Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Scalable high performance de-duplication backup via hash join[*]

Tian-ming YANG, Dan FENG[†‡], Zhong-ying NIU, Ya-ping WAN

(*Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*)

[†]E-mail: dfeng@hust.edu.cn

Received July 22, 2009; Revision accepted Nov. 1, 2009; Crosschecked Dec. 31, 2009

Abstract: Apart from high space efficiency, other demanding requirements for enterprise de-duplication backup are high performance, high scalability, and availability for large-scale distributed environments. The main challenge is reducing the significant disk input/output (I/O) overhead as a result of constantly accessing the disk to identify duplicate chunks. Existing inline de-duplication approaches mainly rely on duplicate locality to avoid disk bottleneck, thus suffering from degradation under poor duplicate locality workload. This paper presents Chunkfarm, a post-processing de-duplication backup system designed to improve capacity, throughput, and scalability for de-duplication. Chunkfarm performs de-duplication backup using the hash join algorithm, which turns the notoriously random and small disk I/Os of fingerprint lookups and updates into large sequential disk I/Os, hence achieving high write throughput not influenced by workload locality. More importantly, by decentralizing fingerprint lookup and update, Chunkfarm supports a cluster of servers to perform de-duplication backup in parallel; it hence is conducive to distributed implementation and thus applicable to large-scale and distributed storage systems.

Key words: Backup system, De-duplication, Post-processing, Fingerprint lookup, Scalability
doi:10.1631/jzus.C0910445 **Document code:** A **CLC number:** TP309.3

1 Introduction

Disk-based de-duplication storage is emerging as a key technology to fight against the data growth in data protection (Quinlan and Dorward, 2002; You *et al.*, 2005; Eshghi *et al.*, 2007; Rhea *et al.*, 2008; Zhu *et al.*, 2008; Dubnicki *et al.*, 2009; Lillibridge *et al.*, 2009). By eliminating duplicate data across the system, a disk-based de-duplication storage system can achieve far more efficient data compression than tapes. Although the actual compression rate depends on a number of variables such as the de-duplication granularity, the type of data processed, and the frequency of change rate or backup policy employed, a greater than 20:1 compression rate can be generally achieved (Zhu *et al.*, 2008). This high compression rate dramatically reduces the storage requirements for data protection, making it more cost-effective and practical to build a massive disk-based storage system for backup and archiving.

The most widely used de-duplication method is to divide a file or stream into chunks and drastically reduce the storage required for those chunks by identifying and coalescing duplicate chunks within and between files. Specifically, each chunk is stored and addressed by the hash value of its content, called the ‘fingerprint’ of the chunk, to ensure that only a ‘unique’ chunk is stored on disk. A disk index is used to establish mapping between fingerprint and the location of its corresponding chunk on disk. The key performance challenge of this approach is to reduce the significant disk input/output (I/O) overhead as a result of constantly accessing the disk index to search for the data chunks. Considering that the location of a coming fingerprint in the index is essentially random

[‡] Corresponding author

[*] Project supported by the National Basic Research Program (973) of China (No. 2004CB318201), the National High-Tech Research and Development Program (863) of China (No. 2008AA01A402), and the National Natural Science Foundation of China (Nos. 60703046 and 60873028)

and the entire index is usually too large to fit in memory, without another technique, the performance of de-duplication would be limited to the random I/O performance of the index disks, which, for current technology, is a few hundred accesses per second. The Venti system (Quinlan and Dorward, 2002), for example, reported a throughput of less than 6.5 MB/s, even with a redundant array of inexpensive disks (RAID) of eight disk drives for index lookups in parallel.

There are two main methods of implementing de-duplication: inline and post-processing. Inline de-duplication is de-duplication where the data is de-duplicated before it is written to disk, as opposed to post-processing de-duplication where the data is first accumulated in a temporary on-disk staging area and then de-duplicated in batch mode. One advantage of inline de-duplication is that the data is de-duplicated on the fly; no extra disk space is required to temporarily hold the raw data, thereby maximizing cost-savings achieved by reduced disk requirement.

Existing inline approaches such as the data domain file system (DDFS) (Zhu *et al.*, 2008) and sparse indexing (Lillibridge *et al.*, 2009) exploit duplicate locality to avoid the fingerprint lookup disk bottleneck when done on a large scale. The duplicate locality is the tendency for chunks in backup streams to reoccur together. That is, when chunks A, B, and C appear consecutively in today's backup stream, tomorrow when chunk A appears, chunks B and C follow with a high probability. Traditional stream-based backup workloads, such as large directory trees coalesced into a tar file, or data generated conforming to legacy tape library protocols, contain large stretches of repetitive data among streams generated on a daily or weekly basis, hence presenting well duplicate locality; i.e., identical or similar files (in other words, a great number of chunks) tend to reoccur together. By exploiting duplicate locality, inline techniques can achieve reasonable de-duplication throughputs under these workloads using an appropriate number of random access memory (RAM) for a given system scale.

DDFS (Zhu *et al.*, 2008) uses a combination of the in-memory Bloom filter (Bloom, 1970) and the cache technique to improve the performance of index lookups. The in-memory Bloom filter filters most of the new fingerprints before querying the disk index, and the cache technique exploits duplicate locality to prefetch groups of chunk fingerprints that are more likely to be accessed together in the near future. Using these techniques, DDFS has achieved backup throughputs of over 100 MB/s. The RAM overhead is determined by the required system physical capacity. For an expected chunk size of 8 KB, it requires a 1 GB in-memory Bloom filter to store 2^{30} fingerprints of about 8 TB physical storage, which results in a reasonably low false positive rate of 2% (Broder and Mitzenmacher, 2005; Zhu *et al.*, 2008). In order to keep the same 2% false positive rate, the size of the in-memory Bloom filter must linearly increase with the system capacity. For example, a 1-perabyte physical capacity will need an at least 120 GB in-memory Bloom filter.

Sparse indexing (Lillibridge *et al.*, 2009) performs de-duplication by dividing an incoming backup stream into relatively large segments and de-duplicating each segment against only a few of the most similar previously-stored segments. To identify similar segments, Lillibridge *et al.* (2009) chose a small portion of the chunks in the segment as samples, and exploited the sparse index to map these samples to the existing segments in which they occurred. Using a very low sampling rate (e.g., one sample roughly every 64 chunks), the sparse index can be made sufficiently small to reside in the server memory so that only a few disk seeks are required per segment and thus the fingerprint-lookup disk bottleneck is avoided.

Both DDFS and sparse indexing, however, are centralized systems and do not do global de-duplication in a distributed environment. Moreover, both systems perform poorly in the case of poor duplicate locality: with DDFS, the index cannot be cached effectively and thus throughput deteriorates, whereas with sparse indexing, the sampling becomes ineffective and de-duplication quality degrades. Therefore, both approaches are unsuitable for low-locality backup workloads in which the probability that identical or similar files reoccur together in a given window of time is very low, because they mainly consist of individual files (not large data streams) that arrive in a random order from disparate sources. Several scenarios generate such workloads: network-attached storage (NAS) clients that make file backup requests via an NFS/CIFS interface; email servers that periodically backup emails received from thousands or tens of thousands of users; continuous data protection (CDP), where files are backed up in the order that they have changed. In fact, there has been research focusing on random file de-duplication backup recently. Extreme binning (Bhagwat *et al.*, 2009) is such a backup system that provides inline chunk-based de-duplication for random files with a poor duplicate locality. It exploits file similarity instead of locality, and makes only one disk access for the fingerprint lookup per file, which gives a reasonable throughput and can scale gracefully to multiple nodes. But extreme binning is designed specially for file-based backup and cannot be applied to traditional stream-based backup where individual files and directories are encoded into large image files. It would be desirable to have a de-duplication backup system that does not rely on workload characteristics to achieve a high throughput, while allowing for easy scale out for large scale distributed environments.

In this paper, we present Chunkfarm, a post-processing de-duplication backup system, designed to improve capacity, throughput, and scalability for de-duplication. Chunkfarm performs de-duplication backup in two phases. In phase-1, files (which can be ordinary files or large backup images, e.g., tar files) are transmitted from the backup client, and temporarily stored on the local disk of the backup server. In phase-2, the backup server reads files from its local disk, performs de-duplication to write new chunks to a chunk repository, and then updates the new fingerprints to the disk index.

To improve throughput in phase-2, Chunkfarm performs fingerprint lookup and update in-batch using hash join algorithms (Shapiro, 1986) called 'index lookup in-batch (ILB)' and 'index update in-batch (IUB)'. ILB and IUB exploit memory cache and the disk index property to judiciously turn the notoriously random and small disk I/Os of fingerprint lookup and update into large sequential disk I/Os, hence achieving a high de-duplication throughput. Compared with DDFS and sparse indexing, this approach has the advantage that both its de-duplication quality and throughput are insensitive to the locality of the input data; it hence works fairly well under both traditional stream-based workloads and random file workloads. It also consumes fewer expensive RAM resources than the above inline solutions, especially DDFS, to support an equivalent level of system capacity while maintaining a high throughput. More importantly, by decentralizing fingerprint lookup, ILB and IUB can support a cluster of servers to perform de-duplication backups in parallel, thus rendering Chunkfarm highly efficient, adaptive, and scalable.

The disk index should be designed appropriately small since a smaller disk index can help improve the batch process efficiency due to the fact that it consumes less time to scan through the index. We have analyzed the index overflow probability to help select an appropriate index bucket size that can effectively improve the disk index utilization to enable a relatively small disk index for a given system backup capacity.

2 System architecture

The Chunkfarm (Fig. 1) uses a cluster of backup servers to provide large-scale and high-performance data backups. A metadata server (MDS) is designed to take charge of metadata management, the deployment of security mechanisms, and the control of system-wide activities such as object scheduling and load balancing among backup servers. A user can define job objects through the MDS to backup their data to the system automatically. A backup job object

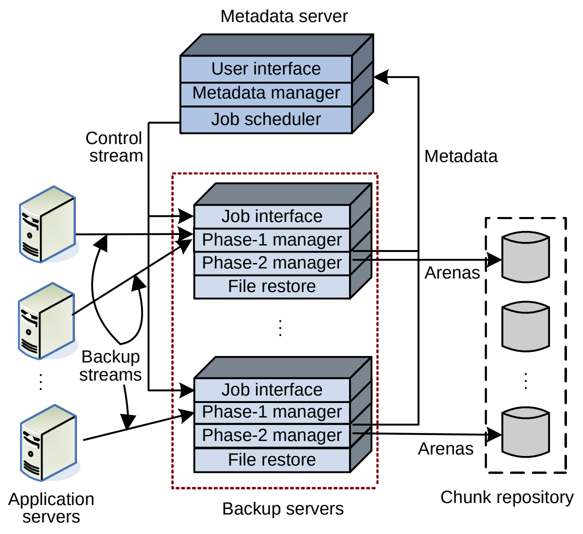

Fig. 1 Architecture of Chunkfarm

includes at least three attributes, namely, (1) a client attribute that specifies a backup client, which runs as a background daemon on the application server that has data to be backed up, for the job, (2) a dataset attribute that specifies the list of files (maybe backup image files) and directories needing backup on the application server, and (3) a schedule attribute that specifies when the backup job should be scheduled to run. When initiating a backup job, the MDS assigns an appropriate backup server to run the job based on load balancing consideration.

The backup server runs in two phases. The first phase is performed by the phase-1 manager, which runs a backup job to receive backup stream (files) from the backup client, divides files into variable-sized chunks using the content-defined chunking algorithm (CDC) (Muthitacharoen *et al.*, 2001), computes the SHA-1 (Secure Hash Standard, 1995) hash (160 bits) of each chunk as its fingerprint, and then temporarily appends the fingerprint h and its data chunk $D(h)$ to a local on-disk chunk log as the form $<h, D(h)>$. In the process, the file index is built and sent, together with the file metadata, to the MDS. A file index, which facilitates retrieving the file from the system, is a sequence of fingerprints that map to the file chunks. In order to eliminate the internal duplicate chunks in the backup stream, the phase-1 manager caches the new fingerprints in the memory and if an incoming fingerprint h is found in the cache, its data chunk $D(h)$ is discarded. During data transmission, the generated fingerprints are also collected to a file called the 'undetermined fingerprint file', which represents the fingerprints needing to be further identified through index lookups.

The second phase is initiated by the MDS, and executed by the phase-2 managers of the backup servers. First, the system performs ILB to identify new chunks; ILB avoids the random small disk I/Os for fingerprint lookups and thus significantly improves the disk index lookup efficiency. After the ILB process, each phase-2 manager obtains its own fingerprint-lookup results. Secondly, each phase-2 manager reads files from its local on-disk chunk logs and refers to the ILB result to store new chunks to the chunk repository efficiently. Finally, the system performs IUB to write new fingerprints to the disk index.

The chunk repository can be built on a cluster of storage nodes; it provides a global disk-based storage pool which consists of fixed-sized arenas for backup servers to store new chunks. An arena is self-described in that a metadata section located before the data section of the arena stores metadata describing the chunks stored in the data section. The chunk metadata, which locates a particular chunk in its arena, includes the fingerprint, chunk size, and storage offset of this chunk. Each arena is filled with a large number of data chunks in an append-only manner and is sized to facilitate operations such as allocating, migrating among different storage nodes, and copying to removable media. An arena in the chunk repository is identified uniquely by its arena ID. A disk index is used to establish mapping between a chunk fingerprint and the arena holding the chunk in the repository.

2.1 Disk index

Chunkfarm uses a disk index, as shown in Fig. 2, to locate a chunk within the chunk repository. The disk index is implemented as a hash table that contains fixed-sized buckets with each bucket containing a certain number of entries and each entry storing a fingerprint hashed to the bucket. Since the fingerprint itself is essentially random, for an index containing a total of 2^n buckets, we simply take the first n bits of a fingerprint as the bucket number to map this fingerprint to a corresponding bucket. Owing to the good randomness resulting from the SHA-1 algorithm, the fingerprints will be distributed to the index buckets in a sufficiently uniform manner. And given a sufficiently large number of appropriately sized buckets, the index can achieve a relatively high utilization before it begins to overflow. Improving the disk index utilization can not only reduce the metadata storage overhead but also improve the Chunkfarm backup

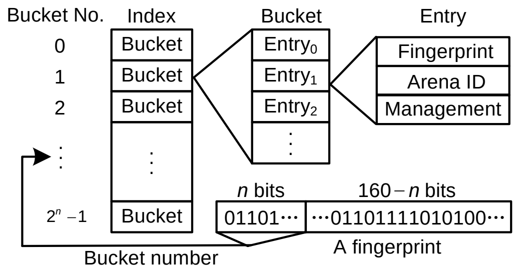

Fig. 2 Structure of the disk index

performance, since a small disk index can effectively reduce the time overhead of the sequential index lookup and update.

Supposing a disk index with a total of 2^n (n>20) buckets with each bucket having a limited capacity of b (b>1) fingerprints, and that fingerprints are inserted into the disk index in a uniform manner. That is, the probability of a fingerprint being inserted into any given bucket is $1/2^n$. Let C be the event that there exists a bucket that overflows before a total of $\eta\times b\times 2^n$ ($0<\eta<1$) fingerprints have been inserted into the disk index, with η being the disk index utilization. Since 2^n is sufficiently large, the number of fingerprints inserted into a given bucket is subject to the Poisson distribution with parameter ηb. Then, the probability that event C happens can be expressed as

$$\Pr(C) < 2^n\left(1-\sum_{k=0}^{b}\frac{(\eta b)^k \mathrm{e}^{-\eta b}}{k!}\right). \tag{1}$$

We use Eq. (1) to estimate the upper bound of Pr(C) for a 512 GB disk index under different bucket sizes and disk index utilizations, and the results are shown in Table 1. We construct the disk index buckets using disk blocks with each disk block (usually 512 bytes in size) storing up to 16 fingerprint entries (an entry is 32 bytes with 20 bytes for the fingerprint, 5 bytes for arena ID, and the remaining 7 bytes for entry management such as the chunk reference counter). Then, for a given bucket size listed in Table 1, its corresponding parameters b and n can be determined. For example, an 8 KB bucket contains 16 disk blocks and thus can store up to 256 fingerprint entries, giving rise to b=256 and n=$\log_2$(512 GB/8 KB)=26. The result obtained from Eq. (1) based on these parameters implies that, for a 512 GB disk index with 8 KB-sized buckets, the probability that there exists a bucket that overflows before the disk index utilization reaches 65% is less than 1.07%. In other words, if a bucket overflows, then with a very high probability (over 98.93%) the disk index utilization is over 65%.

Based on the results in Table 1, we have selected 8 KB as the bucket size for the Chunkfarm disk index in order to achieve over 65% disk index utilization for a backup server that may in turn support up to 512 GB disk index. For an expected chunk size of 8 KB, a 512 GB-sized Chunkfarm disk index can support a system physical capacity of over 83.2 TB (=(512 GB/8 KB)×256×8 KB×65%).

Table 1 Calculated upper bound of disk index overflow probability Pr(C)

Bucket size (KB)	η (%)	Pr(C)< (%)
0.5	13	51.00
1	25	7.30
2	41	6.71
4	54	2.21
8	65	1.07
16	75	1.81
32	82	1.25
64	87	1.05

η: disk index utilization; Pr(C): disk index overflow probability

The additional computation overhead due to searching a fingerprint in a large 8 KB-sized bucket, which can contain up to 256 fingerprints, is negligible for modern processors. We have tested the speed of fingerprint lookup in main memory using an Inter Xeon DP 5365 3.0 GHz CPU running at 47% utilization, and have achieved a speed of 1.944 million fingerprints per second even with each fingerprint requiring 256 comparison operations. It is precisely this high speed of in-memory fingerprint lookup that motivates one to use large-sized bucket for the disk index to compensate for the relatively low speed of disk I/O for the sequential index lookup and update. The large 8 KB-sized bucket has almost no adverse impact on the performance of the random on-disk fingerprint lookup. Since the time overhead of a random small disk I/O stems mainly from the disk seek rather than data transfer, the time overhead of a random 8 KB disk I/O is almost the same as that of a random 512-byte disk I/O. Moreover, since the random disk I/Os for fingerprint lookup in Chunkfarm occur only during data restoration, the locality preserved caching (LPC) (Zhu *et al.*, 2008) read cache can be used to eliminate most of these random disk I/Os.

3 De-duplication backup

3.1 Index lookup in-batch

The ILB algorithm is derived from the hash join algorithm (Shapiro, 1986) which computes the equijoin of relations in database systems. Here, the goal of the ILB is to compare efficiently two sets of fingerprints labeled S and R, for each fingerprint in S, checking to see if it also exists in R. S is the set of undetermined fingerprints, and R is the set of fingerprints contained in the Chunkfarm disk index, where $|S|\leq|R|$. We use an index cache that is implemented as an in-memory hash table with 2^m buckets to store S. A fingerprint is stored to a certain index cache bucket using its first m bits as the bucket number. Then S *is* automatically partitioned into 2^m consecutive disjoint subsets, i.e., buckets: $S=S_0\cup S_1\cup\cdots\cup S_{2^m-1}$, $S_i\cap S_j=\varnothing$ ($i, j\in\mathbb{Z}$, $0\leq i\neq j\leq 2^m-1$). For a Chunkfarm disk index with 2^n ($n\geq m$) buckets, R is also partitioned into 2^m consecutive disjoint subsets:

$$R=R_0\cup R_1\cup\cdots\cup R_{2^{m-1}},\ \ R_i\cap R_j=\varnothing,$$

where R_k (k=0, 1, …, 2^m-1) contains 2^{n-m} consecutive disk index buckets, namely bucket $2^{n-m}k$ to bucket $2^{n-m}(k+1)-1$. Since we use the first n bits of a fingerprint as the bucket number to hash the fingerprint to the corresponding disk index bucket, the above partitioning satisfies the following condition: for any fingerprint $h\in S_k$ (k=0, 1, …, 2^m-1), if $h\in R$, then $h\in R_k$ (since all the fingerprints in R whose first m bits constitute the binary number k are stored in R_k). Therefore, for the fingerprint lookups, the algorithm simply needs to sequentially read a large bulk of consecutive buckets (i.e., subsets R_k, k=0, 1, …, 2^m-1) from the disk index to the memory to sequentially check the index cache buckets (i.e., subsets S_k). Note that it can sequentially read tens or even hundreds of buckets per I/O. The data transfer rate of sequential large disk I/Os is generally over one order of magnitude faster than that of random small disk I/Os. During the checking process, the algorithm does the following: if a fingerprint in the index cache bucket is found in the corresponding disk-index bucket which has been read to the memory, then it is duplicate—its index cache node is deleted from the index cache; otherwise, its node is retained in the index cache to indicate that it is a new fingerprint to the system. After the completion of the entire lookups, all the new fingerprints are retained in the index cache and will be used in the chunk storing process.

ILB is highly scalable, as it intrinsically supports a cluster of backup servers to perform parallel fingerprint lookups. In a large-scale Chunkfarm system with 2^w backup servers, the disk index is divided into 2^w parts with index part k (k=0, 1, …, 2^m-1) locating on the local disk of backup server k, which maps the fingerprints whose first w bits constitute the binary number k. These 2^w backup servers cooperate to perform parallel index lookup (PIL), as described below. First, each backup server's undetermined fingerprints are divided into 2^w subsets according to their first w bits. Then, these 2^w backup servers exchange their subsets to ensure that backup server k processes the fingerprints whose first w bits constitute the binary number k. After the exchange is finished, backup server k uses its local index part k to perform ILB. Since 2^w ILBs are being performed in parallel, the efficiency of PIL can be significantly higher than that of ILB. After the completion of PIL, the 2^w backup servers exchange their lookup results to ensure that each backup server gets its own lookup results. In the process, a new fingerprint is sent only to one backup server even though it is shared by multiple backup servers. This, in turn, eliminates duplicate chunks among multiple backup servers.

3.2 Storing new chunks to the chunk repository

After completing the index lookups, the backup server sequentially reads fingerprints and chunks from the chunk log and stores new chunks to the chunk repository. It first requests a new arena ID from the chunk repository, and then initializes an empty in-memory arena for the ID to store new chunks. When an in-memory arena is full, it is flushed to the chunk repository and then another new arena ID is requested. In the chunk storing process, the backup server refers to the index cache for fingerprint checking. Specifically, for a fingerprint unit $<h, D(h)>$ read from the chunk log, the algorithm does the following:

1. Check whether fingerprint h is in the index cache. If there, check whether its corresponding arena ID is null. If arena ID is null, copy the current arena ID to h's index cache node and write $<h, D(h)>$ to the current arena; otherwise, discard $<h, D(h)>$.

2. If fingerprint h is not in the index cache, discard $<h, D(h)>$.

The chunk storing process can be very efficient since data is read from the chunk log and written to the chunk repository sequentially. This sequential process approach also preserves chunk locality that helps improve read performance.

3.3 Index update in-batch

After the chunk storing process is completed, all the new fingerprints in the index cache have got their arena IDs, and then the disk index can be sequentially updated using these fingerprints and their arena IDs. The principles of IUB are similar to those of ILB. Like ILB, IUB naturally supports a cluster of servers to perform parallel index update (PIU), which is a similar process to PIL.

In the process of IUB, the disk index can become full; that is, there exist buckets that overflow, in which case, Chunkfarm automatically scales up the index by performing the capacity enlarging algorithm that contains only simple bucket-copying operations. Specifically, constructing a new index with 2^{n+1} buckets from an old index with 2^n buckets, the enlarge algorithm does the following: copying the entries in bucket k (k=0, 1, …, 2^m-1) of the old index to buckets $2k$ and $2k+1$ of the new index to ensure that buckets $2k$ and $2k+1$ store the entries whose fingerprints' first $n+1$ bits constitute the binary numbers $2k$ and $2k+1$, respectively. In addition, if the index also becomes so large in size that it becomes a performance bottleneck, Chunkfarm can further divide the index into multiple parts to be distributed among more backup servers.

4 Experimental evaluation

We have implemented a prototype Chunkfarm in Linux and a prototype DDFS according to Zhu *et al.* (2008) for the purpose of performance comparison. Since the only publicly available DDFS paper did not describe how the disk index is updated, nor could we obtain the detailed updating method from the authors due to proprietary reasons, we implemented the DDFS update by using a sufficiently large in-memory write buffer to collect new fingerprints during backup and writing them to the disk index using the IUB technique after the backup process is finished.

In this section, we evaluated Chunkfarm using several experiments. First, we compared Chunkfarm with DDFS in terms of write throughput using real world data. Note that for Chunkfarm, we focused only on backup phase-2 write throughput since it does actual de-duplication. Next, we evaluate the ILB and IUB algorithms, and compare with DDFS in terms of system backup capacity supported under the same main memory overhead. Finally, we measure the aggregate throughput of multi-server Chunkfarm deployments to check the system scalability.

In our experiments, the Chunkfarm MDS, backup servers, the chunk repository, and the DDFS backup server ran on an 18-node Linux cluster, in which each node was a computer with an Intel Xeon 3.0 GHz CPU, 4 GB RAM, two 1-Gb network interfaces cards (NICs) and one Highpoint RocketRAID 2220 controller attached to eight SATA disks.

4.1 Experimental results on real world data

In this experiment, the Chunkfarm used one backup server with 1 GB memory for the index cache for ILB and IUB, while DDFS used 1 GB memory for the Bloom filter and another 320 MB memory for fingerprint cache—with 256 MB for write buffer and the remaining 64 MB for the LPC cache. Both Chunkfarm and DDFS used 32 GB disk index.

We used two different workloads mainly collected from the heterogeneous unified storage system (HUSt) (Zeng *et al.*, 2006), a massive storage system which was built at the Wuhan National Laboratory for Optoelectronics (WNLO). Workload-1 contained four backup streams each of which was made up of 10 backup versions in their chronological order from a HUSt server, where significant amounts of data remained unchanged between adjacent versions. Workload-2 is collected from different personal workstations at the WNLO Information Storage Division, which were running Windows and frequently accessed the HUSt servers to download or share files. A large percentage of files in workload-2 were downloaded from different HUSt servers, including geographic information system (GIS) pictures, development sources and documentations, meeting records and technology discussions, and research papers and reports in various formats and languages. The workload-2 was also divided into four backup streams, each containing 10 backup versions. We used four personal computers (PCs), each of which ran a backup client to send a backup stream to the server. A workload was running when its four backup streams were written to the de-duplication system in parallel, each stream in the version order. We call the four versions that had the same version number in a workload a 'version set'.

We first ran workload-1 and then workload-2 on Chunkfarm and DDFS respectively. A total of 1.27 TB and 0.68 TB logical data (data in user's or application's perspective) were backed up in workload-1 (an average version size of about 32.5 GB) and workload-2 (an average version size of about 17.4 GB) respectively; the amounts of physical data stored in Chunkfarm and DDFS were roughly the same, about 151 GB and 45 GB, achieving data compression ratios of about 8.61 to 1 and 15.47 to 1 under workload-1 and workload-2, respectively.

Fig. 3 shows the write throughput of Chunkfarm backup phase-2 under workload-1 and workload-2, compared with that of DDFS. The Chunkfarm result was calculated using the amount of time the ILB and chunk storing process consumed divided by the amount of data processed. For both Chunkfarm and DDFS results, the disk index update time was not taken into account since IUB was done after the backup. In the Chunkfarm experiment, ILB ran a total of seven times, with four times under workload-1 and three times under workload-2. The average time spent on an ILB process was about 2.53 min. As can be seen in Fig. 3, Chunkfarm delivered high write throughputs in backup phase-2 for all the version sets in both workload-1 and workload-2. It achieved a write throughout of about 221 MB/s, which was the sustained read throughput of the chunk log, for the first version set in workload-1, and stayed around 197 MB/s for almost all the remaining version sets. The first version set in workload-1 induced a relatively high write throughput, because it did not need to perform ILB during the initial deployment of the system when the disk index was still empty. DDFS achieved a write throughput around 207 MB/s, which was the sustained throughput of the network card in our experiment, for all the version sets in workload-1. This is because that there exist a lot of duplicate localities within workload-1 at the small number of megabytes scale. And the container (8 MB-sized containers were used in DDFS in the experiment according to Zhu *et al.* (2008)) mechanism has well exploited this scale of locality to effectively improve the LPC cache (Zhu *et al.*, 2008) hit rate, combined with the Bloom filter that identifies new fingerprints, the number of random on-disk I/Os for fingerprint lookups has been dramatically reduced. However, write throughput of the DDFS under workload-2 was relatively low and quite fluctuant, with some version sets higher than 100 MB/s and some version sets lower than 50 MB/s. The lowest DDFS throughput was just 34 MB/s, which occurred on the sixth version set of workload-2. By further analyzing the version set, we found that a large proportion of its files were smaller than 100 KB and had their duplicate copies scattered in various parts of the workload-1 versions which have been stored in the system. This small scale of duplicate locality (several tens of kilobytes scale) significantly reduced the efficiency of the LPC cache since pre-fetching a container of fingerprints to the LPC cache cannot create a sufficient number of subsequent cache hits. As a result, the number of random disk I/Os for fingerprint lookup due to cache misses increased, and then the de-duplication write throughput decreased. This indicates that DDFS write throughput is sensitive to the scale of internal duplicate locality of the backup workloads. Therefore, DDFS is unsuitable for random file backups. Unfortunately, we cannot ensure that a backup workload always holds a lot of duplicate localities at several megabytes scale since most of the practical files are no more than 100 KB in size (Tanenbaum *et al.*, 2006; Agrawal *et al.*, 2007).

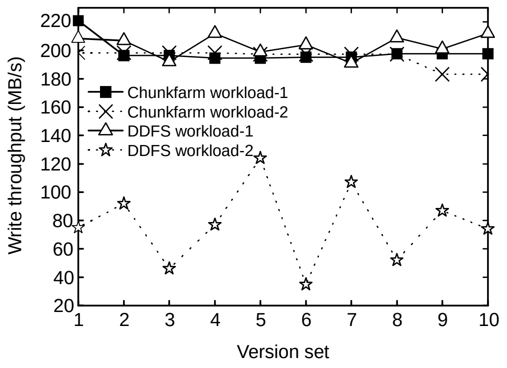

Fig. 3 Write throughput versus version set of Chunkfarm backup phase 2 and DDFS under workload-1 and workload-2

4.2 Performance of ILB and IUB

In this subsection, we use a synthetic dataset to evaluate the batch process algorithms and measure system throughput under different-sized disk indexes. In the test, we used five different Chunkfarm configurations identified by the size of the disk index, namely 32, 64, 128, 256, and 512 GB disk indexes. Each configuration was scheduled to run in four different modes: Chunkfarm-1 GB, Chunkfarm-2 GB, and Chunkfarm-3 GB perform ILB and IUB, using 1 GB, 2 GB, and 3 GB memory index caches, respectively, while Chunkfarm-random performed random disk I/Os instead of ILB and IUB to identify new chunks and update index whenever an arena has been stored to the chunk repository.

Before each test, the client PC was scheduled to run to send data to the Chunkfarm backup server to generate a sufficient number of fingerprints to fill the index cache. We implemented the index cache as an in-memory hash table with each bucket storing an array of fingerprint entries. We stored a fingerprint entry using 32 bytes with 5 bytes for the arena ID, 20 bytes for the fingerprint, and the remaining 7 bytes for cache management. Using this type of index cache, 1 GB memory can store about 33 million fingerprints, 255 GB worth of 'unique' data chunks for an expected chunk size of 8 KB.

Figs. 4 and 5 show the time overheads and the efficiencies of ILB and IUB respectively. We found that the time overheads of ILB and IUB were related only to the disk index size and the disk transfer rate and were independent of the number of fingerprints processed (i.e., the size of the index cache being used). With the disk index size increasing from 32 to 512 GB, the ILB and IUB time increased from around 2.5 min and 6.2 min to around 39 min and 97.5 min, respectively. This is because the central processing unit (CPU) did not become a bottleneck. ILB and IUB are effective techniques to improve the efficiency of fingerprint lookup and update. As Fig. 5 shows, with a 32 GB disk index and a 3 GB in-memory index cache (ILB-3 GB, IUB-3 GB), the measured maximum speeds of ILB and IUB were about 627 000 and 267 000 fingerprints per second, respectively, a speedup factor of 1205 and 989, respectively, over the random on-disk fingerprint lookup and update, which achieved corresponding speeds of about 520 and 270 fingerprints per second, respectively. Even with a 512 GB disk index and 1 GB in-memory index cache (ILB-1 GB, IUB-1 GB), the ILB and IUB still achieved speeds of about 13 760 and 5651 fingerprints per second, over 26 and 20 times faster, respectively, than the random on-disk fingerprint lookup and update speeds. Obviously, using a larger index cache to store more fingerprints further improves the batch process efficiency until the CPU eventually becomes a bottleneck.

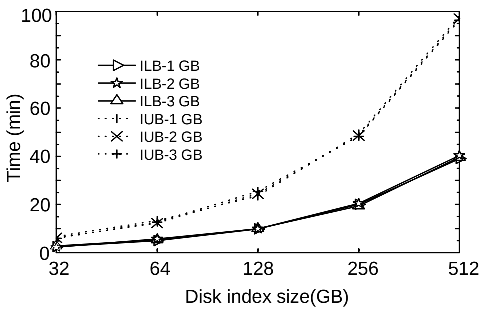

Fig. 4 Time overhead of ILB and IUB

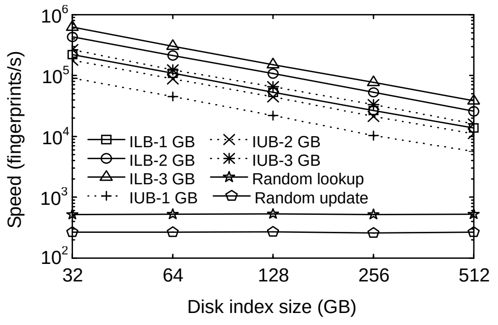

Fig. 5 The efficiency of disk index lookup and update

Fig. 6 shows the write throughput of the four Chunkfarm modes under different-sized disk indexes (32, 64, 128, 256, and 512 GB) which in turn correspond to different system capacities supported (5.2, 10.4, 20.8, 41.6, and 83.2 TB) (Section 2.1). Chunkfarm-random had very low throughput at about 5 MB/s near the Venti system (Quinlan and Dorward, 2002). In contrast to Chunkfarm-random, the other Chunkfarm modes all achieved high throughputs above 100 MB/s except for Chunkfarm-1 GB with a 512 GB index, which was about 75 MB/s. This performance achievement was mainly obtained from the ILB that effectively improved the fingerprint lookup efficiency. As can be seen from Fig. 6, Chunkfarm-1 GB achieved a write throughput of about 113 MB/s under a 256 GB-sized disk index, while Chunkfarm-2 GB and Chunkfarm-3 GB achieved write throughputs of about 112 MB/s and 134 MB/s, respectively, under a 512 GB-sized disk index. This means that a single-server Chunkfarm can support a physical capacity of more than 40 TB using a modest amount of physical memory such as 1 GB meanwhile maintaining a relatively high throughput. And by doubling the size of the in-memory index cache for ILB and IUB, the physical capacity that Chunkfarm can support will also be doubled while achieving roughly the same write throughput.

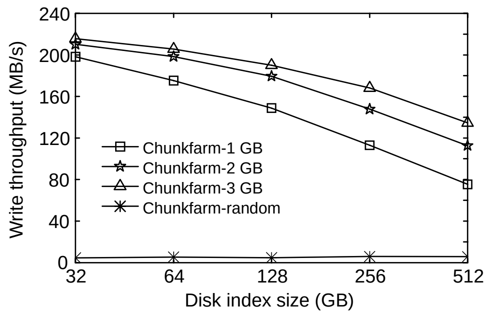

Fig. 6 Chunkfarm write throughput under different-sized disk indexes

In contrast to Chunkfarm, using the same amount of memory, DDFS can support only a relatively small physical capacity. Unlike Chunkfarm, which consumes memory space mainly for ILB and IUB, DDFS consumes memory space mainly for the Bloom filter, which represents the fingerprint set of the whole system in order to quickly determine whether an incoming fingerprint has been stored in the system with a low false positive probability. Supposing an *m*-bit Bloom filter, which represents the system fingerprint set with *n* fingerprints and *k* independent hash functions, the false positive probability of the Bloom filter is $\rho = (1-\mathrm{e}^{-kn/m})^k$. Given $k=(m/n)\ln 2$, the minimum false positive probability is equal to $(1/2)^k$ or $0.6185^{m/n}$ (Broder and Mitzenmacher, 2005). Then, using a 1 GB in-memory Bloom filter to support one billion fingerprints (for an expected chunk size of 8 KB, one billion fingerprints imply a physical backup capacity of about 8 TB) such that m/n=8, the minimum false positive probability will be about 2%. If a 1 GB in-memory Bloom filter is used to support a 16 TB physical capacity such that m/n=4, the minimum false positive probability will quickly increase to about 14.6%. This high false positive probability will inevitably result in a high percentage of small random disk I/Os for fingerprint lookups, thus significantly degrading the DDFS performance. Therefore, using 1 GB of memory space, DDFS can support a system backup capacity of no more than 8 TB.

4.3 Performance of multi-server deployment

In this subsection, we evaluate Chunkfarm scalability by running the system in variable multi-server deployments. Specifically, we ran the system under a total of 15 different modes denoted as (x, y), where x (x=2, 4, or 8) represents the number of backup servers used, y (y=32, 64, 128, 256, or 512) represents the size (in GB) of index part each backup server holds. The system first ran under (2, 32), then by enlarging each index part to 64 GB using the capacity enlarge algorithm (Section 3.3) the system was scaled up to (2, 64), and then by dividing each index part into two 32 GB parts the system was transferred to (4, 32). Then by merging the four 32 GB index parts to two 64 GB index parts and enlarging each of them to a 128 GB index part, the system was moved to (2, 128), etc., until the system ran under (8, 512) mode. The mode-to-mode transition was basically index operations (enlarging, dividing, or merging) that can be completed more easily with no impact on the previously stored data in the chunk repository. In the experiment, each backup server used 1 GB memory index cache for PIL and PIU. In addition, we used a total of eight nodes to construct a chunk repository to store arenas. The test data was generated by the backup servers themselves for simplicity, since the main purpose of this experiment was to measure the aggregate throughput of backup phase-2 not that of the backup phase-1.

Figs. 7 and 8 show the measured PIL speed and aggregate throughput of Chunkfarm backup phase-2 of the 15 run modes, respectively. The number of backup servers was varied along the x-axis, and the size of index part each server holds was displayed in the legend. The aggregate throughput of Chunkfarm backup phase-2 was calculated using the amount of time the PIL and parallel chunk storing consumed divided by the amount of data processed. As expected, both the PIL speed and the aggregate write throughput of Chunkfarm grew rapidly with the increment of the number of servers and decrement of the size of index part. Under (2, 512), it got the slowest lookup speed of about 27 000 fingerprints/s and aggregate write throughput of 146.2 MB/s, while under (8, 32) it achieved the highest lookup speed of 1 730 000 fingerprints/s and aggregate write throughput of 1505 MB/s. Although the larger disk-index part suffers from a relatively low write throughput, it supported larger system backup capacity. According to the expected throughput and physical capacity, the appropriate number of servers and size of index part can be selected. Selecting (8, 512) mode, for example, the system held a disk index 512×8 GB in size, and hence can support a physical capacity of over 83.2 TB×8 =665.6 TB and maintain an aggregate write throughput of about 508 MB/s.

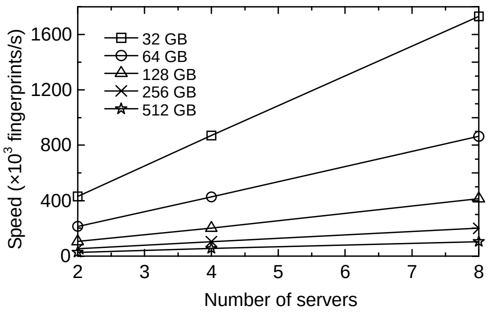

Fig. 7 Aggregate speed of parallel index lookup

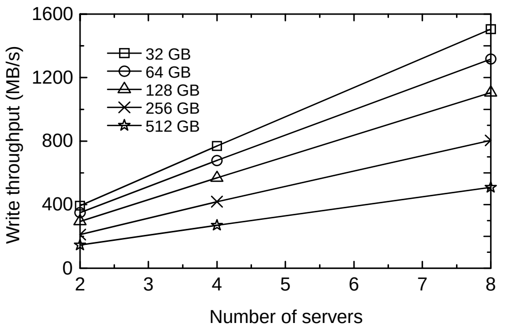

Fig. 8 Aggregate write throughput of multi-server Chunkfarm

4.4 Discussion

The main disadvantage of Chunkfarm, as a post-processing de-duplication system, is that it requires additional disk space to keep phase-1 data and increases non-backup-window processing time during phase-2. The additional disk space overhead α can be represented as $\alpha=B\times T/P$, where B is the data transfer rate, T is the backup window, and P is the system physical capacity. Assuming an uninterrupted 8-h backup window overnight with a sustained data transfer rate of 200 MB/s, a Chunkfarm backup server would need a 5.5 TB disk log to temporarily hold the data in phase-1. If the server holds a 512 GB disk index, which supports a physical capacity of over 83.2 TB, the overhead α is reasonably low, about 6.6% (5.5 TB/83.2 TB). However, for a weekend 24-h backup window with higher data transfer rate, 1 GB/s, and a lower physical capacity, for example 41.6 TB (using 256 GB disk index), the overhead α will be unacceptably high, over 200%. Fortunately, we can deploy multiple backup servers with some servers running in phase-1 and other servers running in phase-2, which can effectively reduce the overhead α while providing a 24-h backup window for users. In this case, each backup server needs only to maintain a relatively small disk log (e.g., 2–4 TB) for phase-1. If the disk log of a backup server is fully filled by backup jobs, the backup server turns to phase-2 to free up the entire disk log space while the backup jobs are redirected to other backup servers. In future work, we plan to use dedicated index servers (instead of locating the disk index to the backup servers) for fingerprint lookup and update, and the disk log of each backup server will be divided into appropriately sized (e.g., 0.5–1 TB) slots. When a backup server fully fills a slot, it initiates a lookup request for that slot to the index servers. During idle or light-loaded periods or when all the slots are full, the backup server writes full slots to the chunk repository using the lookup results received from the index servers. In this approach, an effective pipelined scheduling strategy will be employed to achieve high sustained throughputs with acceptable additional disk space overhead.

While on-demand data deletion is relatively simple for an ordinary storage system, it becomes difficult if there is duplicate elimination. In a de-duplication storage system, before actually deleting a physical chunk, the following two conditions must be met to ensure data consistency and integrity: (1) no old files pointing to it, and (2) no identical chunks being duplicate-eliminated against it at the moment. To meet condition (1), we maintain a reference counter, which is stored in a corresponding disk index entry, for each physical chunk that counts the number of files in the system pointing to this chunk. After storing/deleting a file to/from the system, the reference counters of all the ‘unique’ chunks in the file are incremented/decremented by one. Only physical chunks whose reference counters are zero are candidates for deletion. To meet condition (2), we perform the deletion after the de-duplication phase-2 has finished, and ensure no de-duplication phase-2 launched during the deletion. The deletion is done as follows: first, all the deletion requests are performed by deleting their files (deleting file index and metadata from the MDS) and decrementing corresponding chunk reference counters, and then physical chunks whose reference counters are zero are deleted from the arena. Then, free space fragmentation is cleaned by chunk migrating and arena rewriting. Finally, the disk index is updated to track chunk migrations. Needless to say, the deletion operation is expensive and it hence should not be done unless there are sufficient amounts of deletion requests accumulated. It should be noted that, for an inline de-duplication system to perform deletion while new data are being written to the system, condition (2) is difficult to meet since the system may eliminate duplicates using chunks which are truly being deleted in the process. To solve this problem, inline de-duplication systems such as Hydrastor (Dubnicki *et al.*, 2009) impose a read-only phase for deletion.

While de-duplication conserves space, it may reduce data reliability. Now that files share chunks due to de-duplication, the loss of a heavily-shared chunk may result in a disproportionately large data loss. This implies that chunks with larger reference counts are inherently more valuable than others. To protect the more valuable data, we should reintroduce redundancy beyond a simple redundant array of inexpensive disks (RAID), so that more valuable chunks are stored with a higher level of redundancy than less valuable chunks to achieve high reliability with minimal additional space overhead. The erasure resilient coding techniques such as the Reed Solomon codes (Blomer *et al.*, 1995) can be used to protect chunks with different levels of redundancy.

5 Conclusion

This paper introduces Chunkfarm, a post-processing de-duplication backup system, which performs de-duplication backup in-batch to deliver high write throughput and scalability. Compared with current mainstream solutions such as DDFS, Chunkfarm achieves stable write throughput which is not influenced by the duplicate locality of the input data. Using the same amount of memory, a single-server Chunkfarm can support a system capacity that is five times larger than that of DDFS while achieving a write throughput of over 100 MB/s. More importantly, by partitioning fingerprint lookup across multiple servers, Chunkfarm is shown to scale cost-effectively in both write throughput and physical backup capacity, achieving an aggregate throughput of 508 MB/s and supporting 665.6 TB physical backup capacity with eight backup servers.

References

Agrawal, N., Bolosky, W.J., Douceur, J.R., Lorch, J.R., 2007. A Five-Year Study of File-System Metadata. Proc. 5th USENIX Conf. on File and Storage Technologies, p.31-45. [doi:10.1145/1288783]

Bhagwat, D., Eshghi, K., Long, D.D.E., Lillibridge, M., 2009. Extreme Binning: Scalable, Parallel Deduplication for Chunk-Based File Backup. Proc. 17th IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. [doi:10.1109/MASCOT.2009.5366623]

Blomer, J., Kalfane, M., Karpinski, M., Karp, R., Luby, M., Zuckerman, D., 1995. A XOR-Based Erasure Resilient Coding Scheme. International Computer Science Institute Technical Report, No. TR-95-048.

Bloom, B., 1970. Space/Time trade-offs in hash coding with allowable errors. *Commun. ACM.*, **13**(7):422-426. [doi:10.1145/362686.362692]

Broder, A., Mitzenmacher, M., 2004. Network applications of Bloom filters: a survey. *Internet Math.*, **1**(4):485-509.

Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., Welnicki, M., 2009. Hydrastor: a Scalable Secondary Storage. Proc. 7th USENIX Conf. on File and Storage Technologies, p.197-210.

Eshghi, K., Lillibridge, M., Wilcock, L., Belrose, G., Hawkes, R., 2007. Jumbo Store: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service. Proc. 5th USENIX Conf. on File and Storage Technologies, p.22-37.

Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., Camble, P., 2009. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. Proc. 7th USENIX Conf. on File and Storage Technologies, p.111-123.

Muthitacharoen, A., Chen, B., Mazieres, D., 2001. A Low-Bandwidth Network File System. Proc. 18th ACM Symp. on Operating Systems Principles, p.174-187. [doi:10.1145/502034.502052]

Quinlan, S., Dorward, S., 2002. Venti: a New Approach to Archival Storage. Proc. USENIX Conf. on File and Storage Technologies, p.89-101.

Rhea, S., Cox, R., Pesterev, A., 2008. Fast, Inexpensive Content-Addressed Storage in Foundation. Proc. USENIX Annual Technical Conf., p.143-156.

Secure Hash Standard, 1995. Department of Commerce/ NIST, National Technical Information Service, Springfield, VA, USA.

Shapiro, L.D., 1986. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, **11**(3): 239-264. [doi:10.1145/6314.6315]

Tanenbaum, A.S., Herder, J.N., Bos, H., 2006. File size distribution on UNIX systems: then and now. *ACM SIGOPS Oper. Syst. Rev.*, **40**(1):100-104. [doi:10.1145/1113361.1113364]

You, L., Pollack, K., Long, D.D.E., 2005. Deep Store: an Archival Storage System Architecture. Proc. 21st Int. Conf. on Data Engineering, p.804-815. [doi:10.1109/ICDE.2005.47]

Zeng, L.F., Zhou, K., Shi, Z., Feng, D., Wang, F., Xie, C.S., Li, Z.T., Yu, Z.W., Gong, J.Y., Cao, Q., *et al*., 2006. HUSt: a Heterogeneous Unified Storage System for GIS Grid. Proc. ACM/IEEE Conf. on Supercomputing, p.325-338. [doi:10.1145/1188455.1188798]

Zhu, B.J., Li, H., Patterson, H., 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. Proc. 6th USENIX Conf. on File and Storage Technologies, p.269-282.