The Journal of Systems and Software 86 (2013) 744–750

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A RAMCloud Storage System based on HDFS: Architecture, implementation and evaluation

Yifeng Luo [a], Siqiang Luo [a], Jihong Guan [b], Shuigeng Zhou [a,*]

[a] Shanghai Key Lab of Intelligent Information Processing, and School of Computer Science, Fudan University, Shanghai, China
[b] Department of Computer Science and Technology, Tongji University, Shanghai, China

ARTICLE INFO

Article history:
Received 20 March 2012
Received in revised form 15 October 2012
Accepted 6 November 2012
Available online 1 December 2012

Keywords:
Cloud computing
RAMCloud
HDFS
Cloud Storage

ABSTRACT

Few cloud storage systems can handle random read accesses efficiently. In this paper, we present a RAMCloud Storage System, *RCSS*, to enable efficient random read accesses in cloud environments. Based on the Hadoop Distributed File System (HDFS), RCSS integrates the available memory resources in an HDFS cluster to form a cloud storage system, which backs up all data on HDFS-managed disks, and fetches data from disks into memory for handy accesses when files are opened for read or specified by users for memory storage. We extend the storage capacity of RCSS to that of the substrate disk-based HDFS by multiplexing all the available memory resources. Furthermore, RCSS supports MapReduce, which is a popular cloud computing paradigm. By serving data from memory instead of disks, RCSS can yield high random I/O performance with low latency and high throughput, and can achieve good availability and scalability as HDFS.

1. Introduction

The I/O performance of disk-based storage systems has been far outstripped by computing facilities, and the situation is getting worse in cloud environments. Most existing cloud storage systems are based on disks, and can efficiently support sequential read accesses over massive datasets. However, few can handle random read accesses efficiently, especially for WORM (Write Once Read Many) applications that can yield a huge number of random accesses, and require low latency and high throughput.

Ousterhout et al. (2011) proposed the idea of RAMCloud for cloud environments, aiming at aggregating all available RAMs in a big cluster to host massive online data to meet the needs of large-scale Web applications, while relegating disks to a backup/archival role. They expected to achieve efficient RPC APIs and high I/O performance by implementing specialized protocols and deploying optimized network infrastructures. For example, they tried to achieve low round-trip network latency of μs scale by using expensive Infiniband switches. While most built datacenters use Ethernet/TCP/IP based network infrastructures, it is difficult for the existing datacenters to deploy such a RAMCloud system. Furthermore, it is a fresh-new storage architecture, to migrate existing applications onto it would be an onerous job.

Many commercial companies, including Yahoo! and Facebook, have deployed HDFS (Shvachko et al., 2010), an open-source implementation of the Google File System (Ghemawat et al., 2003), as their massive storage systems, as it is easy to deploy HDFS in existing Ethernet-based datacenters. HDFS is a part of the famous open-source project — Hadoop (Venner, 2009; Apache, 2007). Hadoop also includes an open-source implementation of MapReduce (Dean and Ghemawat, 2004), which is now one of the most popular cloud computing paradigms for processing analysis and transformation jobs over large-scale massive datasets. HDFS is highly fault tolerant while it runs on inexpensive commodity servers, and can deliver high aggregate performance to a large number of concurrent clients, especially for WORM (Write Once Read Many) applications.

However, HDFS is unable to provide efficient support for applications with a huge number of random read accesses, as its basic storage devices are magnetic disks. For example, HBase (Apache, 2007) — an HDFS-based key-value store, often accesses data randomly, while the underlying HDFS is unable to provide low-latency and high-throughput I/O performance. A typical HBase application is to employ HBase to keep records of a massive amount of user events of a large e-business company for offline analysis, which would yield a lot of random lookups to track all the actions of users across time, as it is very important to understand how users interact with the company's website at large.

In this paper, we present *RCSS*, an HDFS-based *R*AMCloud *S*torage *S*ystem, to improve random I/O performance for HDFS-based applications. RCSS can accommodate all existing running HDFS-based applications without any modification to the

* Corresponding author.
E-mail addresses: luoyf@fudan.edu.cn (Y. Luo), sqluo@fudan.edu.cn (S. Luo), jhguan@tongji.edu.cn (J. Guan), sgzhou@fudan.edu.cn (S. Zhou).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.11.025

application programs. We design the architecture of RCSS, and implement the system. We also give an extensive performance evaluation on the system.

The rest of this paper is organized as follows. Section 2 gives an overview on HDFS. Section 3 presents a simple memory storage solution for HDFS, which is called RAM-HDFS. Section 4 provides the details of design and implementation of RCSS. Section 5 presents the experimental evaluation of RCSS. Section 6 surveys the related research work, and finally Section 7 concludes the paper.

2. HDFS

As RCSS is built on top of HDFS and employs HDFS as its backup facility, we give a short introduction to HDFS in this section to help understand how RCSS harnesses HDFS to ensure data availability, durability and scalability, which are the key features to a cloud storage system.

HDFS has a master/slave architecture, as shown in Fig. 1. An HDFS cluster consists of one NameNode and multiple DataNodes, where the NameNode serves as the master, and the DataNodes run as the slaves. An HDFS file consists of file blocks and metadata about mapping information of data blocks. NameNode manages all metadata, namely the namespace of the HDFS, and DataNodes manage data blocks in HDFS. A data block usually is replicated across multiple DataNodes for fault-tolerance and access performance. By default, three replicas are stored. NameNode and DataNodes are fully connected and can communicate with each other using TCP-based protocols.

The HDFS namespace is a hierarchy of files and directories. The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes (the physical location of file data). HDFS keeps the entire namespace in memory. The NameNode keeps a persistent record of namespace and logs the modifications of namespace in its local native file system. During restarts, the NameNode restores the namespace in memory by reading the namespace from local native file system and replaying the logs.

When an HDFS client reads a file from HDFS, the client first contacts the NameNode to get the information about DataNodes from which the required file blocks are available. Usually, the DataNodes at which a file block stays are sorted according to their distances to the client, and the client retrieves the file block according to this sorted order. If the replica on the first DataNode becomes unavailable, the next one in the sorted DataNode list will be tried. When a client writes data into HDFS, the client first contacts NameNode, which creates and records the metadata about this file, and then returns to the client the information about the DataNodes at which the blocks of this file will be stored. Finally, the client starts a data stream to write the file blocks to the corresponding DataNodes. When a DataNode receives a block from the data stream, it mirrors the stream out to the next DataNode where a replica of this block will be stored. And from this replicating DataNode, the stream will be continuously mirrored out to the next DataNode until the block reaches the specified number of replicas.

To sum up, HDFS is designed to take DataNode failures as the normalcy rather than exceptions, so as to provide high fault-tolerance. HDFS can deliver high aggregate performance to a large number of concurrent clients, especially for WORM (Write Once Read Many) applications.

3. RAM-HDFS: a simple solution and its drawbacks

There exists a very simple solution to implement a memory storage system in HDFS, and we call it *RAM-HDFS* in this paper. To store all data in memory, we can simply make the HDFS storage directories reside on RAM devices instead of disk devices, and only use local disks for backup to ensure data durability and availability in case of node or system reboots. When an HDFS cluster or a node starts up, data can be restored from local disks into memory. However, RAM-HDFS has the following drawbacks, which make RAM-HDFS not a viable cloud storage solution:

1 Its storage capacity is limited to the total memory of all cluster nodes. As a cloud storage system, its storage capacity must be very large, and the storage capacity limitation would make RAM-HDFS incompetent to be a cloud storage system. Assuming a company owns a cluster that consists of 5000 commodity servers, and each server can spare 30 GB memory resources for RAM-HDFS, the whole storage capacity can only add up to 150 TB. What is more, considering redundancy for data availability, HDFS usually stores three replicas for a data block, and this would limit the storage capacity for application data to be 50 TB. A storage capacity of 50 TB would sometimes even not suffice to host the data for just one large-scale application. And this situation would get worse for small or middle-class organizations, which cannot afford big clusters but still have to explore massive datasets.
2 Restoring data from local disks to RAMs is time-consuming, sometimes even unbearable when the stored application data consist of a huge number of small blocks. Assuming each DataNode in RAM-HDFS has stored 30 GB data, and the bandwidth of restoring data from local disks to RAMs is 100 MB/s, to restore the 30 GB data from disks to RAMs would take 300 s. If the 30 GB data consist of a huge amount of small blocks, this process would last several hours.
3 Occupied RAMs are not available to other applications before the files occupying the RAMs are moved. Deleting files from RAM-HDFS includes deleting the data associated with these files from both RAMs and backup disks, and data need to be reloaded to RAM-HDFS before the deleted files can be accessed next time. Usually, the data stored in a cloud system belong to many different applications, and not all data are accessed simultaneously by these applications. RAMs can be released once the files will not be accessed soon, and other applications can use them immediately. As a matter of fact, a lot of data are rarely accessed, and these data do not need to be stored in memory.
4 To store data in RAM-HDFS will be slower than to store data in HDFS because data have to be backed up to disks. When writing a file block into HDFS, the DFSClient will initiate a pipelined output stream. By default, three DataNodes will be in the pipeline, and the block should be written to all the three DataNodes before the output stream can be closed. As for RAM-HDFS, while mirroring out the block into a down-stream DataNode, the DataNode should write the block to its RAMs and local backup disks. The output stream cannot be closed until all DataNodes have this block all backed up to their disks. This process would make the write performance of RAM-HDFS even worse than that of disk-based HDFS, which is poor because data durability and availability should be ensured.

4. RCSS: the design and implementation

Based on HDFS, we implement RCSS that overcomes the drawbacks of RAM-HDFS. By using HDFS as its backup facility, system robustness and data availability of RCSS are guaranteed by HDFS. Data are initially written to the disks of substrate HDFS, and fetched into RAMs only when they are to be accessed soon. In this section, we present the details of the design and implementation of RCSS, including architecture and implementation techniques.

4.1. Namespace management

This subsection describes how RCSS harnesses HDFS to manage its namespace and discriminate RCSS files from regular HDFS files.

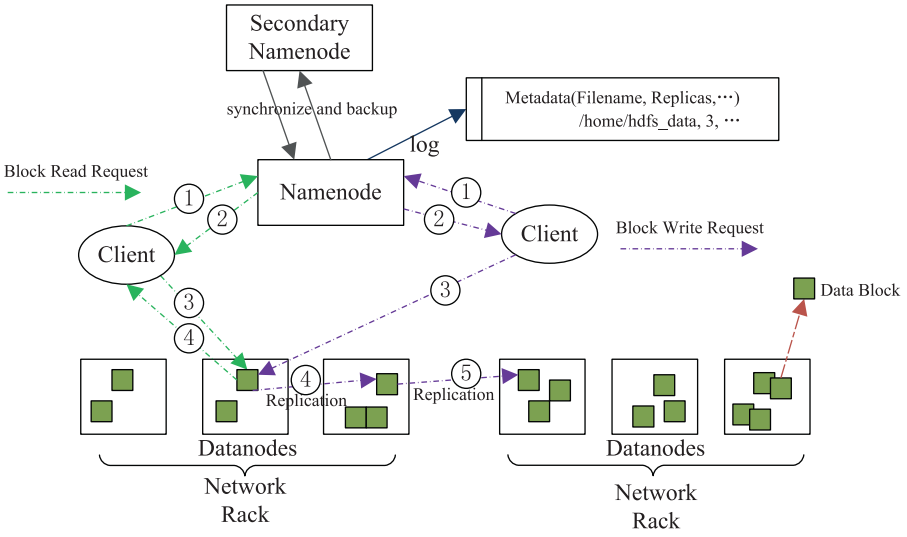

Fig. 1. The architecture of HDFS.

We create a directory in HDFS as the root directory of RCSS, and place all other RCSS directories and files under this root directory. In this way, the namespace of RCSS is treated as a subspace of HDFS namespace. As a result, RCSS files are visible to the HDFS clients. When a user is to access a file, RCSS first decides whether or not the file lies in the RCSS namespace. If the file is in the RCSS namespace, RCSS will handle the I/O requests for the file; Otherwise, HDFS will handle the I/O requests as regular HDFS requests. Employing HDFS to manage RCSS namespace has the following merits:

1 RCSS and HDFS can be managed in a unified framework. Because RCSS namespace is organized as a subspace of HDFS, RCSS files are not different from the regular HDFS files, except that RCSS files can be served from RAMs, and users can use HDFS shell commands or APIs to manage or access RCSS files.
2 Much flexibility can be achieved. As time evolves, the "hotness" or "coldness" of data will change. When cold data in HDFS get hot, we may want to store them in RCSS so as to boost applications; when hot data in RCSS get cold, we may want to move them out of RCSS so as to clear out RCSS storage space for other hot data. To exchange data between RCSS and HDFS is very simple, because it involves only metadata modifications, which can be done efficiently. To import HDFS files into RCSS storage space, we just need to move the designated files or directories to the RCSS root directory; and to clear files or directories out of RCSS storage space, we just need to move the files out of the RCSS root directory.
3 MapReduce is supported by RCSS. As RCSS storage space is an HDFS subspace, MapReduce applications are naturally supported. It is important to support MapReduce, as it is the most popular cloud paradigm for processing large datasets.
4 The storage capacity of RCSS is extended to that of the substrate disk-based HDFS, which makes RCSS a real cloud storage system. RCSS can provide comparable I/O performance to RAM storage, while still possess an extended storage capacity equal to that of the disk-based HDFS.

4.2. Architecture

Fig. 2 shows the architecture of RCSS. RCSS shares NameNode and DataNodes with HDFS. The NameNode consists of the HDFS NameSpace Manager (HNM) and the RCSS RAMBlock Manager (RRM). HNM manages the whole namespace of HDFS, including the namespace of RCSS. RRM manages the global information about blocks in memory. Each DataNode consists of a Local DiskBlock Manager (LDM) and a Local RAMBlock Manager (LRM). LDM manages the data blocks residing on the DataNode's disks, and LRM manages the blocks in RAMs of the DataNode. As HNM and LDM are implemented in HDFS, here we mainly introduce RRM and LRM.

RRM runs on the NameNode, it consists of five modules: *Request Handler* handles all RCSS I/O requests; *RAMInfo Keeper* tracks all blocks stored in RAMs; *RAMBlock Evictor* clears data blocks out of RAMs when there are not enough free RAMs to host new files; *RAMBlock Scheduler* schedules the RAM storage of data blocks; *TargetNode Chooser* selects DataNodes to host a file in RAMs. Each DataNode has an LRM to manage the data blocks stored in its RAMs. LRM stores a block in RAMs and records its location when the DataNode is chosen to host the block in its memory. LRM moves a block out of RAMs and deletes the record about its location when the DataNode is chosen to move a block out of its RAMs.

Note that the aim of RCSS is to speed up random read accesses in HDFS, while application data can be discriminated as frequently accessed and rarely accessed according to domain knowledge or data access history, so it would not make much sense to store rarely accessed files in RAMs, and storing only frequently accessed files in RAMs can yield the most speedup of applications. We expect that users can prioritize RCSS files according to their access frequencies. Files that may be accessed more frequently can be assigned with higher priorities, while files accessed less frequently can be assigned with lower priorities. When there is not enough free memory to host new files, RCSS moves out in-memory files according to file priorities: files with lower priorities will be moved out before files with higher priorities. So files with higher priorities will have more chances to reside in memory than files with lower priorities.

When a user creates a file in RCSS, the metadata of the file are added to the RCSS namespace hierarchy and persisted into the underlying substrate HDFS namespace. The user can associate a priority, ranging from 1 to 5, with the file to specify its access frequency. Here, a priority 5 implies that the file may be accessed most frequently so that it should be kept in memory as long as possible. A priority 1 implies that the file has the lowest probability that it would be accessed frequently so that it can be moved out of memory once RCSS dose not have enough free RAMs to host more files. The priority of a file is set to 3 by default if no user sets it.

RCSS writes files directly to the underlying substrate HDFS, instead of the RAMs in the cluster. The whole *write* process is the same as writing a regular HDFS file in a pipelined way. So this

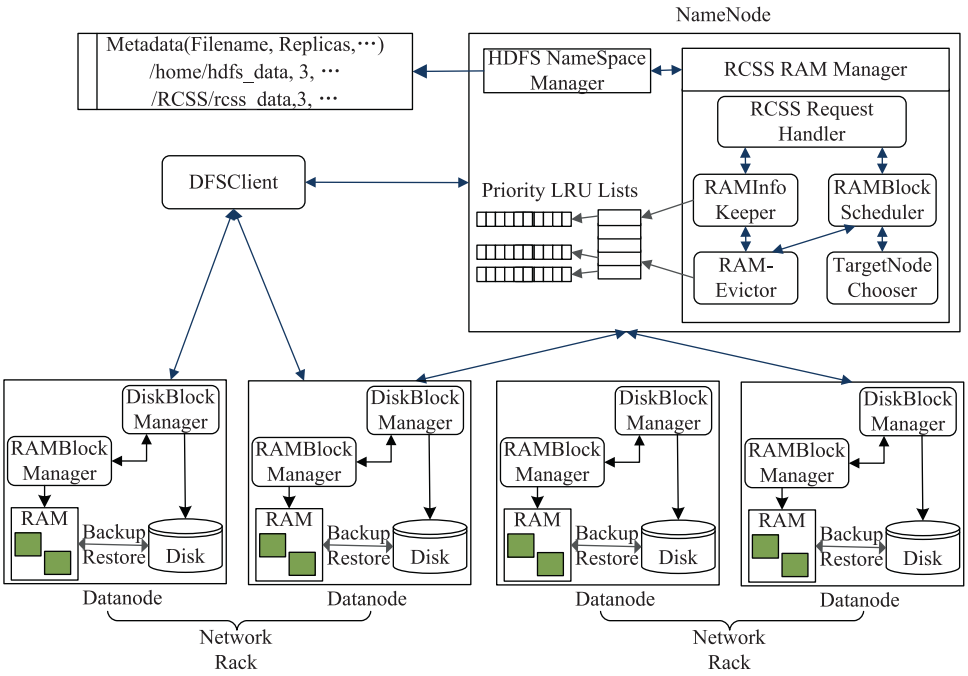

Fig. 2. The architecture of RCSS.

process is different from that of writing data to RAM-HDFS that involves writing data to both RAMs and backup disks. When the file is opened for read or an HDFS shell command we implement is issued, which means that the file is to be accessed soon, RCSS fetches the corresponding file from disks into memory.

RAMBlock Scheduler schedules the RAM storage for blocks of a file. When a new file is to be loaded into RAMs from disks, DFSClient enquires the RAMBlock Scheduler for DataNodes to whose RAMs the data blocks will be written. If there are not enough DataNodes having enough free RAMs to host blocks of the file, RAMBlock Scheduler informs RAMBlock Evictor to clear enough files out of RAMs. RAMBlock Evictor clears files based on two rules: (1) files with lower priorities are evicted before files with higher priorities; (2) files of similar priority are queued in an LRU list, and the least recently accessed files are evicted before the most recently accessed ones. The eviction process continues till RCSS has enough memory to host the new file.

TargetNode Chooser is responsible for choosing the target DataNodes to whose RAMs each data block should be written. RAMBlock Scheduler consults TargetNode Chooser so as to optimize the schedule of RAM storage for all blocks to minimize data transfer across the network. If RCSS has enough free memory to host the file, the whole process consists of the following three steps as shown in Fig. 3:

- Step 1: schedule as many blocks as possible to their local DataNodes;
- Step 2: schedule as many blocks as possible to rack-local DataNodes;
- Step 3: schedule all the remaining blocks to other rack-remote DataNodes.

RAMInfo Keeper keeps the records describing which DataNodes have hosted which data blocks in their memory.

RCSS Request Handler handles the I/O requests when a user reads an RCSS file. When DFSClient reads an RCSS file that has not been stored in RAMs, RCSS Request Handler will automatically fetch all the file blocks into RAMs first, and then accesses the file blocks in RAMs. When DFSClient reads an in-memory RCSS file, RCSS Request Handler first checks with RAMInfo Keeper and HDFS NameSpace Manager for information about the DataNodes that have hosted replicas of file blocks in their RAMs. As there exist multiple DataNodes that may contain a replica of a block in RAMs or on disks, RCSS sorts the DataNodes according to strategy listed in Table 1, whereby DFSClient accesses each data block by the sorted sequence of DataNodes. If accessing the block from a DataNode in the sorted sequence fails, the next DataNode will be tried.

For all DataNodes that contains an in-memory or on-disk replica of a block, we sort the DataNodes according to the strategy listed in Table 1. As we can see from Table 1, not all DataNodes that host in-memory replicas come before DataNodes which host on-disk replicas. DataNodes are sorted according to the following two rules:

1 DataNodes with smaller distances to DFSClient come before DataNodes with larger distances;
2 For all DataNodes with equal distances to DFSClient, DataNodes with in-memory replicas come before DataNodes with on-disk replicas.

The rules above ensure that accessing an in-memory replica from a DataNode dose not induce more network contention than from any other DataNode. So a local DataNode with an on-disk replica comes before a rack-local DataNode with an in-memory replica, and a rack-local DataNode with an on-disk replica comes before a rack-remote DataNode with an in-memory replica.

Table 1
DataNodes sorting strategy.

Order	Relative location of replicas
1	RAM local
2	Disk local
3	RAM Rack-local
4	Disk Rack-local
5	RAM Rack-remote
6	Disk Rack-remote

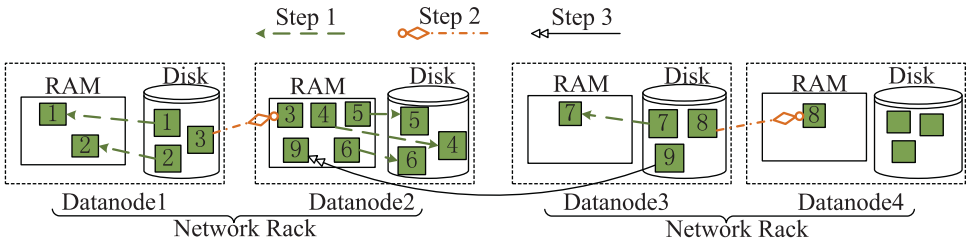

Fig. 3. The illustration of storage schedule for RAM blocks. DataNode1 can host two blocks in RAMs, DataNode2 can host five blocks, DataNode3 and DataNode4 can only host one block respectively. Step 1: Blocks 1–2, 4–6 and 7 are scheduled to their local DataNodes; Step 2: Blocks 3 and 8 are scheduled to rack-local DataNodes; Step 3: Block 9 is scheduled to a rack-remote DataNode.

4.3. User interface

RCSS integrates all the available free memory resources in an HDFS cluster to form a unified storage system for upper-level users. Users can manipulate files in RCSS by issuing HDFS shell commands with "-ram" option, to specify that the target files are RCSS files. For example, issuing command "mkdir -ram" will make a directory in RCSS. Because the user interface of RCSS is similar to that of HDFS, users familiar with HDFS interface can operate the RCSS system without any difficulty.

We implement two additional HDFS shell commands: *copyToRam* and *deleteFromRam* to fetch files into memory from disks and move files out of memory to release the occupied RAMs. One more important feature for users familiar with HDFS is that RCSS supports all HDFS APIs. Users can employ RCSS to boost their HDFS-based applications, including MapReduce applications.

4.4. Data durability and availability

HDFS accommodates mechanisms of backup, synchronization and monitoring. RCSS relies on HDFS to guarantee its data durability and availability.

For a large HDFS cluster, multiple copies of namespace are stored under different directories, which usually reside on different disk devices. A secondary NameNode synchronizes and backs up the namespace from the master NameNode. HDFS namespace can be imported and recreated when the master NameNode loses the original copy of namespace. As RCSS namespace is a subspace of HDFS namespace, RCSS can function normally as long as the whole HDFS cluster continues to function normally.

As for a data block in RCSS, multiple replicas of it are backed up across the HDFS cluster for fault-tolerance. By default, three replicas are stored for each block, where two replicas are stored on two different DataNodes in the same rack, and a third replica is stored on another DataNode in a different rack, so that at least one replica is available in the case when a rack fails. The NameNode periodically checks the statuses of replicas of all blocks. If some replicas become unavailable or corrupted, the NameNode will schedule a replication process to ensure that enough valid replicas are available. So the availability of data blocks stored in RCSS is as high as in HDFS.

5. Experimental evaluation

We used Hadoop 0.20.2 to deploy an HDFS cluster consisting of 31 commodity PCs, with one node serving as NameNode and the rest 30 ones serving as DataNodes. Each node has an Intel Core2 2.93 GHz CPU, 4 GB memory, a 500 GB IDE disk, and Ubuntu Linux OS. The cluster has a two-level network hierarchy, where the core switch and two access switches are all 100 Megabit NetGear Ethernet switches. Each DataNode spares 3 GB memory for RAM-HDFS/RCSS and 80 GB disk capacity for HDFS, so the storage capacity of RAM-HDFS is 90 GB, and RCSS owns a storage capacity of 2.4 TB and has 90 GB memory for RAM storage. As the goal of RCSS is to improve random read performance, we did not measure the sequential read performance of RCSS, but measured only the random read performance of RCSS, and compared it with disk-based HDFS and RAM-HDFS.

Firstly, we measured the write performance of RCSS. The results are demonstrated in Fig. 4(a) and (b) for small and large files respectively. As RCSS directly writes all data into the underlying HDFS, the write performance of RCSS is the same as that of HDFS. Writing data to RAM-HDFS is slower than to HDFS and RCSS, because DataNodes in RAM-HDFS need to write data to RAMs and local disks for backup, that is to say, RCSS and HDFS do not need any write to RAMs, while RAM-HDFS needs one more write to RAMs for each file. We can see, for small files, writing a file to RAMs incurs an average overhead around 1 ms, so the total overhead of writing 10,000 small files to RAMs is around 10 s; For large files, the average overhead for writing 1 GB data to RAMs is around 1 s.

We then measured the random read performance of RCSS under three circumstances: random reads of small files ranging from 2 kB to 10 kB, random seek-and-reads of 1 kB-sized data from large files ranging from 4 GB to 24 GB, and random reads for 1 kB-sized rows from two tables in HBase. We used YCSB (Cooper et al., 2010) to evaluate the read performance of HBase. The results are shown in Fig. 4(c), (d) and (e). We can see that RCSS has similar read performance to that of RAM-HDFS, as both directly serve data from RAMs.

For random reads of small files, it takes around 2 ms to read a file from RCSS/RAM-HDFS, and takes 5–9 ms to read a file from HDFS, so RCSS/RAM-HDFS can yield 2.5–4.5 times higher throughput than HDFS. The average cost for reading a file remains stable for different file sizes when data are served from RCSS/RAM-HDFS, and the average cost for reading a file increases as the size of files increases when data are served from HDFS. As for random seek-and-reads from large files, RCSS/RAM-HDFS can yield 3.7–9.25 times higher throughput than HDFS. The average cost for reading 1 kB-sized data remains stable around 2.5 ms for RCSS/RAM-HDFS, and grows from 9.7 ms to 19.8 ms for HDFS as the file size increases from 4 GB to 24 GB. As for random reads from HBase tables, RCSS/RAM-HDFS can yield 3 times higher throughput than HDFS. The average cost of reading a 1 kB-sized row for RCSS/RAM-HDFS is around 3 ms while the cost for HDFS is 10.9 ms and 15.5 ms for the two HBase tables respectively.

Fig. 4(f) shows the latency of read operations over HBase tables when the system throughput of HBase changes. As we can see that the latency of read operations over disk-based HBase increases slightly from 11 to 13 ms when system throughput increases to 3000 operations per second, which is the maximum throughput that our disk-based HBase cluster can provide, while the latency of read operations over RAM-based HBase remains stable at 3 ms when system throughput increases.

Finally, we measured the overhead introduced by network infrastructure for random read accesses. We compared the

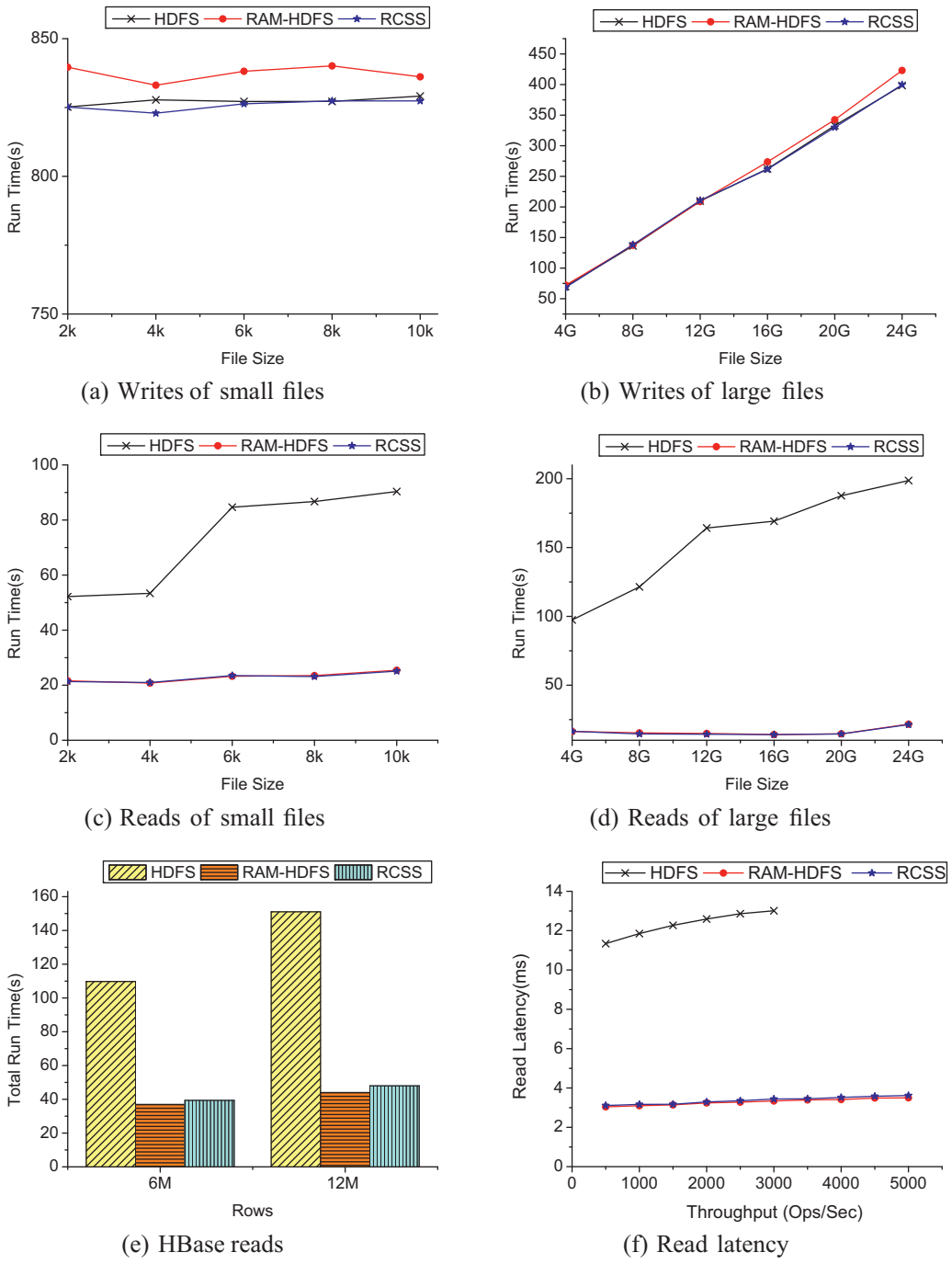

Fig. 4. Experimental results. (a) Run time of 10,000 writes of small files; (b) run time of 1 write of large files; (c) run time of 10,000 random reads of small files; (d) run time of 10,000 random seek-and-reads in a large file; (e) run time of 10,000 random reads of 1 kB-sized rows from two HBase tables of different sizes; (f) latency of read operations when system throughput changes.

random read performance of HDFS with two different configurations: one is configured to serve small files from a local DataNode, and one is configured to serve small files from a remote DataNode, so the performance difference between the two configurations is incurred by network. The results are shown in Fig. 5. The average round-trip network overhead for each read access is about 2 ms. We can see that network overhead is the major cost of reading data from RCSS/RAM-HDFS, while the major cost of reading data from HDFS is disk seeking. If the substrate HDFS is deployed on network infrastructures with much less round-trip overhead, RCSS can yield higher throughput than the experimental results, and HDFS-based applications can gain more speedup from RCSS.

6. Related work

The low-latency requirement for storage system is becoming more and more pressing, and much research effort has been put on employing RAMs to improve performance for disk-based storage systems. Rio Vista (Lowell and Chen, 1997) demonstrated the latency benefits of a DRAM-optimized storage system. Multiple main-memory databases (DeWitt et al., 1984; Garcia-Molina and Salem, 1992), such as Dali (Jagadish et al., 1994), MonetDB (Zukowski et al., 2005), VoltDB (Stonebraker et al., 2010), were proposed to provide high-speed accesses, but their capacities are limited. Multiple key-value stores, such as Riak

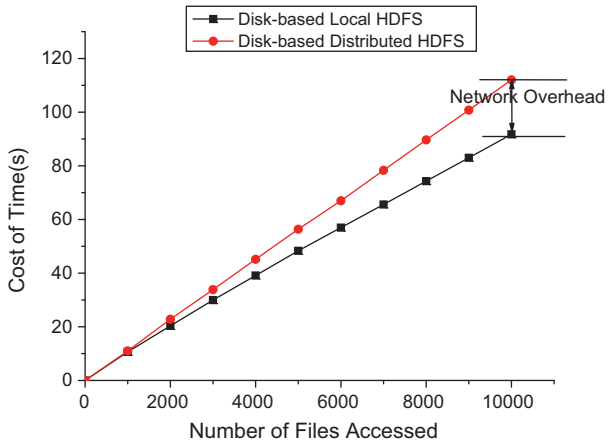

Fig. 5. Network overhead.

(Basho Technologies, 2009), Redis (Sanfilippo and Noordhuis, 2010), Scalaris (Schtt et al., 2008), Tokyocabinet (FAL Labs, 2009), Memcached (Fitzpatrick, 2003), Voldemort (Linked in, 2010), can store data in RAMs to offload back-end storage systems. H-Store (Kallman et al., 2008) operates on a distributed cluster of shared-nothing machines where the data reside entirely in main memory. Bigtable (Chang et al., 2006) allows entire column families to be loaded into memory, where they can be read without any disk accesses. Ousterhout et al. (2011) proposed RAMCloud as a general-purpose online storage system for large-scale Web applications.

RCSS proposed in this paper aims at improving the random I/O performance of the Hadoop distributed file system by employing the memory in a cluster, with which upper-level HDFS-based applications can achieve substantially boosted I/O performance.

7. Conclusions and future work

In this paper, we introduced RCSS that integrates the available memory resources in an HDFS cluster to form a massive RAM-Cloud Storage System by employing disk-based HDFS as the backup facility. By serving data from RAMs, RCSS can achieve high I/O performance with low latency and high throughput for random read accesses. Experimental results validate the efficiency of RCSS.

However, RCSS needs users to manually specify which files should be stored in memory, while this manual selection may be cumbersome. So our future work will focus on implementing an automatic mechanism to enable data migration between disks and RAMs. This mechanism needs to dynamically identify "hot" data and automatically migrate the data from disks into RAMs for fast accesses. RCSS takes a file as the unit of migration and demotion, while a large file may contain only a few "hot" blocks, and fetching the whole file into RAMs may not be optimum, so the dynamic identification and automatic migration of hot data should be implemented at a finer grain, maybe taking data blocks as the units of migration and demotion is a better option.

Acknowledgements

This work was supported by the Research Innovation Program of Shanghai Municipal Education Committee under grant no. 13ZZ003. Jihong Guan was supported by the "Shuguang" Scholar Program of Shanghai Education Development Foundation.

References

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R., 2006. Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th Symposium on Operating System Design and Implementation, Seattle, WA, USA, November.

Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R., 2010. Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing(SOCC), Indianapolis, IN, USA, June.

Dean, J., Ghemawat, S., 2004. MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating System Design and Implementation, San Francisco, CA, USA, December.

DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., Wood, D., 1984. Implementation techniques for main memory database systems. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data(SIGMOD), Boston, MA, USA, June.

Garcia-Molina, H., Salem, K., 1992. Main memory database systems: an overview. IEEE Transactions on Knowledge and Data Engineering 4 (6), 509–516.

Ghemawat, S., Gobioff, H., Leung, S.T., 2003. The Google File System. In: Proceedings of the 19th Symposium on Operating Systems Principles, Lake George, NY, USA, October.

Hadoop. http://hadoop.apache.org

Hbase. http://hbase.apache.org

Jagadish, H., Lieuwen, D., Rastogi, R., Silberschatz, A., Sudarshan, S., 1994. Dali: a high performance main memory storage manager. In: Proceedings of the 20th International Conference on Very Large Data Bases(VLDB), September, Santiago de Chile, Chile.

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D., 2008. H-store: a high-performance distributed main memory transaction processing system. In: Proceedings of VLDB Endow. Vol. 1 No. 2, pp. 1496–1499.

Lowell, D., Chen, P., 1997 October. Free transactions with Rio Vista. In: Proceedings of the 1997 Symposium on Operating Systems Principles (SOSP), Saint-Malo, France.

Memcached. http://memcached.org

Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazires, D., Mitra, S., Narayanan, A., Ongaro, D., Parulkar, G., Rosenblum, M., Rumble, S., Stratmann, E., Stutsman, R., 2011. Communications of the ACM 54 (7), 121–130.

Redis. http://redis.io

Riak. http://basho.com

Schtt, T., Schintke, F., Reinefeld, A., 2008. Scalaris: reliable transactional P2P key/value store. In: Proceedings of the 7th ACM SIGPLAN Workshop on Erlang, Victoria, BC, Canada, September.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. The Hadoop Distributed File System. In: Proceedings of the 26th Symposium on Mass Storage Systems and Technologies, Incline Village, Nevada, USA, May.

Tokyocabinet. http://fallabs.com/kyotocabinet

J. Venner, Pro Hadoop. Apress, June, 2009.

Voldemort. http://project-voldemort.com

VoltDB. http://voltdb.com

Zukowski, M., Boncz, P., NES, N., Heman, S., 2005. MonetDB/X100—a DBMS in the CPU cache. IEEE Data Engineering Bulletin 28 (2), 17–22.

Yifeng Luo received his MS degree in Computer Science from Tsinghua University, China, and BS degree in Computer Science from Harbin Institute of Technology, China. He currently is a PhD student of Fudan University, China, majored in Computer Science. His current research interest focuses on cloud storage and computing.

Siqiang Luo received his Bachelor degree in 2010 from School of Computer Science, Fudan University, China, and he is pursuing his Master degree in the same university. His research interests include distributed computing, spatial keyword search.

Jihong Guan is now a professor at Department of Computer Science & Technology, Tongji University, Shanghai, China. She received her Bachelor degree from Huazhong Normal University in 1991, her Master degree from Wuhan Technical University of Surveying and Mapping (merged into Wuhan University since Aug. 2000) in 1991, and her PhD from Wuhan University in 2002. Before joining Tongji University, she served in the Department of Computer, Wuhan Technical University of Surveying and Mapping from 1991 to 1997, as an assistant professor and an associate professor (since August 2000) respectively. She was an associate professor (Aug. 2000–Oct. 2003) and a professor (Since Nov. 2003) in the School of Computer, Wuhan University. Her research interests include databases, data mining, distributed computing, Bioinformatics, and geographic information systems. She has published more than 100 papers in domestic and international journals and conferences.

Shuigeng Zhou is now a professor at School of Computer Science, Fudan University, Shanghai, China. He received his Bachelor degree from Huazhong University of Science and Technology (HUST) in 1988, his Master degree from University of Electronic Science and Technology of China (UESTC) in 1991, and his PhD of Computer Science from Fudan University in 2000. He served in Shanghai Academy of Spaceflight Technologies from 1991 to 1997, as an engineer and a senior engineer (since August 1995) respectively. He was a post-doctoral fellow in the State Key Lab of Software Engineering, Wuhan University from 2000 to 2002. His research interests include data management, data mining and machine learning, and Bioinformatics. He has published more than 100 papers in domestic and international journals (including IEEE TKDE, IEEE TPDS, IEEE/ACM TCCB, Bioinformatics, BMC Bioinformatics, DKE, PRE, EPL and EPJB etc.) and conferences (including SIGMID, SIGKDD, SIGIR, SODA, VLDB, ICDE, ICDCS, EDBT, BIBM, and BIBE etc.). Currently he is a member of IEEE, ACM and IEICE.