wei, "A digital library architecture supporting massive small files and efficient replica maintenance," in *Proceedings of the 10th Annual Joint Conference on Digital Libraries (JCDL '10)*, pp. 391–392, June 2010.
- [8] E. H. Neilsen Jr., "The Sloan Digital Sky Survey data archive server," *Computing in Science & Engineering*, vol. 10, no. 1, pp. 13–17, 2008.
- [9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.
- [10] M. Shahabinejad, M. Khabbazian, and M. Ardakani, "An efficient binary locally repairable code for hadoop distributed file system," *IEEE Communications Letters*, vol. 18, no. 8, pp. 1287–1290, 2014.

- [11] Amazon-S3, “Amazon simple storage service (amazon s3),” 2009, <http://www.amazon.com/s>.
- [12] X. Zhao, Y. Yang, L. Sun, and H. Huang, “Metadata-aware small files storage architecture on hadoop,” in *Web Information Systems and Mining*, vol. 7529 of *Lecture Notes in Computer Science*, pp. 136–143, Springer, Berlin, Germany, 2012.
- [13] Hadoop Archives, Hadoop archives guide, 2011, <http://hadoop.apache.org/docs/current/hadoop-archives/HadoopArchives.html>.
- [14] SequenceFile, Sequence file wiki, 2011, <http://wiki.apache.org/hadoop/SequenceFile>.
- [15] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, and Y. Li, “A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by PowerPoint files,” in *Proceedings of the IEEE 7th International Conference on Services Computing (SCC '10)*, pp. 65–72, IEEE, Miami, Fla, USA, July 2010.
- [16] X. Liu, J. Han, Y. Zhong, C. Han, and X. He, “Implementing WebGIS on hadoop: a case study of improving small file I/O performance on HDFS,” in *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER '09)*, pp. 1–8, New Orleans, La, USA, September 2009.

Hindawi

Submit your manuscripts at
<http://www.hindawi.com>