A Case for Packing and Indexing in Cloud File Systems

Saurabh Kadekodi[1], Bin Fan[2], Adit Madan[2], and Garth A. Gibson[1]

[1]Carnegie Mellon University
[2]Alluxio Inc.

CMU-PDL-17-105
October 2017

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

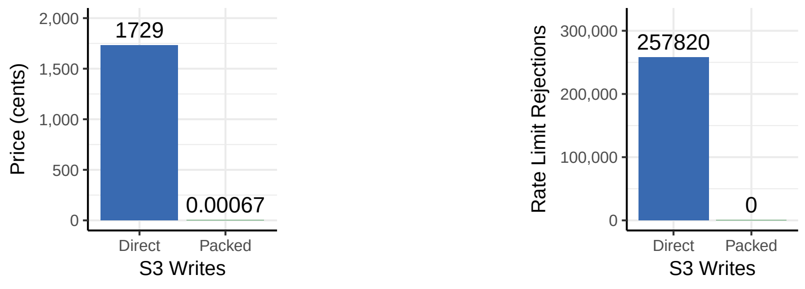
The amount of data written to a storage object, its write size, impacts many aspects of cost and performance. Ideal write sizes for cloud storage systems can be radically different from the write, or file, sizes of a particular application. For applications creating a large number of small files, creating one backing store object per small file can not only lead to prohibitively slow write performance, but can also be cost-ineffective because of the current cloud storage pricing model.

This paper proposes a packing, or bundling, layer close to the application, to transparently transform arbitrary user workloads to a write pattern more ideal for cloud storage. Implemented as a distributed write-only cache, packing coalesces small files (a few megabytes or smaller) to form gigabyte sized blobs for efficient batched transfers to cloud backing stores. Even larger benefits in price / cost can be obtained.

Our packing optimization, implemented in Alluxio (an open-source distributed file system), resulted in >25000x reduction in data ingest cost for a small file create workload and a >61x reduction in end-to-end experiment runtime.

Acknowledgements: This research was supported in part by the DOE and Los Alamos National Laboratory, under contract number DEAC52-06NA25396 subcontract 153593-1 (IRHPIT2), the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE, www.nmc-probe.org), and Seagate through the CMU Data Storage Systems Center. We also thank the member companies of the PDL Consortium - Actifio, American Power Conversion, EMC Corporation, Facebook, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NetApp Inc., Oracle Corporation, Samsung Information Systems America, Seagate Technology, Symantec Corporation and Western Digital.

Keywords: distributed file systems, cloud storage, packing, indexing



(a) (b)

Figure 1: An experiment storing 24.4 GB of data in S3 via 32 parallel clients. Figure 1a shows the overall price comparison with and without application side packing. The packed representation reduced the data ingest price by >25000x, explained in Section 8. Figure 1b shows the number of retries performed with and without application side packing; retries occur when S3 imposes rate limiting.

1 Introduction

Cloud based object stores, like Amazon S3, Google Cloud Storage (GCS) or Microsoft Azure Blob Storage [6], scalably store and access large amounts of data. Cloud storage systems, however, do not perform well with a large amount of small files owing to small transfer sizes. Moreover they incur high costs due to their operation based cost model. Larger object sizes can not only perform better but also reduce the total ingest cost by reducing the number of operations. We handle the costly and inefficient *small cloud writes* problem by in-situ packing of small files to form large blobs and building of a client-side index of packed file locations. Our best result shows a >61x improvement in throughput and end-to-end runtime and perhaps more importantly a surprising >25000x reduction in the price charged for ingesting data into Amazon S3. This result was achieved when we replaced individual 8KB PUT requests by one PUT request for every approximately 1GB packed blob.

Packing and indexing draws inspiration from batching writes and performing large sequential transfers; optimization techniques traditionally employed in disk arrays and still valid today with data centers using HDDs as the primary storage media [5]. Packing also reduces the total requests made against S3, consequently, reducing the price for writing data to the cloud backend stores. The cost reduction observed for data creation can be as much as a factor of the number of small files packed in a blob. As indicated in Figure 1b, packing also eliminates the possibility of hitting the S3 rate limit (see Section 2.4). This experiment is described in Section 8.

We developed our packing and indexing as a pluggable module extending Alluxio (previously Tachyon) [12] - an open-source distributed file system that is co-located with compute and can ensure data is finally persisted to potentially remote stores like S3, GCS etc. With this packing extension, Alluxio is able to serve big-data workloads (e.g., Spark, MapReduce) via its HDFS-compatible interface, while optimizing the performance and price with data buffering and coalescing before uploading to the backend cloud storage. Further details of implementation within Alluxio are specified in Section 6.

2 Motivation for Packing

The cloud storage pricing strategy is disproportionately skewed against issuing small write requests. According to the current Amazon S3 pricing scheme, the cost required to upload a particular amount of data can easily exceed the cost required to store the data for multiple months if the transfer size is not chosen well. For example, the cost of uploading 1GB data by issuing tiny (4KB) PUT requests is approximately

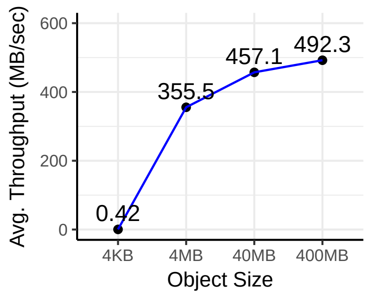


Figure 2: This graph reports average throughput (MB/s) seen by clients (workload generators) creating 12.5GB using object sizes ranging from 4KB through 400MB. Larger (MB-sized) objects saw a 300x improvement over smaller (KB-sized) objects.

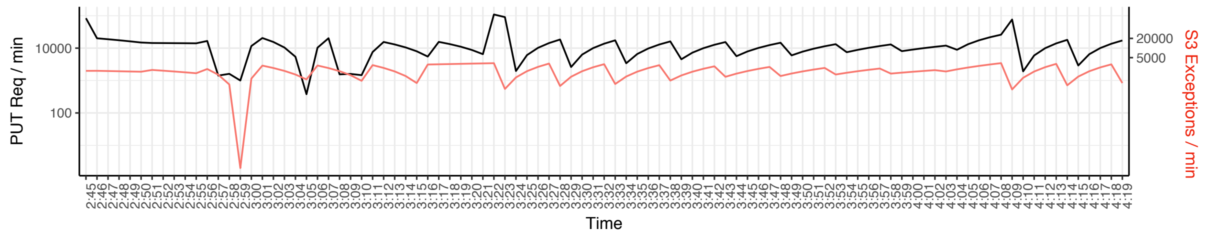


Figure 3: This graph shows a 95 minute window of the PUT request rate and S3's throttling exceptions per min for the application run creating 12.5GB using 4KB files. This data is captured using Amazon CloudWatch [8]. As the request rate exceeds a few hundred requests a minute, S3 fails some in-flight PUT requests by replying with a rate-limit exception code. This forces the application to retry failed requests after which it again issues new PUT requests with a rapid rate leading to more throttling exceptions. We see this behaviour repeat throughout the run.

57x as costly as storing 1GB data in S3 for a month.

2.1 Cloud Storage Pricing

Cloud services are typically pay-per-use and are categorized under the umbrella of infrastructure-as-a-service (IAAS). The pricing model for cloud storage involves three major components:

- Data stored per unit time
- Data moved (bytes read / written)
- Requests serviced

There is a skew in the price of writes versus reads, with writes being at least 10x higher than reads [2, 11]. Packing reduces the number of operations performed for data ingest by a few orders of magnitude.

2.2 Performance

We benchmarked the performance benefit from packing by devising a workload involving 32 clients writing simultaneously to Amazon S3 using 4 Alluxio worker nodes spawned in Amazon EC2 to measure the improvement in performance due to large object sizes. We measure the throughput and latency of writing 12.5GB of data in varying object sizes (4KB, 4MB, 40MB and 400MB). With 4KB, we are generating 100K objects per client and with 400MB we are generating 1 object per client. The larger objects are aimed at simulating small files packed into larger blobs. Figure 2 (MB/s measured from the application for varying object sizes) shows that MB-sized objects are created approximately 300x faster than KB-sized objects.

2.3 Metadata Efficiency

Cloud backends perform poorly when querying a desired object from millions of metadata entries [22]. In the HPC context, this problem has been handled by putting the burden of metadata transactions and lookups on the client [23] to prevent the backend services from hitting its so called *metadata wall* [1]. Client-driven packing and indexing of small files could prove to be an analogous solution to prevent metadata explosion in commodity clouds.

2.4 Rate Limiting

Cloud storage vendors rate limit the operations coming from a busy client. Amazon S3's best practice guideline states that if an application rapidly issues more than 300 PUT/LIST/DELETE requests per second, or more than 800 GET requests per second to a single bucket, S3 may limit the request rate of the application for an unknown amount of time [3]. The mechanism used to limit request rates is to fail requests with a specific "you are throttled" code exception.

Figure 3 reports the PUT request rate and S3's rate limit exception rate per minute for a 95 minute window of an experiment creating 12.5GB data using 4KB files. We see a strong correlation between increased PUT request rate and increased S3's request rejection rate due to throttling. On rejection, the application is forced to retry the PUT request leading to lower application throughput.

Packing can substitute hundreds if not thousands of requests with a single request, thus increasing the effective bandwidth of data ingest into cloud storage backends by a few orders of magnitude, because data rate throttling, if any, does not match request rate throttling. We verified this by writing 12.5GB data with MB-sized objects, completing in under one minute, compared to more than eight hours taken to write the same amount of data using 4KB objects. Intelligent packing choices have the potential to shift the rate-limiting performance bottleneck from cloud storage ingest to network traffic limits or even the application's data generation rate.

3 Related Work

BlueSky [21] is an enterprise level log-structured file system backed by cloud storage. Vrable et al. identify the quickly escalating price of pushing small increments to the cloud (in their work, log appends) and build larger transfer units of about 4MB each for efficiently uploading data to the cloud. Unlike BlueSky's focus on WAN access to cloud storage, Parallel NFS (pNFS) [19] exposes clients to local (i.e., on-premise) cloud storage. Storage servers can export a block interface, an object interface, or a file interface; pNFS clients transparently convert NFS requests to the appropriate lower-level access format. Blizzard [14] is a high-performance block store that exposes cloud storage to cloud-oblivious POSIX and Win32 applications. It is motivated to speedup random IO heavy workloads and also designed and implemented for single-machine applications.

Our self-defined blobs are inspired from the Data Domain Deduplication File Systems [24] fixed sized immutable self-describing containers packed with data segments and a segment index to identify packed data. Containerizing increased their throughput on their storage media - a RAID disk array. Venti [16], an archival file system constructs self-contained array of data blocks called arenas, analogous to blobs in our work. Li et al. [13] identify the traffic overuse problem in the cloud storage context by analyzing several cloud applications for their data update efficiency. The provide a middleware solution called update-batched delayed synchronization (UDS) to batch updates before passing them to clients that perform cloud

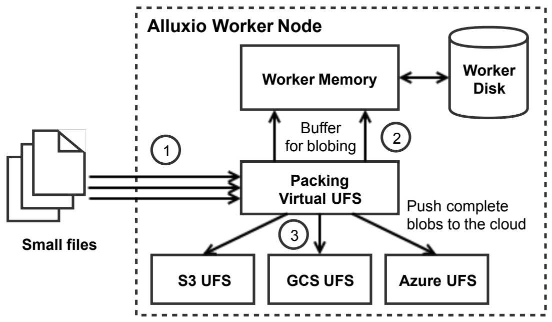


Figure 4: The Packing under file system module sits in an file system worker node. It receives small files as input from clients. It buffers them in the node-local memory and disk(s) to construct a sizable blob. It chooses from among buffered files, constructs a blob and pushes it to the cloud using one of the cloud under file systems.

synchronization. Although ideologically similar, their work focuses on minimizing frequency of updates through batching while we focus on extracting maximum throughput and minimizing cost in the data plane via packing.

Cumulus [20] comes closest to our work with their aggregation of small files to take cost aware file systems backups in the cloud. The difference is in the intended workload, deployment context and the design. Cumulus is designed for backups in thin-clouds environments that only support full-file operations. Moreover, Cumulus primarily deals with static datasets to be uploaded in the most efficient manner to the cloud. Our packing solution is in-situ and meant for active file system environments with streaming data and aggressive throughput / latency requirements

4 Design

4.1 Blob

A packed blob is a single immutable self-describing object that contains several small files and / or slices (contiguous byte-ranges) of large files. A packed blob has two parts - the blob body and the blob footer. The blob body contains concatenated byte ranges of files. We can customize the packing policy used to choose the files to pack in a particular blob to suite the read pattern. The blob footer contains a list of *blob extents*. A blob extent maps the logical byte-range of a file that was packed to the physical byte-range in the blob body. We call this footer the *embedded index* of the blob. Thus, we can fetch the complete data of all files using the embedded indexes of all packed blobs.

4.2 Blob Descriptor Table (Global Index)

While an embedded index is capable of mapping the files it contains, the process of bootstrapping with only embedded indexes could entail pre-reading the footers of all blobs, potentially a very expensive task. Instead we maintain a redundant global index that stores all mappings of all blobs. We call this the *blob descriptor table (BDT)*. BDT consistency, made harder because blob creation is decentralized, is discussed in Section 5.

4.3 Working

This subsection illustrates the interaction between the various packing components in a master-worker setup. Similar to the Google File System [10] and HDFS [4], let us assume a canonical distributed file system setup with one master and *k* worker nodes. The I/O path is between the clients and the workers while the distributed file system metadata is owned and managed by the master node. The worker nodes eventually push user data to a cloud storage system like S3. Thus, all blobs are constructed locally at worker nodes and subsequently pushed to the cloud. Each worker ends up packing and pushing multiple blobs, each containing an embedded index as shown in Figure 4. The master node maintains a BDT which we implement using Google's LevelDB [9] for bounded memory consumption.

Sufficient data has to accumulate to ensure sizeable blobs (GB-sized). This implies buffering of data yet to be packed. Thus, semantically, data waiting to be packed cannot be considered *durable* until it is uploaded to the cloud as a part of one or multiple blobs. Workers choose from buffered data using a particular packing policy (specified at mount time) to build a blob. Once data is copied to the blob file, an embedded index is constructed which is nothing but a list of blob extents that form the blob, and this index is appended to the blob as its footer. The name of the blob is carefully selected as a combination of the following attributes:

- **The worker identifier** - to indicate the ownership of the blob to the other workers. The identifier is usually the worker's IP address.
- **Footer offset** - the embedded index byte offset in the blob. This is an important fault tolerance requirement, see Section 5.
- **Timestamp** - the creation timestamp of the blob disambiguating it from other blobs created from the same worker having the same footer offset.

The blob identifier is the key used to store the blob in the cloud backend. The data is truly durable when the blob is successfully pushed to S3. After pushing a blob, the worker updates the master's BDT with the blob extents belonging to the pushed blob to assist future reads.

The BDT at the master node maintains the location of every file before and after it is packed. Thus, whenever a client writes a file to a worker, the worker synchronously updates the master BDT to claim ownership of the file. Once packed and pushed, the worker updates the previous location of the file with the blob extent of the file.

Reading data can mean different things depending on whether the data has been packed and pushed or is still waiting to be packed. Since each worker *owns* a file before it is packed, a read request issued to the owning worker for buffered data is fulfilled locally. If a read request is issued at a different worker, it first queries the master BDT for the current location of the file. For an unpacked file, the master BDT returns the identifier of the owning worker following which a worker-worker communication takes place exchanging the required data to fulfill the read. For a packed file, the master BDT returns the blob extent of the file using which the worker performs a range-read at the appropriate byte offset in an S3 object (packed blob) to get the file data.

Deletions and renames can be handled effectively using a file system journal, although, deletes can leave holes in packed blobs making a case for garbage collection at some point. We leave the deletions and renames to be handled at a later stage since our current focus was to expedite small file writes.

5 Fault Tolerance

Whatever is pushed to the cloud can be recovered is the invariant maintained in the packing design. Let us see the recovery techniques for different fault scenarios.

5.1 Master Dies

The master periodically backs up the BDT to the same cloud backend that is storing the blobs. The recovery strategy is to load the latest backup and iterate through the embedded indexes of the blobs committed after the last backup. This updates the master with all the blobs, and hence the location of all the packed files. Moreover, since the packing master does not hold any local state, it performs a lossless recovery. It is important to note that the frequency of master BDT backups only affects recovery performance and does not impact correctness. Thus, even though the embedded index adds an overhead in terms of space, it prevents flooding the master with synchronous packing updates from the workers and also plays a crucial role during recovery.

5.2 Worker Dies

In the case of a worker failure, the files being buffered for packing are lost and cannot be recovered. Note that partial recovery of buffered files may be possible if they were stored on local disks at the worker, and the worker restarts. We do not provide an algorithm for this case and conservatively assume all unpacked files are lost. In the case of files packed into blobs, if the necessary blob extents were updated in the master's BDT successfully, then there is no extra recovery process. Reloading the latest BDT backup should give us the locations of packed files. For blobs whose extents were not updated in the BDT requires reading their embedded indexes and inserting their blob extents in the BDT during recovery.

6 Implementation

We implemented packing based on Alluxio, which is a user-level distributed file system that has the ability to cache data and eventually store / retrieve data from a multitude of backends that include local disks, cloud backends, hybrid setups, etc. Alluxio follows the hierarchical master-worker architecture described in section 4.3.

6.1 Packing as an Alluxio *Under File System*

Alluxio supports accessing multiple and possibly different backing stores through its under file systems (UFS) abstraction. The packing layer is also implemented as a UFS module. As a result, the packing layer can serve all existing Alluxio applications (e.g. Spark, MapReduce and etc) without any code or configuration change in applications. Note that, this packing UFS only buffers data within itself and eventually pushes the packed blobs and BDT backups to a cloud under file system like S3.

The packing UFS is *mounted* to an Alluxio file system path, with various configuration such as the maximum blob size, data buffering timeout, number of packing threads, the packing policy and the underlying UFS that will store the data. Thus, all files written to the packing mount point in Alluxio are subject to packing via the specified packing policy.

6.2 Worker-local BDT

As described in section 4.3, every file creation results in two master BDT RPCs. Moreover, every read operation requires to consult the master BDT to locate the file being queried. To prevent overloading master in a large cluster, each worker also maintains a local BDT to store a redundant copy of the blob extents (of blobs packed by only that worker) which are also additionally stored in the master BDT. This prevents the master from getting involved in every packed read issued to workers owning the requested file. Since clients usually persist the connection with a worker, this optimization reduces a lot of traffic to the master node. Note that since the master has a copy of all the mappings, and is getting backed up regularly, the worker-local BDT is purely introduced for performance enhancement and does not need to be backed up.

6.3 Piggybacking Heartbeats

The states of blobs (e.g., packed and pushed) will eventually be synced with the master BDT for consistency. However, before the master learns about a pushed blob, the master is able to redirect read requests to the owning worker by querying its BDT. This allows the worker to update master asynchronously with the states of its blobs. We choose to piggyback the regular alive-heartbeat from the worker to the master with the update preventing additional RPCs and hence avoiding network congestion.

6.4 Multiple Packing Mounts

Since we developed packing as a UFS, it was imperative to support multiple concurrent packing mount points in a single Alluxio cluster installation. This implies one master BDT per packing mount point (since they have to get backed up into the same backing store to maintain the fault tolerance scheme described in Section 5). We can share a worker-local BDT among different packing mount points since they are meant solely for performance enhancement and pose no consistency problems.

7 Practical Tips for Packing

In our packing implementation exercise, we came across several optimization opportunities, the most important of which we highlight here to assist future packing implementations.

7.1 Choosing the BDT Data Structure

The BDT is essentially a map from client files to owning workers or blob extents. In a cloud-based setup with billions of objects, and with the BDT being on the critical path for reads and writes, choosing an appropriate data structure is crucial. Although an in-memory hash table is probably the fastest, the memory bloat due to the number of objects could overwhelm the system. In contrast, a completely disk-based mapping technique (for example, using symbolic links or empty files) would be space efficient but extremely slow as shown in [17]. Keeping these constraints in mind, we chose LevelDB [9], a key-value store that uses LSM trees [15] with bounded memory requirement and efficient disk transfers.

7.2 Buffering Small Files for Packing

Along with storing duplicate blob extents, we also use the worker-local BDTs to buffer small files yet to be packed. It is shown that small files hurt local file system performance [17]. Storing them in LSM trees makes them disk-friendly. We further optimize our BDTs by turning off expensive LevelDB compactions (resulting in very predictable insertion performance) since most of our reads are range-reads to S3 objects. This technique is inspired from similar file system optimizations done in the HPC world by Zheng et al. [23, 18].

8 Evaluation

This section presents ingest benchmarking result for packing in Alluxio with the following configuration: 1 master node, 4 worker nodes, a backend of one Amazon S3 bucket storing blobs. All nodes are Amazon EC2 instances with 128GB RAM and local storage via SSDs. There were 32 workload generators (8 on each worker), each generating 100K, 8KB files for a total workload size of approximately 24.4GB. We report the average of two runs performed with and without packing.

Configuration	*Data Ingest Price ($)*	*Runtime (s)*	*Throughput (MB/s)*	*Rate Limit Retries*
Direct	17.29 (16 for files, 1.29 for retries)	21778	1.16	257820
Packed	**0.00067** (↓ **25000x - no retries**)	**354.5** (↓ **61x**)	**71.46** (↑ **61x**)	**0**

Table 1: Comparing the average performance of two runs of an experiment storing 24.4GB of data as 3.2 million small files (8KB) a few hundred packed *blobs* each of upto 1GB in size.

Each worker has 16 dedicated packing threads and a 5 second timeout triggering packing of files. Since the timeout essentially implies a fault window, 5 seconds was an intentional choice mimicing the journal flush timeout (also a fault window) for mainstream local Linux file systems like Ext4 [7].

We compare this ingest workload applied to the packing version of Alluxio to its non-cached mode that writes files synchronously to S3. The per-client average throughput in Table 1 shows that without packing, the average throughput is a very poor 1.16 MB/sec. This is attributed to frequently hitting S3 rate limits causing thousands of retries to prevent forward progress. When packing is enabled, we see a >61x increase in throughput to about 71 MB/sec. This suggests S3 is not throttling bytes transferred as aggressively as it is throttling the request rate.

Table 1 also shows the price reduction as a result of packing. Our workload creates 3.2 million files. This requires the same number of PUT requests without packing (one for each file). Moreover, we see 257820 rate limit exceptions issued by S3. The rejected PUTs need to be retried, with each retry being another PUT request. In total, our experiment issued approximately 3457820 PUT requests at a cost of 5 cents for 10000 PUT requests, totaling $17.29. In contrast, after packing, we issued only about 104 PUT requests with zero retries and spent less than 0.1 cent.

9 Conclusion

Recently, scalable file system architectures are exploring client-funded metadata management for maximizing metadata performance [23]. The packing and indexing approach in commodity clouds adopts this trend by allowing indexes to be defined on the client side. Moreover, packing also aims at shifting the bottleneck from the expensive and costly operations per second to the network bandwidth by transferring data in large blobs, thus tuning the data path. We observe a dramatic decrease in price due to reducing number of requests and we also avoid getting throttled by cloud backends because of lower frequency of requests.

References

[1] Sadaf R Alam, Hussein N El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel i/o and the metadata wall. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 13–18. ACM, 2011.

[2] Amazon. Amazon s3 price structure. https://aws.amazon.com/s3/pricing/.

[3] Amazon. Amazon s3 request rate limiting. http://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html.

[4] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53, 2008.

[5] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore Tso. Disks for data centers. *White paper for FAST*, 1(1):p4, 2016.

[6] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[7] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

[8] Amazon CloudWatch. Amazon cloudwatch, 2014.

[9] Sanjay Ghemawat and Jeff Dean. Leveldb. *URL:* `https://github.com/google/leveldb`, `http://leveldb.org`, 2011.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[11] Google. Google cloud storage price structure. `https://cloud.google.com/storage/pricing`.

[12] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[13] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient batched synchronization in dropbox-like cloud storage services. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 307–327. Springer, 2013.

[14] James W Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *NSDI*, pages 257–273, 2014.

[15] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[16] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *FAST*, volume 2, pages 89–101, 2002.

[17] Kai Ren and Garth A Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *USENIX Annual Technical Conference*, pages 145–156, 2013.

[18] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 237–248. IEEE, 2014.

[19] Spencer Shepler, M Eisler, and D Noveck. Network file system (nfs) version 4 minor version 1 protocol. Technical report, 2010.

[20] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage (TOS)*, 5(4):14, 2009.

[21] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Bluesky: A cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 19–19. USENIX Association, 2012.

[22] Lin Xiao, Kai Ren, Qing Zheng, and Garth A Gibson. Shardfs vs. indexfs: replication vs. caching strategies for distributed metadata management in cloud storage systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 236–249. ACM, 2015.

[23] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. Deltafs: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 1–6. ACM, 2015.

[24] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 1–14, 2008.