Noname manuscript No.
(will be inserted by the editor)

YuruBackup: A Space-Efficient and Highly Scalable Incremental Backup System in the Cloud *

Quanqing Xu # · **Liang Zhao** · **Mingzhong Xiao** · **Anna Liu** · **Yafei Dai**

Received: date / Accepted: date

Abstract In this paper, we present YuruBackup, a space-efficient and highly scalable incremental backup system in the cloud. YuruBackup enables fine-grained data de-duplication with hierarchical partitioning to improve space efficiency to reduce bandwidth of both backup and restore processes, and storage costs. On the other hand, YuruBackup explores a highly scalable architecture for fingerprint servers that allows to add one or more fingerprint servers dynamically to cope with increasing clients. In this architecture, the fingerprint servers in a DB cluster are used for scaling writes of fingerprint catalog, while the slaves are used for scaling reads of fingerprint catalog. We present the system architecture of YuruBackup and its components, and we have implemented a proof-of-concept prototype of YuruBackup. By conducting performance evaluation in a public cloud, experimental results demonstrate the efficiency of the system.

Keywords Incremental Backup · Data De-duplication · Cloud Storage

1 Introduction

In recent years, cloud storage services become increasingly popular since they provide high reliability and scalability at relatively low cost. Also, cloud storage provider-

Quanqing Xu (✉)
Data Storage Institute, A*STAR, Singapore
E-mail: Xu_Quanqing@dsi.a-star.edu.sg

Liang Zhao · Anna Liu
National ICT Australia, Sydney, Australia

Mingzhong Xiao
Beijing Normal Univeristy, Beijing, China

Yafei Dai
Peking University, Beijing, China

* Yuru means cloud in Australian aboriginal language.

The work was mostly done while the author was with NICTA and visiting Peking University.

s offer ultra large-scale storage space, e.g., there were 905 billion objects in Amazon S3 in the first quarter 2012[1]. Customers need to backup and restore their progressively huge amounts of data to and from cloud storage providers within rather short time. On the other hand, continuously growing volume of data has raised a critical challenge for effectively and efficiently incremental backup through large-scale and high-performance backup systems. Many cloud providers build cloud-scale distributed storage systems, such as Amazon S3, typically consisting of ten thousands of storage nodes. They offer us an excellent chance to construct a large-scale backup system for customers over their cloud storage systems in which the backup system is able to back up at least petabytes of data.

In order to meet the fast-growing demand on backup capacity and performance, the backup systems thus must adopt effective solutions to enhance both storage efficiency and system scalability. Cloud providers that support online storage and long-term archives generally have the need to offer high-performance storage services to millions of users, such as archiving trillions of files for many years. A new generation of backup system over cloud storage has to perform well in system scalability, which makes sure that the backup system is able to meet the progressive demands of backup from a large number of clients. Lack of scalability will eventually make the backup system hard to survive. Also, the vital step for these new backup systems is to use the elasticity mechanism of cloud, such as varying EC2 instances as fingerprint servers with the number of clients.

Data de-duplication technology has emerged as a key solution to space efficiency problems of both storage and bandwidth intensive incremental backup systems [28]. Early de-duplication systems performed mostly file-level or block-level de-duplication by putting a lot of effort into optimizing duplicate lookup. By eliminating duplicate data across the system, a data de-duplication backup system can achieve far more efficient data compression than other backup systems. There are increasingly redundant replicas of data including file-level and block-level, which may be stored in a single machine or a storage cluster across many machines. De-duplication systems take advantage of data redundancy to reduce the underlying space, in which de-duplication working at block-level is much better than that working at file-level [11]. Data de-duplication has two typical advantages including 1) reducing cost, and 2) increasing space-efficiency, e.g., reducing network bandwidth in LBFS [13] and decreasing storage space in Deep Store [27].

In this paper, we present a space-efficient and highly scalable incremental backup system in the cloud named YuruBackup. In YuruBackup, file/chunk (item) fingerprint hash values are calculated in-line, i.e., if it spots an item that has been stored it does not store the new item, just referring to the existing one. However, in-line de-duplication might be slower thereby reducing the backup throughput because hash calculations and lookups take so long. To decrease both backup and restore time, YuruBackup combines source-side de-duplication with target-side de-duplication. Source-side de-duplication ensures that data is deduplicated on the data source, which generally takes place directly within a file system. New files will be periodically scanned for creating hashes and comparing them to hashes of existing

[1] http://aws.typepad.com/aws/2012/04/amazon-s3-905-billion-objects-and-650000-requestssecond.html

items. When items with the same hashes are found, the item copy is removed and the new one points to the old one. The de-duplication process is transparent to the users and backup applications. Target-side de-duplication is the process of removing duplicates of data in the target store. We confirm the effectiveness and efficiency of YuruBackup by conducting an extensive experimental study, and the performance is measured by de-duplication effectiveness and overhead, scalability measurement on fingerprint server cluster, backup performance, and restore performance.

The rest of the paper is organized as follows. Section 2 presents the system architecture of YuruBackup. We describe the *Backup Agent* running in clients in Section 3, and the *Fingerprint Agent* running fingerprint servers in Section 4. In Section 5 we present performance evaluation results of YuruBackup. Section 6 describes related work. In Section 7 we conclude this paper.

2 System Architecture

To develop a new generation of data backup system in the cloud, there are three vital technical challenges that we have to address: 1) increasing scalability to accommodate growing amounts of data; 2) improving space efficiency to reduce costs; and 3) saving bandwidth as efficiently as possible to adapt to the low bandwidth of WAN.

2.1 Architectural overview

To present YuruBackup clearly, we first describe its system architecture briefly, as shown in Figure 1. *Backup Agent* is distributed and installed on client machines, while *Fingerprint Agent* is located in fingerprint servers in private or public data centers. They communicate with each other via an RPC-based component named *Task Agent*. Individual clients maintain their own local fingerprint catalog, while fingerprint servers maintain global fingerprint catalog including chunks and files. In order to make clients run lightly, they utilize a high-performance embedded database BerkeleyDB, while fingerprint servers use a full-featured database system MySQL with master-slave replication over a DB cluster. Since YuruBackup never relies on the correctness of file/chunk fingerprint catalog, it also does not care about whether the fingerprint catalog is available.

YuruBackup avoids any synchronous updates of fingerprint catalog between clients and fingerprint servers, and fingerprint servers reply to clients before inserting new files/chunks into the fingerprint catalog. If the global fingerprint catalog in fingerprint servers loses some hashes of items, clients will simply utilize more bandwidth until the fingerprint catalog comes up to date. The fingerprint server just creates the fingerprint catalog and populates it as clients access files when running without a fingerprint catalog at the beginning. There is a cluster of fingerprint servers running in a private or public cloud for global target de-duplication [18], and *Fingerprint Agent* is deployed in each fingerprint server in 4.3. *Backup Agent* is deployed in each client, which directly backs up its data including file data and metadata to a cloud storage system, e.g., Amazon S3.

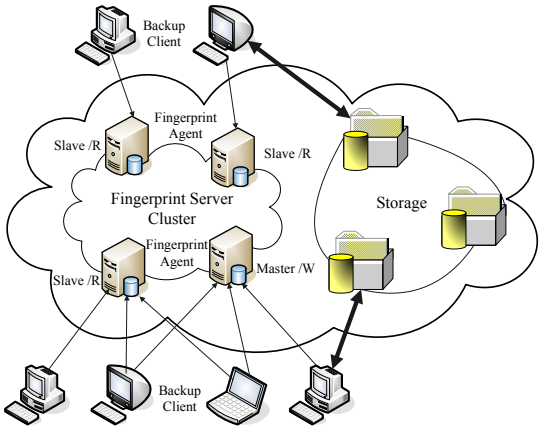


Fig. 1 System Architecture

2.2 Storage Hierarchy

As shown in Figure 2, the storage hierarchy of YuruBackup physically includes 1) chunk that is the smallest storage unit, 2) block that includes a number of continuous chunks with a tag "new" or "old", and 3) collection that consists of a group of blocks. A snapshot consisting of a number of collections is built dynamically according to its description file, and it is an incremental backup providing a point-in-time backup of data, where only the revised sections of a file have been stored since its last snapshot, e.g., if there is a 10 GB file and only 100 MB data has been changed since the last backup, only the 100 MB of changed data will be stored. If a snapshot expires, only its description file and related data are deleted, which is not involved in other snapshots.

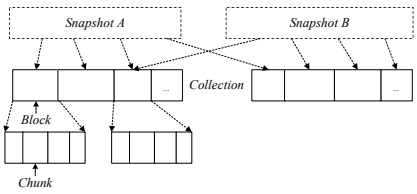


Fig. 2 Storage Hierarchy

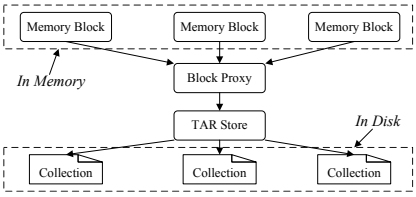


Fig. 3 Mapping blocks from memory to disk

As for organizing blocks in memory and disk, two components *Memory Block* and *TAR Store* are used, as shown in Figure 3. *Memory Block* is utilized to represent a block residing in memory, and it can be built incrementally by organizing continuous new chunks. Finally, it is written into a collection residing in disk using the following two components: *Block Proxy* and *TAR Store*. To facilitate restore without fingerprint servers, there are two types of blocks: data and metadata, to be stored in the same snapshot namespace. *Block Proxy* is employed to manage memory blocks, where a block is represented by a five-tuple: <collectionUuid, blockNo, checksum, start, length>, where a collection is represented by an Uuid, a block has a sequence No.

in the collection and a $checksum$, $start$ and $length$ are the start and length of the block in the collection. Every memory block is mapped to a unique *Block Proxy* that is leveraged to read the metadata of a block in a collection.

Tar is a popular file format[2] used in backup systems, and it can store many files in one tar archive. Files are appended one after the other, and file data is written and kept to be constant except that its length comes up to a multiple of 512 bytes. Each Tar file is preceded by a header containing information including user and group permissions, and dates, and Tar files are compressed as a whole. We thus designed and implemented a component named *TAR Store* that is used to store one or more blocks into a collection. If the collection's size is not less than a given size, it will be written into a tar file by the component to organize blocks efficiently. To avoid the collision of collection names without central authority, randomly generated 128-bit UUIDs are assigned to collections.

2.3 Backup Process

Table 1 lists symbols and their meanings utilized in our algorithms. For a given directory, the backup process reads as shown in Algorithm 1. Firstly, we perform initialization operations for backing up a directory (lines 1-2). There is a loop to deal with all files in the directory (lines 4-9), where a function is used to write a file's incremental backup into one or more collections (line 6), as shown in Algorithm 2. We release resources and close DB connection to the fingerprint catalog (lines 11-12).

Table 1 Symbols and their meanings

Symbols	Meanings
Fp_f/Fp_c	a file f's/chunk c's fingerprint
If_l/Ic_l	local file/chunk *Index Summary*
Ff_l/Fc_l	local file/chunk fingerprint catalog
If_g/Ic_g	global file/chunk *Index Summary*
Ff_g/Fc_g	global file/chunk fingerprint catalog

In file systems, most accesses to files are read-only, which means most files are never changed after their creations [14]. In Microsoft's five-year file metadata study [1], more than two thirds of files have not been modified since they were initially copied into file systems. Most of duplicate files can be found in the local file *Index Summary* that is discussed in Section 3.2 and fingerprint catalog, and the remaining duplicate files will be found in the global file *Index Summary* and fingerprint catalog (line 1), where they are referred to the existing files (line 16). For a changed file, its content-defined chunks are got easily, if a chunk is old it is just referred to the existing chunk (line 8), and it is processed if it is new for the local chunk *Index Summary* and fingerprint catalog (lines 5-6). If chunks are new for the global chunk *Index Summary* and fingerprint catalog, they are processed in a batch (lines 10-12); otherwise, they are referred to the existing chunks in the cloud storage system.

[2] http://en.wikipedia.org/wiki/Tar_(file_format)

Algorithm 1: Backup a directory

Input: A given directory D
Output: A snapshot S

```
1  Create DB connection to fingerprint catalog;
2  Initialize the Metadata Manager M and the TAR store T;
3  Scan D to get a file list L;
4  while L ≠ ∅ do
5      F = L.front();
6      Write F's incremental backup into T;
7      L.pop_front();
8      if T's size >= a given size then
9          Write T into S and clear it;
10 Write T into S; // Process the rest of files
11 Release the Metadata Manager M and the TAR store T;
12 Close DB connection to fingerprint catalog;
13 return S;
```

Algorithm 2: Write a file incremental backup

Input: A given file f
Output: A TAR Store T

```
1  if (Fp_f ∉ If_l || Fp_f ∉ Ff_l) && (Fp_f ∉ If_g || Fp_f ∉ Ff_g) then
2      Compute f's Content-defined chunks C;
3      for c ∈ C do
4          S = ∅;
5          if Fp_c ∉ Ic_l || Fp_c ∉ Fc_l then
6              Insert Fp_c into S;
7          else
8              Refer to the existing chunk;
9
10     Get S' = {c|c ∈ S, (Fp_c ∉ Ic_g||Fp_c ∉ Fc_g)};
11     Store cs (∀c ∈ S') into T in batch;
12     Insert batched Fp_cs(∀c ∈ S') into Ic_l/Ic_g, Fc_l/Fc_g;
13     Refer to the existing chunks (∀c ∈ S − S');
14     Insert Fp_f into If_l/If_g and Ff_l/Ff_g;
15 else
16     Refer to the existing file;
17 return T;
```

2.4 Restore Process

The restore operation is shown in Algorithm 3. For a given snapshot S, its file list F is retrieved from a cloud storage system (line 1). For each file f in F, the local file *Index Summary* If_l and fingerprint catalog Ff_l are exploited to determine if f can be obtained from the client, where most files can be quickly determined if it exists because *Index Summary* is a Bloom filter without disk I/O (lines 3-5). For the remaining files in $F - F'$, the chunk list C of them is retrieved from S's description file (line 7). For each chunk c in C, the local chunk *Index Summary* Ic_l and fingerprint catalog Fc_l are used to decide whether it exists (line 10). In order to reduce bandwidth

consumption, all the chunks in $C - C'$ are transferred from the remote cloud storage system to the client in batch mode (line 13). The worst restore performance is on the newest backup since it typically contains the most pointers, while the best restore performance is found on the oldest data, where there are the fewest pointers.

Algorithm 3: Restore a snapshot

Input: A given snapshot S
Output: A restore R

```
1  Retrieve the list F of S's all files;
2  F' = ∅;
3  for f ∈ F do
4      if (Fp_f ∈ If_l) && (Fp_f ∈ Ff_l) then
5          F' += f; // Get f from the local
6  Load F' into R;
7  Retrieve the list C of all chunks for files F − F';
8  C' = ∅;
9  for c ∈ C do
10     if (Fp_c ∈ Ic_l) && (Fp_c ∈ Fc_l) then
11         C' += c; // Get c from the local
12 Get all chunks in C − C' from cloud storage in batch;
13 Load all the chunks into R;
14 return R;
```

3 Backup Client

Backup Agent as a client program provides a functional interface such as backup and restore to users. It is responsible for scanning files, and archiving/restoring them to/from a cloud storage system for backups and restores. It is shown in Figure 4.

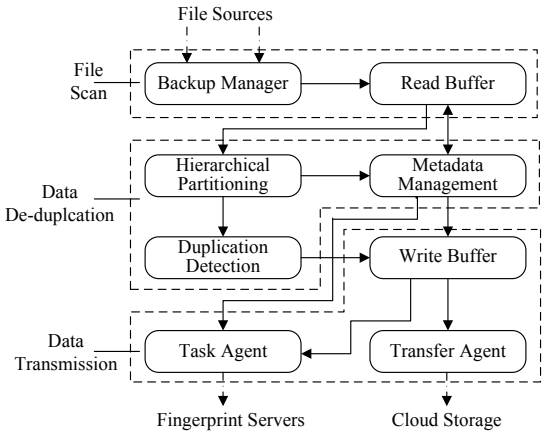


Fig. 4 Backup Agent

3.1 File Scan

Given a directory that is backed up, *Backup Manager* creates a database connection to fingerprint catalog, initializes *TAR Store* and *Metadata Manager*. For each backup, a read buffer in memory is allocated to temporarily buffer some files of the directory. When the read buffer is full, the data de-duplication module will be invoked to deal with the files in the read buffer. By doing so, the number of costly I/O requests to disk will be increasingly reduced. This cycle is repeated until the entire directory has been incrementally stored to disk. Note that *Backup Manager* supports parallelism that multiple threads can concurrently perform the backup function. However, the performance of the backup operation is actually affected by other two bottlenecks: the read/write throughput from/to disk.

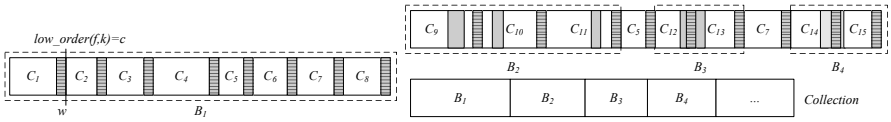
3.2 Data De-duplication

To improve duplicate coverage, de-duplication techniques that perform content-aware chunk boundary calculation have been investigated in many storage systems [28,4]. Such variable-size chunking algorithms leverage different variations of byte-level approaches [13] to improve space efficiency.

3.2.1 Hierarchical Partitioning

To minimize storage overhead, duplicated data across different versions of files must be found quickly, where the files may be in a single client or across multiple clients. The challenge is to find duplicated data across the different versions of files without the knowledge of the underlying structure. We propose a content-defined chunking based hierarchical partitioning approach to efficiently organize chunks, as shown in Figure 5. Content-defined chunking (CDC) [13] can easily deal with it by identifying boundary regions using Rabin fingerprints [15]. Its basic mechanism is shown in Figure 5(a). For each overlapping w-byte sub-string in a file F, the procedure calculating its fingerprint reads as follows. If the low-order k bits of the fingerprint f match a predetermined value c (i.e., $low\_order(f, k) = c$), the offset is marked as an anchor. By doing it, the file F is divided into variable-length chunks by anchors. In YuruBackup, SHA-1 hash values of chunks' contents are utilized to name these chunks that form the basis of file structures to easily share data among different backups.

Figure 5(b) shows how hierarchical partitioning works and what happens to chunk boundaries after a series of edit operations, which are represented by gray shading. Bytes are inserted in chunks c_1 and c_4, producing new and larger chunks c_9 and c_{11}. After a modification in which the anchor is eliminated, chunks c_2 and c_3 of the old file are combined into a new chunk c_{10}. Bytes are inserted in c_6, splitting it into two new chunks c_{12} and c_{13}. Similarly, c_8 becomes c_{14} and c_{15}. Continuous chunks consist of a block, and new blocks are written into a collection. Compared to conventional CDC, hierarchical partitioning reduces I/O requests, e.g., there are only four not nine I/O requests, and easily uses buffer or cache because of its hierarchical structure.



(a) Content-defined chunking (b) Chunk organization

Fig. 5 Hierarchical Partitioning

3.2.2 Metadata Management

Each client stores the fingerprint catalog of previous backups into its local disk, and it can quickly detect which files have been changed and properly reuse data from previous snapshots. In order to reduce network bandwidth and increase backup efficiency, the fingerprint catalog is kept in both clients and fingerprint servers, but it is not necessary for data recovery from a backup. The loss of the fingerprint catalog is not destructive in the clients or the fingerprint servers, but the fingerprint catalog enables various performance optimizations during backups and restores. The client fingerprint catalog stored in a BerkeleyDB database is divided into two parts: 1) local file fingerprint catalog and 2) local chunk fingerprint catalog. The former is used to quickly detect and skip over unchanged files based on 1) modification time for the same file name, and 2) checksum for identical files with different file names. The latter keeps track of recent snapshots and all collections, blocks and chunks stored in them. In this way, data de-duplication is supported, and client data can be restored from backups, while the file/chunk index can be rebuilt from collection data as it is downloaded during the restore.

3.2.3 Source-side Duplication Detection

Source-side de-duplication can reduce the amount of transferred data, thus decreasing the transmission cost. However, it typically does not work with de-duplication appliances because it requires client updates. There are two types of duplication detections in YuruBackup clients, one is Local File-level Duplication Detection (LFDD), and the other is Local Chunk-level Duplication Detection (LCDD) in fingerprint catalog. Before arriving at them, the hashes of files/chunks are passed as arguments to identify and remove duplicate files/chunks via *Index Summary* that is an in-memory compressed Bloom filter [12], and compactly represents the file/chunk fingerprint set in the current client.

If *Index Summary* indicates an item (file/chunk) is not in the index, it is no need to do a further lookup for the item, and it is new and should be stored into *Index Summary*. On the other hand, if *Index Summary* indicates the item belongs to the index, it is actually in the item index with a high probability, but there is no guarantee. *Index Summary* implements the following operations: 1) *Initialize(n, p)*, where n is a predefined number for all the items and p is a desired false positive probability; 2) *Insert(fp)* and 3) *Lookup(fp)*. By testing whether an item is new to the system, *Index Summary* can avoid unnecessary lookups for items that do not exist in the fingerprint catalog. YuruBackup quickly identifies same files via LFDD to avoid the retransmission of data when performing backup to the cloud. After LFDD, YuruBackup breaks

the remaining files into a series of variable-length chunks and determines whether they are duplicates by inspecting the local chunk fingerprint catalog through LCDD.

3.3 Data Transmission

Transfer Agent is used for uploading a snapshot and its description file to the remote cloud storage system. Usually, it needs local storage space for staging files and their metadata before being transferred to the cloud storage system. In order to transfer the files to the remote storage efficiently, they are written into *Write Buffer* as memory files, which are enqueued and transferred asynchronously by *Transfer Agent*. The fingerprint information of the files is uploaded to fingerprint servers via *Task Agent*.

Task Agent is implemented using the RPC (Remote Procedure Call) mechanism, consisting RPC client and server. As an RPC client, *Task Agent* running in YuruBackup clients provides query services to determine whether the fingerprint of a given file resides in a fingerprint server so that it can determine if the file itself is located in the cloud storage system. The RPC mechanism is implemented via message passing over TCP streams for remote communication, or system inter-process communication (IPC) mechanisms for local communication since they can provide a faster local connection than TCP streams. The remote communication utilizes multiple TCP connections to meet the throughput requirements. All the RPC requests from clients are asynchronous and batched, thus minimizing round-trip overhead and improving throughput. For different RPC invocations, each client registers different callback functions to map them, which are used to return the RPC results to the caller as they are available. In order to achieve high backup/restore throughput, the RPC client exploits a simple and fully asynchronous RPC implementation via an event driven and pipelined design.

4 Fingerprint Server Cluster

The cluster of fingerprint servers is the administrative center of fingerprint catalog, which globally manages all fingerprints. It uses a catalog database to keep track of which files and chunks are stored on which buckets in a cloud storage system. It is shown in Figure 6.

4.1 Communication with Clients

High throughput is essential to a highly scalable backup system, so an optimized client-server interface is necessary to improve de-duplication performance and reduce cost. The fingerprint server in YuruBackup receives the hash values of items and delivers index lookup results to the inquiry client. For each index miss, the server will receive new relevant fingerprint linking to its data in the cloud storage system; otherwise it will return the references of the existing items to the client. This process may suffer from a performance bottleneck that is high latency and low communication throughput. The fingerprint server receives a single, batched and asynchronous

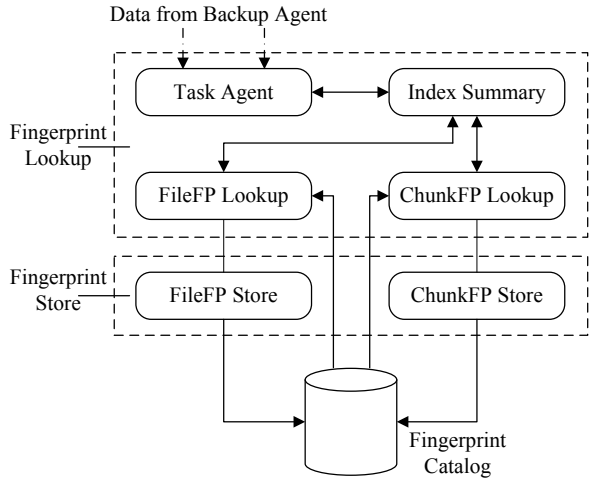


Fig. 6 Fingerprint Agent

lookup RPC from the client, incurring a single RPC round-trip for all n fingerprints. The lookup function delivers the RPC reply and creates references to the containers of the fingerprints that are found on the server. If one or more fingerprints were not found in the current sharding server, the lookup function enqueues the updated request in a queue to the global fingerprint catalog in the whole fingerprint server cluster.

A MySQL database deployed in fingerprint servers is utilized by YuruBackup to store the global fingerprint catalog, thus identifying and locating duplicate files/chunks. The global fingerprint catalog must be updated whenever a file is changed in a client. Significant overhead could potentially be caused due to synchronizing the global fingerprint catalog with file systems of clients. To avoid synchronization problems, YuruBackup never relies on the correctness of the fingerprint catalog. It recomputes the SHA-1 hash of any file/chunk before using it to restore a backup. It is not necessary to worry about crash recovery because YuruBackup does not rely on database integrity, which in turn makes YuruBackup avoid expensive synchronous database updates.

4.2 Global Fingerprint Lookup and Store

In large-scale distributed backup systems, a fast and scalable fingerprint lookup service has to be provided, and multiple fingerprint servers are deployed to enhance scalability. As deployments of data de-duplication are applied to PB-scale data, YuruBackup has to make target-side de-duplication scalable by deploying *Fingerprint Agent* that could share items' fingerprints, thus improving de-duplication efficiency. Source-side de-duplication works with this global target-side de-duplication to minimize network traffic, but there is a balance between transfers of fingerprints of redundant files/chunks from a local fingerprint catalog and updates of the global fingerprint catalog. In YuruBackup, global target-side de-duplication is utilized to look

up incoming files/chunks, some files/chunks are duplicates to an existing file/chunk and a pointer to that original one was recorded. The removal of duplicate chunks represents a real step forward over file-level de-duplication. There are some widely varying results, depending upon the method used to make the comparison.

In the RPC server, *Index Summary* is utilized to reduce the number of times of accessing disk and save bandwidth to look for a duplicate item. There are two components to use *Index Summary*, i.e., Global File-level Duplication Detection (GFDD) and Global Chunk-level Duplication Detection (GCDD), which are similar to LFDD and LCDD, and are deployed in the cluster of fingerprint servers as discussed in Section 4.3. Both of them compare the fingerprints of incoming items with those already stored in the global fingerprint catalog to identify and remove duplicate items. They can become a potential performance bottleneck if the cluster of fingerprint servers is not scalable. At the same time, there is a counter in the cluster to determine if a data item (file/chunk) is popular according to the Zipfian distribution. Only popular data items are stored in the cluster of fingerprint servers because of two aspects: 1) lookup performance and 2) storage cost.

4.3 Highly Scalable Cluster of Fingerprint Servers

To make large-scale distributed backup systems scalable and available, fingerprint catalog is administered by multiple distributed fingerprint servers. A load-balancing DB sharding (partitioning) and replication architecture is designed and implemented for fingerprint server cluster as shown in Figure 7, where the slaves are used for scalable DB reads and the SQL nodes are utilized for scalable DB writes with the load balancer. The load balancer shares the workload as evenly as possible among a group of servers, and then routes incoming read and write requests to the least busy available slave and SQL node respectively. Load balancing comes across multiple nodes in the sharding and replication architecture, where there is a master-master replication pair with many slaves in a MySQL cluster. DB replication is deployed to provide state-of-the-art database scalable reads, and deliver high availability by offering a way to mirror data across multiple nodes for tolerating failures. In YuruBackup, there is a load balancer that is aware of which nodes are readable and writable. Writes are the bottleneck of DB replication with which it is hard to scale writes, and the only way to scale writes is to partition data.

In YuruBackup, MySQL master-master replication is used, where one master *A* is configured as follows: *auto_increment_offset=1*, *auto_increment_increment=2* in its configuration, while the other one *B* is configured as follows: *auto_increment_offset=2*, *auto_increment_increment=2* in its configuration. Note that it can scale up to N ($N > 2$) master servers by modifying their configurations, but it will bring performance issues because of $\frac{N(N-1)}{2}$ connections among the masters. When a single-node MySQL cannot store fingerprint catalog so that sharding must be used, the single-node MySQL will be extended a MySQL cluster[3] based on the NDB cluster engine. Therefore, this cluster architecture is scalable for both reads and writes on the global fingerprint catalog.

[3] http://www.mysql.com/products/cluster/

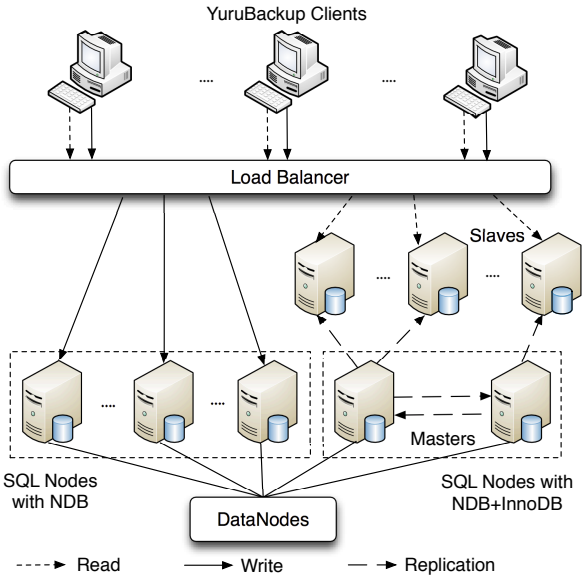


Fig. 7 Highly Scalable Cluster of Fingerprint Servers

5 Performance Evaluation

In this section, we evaluate the performance of YuruBackup in four parts. We have built the YuruBackup prototype system in C++, which consists of about twelve t-housand lines of code, and used real-world data sets to evaluate its performance. We first empirically evaluate the de-duplication performance including effectiveness and overhead. We second present scalability measurement of reads and writes on finger-print catalog atop AWS (Amazon Web Services) S3 and EC2. In the third part, we measure the backup performance of YuruBackup atop AWS. Lastly, we also mea-sures the restore performance of YuruBackup like its backup performance. Table 2 summaries eight datasets used in our experiments. *Backup Client* is deployed on an Intel Core 2 PC (2.50GHz) with 4GB RAM, a 500-GB 7200-RPM Seagate SATA hard disk and a 10/100 Mbps Ethernet port, running 32-bit Ubuntu 12.04.

We empirically evaluate YuruBackup and compare its performance with other two incremental backup systems: 1) rdiff-backup [8] that uses libsync delta encoding and stores increments as diffs; and 2) Brackup [6] that utilizes an optimized scheme including rsync-like deltas and better reclamation of storage space for incremental backups. All the experiments are repeated three times, and the average results are reported.

5.1 De-duplication Performance

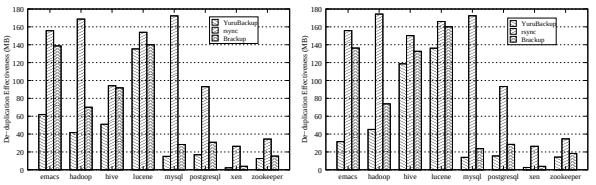
De-duplication performance includes de-duplication effectiveness and hierarchial par-titioning I/O efficiency.

Table 2 Datasets

Dataset	Version	Size (MB)	Overlap	
			Files	Data (MB)(%)
emacs	23.2a	155.4	957	15.7 (10.09)
	23.3a	155.9	980	15.9 (10.17)
	23.4	155.9		
hadoop	1.0.1	167.1	3517	96.8 (56.32)
	1.0.2	171.9	3517	95.1 (53.60)
	1.0.3	177.5		
hive	0.7.0	143.7	3720	49.1 (34.10)
	0.7.1	143.9	2763	15.9 (12.16)
	0.8.0	161.4		
lucene	3.3.0	136.9	208	13.3 (8.51)
	3.4.0	156.4	206	4.9 (2.91)
	3.5.0	168.7		
mysql	5.1.61	173.0	7877	143.1 (82.64)
	5.1.62	173.2	8038	148.1 (85.45)
	5.1.63	173.4		
postgresql	9.1.2	92.6	3773	66.1 (70.02)
	9.1.3	94.1	3728	64.4 (68.36)
	9.1.4	94.2		
xen	4.1.0	45.2	4276	41.7 (92.20)
	4.1.1	45.2	4143	40.7 (89.99)
	4.1.2	45.2		
zookeeper	3.4.1	35.7	1205	20.2 (56.56)
	3.4.2	35.7	1126	16.4 (45.58)
	3.4.3	35.9		

5.1.1 De-duplication Effectiveness

De-duplication effectiveness is of crucial importance to both cloud storage providers and customers. Providers expect less data stored in their clouds to reduce data storage and management costs, whereas customers prefer to transfer less data to make backup/restore time shorter. Figure 8 presents de-duplication effectiveness on eight datasets using three different de-duplication systems. Note that rdiff-backup uses rsync and its statistical result is based on compressed metadata, so we utilize rsync as YuruBackup's competitor instead of rdiff-backup.



(a) First De-duplication (b) Second De-duplication

Fig. 8 De-duplication Effectiveness

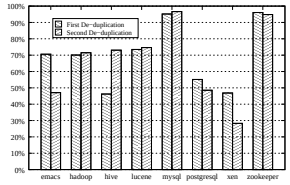
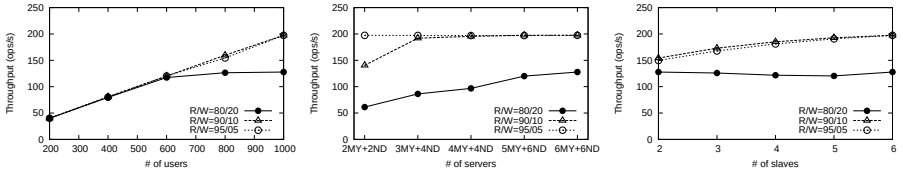


Fig. 9 Hierarchical Partitioning I/O Efficiency

Both Figure 8(a) and Figure 8(b) illustrate that YuruBackup has much better de-duplication effectiveness than rsync and Brackup for the given eight datasets. We define *speedup* using the following formula: $S = \frac{A}{B}$, where A is the de-duplication effectiveness of YuruBackup or Brackup, B is the de-duplication effectiveness of rsync. The speedups of both YuruBackup and Brackup reach the highest points of 12.04 and 6.74 at the xen dataset. The speedup of YuruBackup bottoms out at $lucene$, while the speedup of Brackup is the lowest point at $hive$. Their speedups are 5.17



(a) Varying # of users (b) Varying # of servers (c) Varying # of slaves

Fig. 10 Scalability on Fingerprint Server Cluster. a) with 6 slaves, 6 MySQLDs and 6 NDBDs, b) with 1,000 users and 6 slaves, and c) with 1,000 users, 6 MySQLDs and 6 NDBDs.

and 2.97 respectively. YuruBackup is much better than rsync and Brackup in de-duplication effectiveness because it uses the variable-size chunking approach, which is more effective than the fixed-size chunking one.

5.1.2 Hierarchical Partitioning I/O Efficiency

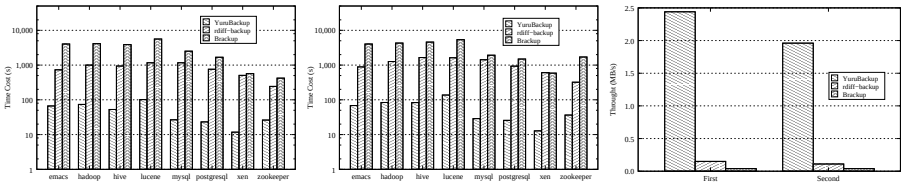
Figure 9 shows the reduced I/O ratio in the first and second de-duplications. The reduced I/O ratio is calculated as follows: $\frac{C-B}{C}$, where C is the number of chunks and B is the number of blocks. By using this metric, we can measure how many I/O requests hierarchical partitioning can reduce compared to the conventional content-defined chunking approach. From Figure 9, we can see that hierarchical partitioning can reduce I/O requests by 68 percent in average. The reduced I/O ratio is the best at *mysql*, more than 95% in both de-duplications, while it is the worst at *xen*, 46.8% and 28.3% in the first and second de-duplications respectively.

5.2 Scalability Measurement on Fingerprint Server Cluster

We have customized the Cloudstone[4] benchmark to measure the scalability of fingerprint server cluster to simulate that many backup clients send massive read/writes requests to the fingerprint server cluster because we do not have a real workload trace of cloud incremental backup. Cloudstone [16] is mainly designed as a performance measurement tool for Web 2.0 applications including many complicated database operations such as *join*, which take much more computations and I/Os than our fingerprint management mechanism. In the following three experiments, a large number of read/write requests on fingerprint entries are submitted to the fingerprint server cluster, where there are 10 minutes ramp-up, 20 minutes steady stage and 5 minutes ramp-down. Slaves connect to a hot-backup cluster node that does not take any requests. All the clients start up simultaneously and send their requests to the fingerprint server cluster concurrently.

We first vary the number of users when there are 6 slaves, 6 MySQL servers and 6 MySQL cluster data nodes. The content of the fingerprint catalog is fixed and evenly range partitioned over all fingerprint servers. Figure 10(a) depicts that the fingerprint server cluster can always maintain a stable and relatively good throughput, no matter how the total request number and the user number vary in a real cloud environment.

[4] The source code of the customized Cloudstone is available on http://code.google.com/p/clouddb-replication/.



(a) First Incremental Backup (b) Second Incremental Backup (c) Backup Throughput

Fig. 11 Backup Performance Comparison in a Public Cloud (AWS)

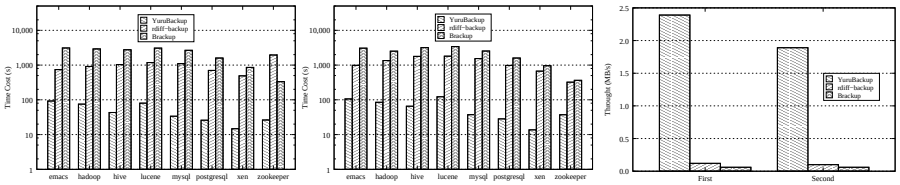
As the total number of read/write requests is fixed, more write requests result in longer processing time since write is always more expensive than read because of locking. It always leads to the worst performance when *R/W* is 80/20, followed by 90/10, and 95/05 that generates the best scalability.

We fix the number of users, vary the numbers of MySQL servers and cluster data nodes when there are 6 slaves. As shown in Figure 10(b), the total throughput increases almost linearly where *R/W* is 80/20 since the number of servers in MySQL Cluster increases, causing each server hosts a smaller portion of the fingerprint catalog. It brings two benefits: 1) decreasing the average consistency overhead of entries and 2) reducing the time spent on entry search. In addition, we change the number of slaves when there are 1,000 users, 6 MySQL servers and 6 MySQL cluster data nodes. Figure 10(c) shows that the throughput rises by increasing the number of slaves when the read/write ratio is no more than 90/10, which fits in with incremental backup systems. From both Figure 10(b) and Figure 10(c), we can conclude that we need to deploy more slaves in MySQL Replication and less servers in MySQL Cluster in YuruBackup to make read/write requests scalable.

5.3 Backup Performance

5.3.1 Backup Window

Backup window means the time cost spent on backing up a given dataset, depending on the transferred dataset's size and available network bandwidth. Figure 11(a) and Figure 11(b) present the first and second incremental backup window results in a public cloud, i.e., AWS. From both of figures, we can see that YuruBackup is still the best one in backup window among the three systems. YuruBackup takes 47.61 seconds and 57.28 seconds in the first and second incremental backups respectively, while rdiff-backup takes around 783 and 1029 seconds respectively, and Brackup takes about 2720 and 2692 seconds respectively. There are several reasons for this. Firstly, YuruBackup's backup client uses *Read Buffer* to reduce I/O operations, compressed Bloom filter and BerkeleyDB to locate items fast, content-defined chunking with hierarchical partitioning to do source-side de-duplication, and an efficient transmission mechanism including batched RPC and parallel uploading/downloading. Secondly, Yurubakup's fingerprint server utilizes global *Index Summary* and global target-side de-duplication to identify items using a single, batched and asynchronous lookup RPC for n FPs. In addition, the storage hierarchy used in YuruBackup packs small files into a file before uploading to S3, which greatly reduces backup latency.



(a) First Incremental Restore (b) Second Incremental Restore (c) Restore Throughput

Fig. 12 Restore Performance Comparison in a Public Cloud (AWS)

5.3.2 Backup Throughput

Figure 11(c) shows that YuruBackup has excellent speedups in a public cloud, while the speedup is $S = \frac{A}{B}$, where A is the throughput of YuruBackup, B is the throughput of rdiff-backup or Brackup. The speedups are around 16 and 54 respectively in the first incremental backup, while they are 17.5 and 44 respectively in the second incremental backup.

5.4 Restore Performance

Restore performance is also important to backup systems. Most backup systems should pay attention to the issue of restore performance [9] since the goal of backup is to restore when it is necessary. We have to evaluate restore performance carefully when reviewing backup systems.

5.4.1 Restore Window

Restore window represents how long a restore of snapshots takes. Forward referencing avoids the restore problems inherent in most de-duplication systems. Figure 12(a) and Figure 12(b) present the first and second incremental restore window results in a public cloud, i.e., AWS. In the first incremental restore, the restore window of YuruBackup (48.9 seconds) is only 5.4% and 2.8% of rdiff-backup's and Brackup's restore windows respectively. YuruBackup's restore window (61.9 seconds) is only 5.8% and 3.4% of rdiff-backup's and Brackup's restore windows in the second incremental restore. The reason for this is that YuruBackup concerns that how redundant files/chunks are found and how their information is organized and stored better. YuruBackup exploits compressed Bloom filter and Berkeley DB to fast locate items. It decreases the number of traversed pointers, and reduces I/O operations during the process of restoring a backup. The more pointers there are, the more fragments the data becomes as more data is online. It also reduces the fragments caused by pointers better than the other two systems. Besides the reasons mentioned above, YuruBackup utilizes a reasonable and effective storage hierarchy discussed in Section 2.2 to make I/O operations as few as possible to S3[5], and uses *Write Buffer* to decrease I/O requests.

[5] Uploading/downloading a file to S3 is an I/O operation.

5.4.2 Restore Throughput

In backup systems, the backup throughput is paid more attention to than the restore throughput because the restore correctness is the main goal. However, restore is also an important operation and we want to ensure that YuruBackup provides sufficient throughput. Figure 12(c) presents restore throughput comparisons among YuruBackup and its competitors in a public cloud respectively. YuruBackup is much better than rdiff-backup and Brackup in restore throughput. The throughput of YuruBackup is 18.5 and 36 times those of rdiff-backup and Brackup in a public cloud.

6 Related Work

6.1 Data De-duplication

As a standard incremental transfer tool, rsync [21] that is used by Dropbox explores schemes to find identical subsets in two different versions of the same file, but it cannot easily capture the same data between them. Data de-duplication as a space-efficient approach, is being exploited widely in backup systems [28,4,7,23], which are increasingly being deployed to reduce cost and increase space-efficiency. The opportunities become scarce to optimize de-duplication efficiency. We thus have to face the challenge that is to design de-duplication systems effectively addressing the capacity, throughput and management requirements of PB-scale data. Existing literatures [28,4,17] on data de-duplication rely on workloads consisting of daily full backups, which represent the most attractive scenario for de-duplication since the content of file systems itself does not change rapidly. The size of historical data quickly exceeds that of the running system because of frequent and persistent backups.

Many organizations have dramatic increases in total storage system costs [10] in spite of obvious decline at storage cost per GB. They thus are very interested in reducing the total storage costs, which has caused de-duplication techniques both in academia and industry. The Data Domain File System (DDFS) [28] addressed the disk bottleneck by introducing a series of optimizations including Bloom filter, Stream-Informed Segment Layout (SISL) and Locality Preserved Caching (LPC). However, the system can support a limited amount of raw storage, and is limited by network performance, since duplicate detection is performed only at the server, which is called *target/server de-duplication*. Additionally, it is not clear whether DDFS can perform really scalable resource reclamation. HYDRAstor [4] achieves good scalability using a highly distributed and hierarchical model, where each node holds a few tens of TB of storage. It yields a high backup throughput, but it costs too high because of its highly distributed costly system architecture.

6.2 Cloud Storage and Backup

Cloud storage supports EB-scale data or trillions of files [24], and even cloud-scale data management [3]. Traditionally, users back up their data by storing multiple replicas, where the backup window is often restricted by the volume of data. Further

complexity is caused if remote users have data retention requirements, which mean historical data must be stored into tape. Cloud backup services are attracting much more attention than traditional ones from both the industry community [5,19,2] and the academic community [22,20] as cloud computing is increasingly popular. As a cloud backup system, Cumulus [22] leverages the source-side de-duplication based on chunk-level to remove duplicates from transmission for backups but not for restores. CABdedupe [20] is a causality-based de-duplication performance booster for both backups and restores in the cloud by enabling the removal of the unmodified data from transmission for backups and restores.

To improve backup performance and decrease backup cost, cloud backup systems such as EMC Avamar [5], Asigra [2] and Symantec NetBackup [19], use the source de-duplication technology to remove redundant data from transmission. Although they have done backup well, they do not pay attention to the problem that happens in restores over the low bandwidth network. Existing cloud recovery solutions either transfer and restore data locally over high bandwidth network by deploying complete servers in the clients [5], or even transfer the data from providers to costumers by vehicles to avoid transmission in the Internet [2], or feature built-in wide area network replication so that backups from the source can be classified and selectively replicated to the recovery site for restore at any time [19].

7 Conclusion and Future Work

We have designed and implemented YuruBackup that is a space-efficient and highly scalable incremental backup system in the cloud. In *Backup Client*, read buffer is for reading files before data de-duplication, and write buffer is for writing files after data de-duplication, thus reducing I/O requests. Using compressed Bloom filter in the first round and Berkeley DB in the second round is to fast locate items. Source-side de-duplication with hierarchical partitioning improves space-efficiency when performing backups, and data transmission using batched RPC and parallel uploading/downloading decrease the communication overhead of backups and restores. *Fingerprint Server Cluster* allows to add one or more servers dynamically to deal with increasing requests from clients. It efficiently communicates with *Backup Client* via RPC, and provides global fingerprint lookup to backup clients, thus facilitating backups and restores. By conducting performance evaluation in AWS, experimental results demonstrate the efficiency of YuruBackup including data de-duplication effectiveness, I/O overhead, the scalability of fingerprint server cluster, backup and restore performance. To achieve much more scalable *Fingerprint Server Cluster* in the cloud, to replace MySQL cluster is one possible work with a key-value store on DHT-based publishing and searching [25]. To deploy cooperative cache [26] in *Fingerprint Server Cluster* is another possible work for facilitating item retrieval.

Acknowledgements We would like to thank Alan Fekete at the University of Sydney, Yong Khai Leong, Khin Mi Mi Aung and Rajesh Vellore Arumugam at Data Storage Institute, A*STAR for their help, and the anonymous reviewers for their suggestions. This work is supported by the National Basic Research Program of China (973) under Grant. 2011CB302305, and National Science Foundation of China under

Grant No. 61073015. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. Agrawal, N., Bolosky, W.J., Douceur, J.R., Lorch, J.R.: A five-year study of file-system metadata. In: FAST, pp. 31–45 (2007)
2. Asigra: Asigra cloud backup and recovery software. White Paper (2012)
3. Cao, Y., Chen, C., Guo, F., Jiang, D., Lin, Y., Ooi, B.C., Vo, H.T., Wu, S., Xu, Q.: Es^2: A cloud data storage system for supporting both oltp and olap. In: ICDE, pp. 291–302 (2011)
4. Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., Welnicki, M.: Hydrastor: A scalable secondary storage. In: FAST, pp. 197–210 (2009)
5. EMC: Efficient data protection with emc avamar global deduplication software. White Paper (2010)
6. Fitzpatrick, B.: Brackup (2010). URL http://code.google.com/p/brackup/
7. Guo, F., Efstathopoulos, P.: Building a high-performance deduplication system. In: USENIX Annual Technical Conference (2011)
8. rdiff-backup (2009). URL http://www.nongnu.org/rdiff-backup/
9. Lillibridge, M., Eshghi, K., Bhagwat, D.: Improving restore speed for backup systems that use inline chunk-based deduplication. In: FAST, pp. 183–197 (2013)
10. Merrill, D.R.: Four principles for reducing total cost of ownership. four-principles-for-reducing-total-cost-of-ownership.pdf (2011). URL http://www.hds.com/assets/pdf/
11. Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. In: FAST, pp. 1–13 (2011)
12. Mitzenmacher, M.: Compressed bloom filters. In: PODC, pp. 144–150 (2001)
13. Muthitacharoen, A., Chen, B., Mazières, D.: A low-bandwidth network file system. In: SOSP, pp. 174–187 (2001)
14. Policroniades, C., Pratt, I.: Alternatives for detecting redundancy in storage systems data. In: USENIX Annual Technical Conference, General Track, pp. 73–86 (2004)
15. Rabin, M.O.: Fingerprinting by random polynomials. Tech. Rep. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, MA (1981)
16. Sobel, W., Subramanyam, S., Sucharitakul, A., Nguyen, J., Wong, H., Patil, S., Fox, A., Patterson, D.: Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In: CCA (2008)
17. Srinivasan, K., Bisson, T., Goodson, G., Voruganti, K.: idedup: Latency-aware, inline data deduplication for primary storage. In: FAST (2012)
18. 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: FAST, pp. 161–174 (2013)
19. Symantec: Symantec netbackup appliances: Key considerations in modernizing your backup and deduplication solutions. White Paper (2011)
20. Tan, Y., Jiang, H., Feng, D., Tian, L., Yan, Z.: Cabdedupe: A causality-based deduplication performance booster for cloud backup services. In: IPDPS, pp. 1266–1277 (2011)
21. Tridgell, A., Mackerras, P.: The rsync algorithm. Tech. Rep. TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra (1996)
22. Vrable, M., Savage, S., Voelker, G.M.: Cumulus: Filesystem backup to the cloud. In: FAST, pp. 225–238 (2009)
23. Xia, W., Jiang, H., Feng, D., Hua, Y.: Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In: USENIX Annual Technical Conference (2011)
24. Xu, Q., Arumugam, R.V., Yong, K.L., Mahadevan, S.: Drop: Facilitating distributed metadata management in eb-scale storage systems. In: MSST (2013)
25. Xu, Q., Hou, X., Cui, B., Shen, H.T., Dai, Y.: Facilitating effective resource publishing and searching in dht networks. HKIE Transactions **16**(3), 32–41 (2009)
26. Xu, Q., Shen, H.T., Chen, Z., Cui, B., Zhou, X., Dai, Y.: Hybrid information retrieval policies based on cooperative cache in mobile p2p networks. Frontiers of Computer Science in China **3**(3), 381–395 (2009)
27. You, L., Pollack, K.T., Long, D.D.E.: Deep store: an archival storage system architecture. In: ICDE, pp. 804–815 (2005)
28. Zhu, B., Li, K., Patterson, R.H.: Avoiding the disk bottleneck in the data domain deduplication file system. In: FAST, pp. 269–282 (2008)