Multi-structured redundancy

Eno Thereska, Phil Gosset, Richard Harper

Microsoft Research, Cambridge, UK

Abstract

One-size-fits-all solutions have not worked well in storage systems. This is true in the enterprise where noSQL, Map-Reduce and column-stores have added value to traditional database workloads. This is also true outside the enterprise. A recent paper [7] illustrated that even the single-desktop store is a rich mixture of file systems, databases and key-value stores. Yet, in research one-size-fits-all solutions are always tempting and point-optimizations emerge, with the current *theme du jour* being key-value stores [8].

Workloads naturally change their requirements over time (e.g., from update-intensive to query-intensive). This paper proposes research around a *multi-structured* storage architecture. Such architecture is composed of many lightweight data structures such as BTrees, key-value stores, graph stores and chunk stores. The call for modular storage and systems is not dissimilar to the Exokernel [4] or Anvil [10] approaches. The key difference that this paper argues about is that we want these data structures to co-exist in the same system. The system should then automatically use the right one at the right workload phase. To enable this technically, we propose to leverage the existing *N*-way redundancy in the data center and have each of *N* replicas embody a different data structure.

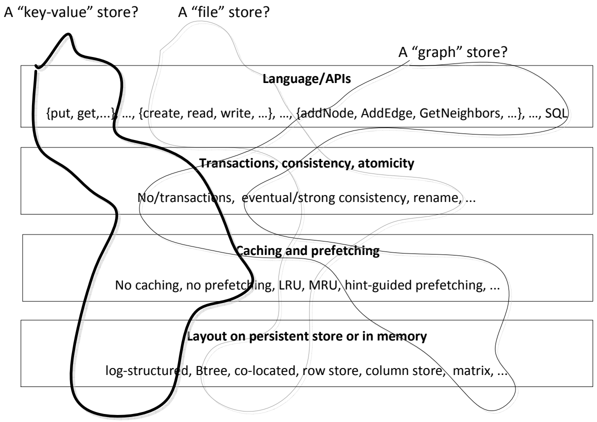

A "key-value" store?
A "file" store?
A "graph" store?
Language/APIs
{put, get,…}, …, {create, read, write, …}, …, {addNode, AddEdge, GetNeighbors, …}, …, SQL
Transactions, consistency, atomicity
No/transactions, eventual/strong consistency, rename, …
Caching and prefetching
No caching, no prefetching, LRU, MRU, hint-guided prefetching, …
Layout on persistent store or in memory
log-structured, Btree, co-located, row store, column store, matrix, …

Figure 1: Can you spot the key-value store?

1 Introduction and motivation

Personal and enterprise storage workload requirements usually vary over time. For example, a home user might first upload 100 photos to her hard drive or to Facebook, requiring good write throughput. Later in the day that same person might sort the photos into albums and tag her friends in them, requiring good read bandwidth and latency and ability to make associations quickly.

In the enterprise, a small business might receive lots of transactions during the day (requiring a fast insert rate and strong consistency) and it might perform analytics on the data at a later time (requiring good streaming bandwidth). The problem is that, while we recognize that workloads change, we over-engineer solutions that are good for only a part of a workload. For example, recent work on key-value stores emphasizes their ability to insert small data quickly with minimum resource requirements [8]. A typical workload, like Facebook's photo example above requires both inserts *and* complex queries.

Figure 1 illustrates another difficulty with evaluating point-solutions in the space. Designing a storage system requires making conscious decisions at many layers (a few are shown in the figure). It is through this figure that sometimes arbitrary point-solutions become apparent. For example, it is somewhat arbitrarily assumed that the workload using a key-value store will only need a simple "put/get" API, while a workload using a file store will need a "create/read/write" API, perhaps implying that one can "write" to arbitrary offsets. A workload using the graph store might need "noSQL" queries that directly manipulate the edges and nodes of a graph structure, or might use SQL.

Then, there are the transaction and the caching layers. It is implicitly assumed that the workload of a key-value store might not have good locality, and hence value-caching or prefetching is not needed, while file caching might be needed. For graph structures, prefetching might be better directed through edge hints. Furthermore, there might be strong locality in the way graphs are processed. And there is the layout layer, with data structures residing in memory or SSDs and disks. Key-value storage optimizations often assume that values will be small and thus a log-structured layout would be optimal in absorbing them. A file storage might have various other layouts since “files” are assumed larger in size than “values”.

What if these assumptions turn out to be incorrect over time? How much of your system design will need to change then? For example, Facebook states that their photo store is for scenarios where ”data is written once, read often, [and is] never modified” [2](page 1). Will that hinder the development of more complex Facebook applications for which those assumptions are not true (e.g., a Photoshop-like application for photo or video editing)? What if the ”value” in a key-value store is better off being stored in a filesystem if it is too large? And what if ”values” in the key-value store are related to one another, making a graph-store a better fit?

1.1 Motivation from database community

Is there “concrete and existing” evidence in industry that shows the problem is real? We list some evidence we have found here, mostly from the database community.

FILESTREAM addition to SQL Server:. Very recently, the SQL Server database added support for having database entries be files in the NTFS file system. This is called the FILESTREAM addition [11]. The hybrid database-like/filesystem-like end solution was appealing for database entries/values larger than 1 MB. It is important to note that the data will either be on the filesystem or on the database, and the structures do not co-exist (this is a single-node SQL Server), but that is still evidence that real users desire a more flexible approach to storing data.

Column-stores: The database community has learned how to have two data structures, a row-store and a column-store, co-exist by keeping one replica as a row-store and one replica as a column-store [1, 13]. Most database vendors nowadays offer both row- and column-stores. This choice affects the layout layer (in Figure 1) only, but it is still an important 2-dimensional choice that users are now offered.

2 Key idea: multi-structured redundancy

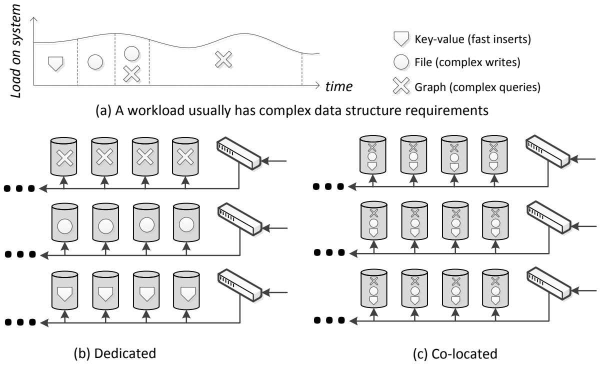
Our key technical idea is to use the existing data redundancy in data center storage systems (e.g., GFS [6]) and specialize each replica to use a different data structure. In a system with *N* replicas, *N* data structures could be supported. The data structures must be equivalent and store the same data, but they might do so in different ways. There is an analogy here with data structures in programming languages: a sorted list and a hash table could store the same elements, and yet one is optimized for range and succession queries, while the other for point-queries. To complete the analogy, it is important to note that we are advocating to keep two copies of the data, one on the list and one on the hash table, and not build a hash table with references to items on a list. Another parallel could be found with N-way programming [3], although the main argument for N-way programming is increased system reliability and not performance.

Figure 2(a) illustrates a workload that, over time, places different demands on the data center infrastructure. The figure illustrates the case when 3-way replication is used. One replica could use a combination of log structured layout, no caching and prefetching, and be used by a ”key-value-like” workload, while another replica could use a co-located layout, LRU caching and deep prefetching, to accommodate a ”file-like” workload while the third replica could use a fully in-memory layout, to accommodate a ”graph-like” workload.

Thus, each replica is specialized well to a different phase of the workload, while working well for a general workload that might not require specialization. For example, the first replica would quickly absorb small-writes in a workload, the second replica is closer to a traditional file system and responds well to reading and writing to large files in directories, while the third replica could do well for analytical workloads. Of course, all replicas need to eventually contain the same updates and an open question (see below) is whether they can stay in sync while still remaining specialized.

Figure 2(b) and (c) show two possible ways to spread the data structures on the available infrastructure. One way (dedicated) is to have dedicated servers and racks to one particular data structure (e.g., one rack could be an in-memory graph store). The other way (co-located) is to let the data structures share resources. The latter method could be better at balancing the load in the system, but the data structures could suffer from interference effects among each other, if there is no performance insulation among them [16].

Comparison with related work: Anvil [10], Stasis [14] and Boxwood [9] share our goal of modular storage and the researchers have already built several building blocks, such as B-trees and key-value stores. The unique observation we make is that workloads need these diverse data structures not in isolation from one-another, but simultaneously co-existing. Hence, we are investigating how to enable this co-existence by using the in-



(a) A workload usually has complex data structure requirements

(b) Dedicated

(c) Co-located

Figure 2: Various ways to assign the data structures on a traditional rack-based data center.

herent redundancy in the data center to provide up to *N* diverse data structures in an *N*-way replicated system. In the database community, SwissBox has called for blurring the lines between a database and OS and for deeper cross-layer optimizations [1]. We hope to identify similar optimizations for our layers (shown in Figure 1).

2.1 Open research questions

There are several open research questions from the above architecture. We have only began exploring them and this section discusses several of them.

Are *N* data structures sufficient?: We have gone from a single-point system specialization (e.g., have a fast key-value store) to an *N*-point specialization. It is likely a workload might require more specialized data structures than *N*. For example, a typical system in which $N = 3$ might be specialized in a key-value structure for fast inserts, a file structure for complex writes and a graph structure for complex queries. The workload might require another specialized structure, like a priority queue where some order is imposed on the data stored. The database community has developed data structures for two types of workload phases, one for transaction processing (a typical benchmark being TPC-C/E) and one for analytics (a typical benchmark being TPC-H). But, even these types of workload phases might require further specialization and other sub-structures like row-stores and column-stores [13]. One could simply increase *N* beyond the default redundancy, but that would require more storage capacity and lead to a decrease in update performance, since more replicas need to be kept consistent.

Could general performance suffer?: One assumption we have made is that replication in the original system is used primarily for reliability, and secondarily for performance. The reason is that data is usually both striped and replicated and read parallelism benefits primarily from striping. However, that might not be true for all workloads. It might be the case that the data center is provisioned for the peak performance of one particular phase of the workload (e.g., a query phase) and *all* servers are needed (using 1 data structure, e.g., the file one) to support that peak performance. In that case, no resources would be left over to allow for the other data structures. There is likely no general answer to this problem, but at a minimum the capacity provisioning tool will have to be more sophisticated. It would need to tradeoff provisioning for one particular phase vs. provisioning for multiple phases that require different data structures.

Speed-matching of updates among data structures: All the data structures need to be at least eventually consistent and sometimes strongly consistent. When an update arrives it must be propagated to all data structures. It is likely some of them are an order of magnitude faster than the others in absorbing updates. For example, the key-value store is optimized for quickly absorbing small writes, whereas a co-located file store will lag behind with such small writes if implemented naively. We want update performance to be roughly uniform among the data structures and query performance to be specialized. Can this be implemented in practice? Will we have to sacrifice strong consistency for it?

Can recovery be uniformly fast?: In addition to speed-matching of updates we want fast recovery. When a server fails, all the data structures residing there will need to be recovered from the replica structures. For example, if a server contains a graph-store with all the graph edges in memory and that server fails, the graph structure needs to be reconstructed from other replicas, e.g., from the key-value store replica and from the file replica. Those stores might not be optimized for quick reading of graph edges. On the other hand, if a server with key-value data structures fails, the in-memory graph store replicas might be very fast to recover it. Hence, we might have non-uniform recovery times. Careful layout of the data structures on resources is thus required.

Could one single (in-memory) structure be always the best?: For some workloads, like the ones discussed in RAMCloud [12], it is likely that one replica of the data could fit in memory and the other $N-1$ replicas are used only for recovery and not for servicing the foreground workload. For such setups, multi-structured replication might make no sense since one of the structures will always be the best for all workload phases.

Interestingly, for small-sized systems of a few servers and for very-large multi-petabyte systems an all-in-memory replica might be impractical for two different reasons. For a small system with tens of servers there aren't enough servers onto which to parallelize the write workload (one copy of which must still go to persistent storage) and as such the SSDs or disks would be a severe write bottleneck. For a very large system the cost of

Data structure	APIs
Key-value	*put(), get(), delete()*
File	*create(), read(), write(), delete()*

Graph	*addNode(), addEdge()*
	getNeighbors(), delete([node/edge])

Table 1: The API into CamFS. Internally these calls are mapped to appropriate caching, prefetching and data layout building blocks.

Petabytes of RAM would be prohibitively large. We take an approach in-between the all-in-memory or nothing-in-memory extremes, directed by the requirements of the data structures. For example, one implementation of our graph data structure in Section 3 has the graph edges fully in-memory, while the rest of the graph is on persistent storage.

Single-server vs. data center implementations: Will the multi-structured approach both scale-up and scale-out? History suggests that a single-server file system (e.g., ext4 or NTFS) is also used in a data center as a basic building block (with a distributed middleware layer on top). A single-server could still support multiple data structures at the lowest layer by partitioning the storage space into *N*. However, performance could suffer. The server would have to keep all *N* data structures consistent and would have to ensure some degree of performance insulation among them (e.g., by partitioning the cache and careful scheduling of requests).

Are workload phases easily and automatically identifiable?: It would be ideal if the system could automatically identify changes in workload phases (e.g., move from insert-heavy to query-heavy), however a starting point is to expose the multiple data structures to developers and let them decide when to use each. This compromise allows us to explore the systems research while allowing for automation (perhaps using machine learning approaches) to happen at a latter stage.

3 CamFS: a vehicle for exploration

To explore the tradeoffs involved, we have started building a storage system we call CamFS. Table 1 shows the current APIs implemented, corresponding to a key-value data structure, a file-based data structure allowing reads and writes at arbitrary file offsets, and a graph data structure. We are experimenting with a single-server (scale-up) version of CamFS where the data structures are on the same server, and a distributed version (scale-out) version where the data structures are distributed.

The building blocks of CamFS are a metadata service, a client library with the above APIs and an I/O coordinator that serializes requests and sends them to the appropriate servers (in the distributed case). A starting point for CamFS' implementation is the Sierra distributed storage system [15] that implements the file APIs only. That gave us a good starting point for having a basic primary-secondary concurrency based protocol, a lightweight metadata service, a recovery mechanism and a distributed journaling/logging service to build on.

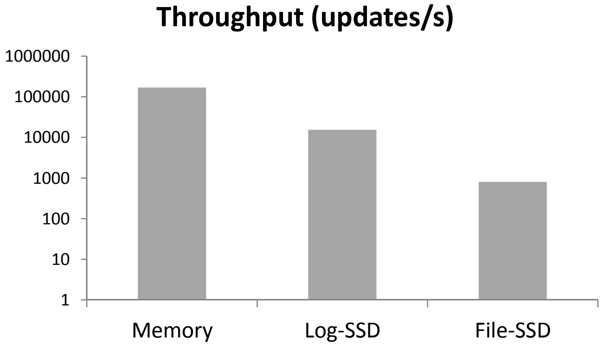


Figure 3: Small $1K$ update performance varies by two orders of magnitude depending on the layout chosen. The y-axis is in log scale.

At the bottom-most layout layer, Sierra exposed fixed-sizes chunks appropriate mostly for large file behavior. We have added a log-structured layout appropriate for absorbing small writes. In the middle layer we are planning to add caching and prefetching algorithms that implement a variety of mechanisms, including LRU, MRU and informed prefetching through hints. In this same layer we are planning to add transactional support not just within, but across the high-level data structures. This way, a developer could, for example, be inserting into the key-value store, adding an edge to the graph store and writing to a file all as part of one transaction.

There are multiple ways to implement the APIs, by choosing the appropriate building blocks. For example, the key-value structure currently utilizes the log-structured layout for small values, but can use the chunk-layout for large values. The graph structure can use the log-structured layout to store edges (the key would be the node and the value would be an adjacency list of all edges out of that node) or it can use an in-memory layout. So one replica of the edges could be in-memory, another could be on a persistent log-structured layout, while a third replica (not yet implemented) could be co-located with the node itself, perhaps by appending the edges to the node's content/file, much like the idea of embedding inodes with the directory entry in C-FFS [5].

Figure 3 shows three particular ways of implementing an update (this could be a put request, a small write, or an update to the graph edge). The updates could be kept in memory, on a log-structured system or in files (using

Setup	MiB/s in isolation	MiB/s combined
Disk(S_1, S_2)	(72,72)	(3.5,3.5)
SSD(S_1, S_2)	(103,103)	(49,49)
Disk(S_1, R_2)	(72,14)	(47,13)
SSD(S_1, R_2)	(103,51)	(84,41)

Table 2: The drop in performance when two workloads run concurrently is more significant for disks. SSDs virtualize better. *S* refers to a streaming, while *R* refers to a random access workload. The performance of each workload 1 and 2 is shown in a tuple (1, 2). For example, two streaming workloads got 72 MiB/s each when running in isolation, but around 3.5 MiB/s each when sharing the disk.

NTFS's data layout). Indeed, all three approaches will likely co-exist in the final system, one for each of the *N* replicas. This figure illustrates just one of the challenges mentioned in Section 2.1: speed matching across replicas with different performance characteristics.

An early decision in CamFS is to only consider SSDs for persistent storage and RAM (and any form of future non-volatile memory that might emerge). SSDs virtualize better than disks, as seen in Table 2[1]. This matters to us because without performance isolation we cannot research and evaluate having multiple data structures on the same server, as the co-located illustration in Figure 2.

4 Summary

Real workloads are complex and multi-phased (e.g., both update and query intensive). As a community we have developed several good point-solutions (e.g., key-value stores and file systems), and we continue to (over-)optimize them, but we do not have good multi-point solutions. This paper makes the case for more research in a robust co-existence of multiple data structures, such as key-value stores, file stores and graph stores. This paper presents a research plan that builds on existing *N*-way data center redundancy to provide up to *N* data structures and it presents a set of open research questions.

5 Acknowledgments

We thank our shepherd Andrew Warfield, the anonymous reviewers and Ant Rowstron, Dushyanth Narayanan and Timothy Zhu for their great feedback.

[1] We used a 500 GiB Western Digital WD5000AAKS-75V0A0 disk and a 256 GiB Kingston V100 SSD for this experiment. The streaming workloads used 64 KiB access sizes to read data. The random-access workload used a 4 KiB access size to write data.

References

[1] G. Alonso, D. Kossmann, and T. Roscoe. SwissBox: An architecture for data processing appliances. In *CIDR*, pages 32–37, 2011.

[2] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Vancouver, BC, Canada, 2010.

[3] L. Chen and A. Avizienis. *N-version programming : a fault-tolerance approach to reliability*, volume 1, pages 3–9. 1978.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 251–266, Copper Mountain, Colorado, United States, 1995. ACM.

[5] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 1997.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, Bolton Landing, NY, USA, 2003. ACM.

[7] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, Cascais, Portugal, 2011. ACM.

[8] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13. ACM, 2011.

[9] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, San Francisco, CA, 2004. USENIX Association.

[10] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with Anvil. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 147–160. ACM, 2009.

[11] Microsoft. Filestream overview. `http://technet.microsoft.com/en-us/library/bb933993.aspx`.

[12] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, Cascais, Portugal, 2011.

[13] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 430–441. VLDB Endowment, 2002.

[14] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 29–44, Seattle, Washington, 2006. USENIX Association.

[15] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of Eurosys'11*, pages 169–182, Salzburg, Austria, 2011. ACM.

[16] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *In Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, 2007.