TÜBİTAK

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2018) 26: 936 – 947
© TÜBİTAK
doi:10.3906/elk-1701-222

Concurrency control algorithms for deduplicated cloud storage

Prabavathy BALASUNDARAM*, **Chitra BABU**
Department of CSE, SSN College of Engineering, Chennai, Tamil Nadu, India

Received: 23.01.2017 • **Accepted/Published Online:** 16.07.2017 • **Final Version:** 30.03.2018

Abstract: Deduplication of data is essential to effectively use cloud storage. As the metadata in deduplicated cloud storage are shared across multiple users, concurrent updates may result in inconsistencies. A coarse-grained locking strategy that has been proposed earlier to overcome this difficulty is not suited for inline deduplication owing to poor performance. In the present work, a fine-grained locking strategy that overcomes this shortcoming is proposed. A metadata structure along with a set of concurrent control mechanisms to accomplish this is presented. This strategy is shown to improve the throughput by as much as 60% with only marginal lock overhead.

Key words: Cloud storage, data deduplication, concurrency issues, locking

1. Introduction

Cloud storage is a service model in which data are maintained, managed, backed up remotely, and made available to users through the Internet. Since redundancy is prevalent in cloud storage, it is essential to utilize the storage space optimally. In order to achieve this, a well-known optimization technique, namely, deduplication [1], is commonly used. This technique basically divides every incoming file into a set of fixed or variable sized blocks [2]. Subsequently, secure hash algorithm 1 is used to find the hash value corresponding to each one of these blocks. These hash values, which are also termed fingerprints, are compared against the fingerprint index to detect duplicates and only one instance of these duplicate blocks is stored. Each entry in the fingerprint index corresponds to a specific block that has its fingerprint, location, and reference count. Since each file is stored as a set of blocks in the deduplicated cloud storage (DCS), that entire set of the constituent blocks is needed to reconstruct a file during read operations. In order to facilitate this, the file recipe maintains the fingerprints of these constituent blocks. These constituent blocks may either be unique ones specific to a file or may be shared across multiple files.

Every access to the DCS consults the metadata, namely, the fingerprint index and the file recipe. During the write operation, every incoming block is compared against the entries in the fingerprint index for its existence. If a block already exists in the storage, its reference count alone is incremented by 1. Otherwise, it is placed in the storage and the fingerprint index is updated with the corresponding entry. During a delete operation, the reference count for every block corresponding to that file is decremented by 1. During a read operation, the file recipe is initially consulted to find out the constituent blocks. Subsequently, the locations of these blocks are obtained from the fingerprint index in order to assemble the entire file.

In the context of DCS, the blocks are typically shared among multiple files. As a consequence, concurrent users may simultaneously access an entry in the fingerprint index corresponding to a block. In general, a read

*Correspondence: prabavathyb@ssn.edu.in

request accesses the location information from the block entry to reconstruct a file. A write or delete request increments or decrements the reference counts. Thus, when concurrent requests attempt to update the reference count at the same time, the value of the reference count may become inconsistent. This problem is referred to as “lost update.”

A periodic garbage-collection process removes the physical blocks in the storage whenever their reference counts become 0. Due to the lost update problem and the consequent inconsistency in the value of the reference count, it is possible that one or more blocks may be garbage-collected prematurely while some files continue to refer to them. This would result in a file reconstruction error due to the unavailability of data blocks while trying to assemble the file back during a read operation. Hence, concurrency control is an important research issue that needs to be addressed in the context of DCS. An earlier work [3] proposes block-level postprocessing deduplication for a live cluster file system where the incoming data are deduplicated only after the data have been written into the disk. In this work, the entire shared fingerprint index is locked by a host during the access to its entries. Such a coarse-grained locking strategy is not suited for the context of the present DCS that performs inline deduplication, resulting in an increase in the response time. Hence, in the present work, a novel fingerprint index structure that is amenable to a fine-grained locking approach along with the necessary concurrency control mechanisms for the basic operations have been proposed.

The rest of this paper is organized as follows. Section 2 summarizes the research related to concurrency control mechanisms in DCS. Section 3 describes the proposed fingerprint structure and the concurrency control algorithms. Section 4 provides the implementation details, the metrics utilized, and the results from a detailed performance analysis that substantiate the proposed strategy. Section 5 concludes the paper with possible future research directions.

2. Related work

Existing literature related to deduplication addresses issues such as the management of fingerprint index, effective chunking mechanisms, reliability and placement of blocks in the disk for deduplicated storage systems that perform inline deduplication. AA-Dedupe [4] analyzes the characteristics of the files and apply a suitable chunking mechanism to minimize the metadata overhead. Efficient indexing [5] and scalable hybrid hash cluster [6] have utilized efficient data structures to maintain the metadata. Guo et al. [7] and Efstathopoulos et al. [8] used a sampling technique to obtain a sample subset of metadata from the disk to improve the read throughput. Chunkstash [9] and sparse indexing [10] utilize efficient I/O devices (solid state drive, flash memory) to maintain the fingerprint index in order to improve the performance of the lookup process in deduplication. Data domain file system [11] utilizes different techniques such as bloom filter and segment informed segment layout for the placement of blocks in order to improve the throughput of the deduplication.

Only a few works have investigated the provision of the concurrency control in the deduplicated storage system. Strzelczak et al. [12] proposed a delete algorithm for the content addressable storage. A block can contain pointers to other blocks. These pointers are the block addresses derived from the block content. If a block is to be deleted, then the reference counter of all blocks pointed by it is decremented by 1. The blocks whose reference counters become zero are treated as garbage and the space associated with such blocks is periodically reclaimed. Clements et al. proposed [3] a decentralized deduplication system for storage area networks clusters, which store virtual machine (VM) images. These VM images are stored in a cluster file system without finding duplicates initially. The fingerprint index is stored in a shared disk. During the idle time of CPU, deduplication is carried out by obtaining the lock on the fingerprint index. Hence, concurrent

requests do not cause any issues related to consistency. However, this approach is not suitable in the context of the present DCS, since it performs inline deduplication.

Improved file sharing and file locking [13] describe the design of a file-sharing–enabled cloud storage system. Concurrency issues that arise due to file sharing are handled by means of file locking and write serialization, which utilizes a queue for placing the write requests to ensure that the writes do not conflict with one another. However, this approach is restricted to file sharing in a nondeduplication system. It emerges that issues related to consistency that arise due to deduplication have hitherto not been addressed.

3. Concurrency control for DCS (CCDCS)

In order to enable concurrency control for DCS, a set of concurrency control algorithms has been proposed in this section. These algorithms utilize the proposed fingerprint index structure that supports concurrent requests.

3.1. Proposed fingerprint structures: striped fingerprint index and enriched file recipe

The traditional fingerprint index consists of a set of entries related to each block that is available in the storage. During concurrent requests, while one request is accessing a set of block entries, it is impossible to restrict another request from updating this set either partially or entirely. This leads to inconsistent metadata. In order to avoid this issue of inconsistency, it is essential to lock every block entry corresponding to a particular file to serve a specific request. As a file may have numerous block entries, locking and releasing every block entry involves considerable overhead. Alternately, the entire fingerprint index can be locked in an attempt to maintain consistency of the metadata. However, this completely eliminates any parallelism, resulting in poor performance. Hence, as a trade-off between the excessive locking overhead and the lack of parallelism altogether, it would be better to lock only the relevant block entries corresponding to a particular file. Since the present fingerprint index structure does not facilitate such fine-grained locking, it is essential to reorganize it.

In this context, this paper proposes a striped fingerprint index (SFI) that builds fingerprint index in two levels. The first level index maintains the hash value of the file path and a pointer to the second level index, which contains entries corresponding to the constituent blocks. A stripe represents an entry in the first level index and its corresponding second level index as shown in Figure 1a.

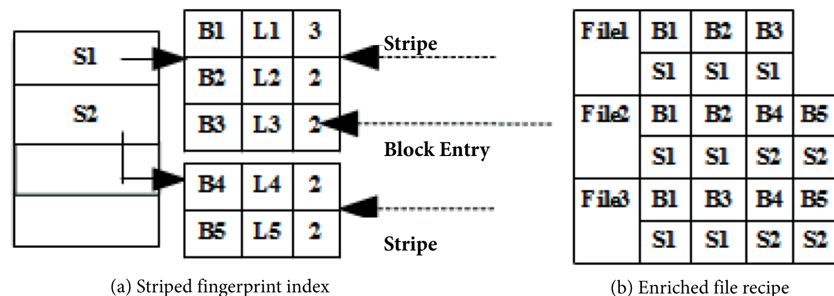

Figure 1. Proposed data structures: a) striped fingerprint index, b) enriched file recipe.

The proposed enriched file recipe (EFR) shown in Figure 1b is a modification of the existing file recipe to support access to the SFI. Since it is essential to obtain the stripe where a constituent block pertaining to a file resides, EFR maintains the stripe information in addition to the fingerprint of every block.

The following example illustrates the proposed indices SFI and EFR. When a file named *File1*, which consists of blocks B1, B2, and B3, needs to be written into the DCS, hash value of the file path (i.e. S1) for *File1* is found and kept in the first level index of the SFI. For every block of a file, the SFI is scanned to determine whether an entry corresponding to that block already exists in the SFI. If there is such a matching entry, the reference count of that entry alone is incremented in the stripe of the file that owns the block. Since these blocks are unique to *File1*, new block entries are created and placed in the second level index of SFI. Similarly, when another file named *File2*, which consists of blocks B1, B2, B4, and B5, arrives, a new stripe is created in SFI with the blocks B4 and B5, since they are the only ones unique to this file. When a file *File3,* which consists of blocks B1, B3, B4 and B5 arrives, a new stripe is not created, since all the blocks constituting this file are already present in the SFI. Hence, reference counts for the corresponding blocks alone are incremented.

3.2. Concurrency control algorithms

This section describes the concurrency control algorithms for the read, write, and delete operations, which are handled by DCS.

3.2.1. Concurrency control algorithm for read request

A read request for any given file consults both the EFR and SFI to reconstruct that file. The proposed algorithm for read request incorporates **Rule 1** to handle concurrency issues that may arise due to content sharing.

Rule 1: A shared lock is obtained on each stripe required by the read request to retrieve the locations of the required blocks.

Rationale: Generally, reading does not require a lock. However, in this context, the shared lock is necessary to prevent other write or delete requests from exclusively locking the same stripe.

Figure 2 illustrates the process for a read request. To read *File2*, the EFR is consulted initially to retrieve the blocks B1, B2, B4, and B5, which constitute that file, and their corresponding stripes S1 and S2. A shared lock is obtained on S1 to retrieve the locations L1 and L2 for the blocks B1 and B2, respectively. This step is repeated for the stripe S2 to retrieve the locations L4 and L5 for the blocks B4 and B5, respectively. Once the locations are available, the blocks are retrieved from the locations L1, L2, L4, and L5 on the disk. They are then assembled according to the order of their occurrence in the EFR to reconstruct the file. Algorithm 1 provides the pseudocode for the read request.

3.2.2. Concurrency control algorithm for write request

During a write operation, blocks of the incoming file are compared against all the stripes available in the storage to determine duplicates.

Algorithm 1 Concurrency control algorithm for read request.

Input: EFR of the file **Result:** Reconstructed file

S := Set of stripes from EFR

for (each stripe $S_i \epsilon$ S) { Obtain Shared-lock (S_i)

Get the locations of all the blocks in EFR Release Shared-lock (S_i) }

Retrieve blocks from storage

Reconstruct the file

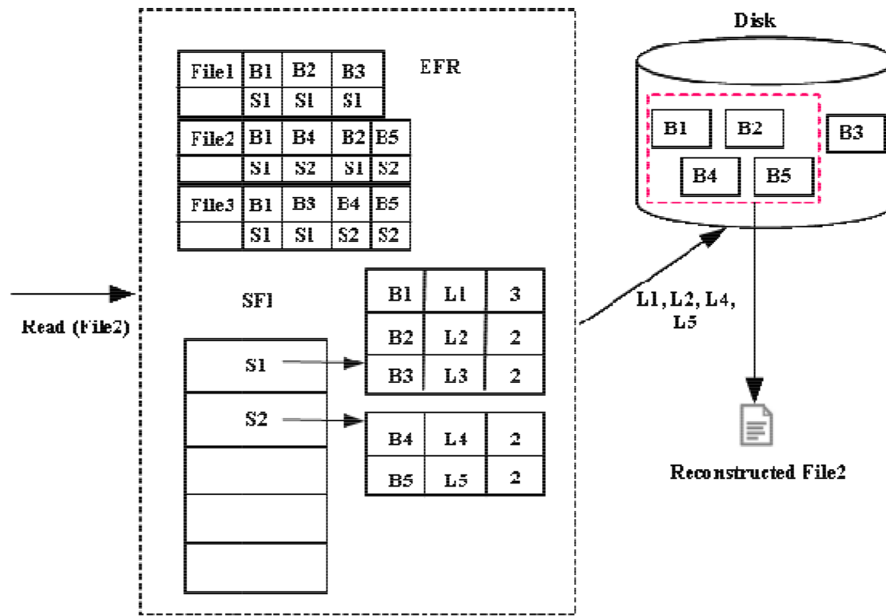

Figure 2. Process for read request.

Rule 2: Each required stripe is locked with an exclusive lock to perform the necessary update on the specific set of block entries. The lock is released once the update is completed.

Rationale: Let us assume two write requests simultaneously try to update the reference count of a particular block entry, for example, B1. Initially, if the reference count is 1, each of the two writes will increment it to 2. However, the actual reference count should have been 3. This will result in a problem when delete requests are issued to two of the three files containing that block. Thus, the reference count for B1 will eventually become 0, while it should actually be 1. The blocks with zero reference counts are removed from the storage by the periodic space reclaim process described in Algorithm 2. Hence, if a read is issued for the remaining file with the block B1, it would result in an error. Thus, it is essential to lock the stripe during a write request in order to avoid a file reconstruction error.

The write request is handled in two ways, depending on whether the file already exists in the system or not.

Case 1: Existing file

Let us assume that an already existing file *File1* in the storage is updated. The blocks of the new and old versions of *File1* need to be compared to find out which blocks have become stale. Based on this information, the reference counts are correspondingly updated.

For example, the old version of *File1* consists of blocks B1, B2, and B3, and the new version of *File1* consists of blocks B1, B2, B4, and B12. The old EFR and the new EFR are compared to determine a set of stale and new blocks. Since B3 in the old set is not a part of the new version of *File1*, its reference count should be decremented. The new block B12 has been added to the existing stripe corresponding to *File1*. The reference counts of the blocks B4 and B12 that belong to the new set should be incremented. Subsequently, the old EFR of *File1* is deleted and the new EFR is written into the storage. During the entire update, an exclusive lock is maintained on S1. Figure 3a shows the procedure for such an update request.

Case 2: File does not already exist in the system

If a file named *File4* is written into the storage, it is first split into a set of blocks B1, B2, B7, and B8.

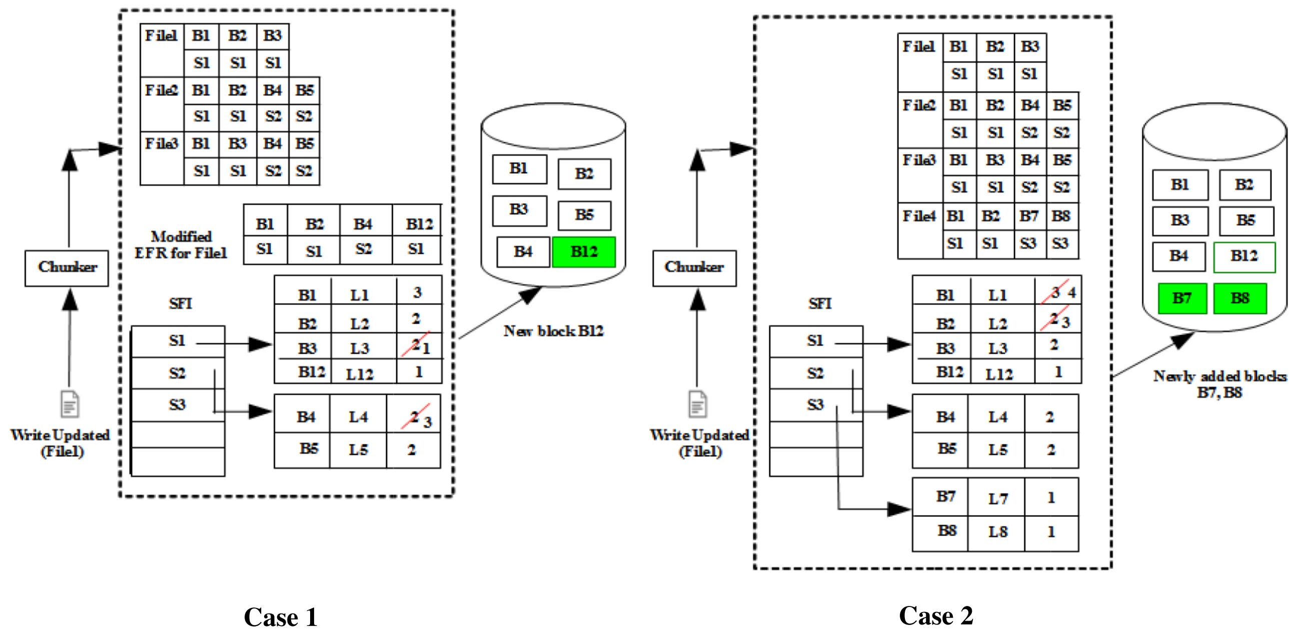

Figure 3. Process for write request: a) Case 1: existing file, b) Case 2: new file.

These blocks need to be checked for their presence among the stripes that belong to the SFI. In this case, S1 is the only such stripe. Hence, S1 is exclusively locked, and the reference counts of B1 and B2 are incremented and then the lock is released. Further, B7 and B8 are new blocks. Hence, a stripe is created for *File4* with B7 and B8 inserted into the second level index as shown in Figure 3b Algorithm 2 illustrates the above two cases with the incorporation of **Rule 2**.

3.2.3. Concurrency control algorithm for delete request

The delete request reads the EFR to determine the list of blocks whose reference counts have to be decremented. Algorithm 3 provides the pseudocode for a delete request by incorporating **Rule 2**.

Rationale: Let us consider a block B1 with the reference count 1. If there are simultaneous write and delete requests to that block, the reference count is read by both the requests as 1. Subsequently, the write request updates the reference count value to 2, and the delete request updates it to 0. Depending on the order of the updates, the reference count can be 2 or 0, while the actual value should have been 1. This incorrect update of reference count might lead to a premature removal of the block by the space reclamation process. Consequently, a file reconstruction error will occur for the read requests that involve this block B1. Hence, it is essential to lock the required stripes during the delete request.

Figure 4 depicts the process of a delete request with an example. If *File2*, consisting of the blocks B1, B4, B2 and B5, has to be deleted, the EFR of that file is first checked to determine the list of stripes which are to be accessed. In this case, the stripes to be accessed are S1 and S2. An exclusive lock is obtained on S1 to decrement the reference counts of the blocks B1 and B2. The process is repeated for S2 to decrement the reference counts for the blocks B4 and B5.

4. Experimental evaluation

This section details the experimental setup, metrics for evaluation, and the results of the experiments.

Algorithm 2 Concurrency control algorithm for write request.

Input: File to be written (*file*), Set of blocks that correspond to *file*

Result: Updated SFI and EFR

B := Set of blocks to be written N := Set of all stripes from SFI

if (file exists) { B_1 := Blocks in old EFR

    B_{new} := B - B_1 // Set of blocks that exist in B but not in B1

    B_{old} := B_1 - B // Set of blocks that exist in B1 but not in B }

else { B_{new} := B }

while (N is not empty) {   /* Find a stripe which is not locked by another request */

    if(some $S_i \epsilon$ N is not locked){ Obtain Exclusive-lock (S_i)

        for (each block B_j in B_{new})

            if(S_i has a block B_j) {

                increment reference count of B_j by 1

                remove B_j from B_{new}} }

    for (each block $B_k \epsilon B_{old}$) { if(S_i has a block B_k) {

            /* As block is no longer part of new file, the reference count needs to be decremented */

            decrement reference count of B_k by 1

            remove B_k from B_{old}}

        release the Exclusive-lock(S_i) }

Remove S_i from N }

if(B_{new} is not empty) {

    if(stripe does not already exist)

        Create a new stripe for the file.

    Insert the block entries corresponding to B_{new} into the existing or new stripe.

    Write the data blocks into the storage }

if (file exists) Update the EFR else Create EFR

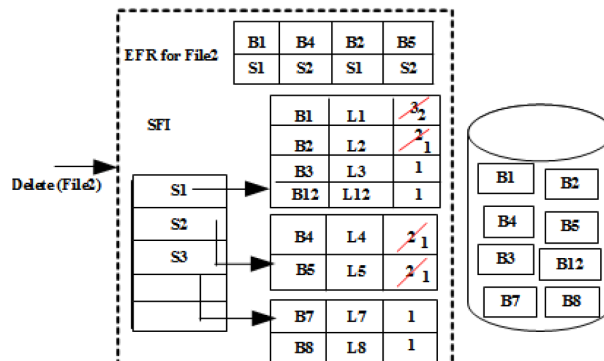

Figure 4. Process for delete request.

Algorithm 3 Concurrency control algorithm for delete request.

```
Input: EFR of the file          Result: Updated SFI
S := Set of stripes obtained from EFR          B := Set of blocks obtained from EFR
while(S is not empty) {          /* Find a stripe which is not locked by another request */
    if (some S_i ϵ S is not locked) { Obtain Exclusive-lock(S_i)
        for(each B_j ϵ B)
            if (S_i has chunk B_j) {
                Decrement the reference count of B_j by 1
                Remove B_j from B }
        Release the Exclusive-lock(S_i) }
    Remove S_i from S }
    Delete the EFR
```

4.1. Experimental setup

Private cloud storage was built using a set of four commodity machines with an open source framework, namely, Eucalyptus [14]. This framework has several components: cloud controller (CLC), cluster controller, node controller, walrus, and storage controller. CLC and walrus are installed in one of the commodity machines with a configuration of Intel Core i7 2.8 GHZ, 4 GB RAM DDR3, and 800 GB HDD. The CLC is responsible for transferring the requests from users to different components of the cloud.

Node controllers are installed in three other commodity machines with a configuration of Intel Core i5 2.8 GHZ, 4 GB RAM DDR3, 500 GB HDD for maintaining the data blocks. Two of these commodity machines are designated to maintain the striped fingerprint indices corresponding to less mutable (LM) and more mutable (MM) types of files, respectively, along with the data blocks. Each SFI has been implemented using the concurrent map of MapDB library. It helps to create an on-disk implementation of the map data structure. Further, it also supports concurrent accesses to it by several parallel threads. Two maps have been utilized to create the SFI. The first-level map contains the file path as key and the second-level map contains the location of the block object, which is the abstraction of the block entry. The walrus component controls the storage, retrieval, and deletion of data in the cloud through put, get, and delete APIs, respectively. The implementations of these APIs without incorporating deduplication are available in putObject(), getObject(), and deleteObject() methods of WalrusManager.java, which resides in the **/edu/ucsb/eucalyptus/cloud/ws** directory.

The put API has been modified to incorporate deduplication process, which divides the stream of data corresponding to a file into a set of fixed or variable sized blocks. Further, fingerprints for these blocks are found by utilizing MessageDigest class. Since walrus knows the addresses of the node controllers that contribute storage and compute resources to the cloud, these fingerprints and data blocks are sent to their respective nodes. Further, the concurrency control algorithm for write request given in Algorithm 2 also has been implemented. The source code was modified using Eclipse SDK 4.3. Apache Ant was used to build this rewritten walrus module to generate eucalyptus-walrus.jar file in the target directory. This newly generated .jar file has been placed in the folder ***/usr/share/eucalyptus/*** by replacing the old file. Subsequently, the services of Eucalyptus are restarted. Once the hosts are up, the put API is invoked to call the modified putObject() method. Similarly,

getObject() and deleteObject() were suitably modified to realize the concurrency control algorithms for read and delete requests given in Algorithms 1 and 3, respectively. Though the cloud has been set up with only four machines, it is possible to scale this environment to a maximum of 256 machines [14].

4.2. Metrics for evaluation

The proposed CCDCS was compared with DCS, which utilizes the basic algorithms for primitive operations, namely, read, write and delete, without involving concurrency control by utilizing a set of common evaluation metrics, namely, the response time, lock overhead, and throughput.

Response time: Time taken to respond to a request.

Lock overhead: Overhead involved in locking and unlocking the stripe for accessing the block entries.

Throughput: Number of requests serviced per unit time. The higher the throughput, the better the performance.

4.2.1. Effect of concurrency control algorithms on response time

A tool called Agent Ransack was used to collect information about real trace of data pertaining to 100 personal workstations to detect the types of files they utilize, and the minimum, average, and maximum size of each type of file. The types of files utilized were .doc, .ppt, .pdf, .txt, .rar, .exe, .mkv, and .mp3 (to name a few) and the minimum, average, and maximum sizes for some types of files are listed in the Table. These files can be categorized as LM and MM.

Table. Summary of types of files.

File type	Min. size	Avg. size	Max. size
DOC	68 K	180 K	1 M
TXT	26 K	606 K	800 K
PPT	126 K	976 K	2 M
PDF	216 K	403 K	2 M
EXE	142 K	298 K	466 K
RAR	876 K	2 M	4 M
MP3	2.6 M	5 M	6 M
MKV	397 M	691 M	1.88 G

LM files: Files that are less prone to modification are less mutable files. Files of these types (.rar, .exe) are known as target types. That is, a user can change the content of these files only by making changes in their source files. Hence, changes made in these files will be less and fixed-size chunking is used to detect duplicates.

MM files: Files that are more prone to modification belong to this category. As the user might change the contents frequently in these types of files (.ppt, .doc, .odp, .odt, .txt), variable-sized chunking is used to detect duplicates.

A representative file of average size has been considered for both LM and MM. The read, write, and delete requests were generated for both DCS and CCDCS to measure the response time for each request. This experiment was repeated 5 times and the average response times were noted and plotted as shown in Figures 5a–5c. The number of blocks generated will be more, as variable-sized chunking is utilized for the MM file. Hence, the average response time for the MM files is longer than that of the LM files. Since CCDCS utilizes lock-based concurrency control algorithms, every request locks and unlocks the stripes that are required for it. Hence, the average response time for any operation with respect to CCDCS is slightly longer than that of DCS.

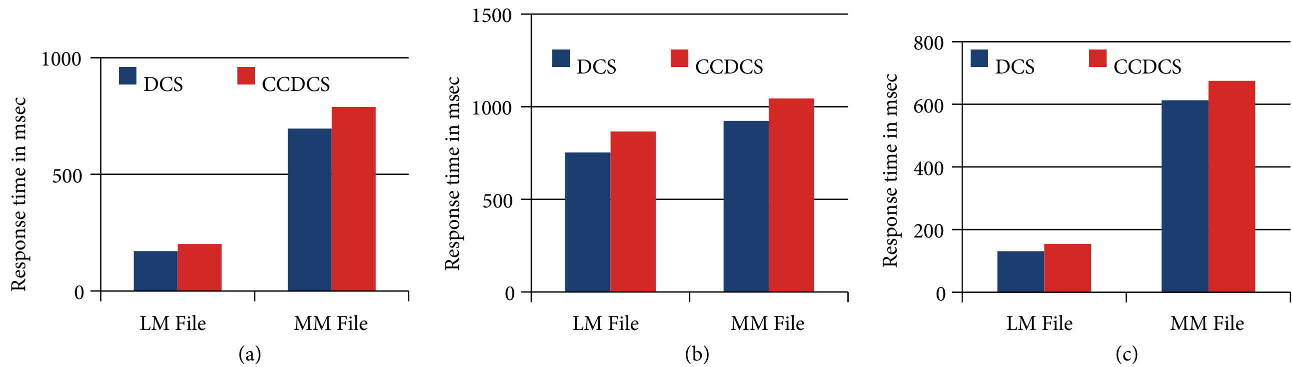

Figure 5. Response time: a) read request, b) write request, c) delete request.

4.2.2. Effect of concurrency control algorithms on lock overhead

A **WorkloadA** that consists of LM files and **WorkloadB** that consists of MM files were considered for this experiment. While LM files are split into a set of fixed size blocks, the MM files are split into a set of variable sized blocks. MM files are frequently modified by the user and this leads to several versions of the same file being stored. These versions share common content. Thus, the number of blocks to be locked is more for MM files than for LM files. The locking overhead is proportional to the time involved in locking and unlocking the stripes, as shown in Figure 6.

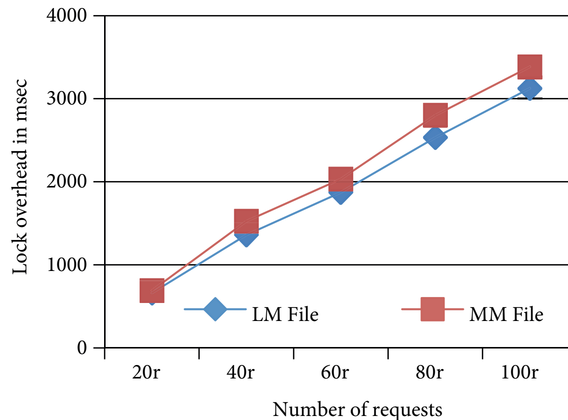

Figure 6. Lock overhead.

4.3. Effect of concurrency control algorithms on throughput

An experiment was conducted to find the throughput in terms of number of requests served per minute for the proposed CCDCS and DCS. It was found by performing load testing. One hundred concurrent requests were generated by using parallel threads. Any request to the DCS needs to contact the SFI. Since SFI supports concurrent access, each request obtains locks on the required stripes according to the algorithms and performs the necessary actions. This experiment considered two workloads, namely **WorkloadC** and **WorkloadD**. The former consisted of LM files with an average file size of 456 KB while the latter consisted of MM files with an average files size of 358 KB. The getObject() method involves reading file recipe, block retrieval, and file reconstruction. A count variable is incremented once this process is completed. A *timeThreshold* variable was set to current time plus 60,000 ms. The count value is noted when the current time reaches *timeThreshold*. The

number of concurrent requests is varied from 20 to 100 to note the read, write, and delete throughput. These experiments were executed five times to determine the average number of requests served and this is plotted in Figure 7a. Similarly, the number of write and delete requests served per minute were determined and plotted in Figures 7b and 7c. MM files generate more blocks than LM files owing to the use of variable-sized chunking. Consequently, the time involved in locking and accessing the blocks is also longer for the MM files. Hence, the throughput for LM files is more than MM files.

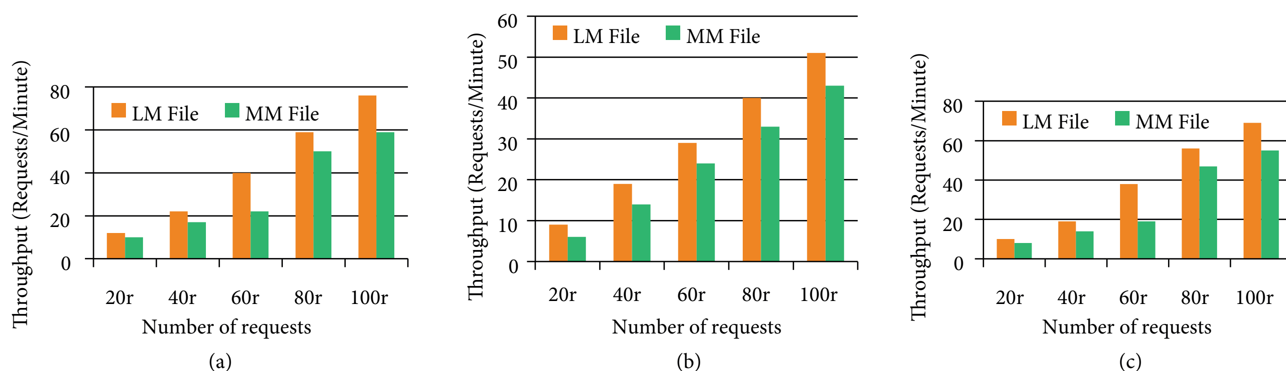

Figure 7. Throughput: a) read request, b) write request, c) delete request.

5. Conclusions

The proposed CCDCS was built by utilizing commodity machines. Primitive operations, namely read, write, and delete operations, were implemented using algorithms 1, 2, and 3, respectively. This helps in allowing concurrent users to access the CCDCS without compromising the consistency of the fingerprint index. Multiple experiments were conducted to compare the performances of the proposed CCDCS and the DCS in terms of lock overhead.

The results clearly demonstrate that the average lock overhead for any file in CCDCS was 14% greater than DCS. Further, the average throughput for CCDCS was increased by about 60% when compared to that of DCS. In this context, though the time taken for executing each request is slightly higher due to the overhead associated with locking, the throughput increases significantly as the system facilitates concurrency.

References

[1] Zeng W, Zhao Y, Ou K, Song W. Research on cloud storage architecture and key technologies. In: 2nd IEEE 2009 Interaction Sciences Information Technology, Culture and Human International Conference; 24–26 November 2009; Seoul, Republic of Korea: IEEE. pp. 1044-1048.

[2] Policroniades C, Pratt I. Alternatives for detecting redundancy in storage systems data. In: USENIX 2004 Annual Technical Conference; 27 June–2 July 2004; Boston, MA, USA: Usenix Association. pp. 73-86.

[3] Clements AT, Ahmad I, Vilayannur M, Li J. Decentralized deduplication in SAN cluster file systems. In: 11th USENIX 2009 Annual Technical Conference; 14–19 June 2009; Santa Clara, CA, USA: Usenix Association. pp. 101-114.

[4] Fu Y, Jiang H, Xiao N, Tian L, Liu F. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In: IEEE 2011 Cluster Computing International Conference; 26–30 September 2011; Austin, TX, USA: IEEE. pp. 112-120.

[5] Thwel T, Thein NL. An efficient indexing mechanism for data deduplication. In: IEEE 2009 Current Trends in Information Technology Conference; 15–16 December 2009; Dubai, United Arab Emirates: IEEE. pp. 1-5.

[6] Xu L, Hu J, Mkandawire S, Jiang H. SHHC: A scalable hybrid hash cluster for cloud backup services in data centers. In: 31st IEEE 2011 Distributed Computing Systems Workshops International Conference; 21–24 June 2011; Minneapolis, MN, USA: IEEE. pp. 61-65.

[7] Guo F, Efstathopoulos P. Building a high performance deduplication system. In: USENIX 2011 Annual Technical Conference; 15–17 June 2011; Portland, OR, USA: Usenix Association. pp. 1-14.

[8] Efstathopoulos P, Guo F. Rethinking deduplication scalability. In: 2nd USENIX 2010 Hot Topics in Storage and File Systems Conference; 22–25 June 2010; Berkeley, CA, USA: Usenix Association. pp. 1-5.

[9] Debnath B, Sengupta S, Li J. ChunkStash: Speeding up inline storage deduplication using flash memory. In: USENIX 2010 Annual Technical Conference; 22–25 June 2010; Boston, MA, USA: Usenix Association. pp. 1-15.

[10] Lillibridge M, Eshghi K, Bhagwat D, Deolalikar V, Trezis G, Camble P. Sparse indexing: Large scale inline deduplication using sampling and locality. In: 7th USENIX 2009 File and Storage Technologies Conference; 24–27 February 2009; San Francisco, CA, USA: Usenix Association. pp. 111-123.

[11] Zhu B, Li K, Patterson H. Avoiding the disk bottleneck in the data domain deduplication file system. In: 6th USENIX 2008 File and Storage Technologies Conference; 28–29 February 2008; San Jose, CA, USA: Usenix Association. pp. 1-14.

[12] Strzelczak P, Adamczyk E, Herman-Izycka U, Sakowicz J, Slusarczyk L, Wrona J, Dubnicki C. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In: 11th USENIX 2013 File and Storage Technologies Conference; 13–15 February 2013; San Jose, CA, USA: Usenix Association. pp. 161-174.

[13] Sundarrajan R, Neelamegam K, Prabagaran VT. Improve file sharing and file locking in a cloud. In: IBM 2010 White Paper; IBM. pp. 1-22.

[14] Nurmi D, Wolski R, Grzegorczyk C, Obertelli G, Soman S, Youseff L, Zagorodnov D. The eucalyptus open-source cloud-computing system. In: 9th IEEE/ACM 2009 Cluster Computing and the Grid International Symposium; May 18 2009; TBD Shanghai, China: IEEE. pp. 124-131.