A Case for Packing and Indexing in Cloud File Systems

Saurabh Kadekodi[1] Bin Fan[2] Adit Madan[2]
Garth A. Gibson[1,3] Gregory R. Ganger[1]
[1]*Carnegie Mellon University* [2]*Alluxio Inc.* [3]*Vector Institute*

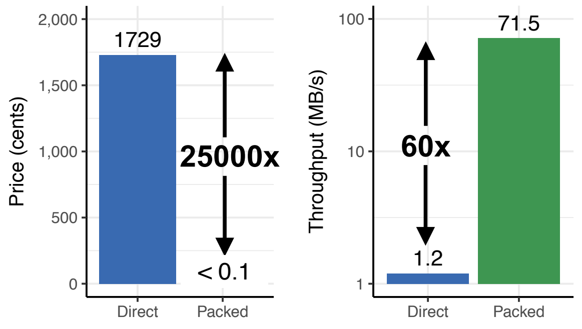
Abstract

Small (kilobyte-sized) objects are the bane of highly scalable cloud object stores. Larger (at least megabyte-sized) objects not only improve performance, but also result in orders of magnitude lower cost, due to the current operation-based pricing model of commodity cloud object stores. For example, in Amazon S3's current pricing scheme, uploading 1GiB data by issuing 4KiB PUT requests (at 0.0005¢ each) is approximately 57× more expensive than storing that same 1GiB for a month. To address this problem, we propose client-side packing of small immutable files into gigabyte-sized *blobs* with embedded indices to identify each file's location. Experiments with a packing implementation in Alluxio (an open-source distributed file system) illustrate the potential benefits, such as simultaneously increasing file creation throughput by up to 60× and decreasing cost to 1/25000 of the original.

1 Introduction and motivation

Cloud file systems provide file system style access to cloud storage. Since the most popular cloud storage model is based on so-called "objects" (arbitrary-sized sequences of bytes identified by an object ID), the straightforward approach is to map files to objects on a one-to-one basis. For example, directories can then be relatively simple mappings of file names to object IDs.

While straightforward to implement, this approach risks both performance and cost consequences. Study after study, along with our own experiences with data analytics storage, show that most files are small (KiB-sized) and that file creation occurs in bursts [1, 4, 5]. Yet, small object creation and write bursts are improper fits for cloud storage, for which both the performance behaviors and the cost models are heavily tilted in favor of using large access units and large objects. In particular, with direct file-to-object mapping, per-operation costs and operation rate throttling dominate other considerations during bursts of small file creation.



(a) Price comparison (b) Throughput comparison

Figure 1: Price and throughput comparison of client-side file packing with direct storage of each file as an object, for creating and writing 24.4GiB worth of 8KiB files to Amazon S3.

This paper illustrates the scale of this issue and the potential benefits from modifying cloud file systems (backed by commodity cloud object stores) to aggressively pack small immutable files into large objects. Using Amazon S3 as a concrete example, Figure 1 shows the benefits of packing 8KiB files into 1GiB objects rather than directly storing each 8KiB file as an individual object. In addition to achieving a >60× increase in throughput, aggressive packing reduces the price to 1/25000 of the original.

Packing files into large cloud objects draws inspiration from the batching of file writes into large sequential disk writes commonly applied in traditional storage systems. But, the benefits we observe are much greater in the cloud context, because cloud object storage systems like S3 are not intended to play the role of traditional file servers. Perhaps to avoid such usage, they heavily penalize workloads with large numbers of small operations—there is a significant per-operation charge, regardless of operation size, and users are explicitly rate limited in terms of $operations/s$.[1] In fact, the throttling

[1]Amazon S3's best practice guideline states that if an application

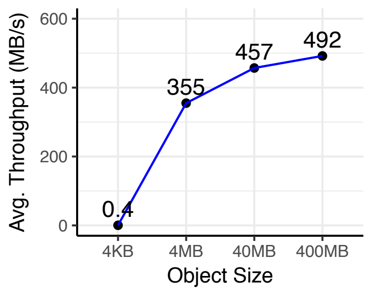


Figure 2: This graph reports average throughput (MiB/s) seen by clients creating 12.5GiB using object sizes ranging from 4KiB through 400MiB. Larger (MiB-sized) objects saw a $>300\times$ improvement over KiB-sized objects.

is so aggressive that PUTing one 4MiB object instead of 1K 4KiB objects can be almost $1000\times$ faster. Figure 2 shows the throughput of 32 clients writing 12.5GiB to an S3 bucket as a function of the object size. So, each client writes hundred thousand 4KiB objects, hundred 4MiB objects, ten 40MiB objects or one 400MiB object. Naturally, there are diminishing returns, but the performance and cost benefits of client-side packing suggests its aggressive use in cloud file storage.

Amazon's Elastic File System (EFS) and DynamoDB are natural alternatives to S3 since EFS is intended for regular file system and big data workloads, while DynamoDB is a NoSQL database meant for KiB-sized data. The EFS pricing model charges only for the space used and not for the data transferred or requests issued. But, the EFS space is charged at \$0.3 per GB-month. On the other hand, DynamoDB charges \$0.25 per GB-month with a write cost per month of \$0.47 for 1-write-per-second and \$0.09 for 1-read-per-second. In comparison, S3 charges \$0.023 per GB and 0.0005¢ per PUT. Suppose we have to store 1TiB worth of 4KiB files per month, this translates to slightly over 100 write requests/s. The monthly charges for S3 -without-packing, EFS, DynamoDB and S3-with-packing (henceforth referred to as just packing) are approximately \$1366, \$307, \$303 and \$24 respectively. Furthermore, the fraction of the operation costs (which in this example means writes or PUTs) in each are 98%, 0%, 16% and <0.001%. As a result, packing is at least $> 10\times$ cheaper than existing alternatives. Even though EFS has zero per-operation cost, packing shifts the dominant cost in accessing S3 objects to storage, instead of operations. Another way to quantify the benefit is to calculate the time it would take for the storage cost of 1GiB data to be equal to its access cost (break-even point). This comes out almost 5 years for S3-without-packing, >5 days for DynamoDB and just >27 seconds for packing! Thus, packing is lucrative for systems like Alluxio that employ client-side caching and try hard to minimize remote cloud operations.

This paper describes a plug-able client-side packing and indexing module designed for Alluxio (previously Tachyon) [8], an open-source distributed file system that is co-located with applications (e.g., Spark computations) and can ensure that memory-resident data is persisted in a multitude of (both remote and local) backing stores. Our optimization applies to Alluxio backed by commodity cloud object stores like Amazon S3, Google Cloud Storage, Microsoft Azure, etc[2]. Behind Alluxio's HDFS-compatable interface, our module packs data into GiB-sized blobs with an index of where each packed file is located in the blob. It also maintains a redundant client-side index, so that most reads can directly GET particular file data; the embedded indices are used to reconstruct this index if the client crashes rather than shutting down cleanly. With packing+indexing Alluxio is able to efficiently serve big-data workloads (e.g., Spark, MapReduce) while also optimizing the performance and price of its backend cloud object stores like S3. The remainder of this paper overviews our module's design, presents more evidence of the effectiveness of such packing, and discusses related work.

2 Design

Our packing and indexing solution resides as a module in the cloud file system which acts as a middleware between applications and the cloud object stores. The packing module uses a packing policy to pack application data into blobs and indexes it before pushing it to the cloud. § 2.1 describes the structures used within the packing module and § 2.2 explains the interaction of these structures for basic file system tasks and fault tolerance. § 2.3 discusses how these design choices help us meet our design goals.

2.1 Packing structures

Blob: A packed blob is a single immutable self-describing object that contains several small files. A packed blob has two parts: the blob body and the blob footer. The blob body contains concatenated byte ranges of packed 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

rapidly issues more than 300 PUT/LIST/DELETE req/s, or more than 800 GET req/s, to a single bucket, S3 may limit the request rate of the application for an unknown amount of time [2]. Figure 3 shows S3's rate limiting in action when creating millions of 4KiB objects.

[2]Amazon S3, Google Cloud Storage and Microsoft Azure have an identical operation pricing structure today i.e. 5¢ per 10K PUT operations and 4¢ per 100K GET operations. Moreover, this operation charge is independent of how much data is transferred.

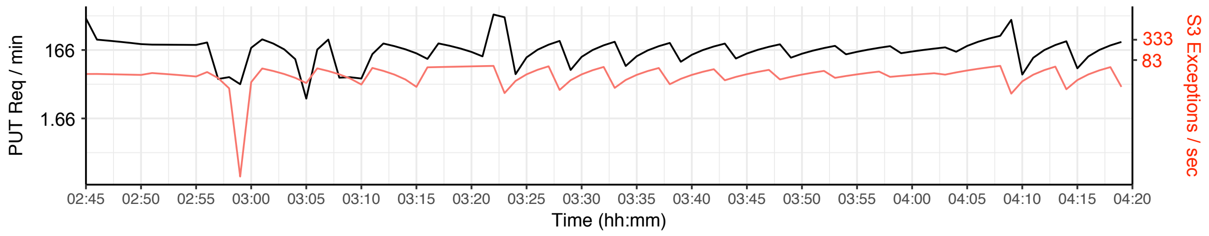


Figure 3: This graph shows a 95-minute window of the PUT request rate (black line; left Y-axis) and S3's throttling exceptions per min (red line; right Y-axis) for bulk creation of 3M 4KiB objects. As the request rate exceeds a few hundred requests a minute, S3 denies some in-flight PUT requests and replies with a "rate-limit" exception code. Such denied requests must be retried, often leading to more throttling exceptions and lower overall throughput.

using the embedded indices of all packed blobs.

Our blob is conceptually similar to sorted string tables (SSTables) [4, 6]. The reason we chose to not use SSTables today is because we wanted to avoid incurring the additional performance cost of sorting key-value pairs and the operation cost of reading the SSTable index before reading data from S3. Constructing raw data blobs allowed us to directly exploit the range-read feature of commodity clouds (see §2.3). Having said this, there is nothing fundamentally wrong with SSTables and it is possible to tweak SSTables for our purpose.

Blob Descriptor Table (BDT): While an embedded index is capable of mapping the files it contains, the process of bootstrapping with only embedded indices could entail pre-reading the footers of all blobs, potentially a very expensive task. Instead we maintain a redundant (as it can be losslessly reconstructed using the embedded indices of all the packed blobs) 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 2.2.

2.2 Operation

We implemented packing as an optimization to Alluxio, which has the canonical distributed file system architecture with one master and *k* worker nodes. Similar to GFS [7] or HDFS [3], the data transfer happens directly between clients and the workers while the distributed file system's metadata is solely managed by the master node. Thus, after file creation, the master delegates the responsibility of a file to a worker, and all blobs are constructed locally at each worker before being pushed to the cloud. A worker ends up packing and pushing multiple self-describing blobs. The master node maintains a BDT which we implemented using LevelDB [6] for bounded memory consumption.

Packing and pushing: Sufficient data has to accumulate to ensure sizeable (GiB-sized) blobs. Thus, buffered data waiting to be packed cannot be considered *durable* until it is uploaded to the cloud as a part of one or more 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 as explained in § 2.1. The name of the blob is carefully selected as a combination of the following attributes:

- **Worker IP:** to indicate the ownership of the blob to the other workers.
- **Footer byte offset:** the embedded index byte offset in the blob. This is an important fault tolerance requirement, see § 2.2.
- **Timestamp:** the creation timestamp of the blob disambiguating it from other blobs created from the same worker having the same footer offset.

The blob name used as the object ID for storing the blob in the cloud backend. After pushing a blob, the worker updates the global BDT with the blob extents belonging to the pushed blob. These extents are the packed data's latest address. The global BDT maintains the latest address of every file throughout the packing process. Thus, prior to packing, a file's address is the worker node on which it is being buffered, and after packing its address comprises of one or more blob extents.

Reads: Reading data can mean different things depending on whether the data has been packed and pushed or is still waiting to be packed. Since the latest address of each file before packing is the worker node it is being buffered on, a read request issued to the correct worker (i.e. the worker on which the file resides before being packed and pushed) is fulfilled locally. If a read request is issued at a different worker, it first queries the global BDT for the current address of the file[3]. For an unpacked file, the global BDT returns the identifier of the correct worker following which a worker-to-worker communication takes place exchanging the required data to fulfill the

[3]In case the worker's IP address (current worker identifier) changes dynamically, we treat it similar to a worker crash, which implies loss of unpacked data, since the master BDT no longer reflects the correct worker. Multiple solutions are possible to solve this problem. The simplest involves piggybacking the worker IP address with its keepalive heartbeat and updating the master BDT whenever there is a change. Another one could be to maintain a separate worker identifier controlled by Alluxio, which stays the same across IP changes.

read. For a packed file, the global BDT returns the blob extent(s) of the file. We then leverage the range-read feature of cloud storage systems to fetch the required bytes from the packed blob.

Deletes and Renames: Packing (essentially buffering) complicates the delete and rename semantics. As our current focus was to expedite small file writes, we sketch an outline for handling deletes and renames of packed objects, and leave its implementation as future work.

When the client issues a delete of a small file, it is unreachable once we remove its mapping from the global BDT. Since there is only one global BDT, and access to it is synchronized, we don't need to worry about inconsistencies or race conditions. The remaining task is to reclaim the space occupied by the deleted small file in the cloud, which we can do so, by tracking the *utilization* of a blob in the global BDT. The utilization is the amount of live (reachable) data that exists in a blob. Once the live data falls below a threshold, we can spawn a background job to fetch live data from packed blobs and repack the data in other blobs being built. The global BDT modification will atomically reflect the latest blob address, following which we can delete the original blob once it is empty.

Handling renames would involve an atomic change-of-name in the global BDT along with piggybacking renames as a part of future blobs' embedded indices. The latter process is required to maintain the invariant that the embedded indices of blobs (read in order of creation) are sufficient to reflect latest metadata of all packed files.

Fault tolerance: Our fault tolerance strategy follows the invariant: ***whatever is pushed to the cloud can be recovered***. In the case of an Alluxio based implementation, two possible failure scenarios can occur, either the master dies or one or more workers die.

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 indices 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 global 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.

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 global 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 indices and inserting their blob extents in the BDT during recovery.

2.3 Discussion: meeting design goals

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. This serves our design goals of ***transparency and modularity***. The packing UFS is *mounted* to an Alluxio file system path, with various configurations 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. A mount-point based access to packing allows it to ***co-exist*** with other packing solutions (on different mount points, with potentially different packing policies) and with non-packing Alluxio solutions.

Worker-local BDT: Every file creation results in two global BDT RPCs. Moreover, every read operation consults the global BDT to locate the requested file. To prevent overloading the 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 stored in the global BDT. This prevents the master from getting involved in every read request for a packed file issued by the client to a worker. Since clients usually persist the connection with a worker, this optimization reduces a lot of traffic to the master node. This optimization ***reduces communication overhead***. Optimizing worker-local BDTs could potentially relieve the global BDT even more, but we leave this exploration for future work. It also ***improves scalability*** because each worker is well-equipped to handle bidirectional I/O traffic from the clients. 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.

Exploiting range-reads: Most commodity cloud stor-

Configuration	*Data Ingest Price ($)*	*Runtime (s)*	*Throughput (MB/s)*	*Rate Limit Retries*
Direct	$17.3 ($16 for files, $1.3 for retries)	>21000	1.2	>250000
Packed	$<$ 0.1¢ ($\downarrow$ 25000×**, no retries**)	**355** ($\downarrow$ 60×)	**71.5** ($\uparrow$ 60×)	**None**

Table 1: Comparing the average performance of two runs of an experiment storing 24.4GiB of data as 3.2M 8KiB files into a few hundred packed *blobs* each of roughly 1GiB in size. The results are means across two runs, with stdev of packed runs and direct runs being $< 7\%$ and $< 19\%$ respectively of the mean for each number. The high fluctuation in direct runs is attributed to the heavy fluctuation in the number of retries caused by how aggressively S3 chooses to rate-limit, affecting both performance and cost.

age services provide range-reads. By downloading only the required bytes from within an object, we satisfy our design goal of ***avoiding large read latencies because of forcibly fetching entire blobs*** while expediting writes.

Packing open files or large files: Today we do not support packing open files or files larger than a blob. One solution approach is to "fake-close" all candidate files when constructing blobs. Thus, both open and large files might end up as different extents packed in multiple blobs. Despite being spread out, we can continue to efficiently read these files by issuing parallel range-reads.

Handling immutable files: Overwrites (also currently unsupported) are complicated to handle since they could partially invalidate a packed blob extent. Along with writing the new data as a new extent in a separate blob, we need to modify the global BDT along with piggybacking the information of the new extent breaks in the original blob extent. Similar to renames, this modification is necessary for ensuring correct global BDT reconstruction and / or recovery using the blobs' embedded indices.

3 Packing + indexing in action

This section presents an 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 8KiB files for a total workload size of approximately 24.4GiB. We report the average of two runs performed with and without packing.

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 mimicking the journal flush timeout (also a fault window) for mainstream local Linux file systems like Ext4 [10].

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 just 1.2MB/s. This is attributed to frequently hitting S3 rate limits causing thousands of retries which prevents forward progress. When packing is enabled, we see a $> 60\times$ increase in throughput to about 71.5MB/s. This suggests S3 is not throttling bytes transferred as aggressively as it is throttling the request rate. This improvement is not quite as dramatic as Figure 2 because that experiment only simulated packing by creating large objects, whereas here we are also accounting for the packing module's overhead.

Table 1 also shows the price reduction as a result of packing. Our workload creates 3.2M files. Without packing, we have to issue one PUT per file. Moreover, we see >250K rate limit exceptions issued by S3. The rejected PUTs need to be retried, with each retry issuing another PUT operation. In total, our experiment issued more than 3.45M PUTs (at a cost of 5¢ for 10K PUTs), totaling $17.3. In contrast, after packing, we issued only about 104 PUTs with zero retries and spent $<$ 0.1¢.

4 Related work

We don't have applications that include application-level packing at hand, but certainly they could be expected to achieve at least the same cost and performance benefits, and perhaps better by exploiting application-level knowledge. The paper makes a case for packing and indexing at the file system level, providing that benefit to unmodified applications that do not include such packing themselves.

Our self-defined blobs are inspired from the Data Domain Deduplication File System's [16] 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 [12], an archival file system also built self-contained array of data blocks called *arenas*, analogous to blobs in our work. Our simple fault-tolerant design is attributed to our self-describing blobs, which in turn are inspired from Data Domain containers and Venti arenas. Li et al. [9] identify the traffic overuse problem in the cloud storage context by analyzing several cloud applications for their data update efficiency. They provide a middleware solution called update-batched delayed synchronization (UDS) to batch updates before passing them to clients that perform cloud synchronization. Although our work is ideologically similar to UDS and Venti, we focus on expediting commodity cloud ingest rate while minimizing cost in the data plane via packing.

BlueSky [15] 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 4MiB each for efficiently uploading data to the cloud. We enlarge the transfer unit size (by creating GiB-sized blobs) to ensure good bandwidth utilization of today's multi-GB networks. Unlike BlueSky's focus on WAN access to cloud storage, Parallel NFS (pNFS) [13] 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 [11] is a high-performance block store that exposes cloud storage to cloud-oblivious POSIX and Win32 applications. It is designed and implemented for single-machine applications to speed up random IO heavy workloads.

Cumulus [14] 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.

5 Conclusion

Cloud file systems backed by cloud object stores should aggressively pack files into cloud objects, at the client, rather than simply mapping each file to a separate cloud object. Cloud object stores apply significant per-operation costs and operation/s rate limits, regardless of the amount of data read or written per operation. In the common case of mostly small files, packing can provide orders of magnitude improvements. For example, our packing and indexing module for the Alluxio distributed file system provides up to $60\times$ higher throughput and $1/25000$ of the original cost for bulk small file creation on Amazon S3. We expect similar behavior for other cloud object storage providers, making client-side packing$+$indexing an important component of cloud file systems.

6 Acknowledgements

We would like to thank our shepherd, John Wilkes, the anonymous reviewers and Alluxio Inc. employees for their feedback and guidance. We thank the members and companies of the PDL Consortium (Alibaba, Broadcom, Dell EMC, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, Intel, IBM, Micron, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Toshiba, Two Sigma, Veritas and Western Digital) for their interest, insights, feedback, and support.

References

[1] Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)* (2007).

[2] Amazon. Amazon S3 Request Rate Limiting. http://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html, 2018.

[3] Borthakur, D., et al. HDFS architecture guide. *Hadoop Apache Project* (2008).

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* (2008).

[5] Davidson, A., and Or, A. Optimizing shuffle performance in spark. Tech. rep., University of California, Berkeley - Department of Electrical Engineering and Computer Sciences, 2013.

[6] Ghemawat, S., and Dean, J. LevelDB. https://github.com/google/leveldb, http://leveldb.org, 2011.

[7] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[8] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *ACM Symposium on Cloud Computing* (2014).

[9] Li, Z., Wilson, C., Jiang, Z., Liu, Y., Zhao, B. Y., Jin, C., Zhang, Z.-L., and Dai, Y. Efficient batched synchronization in Dropbox-like cloud storage services. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (2013).

[10] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium* (2007).

[11] Mickens, J. W., Nightingale, E. B., Elson, J., Gehring, D., Fan, B., Kadav, A., Chidambaram, V., Khan, O., and Nareddy, K. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[12] Quinlan, S., and Dorward, S. Venti: A New Approach to Archival Storage. In *USENIX File and Storage Technologies (FAST)* (2002).

[13] Shepler, S., Eisler, M., and Noveck, D. Network file system (NFS) version 4 minor version 1 protocol. Tech. rep., NetApp, 2010.

[14] Vrable, M., Savage, S., and Voelker, G. M. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage (TOS)* (2009).

[15] Vrable, M., Savage, S., and Voelker, G. M. BlueSky: A cloud-backed file system for the enterprise. In *USENIX File and Storage Technologies (FAST)* (2012).

[16] Zhu, B., Li, K., and Patterson, R. H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *USENIX File and Storage Technologies (FAST)* (2008).