http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 12 (11): 2101-2110, 2013
ISSN 1812-5638 / DOI: 10.3923/itj.2013.2101.2110

LevelStore: A Large Scale Key-value Store for Deduplication Storage System

Dong Wang and Yongjing Lu
College of Information Science and Engineering, Hunan University, Changsha, China

Abstract: The traditional high-performance deduplication systems are focused on improving the data compression ratio and throughput of a single node, but most of them have ignored the scalability which makes it difficult to be applied to large-scale distributed environments. This paper presents LevelStore, a large scale key-value store for deduplication storage system which optimized three indicators including compression ratio, throughput and scalability. LevelStore introduced a hierarchical feature which improved the disk utilization by introducing container compaction and exploited a sorted in-memory buffer to favor large sequential writes. More importantly, LevelStore can be dynamically and cost-effectively scaled up on demand. Experiments for different datasets are presented and the results show LevelStore can not only greatly improved the performance of a single node but also provided a lot of efficient building blocks for distributed implementation and thus applicable to large scale and distributed systems.

Key words: key-value, deduplication, throughput, scalability, petabytes

INTRODUCTION

The explosive increase in the amount of data has raised a critical demand for long-term data protection through large scale and high performance storage and backup systems. According to International Data Corporation, the global amount of data will grow 44 times to 35.2 ZB in 2020. Storage and backup systems are challenged with the task of finding effective solutions to boost both storage efficiency and system scalability to meet the accelerating demand on backup capacity and performance.

The development of disk storage makes the great success of deduplication system, because the disk can be accessed randomly. In the process of deduplication, the system needs to locate the duplicate data rapidly, while the traditional tape can not do this (Kulkarni *et al*., 2004; Jain *et al*., 2005; Zhu *et al*., 2008). The general deduplication method works by dividing files into multiple chunks based on a content-aware chunking algorithm, then a fingerprint (Rabin, 1981; Burrows, 1995) was generated according to the contents of each chunk. Duplicate chunks are identified by comparing their fingerprints with existed chunks', only new chunks are stored.

To get good deduplication efficiency, some deduplication systems have to put the entire index into memory to perform chunk index. Unfortunately, for a large scale system, the index is usually too large to fit in memory, thus lead to the chunk lookup disk bottleneck problem. Zhu *et al*. (2008) have addressed this problem by using a combination of Bloom filter (Bloom, 1970) and locality preserved caching. Keeping Bloom filter in memory, can avoid most unnecessary lookups for chunks that do not exist in the index. But the memory cost grows linearly with the capacity of the system would set a hard limit to the scalability of the system. Moving the hash table into disk can eliminate the limits of scalability, but greatly reduces the performance.

Another important problem is garbage collection. In the deduplication, the same logical data chunks by sharing the same physical data chunk to achieve the purpose of deduplication. When the logical data chunk update or deletion happens, some physical data chunks will not be referenced, we call those chunks garbage. After the deduplication system ran a long period of time, the garbage chunks will increase. This not only wasted disk space, but also reduces the performance of the system, thus introduced the garbage collection to determine who is using a particular data chunk and when it can be reclaimed. Most systems do not provide this function. Guo and Efstathopoulos (2011) proposed a grouped mark-and-sweep method to tackle this problem, but the computational and space complexity of the grouped mark-and-sweep is proportional to the capacity of the system, thus limiting its scalability.

However, a lot of research in this area has focused on optimizing deduplication efficiency and compression ratios (Lillibridge *et al*., 2009; Shilane *et al*., 2012) rather than on methods for improving the throughput and scalability of deduplication, with the exception of Yang *et al*. (2010). They addressed the scalability and

Corresponding Author: Yongjing Lu, College of Information Science and Engineering, Hunan University, Changsha, China

throughput problems by deploying a complex distributed system without making full use of each node resources, thus lead to a very high cost. In addition, they ignored the problem of garbage collection.

To achieve large scale and high throughput and make the best use of limited resources, deduplication storage systems need an underlying high-performance storage system. Key-value store has a better scalability and performance than traditional relational databases and the retrieval and appending operations are highly optimized (DeCandia *et al*., 2007; Lakshman and Malik, 2010), so we using key-value store as the persistent store in deduplication. The performance of the key-value store often governs the performance of deduplication.

In this study, we present LevelStore, a scalable and high-performance storage system. LevelStore can scale to large capacity and achieve high throughput by using a hierarchical model with each storage node holding a few tens of TB of storage to maximize system capacity. The garbage collection mechanism is both scalable and fast enough to keep up with the backup speed.

RELATED WORK

Early deduplication systems have put a lot of effort into optimizing duplicate detection, bandwidth and compression ratios. In particular, EMC Centera Content Addressed Storage (EMC, 2008) and Single Instance Storage (Bolosky *et al*., 2000) eliminate duplicate at file level which can achieve only limited storage saving. Venti (Quinlan and Dorward, 2002) removes duplicate data at fixed-sized blocks by comparing their secure hashes. To tolerate shifted contents, LBFS (Muthitacharoen *et al*., 2001) introduced the variable-size data chunks aiming to improve compression ratio which employs Rabin fingerprint (Rabin, 1981) to divide files into variable-sized data blocks. A stream-informed delta compression is used in (Jin and Miller, 2009) for replication of backup datasets to achieve higher compression.

However, the above studies have been mainly focused on basic methods to achieve more data reduction but not on techniques to achieve high throughput and scalability. Even if these systems make the best of raw storage (Lillibridge *et al*., 2009), system capacity is limited. For instance, Venti (Quinlan and Dorward, 2002) uses caching and striping to improve the index lookup performance. Since fingerprints have no locality, their index cache is not effective. Besides using a straightforward index cache to improve the deduplication throughput, Data Domain (Zhu *et al*., 2008) addressed the disk bottleneck by introducing a series of optimizations, including a Bloom filter and stream-informed segment layout which improve throughput but still suffer poor scalability for large-scale and distributed deduplication environments. A sampling method called sparse index is used in Lillibridge *et al*. (2009) to address indexing scalability restrictions. It uses sampling to probabilistically identify "super-chunks" that are used to perform coarse-granularity deduplication which only need less than half the memory space for an equivalent level of deduplication over Data Domain. However, the sparse index may store some duplicate chunks. For peta-scale systems, decentralizing the sparse index into multiple servers can incur large larger amounts of network I/Os for index lookup which in turn degrades the inline backup performance.

In addition, MAD2 (Wei *et al*., 2010), HYDRAstor (Bobbarjung *et al*., 2006) and Deep Store (You *et al*., 2005) all try to achieve high scalability and availability using a highly distributed storage system, as well as a number of advantageous features that include locality caching, Bloom filters and delta-encoding to get fine grain data deduplication. This design yields a high backup throughput, but another challenge in those systems is garbage collection. Garbage collection need to keep track of data chunk usage and reclaim freed space. Garbage collection in those systems is implemented with a distributed reference counting method which is difficult to maintain correctly and scale without a large performance hit. A grouped mark-and-sweep design was proposed by Guo and Efstathopoulos (2011) to address the challenge in garbage collection. To avoid touching every file and container, grouped mark-and-sweep only keeps track of the changes in file manager which is reliable, scalable and fast, but it may use too much memory according the protection map and can only apply to single-node.

LEVELSTORE ARCHITECTURE

The LevelStore architecture, shown in Fig. 1, uses a hash table, an in-memory memtable and a cluster of containers on disk. The hash table consists of a series of fixed-sized buckets with each bucket containing a series of fixed-sized entries. Each entry stores at least the fingerprint and the corresponding container ID. Each container has a global unique ID. The containers provide a global deduplication storage pool for data chunks.

When a data stream enters the system, it needs to be divided into chunks and a fingerprint is generated for each chunk. Duplicate chunks are identified by comparing the chunk fingerprints represented by the hash values of chunk contents. The new chunks are written to a sorted fixed-sized buffer according to their fingerprints, called memtable. When the memtable fills up, it will be stored

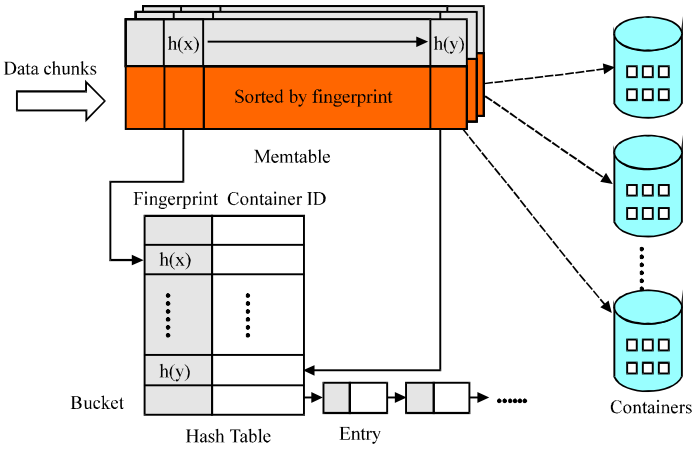

Fig. 1: Architecture of LevelStore

into a container according to the content of memtable. Then a container ID was returned. You should notice that, all the data chunks in the one memtable have the same container ID. Finally, LevelStore updates the fingerprints and container ID to the hash table entries. Fingerprints are automatically sorted into different index buckets in the order of their corresponding bucket numbers (i.e., their first n bits).

Similar to other key-value stores, LevelStore also provides some basic operations including put and get. Additionally, LevelStore provides delete operations. Each put operation inserts an entry into LevelStore. For a get operation, LevelStore searches for the key in hash table first, then memtable and containers at last, it will return the value founded in the youngest place. Delete operation is a little different, it inserts a special entry into LevelStore, the real delete operation happens in the container.

The following describes the hash table, memtable and container in detail and the rationale behind some design choices.

Hash table: Due SHA-1 algorithm's good randomness and collision-resistant, we use it to calculate chunk fingerprints. For a hash table contains 2^n buckets, we only take the first n bits of a fingerprint as the bucket number, thus mapping a fingerprint to its corresponding bucket. Such a simple mapping method enables the scalability of LevelStore, as described below.

Scalability: To support more storage capacity, the hash table capacity expansion can be done through some simple copy operations. For example, enlarging the hash table with 2^n buckets to 2^{n+1} buckets can be done as follows: first, a new hash table with 2^{n+1} buckets should be constructed, then the entries in the old hash table of bucket i (i = 0, 1, …, 2^n-1) are copied to the bucket 2i and 2i+1 of the new hash table, respectively. Such a simple expansion operation enables the system to meet the larger capacity requirements easily.

Considering the fact that the storage capacity scales in size, the entire hash table is usually too large to fit in a server's memory and the memory efficiency is increasingly becoming one of the most important factors for the systems' scalability and overall cost effectiveness. The hash table in LevelStore can be divided into several equal-sized parts with each part being located in a different node to provide parallel index and update. Assuming the hash table is divided into 2^k equal-sized parts with each part being located in the 2^k different nodes, the first k (k<n) bits of fingerprints will be used to index nodes and the remaining n-k bits will be used to mapping the corresponding bucket in each node. Since the fingerprints index and update can be performed in parallel on these 2^k nodes, LevelStore can scale to large capacity and high performance dynamically and cost-effectively by simply adding the number of nodes.

Utilization: The utilization of hash table is closely related to the performance of the system. Standard hashing allows 50% of the table buckets to be occupied before unresolvable collisions occur. Due to the randomness of SHA-1 algorithm, the fingerprints will be distributed to the buckets evenly. Given a sufficiently large number of appropriately sized buckets, the index can achieve a relatively high utilization before it begins to overflow, but it increases the number of fingerprints comparison. So the key to designing an index is to select an appropriate size for the bucket that can meet the demand of the expected index utilization while introducing little additional overhead.

For example, LevelStore uses a 16-bit key fragment as the bucket number. Each bucket consisted of 8 disk blocks with each disk block (usually 512 bytes in size) storing 25 entries (an entry is 20 bytes) and thus can store up to 200 entries. Moreover, a 4 KB bucket allows each bucket to fit in a single cache line.

When the insertion of a fingerprint causes the overflow of a bucket, the system will insert the fingerprint into one of its two adjacent buckets randomly. And if the two adjacent buckets are full, the capacity of hash table needs to be expanded.

Our experiments show that using an 8-way set associative hash table improves space utilization of the table to about 90%. To store more entries in hash table, one can increase the size of the key fragment to have more buckets, increase the associativity to pack more entries into one bucket, or partition the key-space to smaller parts and assign each part to one node.

Memtable: Memtable is a write-friendly in-memory buffer that handles individual put and delete operations which uses Skip List (Pugh, 1990) to keep the entries orderly instead of a balanced tree. Compared with the balanced tree, Skip List is easy to implement and has a higher operating efficiency.

Each container corresponds to a memtable. As deduplication is performing, the new data chunks are inserted into memtable. Usually, the memtable size is fixed 4 MB which is the same as the file in container's level-0. Once memtable fills up, LevelStore will submit it to a container based on the container ID which extracted from the fingerprint. After the memtable storing process is completed, all the fingerprints and their corresponding container IDs are written to hash table. Thus the data persistence is completed.

As the buffer of data chunks, memtable offers several benefits: The fixed-size makes it allocation and deallocation easy. It is possible for LevelStore to perform bulk sequential writes.

Sorting is well-studied. The new chunks can be sorted and sequentially merged into existing sorted container. The space occupied by deleted or stale chunks can be reclaimed when merging happens.

Container: The container has many levels which is composed by sstables. Levels from top to bottom are number-ordered, the first level is numbered level-0, the second level is numbered level-1 and the like. When a memtable appends to container, container manager will convert it to an sstable in level-0 and keeps track of the reference and the specific location of data chunks. As

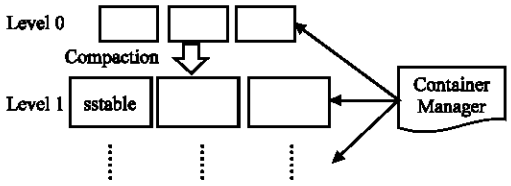

Fig. 2: Design of container

shown in Fig. 2, all the sstable files that make up each level, the corresponding key ranges and other important metadata are managed by the container manager.

Sstable: Sstable is the storage unit of the container which provides a persistent, ordered immutable map from fingerprints to data chunks. Sstable is self-described in that a metadata section includes the descriptors for the stored chunks which located before the data section. They are immutable in that new sstables can be appended and old sstables deleted, but sstables cannot be modified once written.

Sstable size grows exponentially with the increase of levels. But sstable size and number of each level is fixed which can be controlled by LeveStore. The SSTable size is usually a few MB to dozens of MB, in order to store thousands of data chunks in KB. For a data stream, each sstable contains a sequence of its corresponding logical data chunks orderly. It hence creates a spatial locality for the access to achieve a high read throughput.

Compaction: Unlike other systems which cannot avoid duplicated storing incurred by update and deletion, LevelStore uses a simple mechanism to effectively eliminate duplicate chunks due to asynchronous compaction, as described below.

The container manager maintained a timer and the reference of delete operations. When the delete operation reaches a number or the timer reaches the threshold, container manager will compact the sstables in background.

The compaction picks a file from level-L and all overlapping files from the next level-(L+1). Note that if a level-L file overlaps only part of a level-(L+1) file, the entire file at level-(L+1) is used as an input to the compaction and the input sstables can be discarded as soon as the compaction has finished. A compaction merges the picked files to produce a sequence of level-(L+1) files. We switch to producing a new level-(L+1) file after the current output file has reached the target file size. The compaction task is done by enumerating every item in the file of level-L, merging with

Table 1: Merge rule for level-L. K_L and K_{L+1} is the current key from level-L and level-(L+1). "Deleted" means the current item in K_L is a special entry indicating deleted

Comparison	"Deleted"?	Action on K_L	Action on K_{L+1}
$K_L > K_{L+1}$	Any	-	Copy
$K_L < K_{L+1}$	Yes	Drop	-
$K_L < K_{L+1}$	No	Copy	-
$K_L = K_{L+1}$	Yes	Drop	Drop
$K_L = K_{L+1}$	No	Copy	Drop

already sorted files in level-(L+1). Table 1 summarized the merging rules for level-L, either copy or drop action will move the "key pointer" forward, the "key pointer" remains unchanged if nothing applied to the key. During the merge process, the old files are discarded and the new files are added to the serving state. The space occupied by deleted or stale data is reclaimed when compaction happens. Theses lazy deletes trade disk space for large sequential disk I/Os which makes the compaction far more efficient than the conventional random update.

Index: Because the data stored in container is layered, a single get operation may involve multiple disk I/Os. Specifically, for a data chunk D and its fingerprint F, LevelStore checks whether the fingerprint F is in the hash table. If not found there, it returns null. Otherwise it returns the container ID. Then the fingerprint F will be sent to the corresponding container as the request of D. Container Manager will handle the request by searching the fingerprint F by level, then return the chunk D found in the youngest level. You should notice that, for the fingerprint F, there may be multiple data chunks in the container due to the update and deletion operation. The younger the level is, the fresher the data are. For example, the chunks in level-0 are fresher than they in level-1.

This index design supports the dynamic load balancing of LevelStore. Depending on the capacity and performance of the storage node, the different storage node can be assigned different containers. We can control the number of levels, the sstable size and numbers in each level to determine the container capacity. The different capacity of containers determines the proportion of the data traffic between them. This kind of design can boost one node's performance and scalability and does not affect the stability of the upper index which has a higher efficiency.

ANALYSIS

Compared to other key-value store systems, the design of LevelStore has more system parameters, such as the number of containers, sstable size and the maximum levels in container and so on. To illustrate the performance of LevelStore, we build a simple model to analyze. Table 2 presents the design parameters of LevelStore.

Table 2: Design parameters of LevelStore

Symbol	Meaning	Example
M	Memtable size	4 MB
N	No. of files in a level	100
L	No. of levels	10
S	Common ratio	2
P	No. of nodes	2^{20}
Z	Disk speed	100 MB sec^{-1}
T	Compaction throughput	20 MB sec^{-1}
C	Physical capacity	100 GB

Container capacity: The capacity of each container consisted of all the sstables in each level. Sstables in level-0 are generated from memtable, so the file size is the same as memtable. File size of current level is larger than the previous one, usually multiplied by the common ratio. Suppose that there are L levels and N files in each level with the common ratio S, so the capacity a container can support is:

$$C_p = S^0 NM + S^1 NM + ... + S^{L-1} NM = \frac{S^L - 1}{S - 1} NM \quad (1)$$

For instance, a container is built with L = 10 levels, N = 500 files in each level, with memtable size M = 4 MB and the common ratio S = 2, the capacity a container can support is almost 2 TB. We can support more capacity by setting the parameters properly.

Storage nodes: Due to its high scalability, LevelStore can scale up dynamically and cost-effectively by adding containers. If all the containers are stored in one node, when the container number increases, LevelStore overall performance will be greatly decreased. To improve the overall performance, each container in the system can be assigned a storage node, but this method makes the system spending too much. To address the problem, LevelStore uses the concept of "virtual nodes" which was proposed by DeCandia *et al.* (2007), a virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple virtual nodes. With each virtual node contained multiple containers.

Assume the LeveStore has $P = 2^{20}$ containers, using Eq. 1, the total capacity LevelStore can support is:

$$C_t = PC_p \quad (2)$$

For a 2 TB container, a 20-bit container ID can represent a maximum physical backup capacity of 2048 PB, thus is sufficient for a PB-scale storage system.

Compaction: Even though the compaction operation is performed in background asynchronous, we can estimate the throughput of compaction through a simplified model. For example, to compact 1 MB data, LevelStore reads 1 MB data from level-L and about S MB data from level-(L+1), then write S MB data to Level-(L+1). Therefore, one compaction needs (2S + 1) MB data of disk I/O. Assume the disk speed is Z, the throughput of compaction is:

$$T=\frac{Z}{2S+1} \quad (3)$$

According to Eq. 3, disk speed Z is fixed, in order to achieve a higher throughput of compaction, we should select a suitable size for S.

EVALUATION

We have implemented LevelStore in the Linux platform and a ChunkFarm prototype according to Yang *et al.* (2011) for the purpose of performance comparison.

In this section, we evaluated LevelStore using several experiments. First, we evaluate the performance of a single-server LevelStore with ChunkFarm using real world data in terms of data compression ratio. Next, we compare with ChunkFarm in terms of throughput. Finally, we measure the scalability of LevelStore.

We used two main datasets for testing. The dataset-1 is obtained from a server in our laboratory which was made up of 10 backup versions. Each version includes one week of backup data. The server is mainly used for code development, file sharing and experiment. A large percentage of files on the server remained unchanged between adjacent versions. The dataset-2 is a synthetic dataset, in order to obtain a sufficiently large dataset for testing. Because the factors affecting deduplication (data compression ratio) is the distribution and amount of duplicated fingerprints (redundant locality). Discarding data chunks has no impact on the correctness of the test, so we stored everything except the actual data chunks. The dataset contains 10 backup versions too. The 1st version of backup stream does not contain any duplicate fingerprints. Subsequent, the 2nd, 3rd, …, 10th version of backup contain 10, 20, …, 90% duplicate fingerprints.

In our experiments, the LevelStore ran on a 6-node Linux cluster, in which each node was equipped with a 16-core Xeon E5450 at 3GHz with 12288KB of total L3 cache and 32GB RAM at 1333MHz. Our 24 TB disk array consists of 12 disks, 2 TB each. Note that container uses at most two CPUs (one to handle the request and one for background compaction), we deployed 8 containers in each node to maximum each node's capacity. Both LevelStore and ChunkFarm use 1 GB memory for index cache and employ the CDC chunking mechanism with an expected chunk size of 8 KB, while LevelStore used another 256 MB memory for write buffer which contained 64 memtables, that is enough for the containers.

Compression ratios: Figure 3a shows the logical capacity and the physical capacity of LevelStore under dataset-1, compared with ChunkFarm. In the backup of the 1st version, LevelStore and ChunkFarm reduced approximately 3.8 and 3.4% of the logical data, respectively. This is because nothing stored before the first backup, both LevelStore and ChunkFarm can only eliminate duplicate chunks included in the backup stream. After the 1st backup, despite the large variation in the actual number, duplicate chunks elimination becomes extremely effective both in LeveStore and ChunkFarm. This illustrates that a large proportion of data does not change between two adjacent backup versions. The detailed compression ratios are shown in Fig. 3b, LevelStore almost made the same deduplication effect of

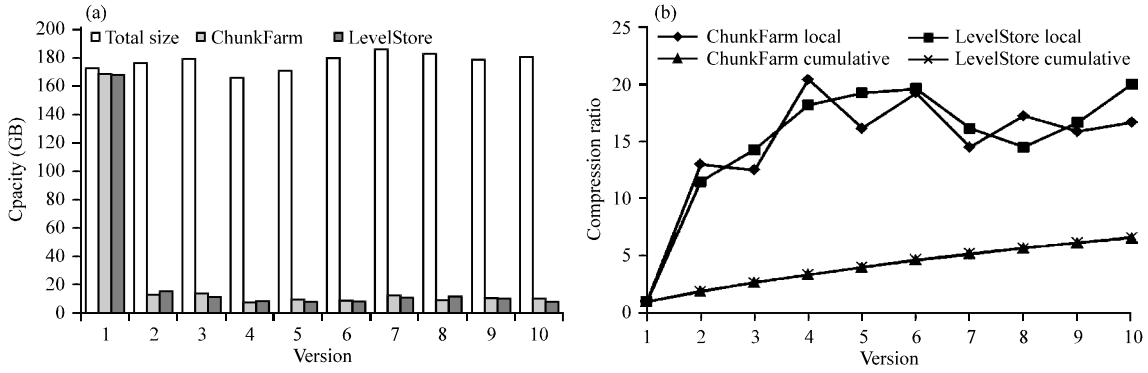

Fig. 3(a-b): Comparison of LevelStore and ChunkFarm under dataset-1 (a) Logical/Physical capacities (b) Compression ratios

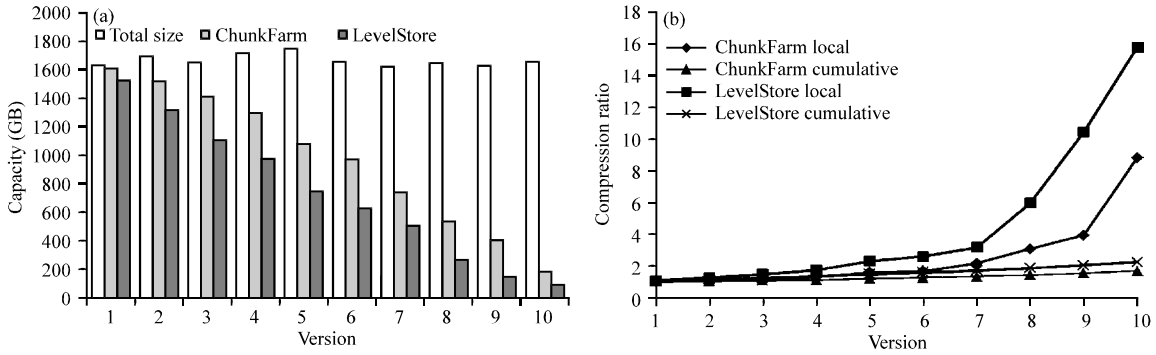

Fig. 4(a-b): Comparison of LevelStore and ChunkFarm under dataset-2 (a) Logical/Physical capacities (b) Compression ratios

ChunkFarm: A total of 1.72 TB logical data were backed up (an average size of about 176.8 GB) and the actual physical data stored in LevelStore and ChunkFarm were roughly the same, about 266.38 GB and 269.13GB, achieving cumulative data compression ratios of about 6.64 to 1 and 6.57 to 1, respectively. During the 10 backup versions, both LevelStore and ChunkFarm's local compression ratios changes quite a bit, whereas the cumulative compression ratios are quite stable.

The results of dataset-2 are show in Fig. 4. Like Fig. 3, duplicate data chunks elimination tends to be ineffective in dataset-2 during the 1st backup, this is because most chunks are new during its initial deployment. In the next nine backups, with the increase of backup version, the duplicate data eliminated by LevelStore and ChunkFarm become more and more which makes the physical data stored by LevelStore and ChunkFarm reduced gradually. At the end of 10th backup, the logical capacity reaches about 16.3 TB and the corresponding physical capacity stored in LevelStore and ChunkFarm were 7.20 TB and 9.68 TB, achieving cumulative data compression ratios of 2.64 to 1 and 1.68, respectively. This is because the significant amounts of data changed between the adjacent versions and addition of new data. But in each backup the compression ratio varies greatly, we can see it from Fig. 4b. For ChunkFarm, the 1st compression ratio is 1.11 to 1 and the 10th compression ratio is 8.80 to 1. For LevelStore, the 1st compression ratio is 1.07 to 1 and the 10th compression ratio is 15.72 to 1. We can see that LevelStore has a higher compression ratio that ChunkFarm, this is because LevelStore can reclaim the deleted or stale data when compaction happens, thus further improved the space utilization of the disk.

Throughput: We used dataset-2 to determine the throughput of LevelStore and ChunkFarm. We used 4 backup streams, each backup stream contains 10 versions of the backup data. There are two main advantages of using the synthetic dataset. The first is that various models can be deployed easily. The second is the distribution of duplicate data makes it convenient to test the throughput on different redundancy load.

Figure 5a presents a read throughput comparison between LevelStore and ChunkFarm. LevelStore achieves over 200 MB sec^{-1} high read throughput for all versions. For ChunkFarm, the read throughput is 196 MB sec^{-1} for the 1st version and stay around 220 MB sec^{-1} for future versions. Throughput keeps steadily as backup version increases. The main reason for the quite stable read throughput in both systems is that future versions have more duplicate data chunks than the first new. The bottleneck of the system has switched from the CPU to network I/O. In contrast to LevelStore, ChunkFarm depends on index update in-batch servicing the outcomes of multiple index lookup in-batch asynchronous thus promotes the backup performance.

Figure 5b shows the write throughput of LevelStore and ChunkFarm. Both systems deliver high write throughput. The write throughput of LevelStore increases from 143 MB sec^{-1} to 219 MB sec^{-1} for version 1 through 6, then sustained 230 MB sec^{-1} for version 7 through 10. ChunkFarm maintains write throughput ranging from the lowest of 106 MB sec^{-1} for version 1 to the highest 217 MB sec^{-1} for version 9. At the end of version 10, ChunkFarm achieves throughput of 213 MB sec^{-1}.

For version 1, the write throughput is lower because most chunks are new and absence of duplicates. Throughput improves steadily as the duplicated chunks increase. As expected, LevelStore keeps a high write throughput at the end of experiment. Such a high performance is mainly attributed to the memtable and compaction which turn the random small writes to sequential write and manage the serialization of containers

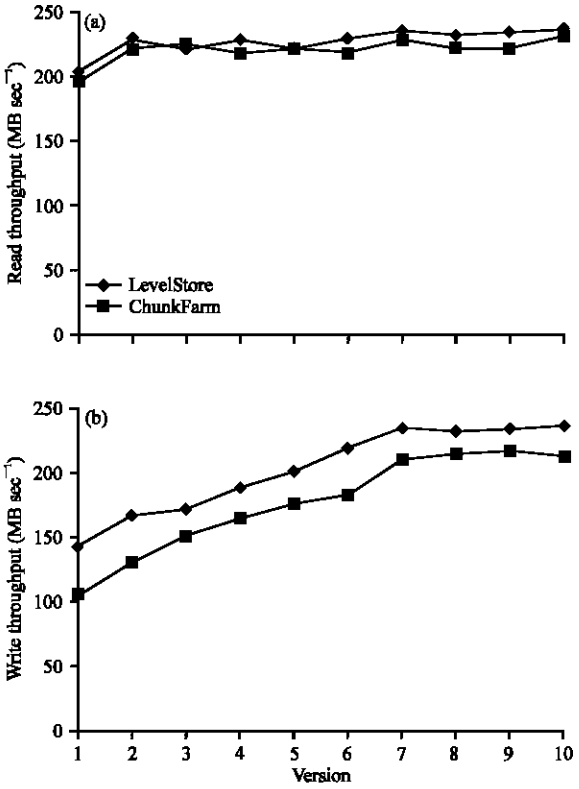

Fig. 5(a-b): Comparison of read and write throughput; (a) Read throughput and (b) Write throughput

to disk. However, write throughput of the ChunkFarm was relatively low. This is because the write throughput of ChunkFarm depends primarily on index update in-batch. Index update in-batch needs comparing the fingerprints then updating the index fetched from disk, the multiple disk I/Os become a performance bottleneck.

The experiments show that when backing up, LevelStore's write throughput is faster than ChunkFarm while maintaining almost the same read throughput as ChunkFarm.

Scalability: To determine the scalability of LevelStore, we first varied the size of disk capacity on one backup server. Figure 6a presents a throughput comparison between LevelStore and ChunkFarm which is obtained by averaging the results of 10 versions. The result indicates that LevelStore can support system backup capacity of 1TB while achieve a throughput of about 230MB sec^{-1}. With the system capacity increases, LevelStore can also maintain a high throughput about 200MB sec^{-1} to support capacity of 24TB. When the system capacity is 1TB, ChunkFarm keeps the almost high throughput with

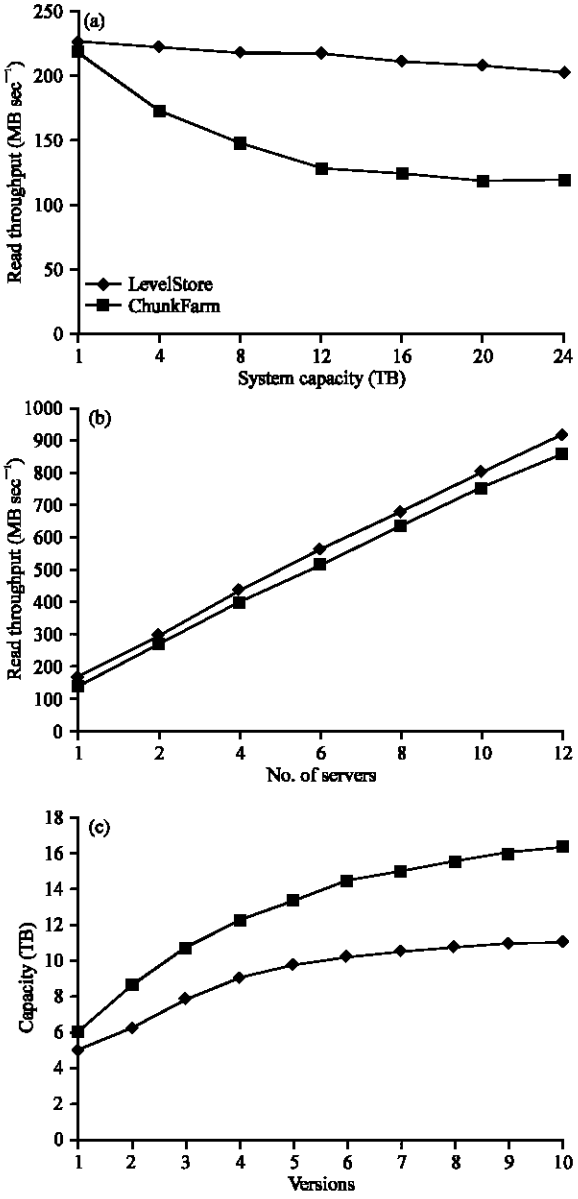

Fig. 6(a-c): Comparison of scalability; (a) Throughput under different system capacities, (b) Aggregate throughput and (c) Comparison of physical capacity

LevelStore. However, throughput declines rapidly with the increasing system capacity. Finally, it achieves throughput of 120 MB sec^{-1}. While system capacity increased by 24 times, LevelStore and ChunkFarm's throughput dropped by 10 and 45%, respectively. This means that the throughput of LevelStore will scale well in terms of system capacity.

Then we measure the aggregate throughput by varying the number of storage nodes. As described

before, we can increase the storage node by introducing "virtual node". Specifically, we have 12 virtual nodes on the 6 servers, each server has 2 virtual nodes. During backup, the backup streams write their synthetic fingerprint versions to the backup servers in parallel, with each stream following its version order.

Figure 6b shows the measured results of throughput under multi-server deployments. As we can see, both systems deliver well scalability, using more backup servers, the aggregate throughput increase linearly. ChunkFarm has negligible performance impact. This is because for multi-server throughput, the system performance is bounded by the disk I/O which overshadowing the effects of CPU overhead.

Figure 6c compares the physical capacity of LevelStore and ChunkFarm with 12 backup servers. For convenient, we leave out the logical capacity. After the deduplication, most duplicated chunks are discarded, both systems' physical capacity increased when backup more and more versions, but ChunkFarm grows faster than LevelStore. At the end of 10th version, LevelStore has backed up about 11.04TB while ChunkFarm has backed up 16.34TB which saves as much as 32% of the space. This is because in addition to the new data, there is deleted data between adjacent backup versions. LevelStore can reclaimed those data in container when compaction happens while ChunkFarm can't.

The results demonstrate that LevelStore is able to scale to high capacity with almost no performance decrease. We are confident that our system would scale to higher capacities, given more resources.

CONCLUSION

This study presents LevelStore, a high performance deduplication storage system which introduced a hierarchy model with other system techniques. LevelStore uses a simple directly index method to locate the chunks which provides fundamental support to system scalability. For incoming data chunks, it takes advantage of the sorted string table to judiciously turn the random small disk I/O to sequential large disk I/O for fingerprint insert and update, yielding excellent deduplication throughput.

The experiments show that the high performance and scalability of the LeveStore. LevelStore can achieve over 180 MB sec^{-1} throughput when supporting 24TB physical capacity with one backup server. While using a cluster of 12 backup servers, LevelStore is shown to scale cost-effectively in both throughput and physical backup capacity, achieving an aggregate throughput of 915 MB sec^{-1} and saving 32% of physical backup capacity.

ACKNOWLEDGMENTS

This study was partly supported by National Natural Science Foundation of China (No. 61272061), the fundamental research funds for the central universities of China (No.531107040263) and Hunan Natural Science Foundation of China (No. 10JJ5069).

REFERENCES

Bloom, B.H., 1970. Space time trade-offs in hash coding with allowable errors. Commun. ACM, 13: 422-426.

Bobbarjung, D.R., S. Jagannathan and C. Dubnicki, 2006. Improving duplicate elimination in storage systems. Trans. Storage, 2: 424-448.

Bolosky, W.J., S. Corabin, D. Goebel and J.R. Douceur, 2000. Single instance storage in windows 2000. Proceedings of the 4th Conference on USENIX Windows Systems Symposium, Volume 4, August 3-4, 2000, Redmond, WA., pp: 2-2.

Burrows, J.H., 1995. Secure hash standard. National Technical Information Service. Springfield, VA., April 1995. http://www.dtic.mil/cgi-bin/GetTRDoc?Location = U2&doc = GetTRDoc.pdf&AD = ADA406543.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati and A. Lakshman *et al.*, 2007. Dynamo: Amazon's highly available key-value store. Proceedings of the Symposium on Operating Systems Principles, October 14-17, 2007, Stevenson, WA., pp: 1-6.

EMC, 2008. Automated archiving made simple, affordable and secure. EMC Centera: Content-Addressed Storage System Data Sheet, pp: 1-3.

Guo, F. and P. Efstathopoulos, 2011. Building a high-performance deduplication system. Proceedings of the USENIX Annual Technical Conference, June 15-17, 2011, Portland, OR., USA., pp: 25.

Jain, N., M. Dahlin and R. Tewari, 2005. TAPER: Tiered approach for eliminating redundancy in replica synchronization. Proceedings of the 4th USENIX Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA., pp: 281-294.

Jin, K. and E.L. Miller, 2009. The effectiveness of deduplication on virtual machine disk images. Proceedings of SYSTOR: The Israeli Experimental Systems Conference, May 2009, Haifa, Israel, pp: 1-12.

Kulkarni, P., F. Douglis, J.D. LaVoie and J.M. Tracey, 2004. Redundancy elimination within large collections of files. Proceedings of the Annual Conference on USENIX Annual Technical Conference, June 9-14, 2004, Boston, MA., pp: 59-72.

Lakshman, A. and P. Malik, 2010. Cassandra: A decentralized structured storage system. Operating Syst. Rev., 44: 35-40.

Lillibridge, M., K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise and P. Camble, 2009. Sparse indexing: Large scale, inline deduplication using sampling and locality. Proceedings of the 7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA., USA., pp: 111-123.

Muthitacharoen, A., B. Chen and D. Mazieres, 2001. A low-bandwidth network file system. Proceedings of the 18th ACM Symposium on Operating Systems Principles, October 21-24, 2001, Banff, Canada, pp: 174-187.

Pugh, W., 1990. Skip lists: A probabilistic alternative to balanced trees. Communi. ACM, 6: 668-676.

Quinlan, S. and S. Dorward, 2002. Venti: A new approach to archival storage. Proceedings of the 1st USENIX Conference on File and Storage Technologies, January 2002, Monterey, CA., pp: 89-102.

Rabin, M.O., 1981. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, May 1981. http://www.xmailserver.org/rabin.pdf.

Shilane, P., M. Huang, G. Wallace and W. Hsu, 2012. WAN optimized replication of backup datasets using stream-informed delta compression. Trans. Storage, Vol. 8 10.1145/2385603.2385606

Wei, J., H. Jiang, K. Zhou and D. Feng, 2010. Mad2: A scalable high-throughput exact deduplication approach for network backup services. Proceedings of the 26th Symposium on Mass Storage Systems and Technologies, May 3-7, 2010, Incline Village, NV., pp: 1-14.

Yang, T., H. Jiang, D. Feng and Z. Niu, 2010. DEBAR: A scalable high-performance deduplication storage system for backup and arching. Processing of the IEEE International Symposium on Parallel and Distributed, April 19-23, 2010, Atlanta, GA., pp: 1-12.

Yang, T.M., D. Feng, Z.Y. Niu and Y. Wan, 2011. Scalable high performance de-duplication backup via hash join. J. Zhejiang Univ. Sci., 11: 315-327.

You, L., K. Pollack and D.D.E. Pollack, 2005. Deep store: An archival storage system architecture. Proceedings of the 21st International Conference on Data Engineering, April 5-8, 2005, Tokyo, pp: 804-815.

Zhu, B., K. Li and H. Patterson, 2008. Avoiding the disk bottleneck in the data domain deduplication file system. Proceedings of the 6th USENIX Conference on File and Storage Technologies, February 26-29, 2008, San Jose, CA., pp: 269-282.