

Position: Short Object Lifetimes Require a Delete-Optimized Storage System

Fred Douglass† John Palmer‡ Elizabeth S. Richards*
David Tao* William H. Tetzlaff‡ John M. Tracey† Jian Yin†

† IBM T.J. Watson Research Center

‡ IBM Almaden Research Center

*U.S. Department of Defense

Abstract

Early file systems were designed with the expectation that data would typically be read from disk many times before being deleted; on-disk structures were therefore optimized for reading. As main memory sizes increased, more read requests could be satisfied from data cached in memory, motivating file system designs that optimize write performance. Here, we describe how one might build a storage system that optimizes not only reading and writing, but creation and deletion as well. Efficiency is achieved, in part, by automating deletion based on relative retention values rather than requiring data be deleted explicitly by an application. This approach is well suited to an emerging class of applications that process data at consistently high rates of ingest. This paper explores trade-offs in clustering data by retention value and age and examines the effects of allowing the retention values to change under application control.

1 Introduction

We are researching the storage system for a highly scalable distributed stream processing system, similar to TelegraphCQ [2] or Medusa [16]. Unlike conventional systems that are typically engineered to have sufficient capacity, this system must be designed assuming its capacity is chronically insufficient. This assumption is appropriate for certain data mining applications in which the product of the available data and the set of potential mining algorithms dwarf any conceivable set of processing resources. In such an environment, the system, or at least its bottleneck resource, is always fully utilized. Disks are typically nearly full and they service an unrelenting stream of requests.

Individual data objects¹ can be of arbitrary size; many will be just a few bytes. While some data will be discarded immediately and never make it to secondary storage, a substantial amount of data will be written to disk, read once or a small number of times, and then quickly be deleted. Depending on system load and priorities,

some data may be deleted before ever being read. A relatively small fraction of the input data will be retained for a long time and read repeatedly. In this environment we observe that as file lifetimes become short, and all other things are equal, Little's Law requires that a fixed-sized storage system will have increasing create/delete rates. Since creates/deletes involve random disk I/O, and disk technology is progressing faster in density than access rate, this will become increasingly important in the future.

Three key notions in the design of our new storage system are *immutability*, *relative valuation*, and *pipelining*. First, data objects are immutable once created.² Thus the only operations on objects that involve their data are to write them initially, read them, or delete them. Second, there are additional operations to affect the metadata of an object, particularly its *retention value* (RV). When an object is created, it is given a current retention value (CRV) that indicates the relative importance of keeping the object, and a function defining how the CRV decays over time; objects therefore naturally age out of the system. Third, applications are designed to take objects along a pipeline, often in an arbitrary order. Rather than an application requesting a specific object and suffering the latency of retrieving that object, most applications will be designed to receive a stream of objects, the order of which is dictated by a resource manager. For example, a web crawler that processes retrieved pages may not care which pages it processes first, only that it processes all recently crawled pages in some order.

RVs are only *hints* to the system about how long to retain an object, not *absolute* guarantees. Thus, unlike traditional file systems that write a file and then ensure the availability of that file until it is deleted or overwritten, our system writes an object and then makes a good-faith effort to retain it in accordance with its specified RV. As objects are processed, their processing can affect the RV of various objects (themselves or others), causing them to be retained for longer or shorter periods. However,

the system is designed with the expectation that updates to existing RVs are relatively uncommon. In the steady state, perhaps 10-20% of objects will change their RV before deletion, but if 80% were to change, making assumptions about the ultimate RV would be unfruitful.

The implications of these load characteristics are substantial. The large number of small objects requires some form of aggregation to amortize I/O overheads. Clustering objects into collections of data, all written contiguously, makes sense from the standpoint of write performance. However, units such as the *segments* used in log-structured file systems (LFS) [11] can suffer from high overheads from garbage collection when the overall storage utilization is moderately high (typically above 80%). If there are no segments without any “live” data, the system must garbage-collect to coalesce live data into fewer segments and create entirely empty segments to be reused. In contrast, deleting an entire empty segment at once, without the need to copy “live” data to a new segment, can improve performance dramatically.

The key to such performance gains is the ability for *applications* to predict at object creation time which objects are likely to be deleted together. By clustering objects into different groups that depend on their anticipated lifetime, the system can create segments that can be reclaimed in their entirety at an appropriate time, without the need for cleaning. We refer to collections of objects that are stored contiguously and deleted as a group as *storage containers*. We refer to this sort of storage system as a *Delete-Optimized Storage System* (DOSS) because it makes deletions extremely fast.

Unsurprisingly, the solution is not nearly this straightforward. As RVs can change, data in a single container may have radically different RVs. A naive approach to this problem would revert to LFS segment cleaning, copying small amounts of live data to enable the reclamation of larger amounts of unused space. Even with optimizations such as hole-plugging [15], these additional disk I/Os have the potential to significantly curtail performance. In DOSS, the system must dynamically choose between violating desired RVs (deleting an object before or after its desired expiration date), proactively migrating data with inconsistent RVs to a different container, or rescuing live data just prior to deletion (as a log-structured system’s segment cleaner does).

2 Retention Policies

Most file systems store a given file until it is explicitly deleted. Some systems that run at or near their storage capacity may migrate inactive files to tertiary store based on their access characteristics [4] or even allow users and applications to specify precise expiration dates for files. For example, IBM’s z/OS has supported file expiration dates for nearly 40 years.

In fact, it is possible to have a range of retention policies. In non-decreasing order of system complexity, one can imagine any of the following:

Infinite retention. All files are kept forever; there is no thought of, or overhead associated with, deletion.

Unlimited retention. All files are kept until explicitly deleted. Deleting an object recycles its resources. Garbage collection of deleted resources can happen any time after deletion, preferably at a time when any overhead can be amortized (for example, segment cleaning in an LFS).

Fixed and immutable expiration. Files are kept until they expire. Allowing deletion prior to expiration is optional; if allowed, then the ability to predict expiration puts an upper bound on resource usage, since a file that is deleted before its expiration can be kept until the expiration occurs. Grouping files with the same expiration timestamp can result in optimizations in an LFS, as it entirely avoids segment cleaning.

Fixed but mutable expiration. Files are kept until they are deleted or expire, whichever comes first, but the expiration date can be modified until the actual deletion occurs. Grouping co-expiring files can help, but the benefits of such grouping are reduced to the extent that expiration timestamps vary over time, as a single storage container may not contain objects with a single expiration timestamp. We return to this issue below.

Approximate expiration. Files are given an approximate expiration time, and a window around that time, which can be modified. Files are deleted at any time within the time window. This approach is similar to *fixed but mutable expiration* but reduces the effect of differences between timestamps of files in the same container.

Fixed-priority expiration. Files are given a static “priority” and are deleted in order of increasing priority to reclaim needed storage capacity. Priorities may be weighted by other factors, such as size or access history, allowing the system to remove one object with slightly higher priority to retain many objects with lower priority.

Variable-priority expiration. Like with the preceding technique, files are given a priority, and relative priorities determine which files are deleted. These priorities can vary over time, for example via an exponential decay, such as the half-life of nuclear decay; even without a simple varying priority such as exponential decay, one can think of the half-life of a priority function as the time by which one expects half the data to be deleted. Priorities may be modified during the lifetime of an object.

Arbitrary priority functions. The most complex ap-

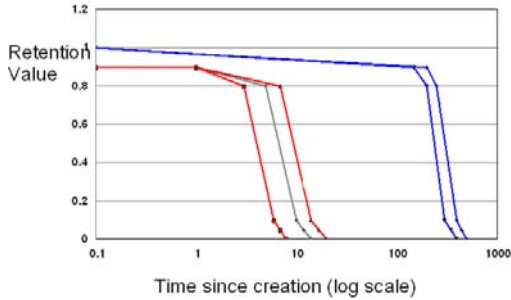


Figure 1: CRV decay curves.

proach is to allow the priority of a file be a function of other system state. For example, the priority of retaining a `.o` object file might be infinite if the corresponding `.c` source file is unavailable, but low if the object file can easily be recreated from the source file on demand.

Our system is starting with variable-priority expiration, which we refer to as *retention values*, with an eye toward arbitrary priority functions. As objects are created, they are annotated with an initial value between 0 and 1, with 1 referring to objects that should be considered permanent. They are also annotated with a decay function that specifies the *anticipated retention decay* (ARD) of the object’s data; the ARD of an exponential decay function would be its half-life. Figure 1 gives an example of decay curves. Also, just as with nuclear decay, a decay function in the context of our system is never a guarantee—just a statistical formulation. Under appropriate circumstances, an object might be deleted long before its nominal RV would suggest, or survive well past the expected point, but with a constant rate of object creation and expiration it should predict the deletion point of most objects within a small window.

CRVs and ARDs (collectively RVs) may be changed at any time. An object whose RV increases should be expected to survive longer, corresponding to another object whose RV places it in the same timeframe for deletion.

The pressure on the system to store its objects varies somewhat over time. When the rate of writes surpasses the rate of deletions, the total storage utilization increases. Over short times discrepancies between the two are to be expected, but eventually they must be in sync. This is accomplished by having a *high water mark* that defines a value between 0 and 1 specifying the current retention level: objects below that value are reclaimed. As available space falls below some minimum, the water mark increases; as it rises above some maximum, the water mark is reduced again.

3 Storage Containers

Our design takes advantage of the combination of high data rates, rapid object deletion, and predictable relative

RVs. Any given combination of initial CRV and ARD is extremely likely to have a steady stream of new data coming into the system. In such cases, they are accumulated in a *storage container* that holds only objects with that RV, and which were created within an ϵ time interval of each other. When the storage container is full, or after an appropriate delay, it is written to disk in a single high-bandwidth operation.

Grouping objects by RV, and writing large containers contiguously, makes writes efficient. There remain two open questions, each of which we address in turn: first, what is the effect of this design on *read* performance; and second, what is the effect of changes in RV?

3.1 Handling Reads

The impetus for the early log-structured file system work was the observation that with large memories absorbing many reads, disk I/O would soon be dominated by writes [9]. However, the development of segmented log-structured systems demonstrated benefits for reads as well as writes, because files that were written together would tend to remain close on disk and read in a single large I/O at a later point [11].

The workload for DOSS is expected to be extremely disk-intensive, with memory caches only absorbing a small fraction of all I/Os. However, its applications will be optimized to accept data often in arbitrary order. There are two ways in which this will be supported. First, applications will often be designed to have data pushed to them. Rather than deciding what data to read, they will allow an external optimizer to read the objects that are the “best” available (due to a combination of factors that include their expected time to live, the performance of reading particular objects, and inter-object dependencies). Even applications that decide on specific objects to read can improve performance substantially by specifying a long list of objects and then accepting them in an order determined by the underlying system [10]. Second, the system will always have more work to do than available resources, so its scheduler can run those applications that have their data immediately available. With rare exceptions for high-priority analysis, should an application need a specific object read from disk, the added latency for that application is unimportant as long as the system as a whole makes constant progress.

3.2 Handling Changing RVs

Thus far, we have described storage containers as being written once, with a number of objects having common RV (both their initial CRV and ARD), and then being deleted when their value falls low enough. What happens when RVs change?

Recall that all RVs are relative, non-absolute, and

application-specified. Any object can be deleted at any time if its CRV is below 1. (If applications choose to set so much data at the absolute CRV of 1 that the system runs out of space, an exception³ is triggered.) An application that wishes to increase the relative value of an object can modify it to have a higher RV, and the system should endeavor to keep the object an appropriately longer interval. There are three approaches to consider, diagrammed in Figure 2.

One approach is to eagerly move any object that has its RV change, inserting it into a new storage container with an appropriate overall RV. A consideration here is that occasional changes to RVs may not have the same steady-state behavior as a constant stream of external inputs, leading to a storage container being written when it is largely empty or, conversely, being kept in memory while the system attempts to fill it.

A variant of this approach is to write the changed object into an existing container. This can be done if an appropriate container has space, either because other objects have been deleted or moved, or because some space has been reserved in the first place. Filling holes in existing containers would be similar to hole-plugging in HP Autoraid [15], and would suffer some performance issues due to decreased amortization of the overhead of each I/O. This variant is shown in Figure 2(b).

Another approach is to affect the entire container in which the object resides. A simple policy is to delete a container when the highest-value object within it is below the high water mark, but this risks dramatically increasing overall storage usage, or increasing the pressure on the recycler such that it deletes containers without such outliers much earlier than the applications desire. Small changes in policy can be handled in this fashion, however. This variant is shown in Figure 2(c), with the middle container getting a lower RV.

Still another approach is to ignore a change entirely, or to note it to await a large enough aggregate change, as with the middle container in Figure 2(b). Since all RVs are merely hints, it is acceptable to delete something “prematurely” if keeping it longer would present a hardship to the system as a whole. Thus a single object with CRV of 0.7 and ARD half-life of a day might remain in a container with CRV of 0.6 and ARD of 12 hours, but changing a second object to 0.7 might trigger copying the two objects or adjusting the entire container.

4 Related Work

Our work is partly inspired by log-structured file systems (LFS) [11]. In LFS, the high cost of segment cleaning can be a major performance drawback [13]. Our work can be thought of as an optimization to LFS for systems in which applications can supply hints on object deletion times, or the system can infer object lifetimes with

some precision. We take advantage of such information to group objects with similar deletion times to reduce the cost of segment cleaning (ideally to nothing).

There have also been other attempts to reduce segment cleaning cost for traditional file workloads. Blackwell et al. [1] proposed to clean segments during projected disk idle times to reduce the performance impact of cleaning. Menon and Stockmeyer [8] proposed to clean only aged segments to reduce the number of live objects that must be moved during the process, and Wilkes et al. [15] proposed filling holes in live segments to avoid copying entire segments. While these systems addressed ways to minimize the impact of LFS segment cleaning, DOSS attempts to avoid traditional cleaning entirely.

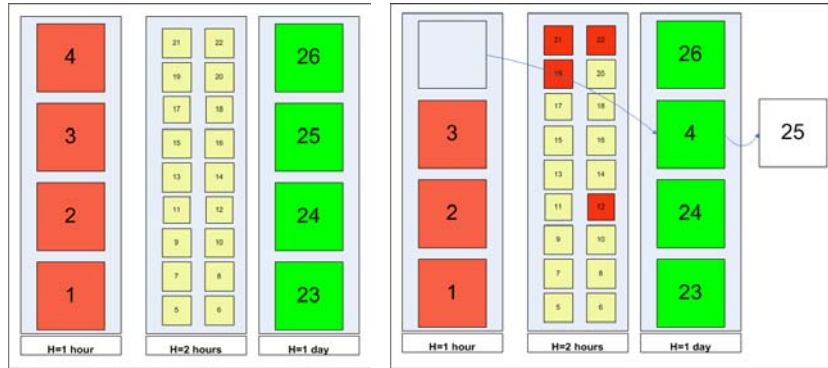
More recently, Wang and Hu [14] described a system that segregates active and inactive data into different segments, so that long-lived data tend to die slowly while short-lived data can be cleaned at minimal cost. More data in a cleaned segment would be dead, resulting in less write overhead than the original LFS approach, but cleaning would still be necessary.

Log-structured systems are not the only ones to group objects and support contiguous writes. Traditional UNIX file systems such as FFS [7] have attempted to store files within a directory near each other, to optimize read performance. Microsoft Windows NTFS systems run defragmentation software to coalesce file blocks to make writes more efficient (through large contiguous free space), reads more efficient (by storing files contiguously), and future defragmentation more efficient (by storing long-unmodified files together where the defragmentation process can ignore them) [5]. By comparison to defragmentation, our system is intended to use hints provided by applications to place most data in a place where it can later be deleted efficiently, rather than performing frequent full-disk scavenging to coalesce files and free blocks.

For compliant applications, our system will reduce read costs by scheduling many reads together, reorganizing accesses to push data to applications. Being able to read many objects can reduce disk head movement and thus increase effective disk bandwidth. Iyer and Druschel [6] proposed to deliberately delay the service of disk requests to allow more disk requests to be scheduled together; their anticipatory disk scheduling is applicable to our system.

Our system exploits application-supplied object deletion times. Other kind of application-supplied information such as read patterns have been exploited to optimize buffer management in file systems [10], and can be incorporated.

Gärtner, et al. [3] examine how to do bulk deletion in a relational database. While their problem is similar to



(a) Fresh containers.

(b) Containers after some adjustments.



(c) Containers after additional adjustments.

Figure 2: Part (a) shows three containers after being filled. They have ARDs of 1 hour, 2 hours, and 1 day, respectively. Each object is numbered for reference. In (b) objects 12, 19, 21, and 22 are changed to expire earlier, but they are left in the container with the higher ARD; Object 25 is deleted explicitly and later replaced by object 4, which has been given a higher RV than before. (c) demonstrates recategorizing an entire container, with nearly all the objects shifting from 2 hours to 1 hour—presumably a rare occurrence.

ours in some respects, they have a different focus: how to update database indices efficiently in the face of large numbers of concurrent deletions. Rather than updating the indices as each item is deleted, they delete numerous data items and then update (or recreate) the indices in a single operation. Those techniques may apply to updates to metadata in our system, deferring them until they can be done in bulk.

Finally, the Elephant file system [12] allows user-defined policies to control when files are permanently deleted. Users might want to retain only the most recent of each `.o` file but keep older “landmark” versions of `.c` files. Elephant assumes that space is relatively cheap, and focuses on recovering from accidental errors such as overwrites and deletions, while DOSS lets user-defined policies prioritize deletions and proactively place objects on disk according to their expected retention.

5 Conclusions

Early file systems assumed that data would be read from disk many times before being deleted and optimized the disk layout for reading. With the advent of large cache memories, systems such as LFS [11] were created to optimize for writes, assuming that reads would be handled in memory. Such systems have other overheads relating to garbage collection, necessary to provide large sequences of contiguous free space [13].

The new breed of distributed stream processing systems [2, 16] places unusual demands on a storage system. We are researching a new storage system that needs to not only make disk reads and writes efficient, but also handle substantial rates of data creation and deletion. Furthermore, unlike other systems, data items are retained based on *relative values* rather than until they are explicitly deleted. In this paper, we have discussed the tradeoffs in clustering data based on their retention val-

ues and the effects of permitting changes to these values prior to deletion. Our research is in the early stages, but we believe the requirements our overall system places on its storage subsystem, and the delete-optimized clustering approach we are taking, will enable it to scale dramatically beyond traditional file systems.

References

- [1] T. Blackwell, J. Harris, and M. Seltzer. Heuristic Cleaning Algorithms in Log-Structured File Systems. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.
- [2] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.
- [3] A. Gärtner, A. Kemper, D. Kossmann, and B. Zeller. Efficient bulk deletes in relational databases. In *Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, April 2001. IEEE.
- [4] Bezalel Gavish and Olivia R. Liu Sheng. Dynamic file migration in distributed computer systems. *Communications of the ACM*, 33(2):177–189, 1990.
- [5] Raxco Software, Inc. A tutorial on disk defragmentation for windows nt/2000/xp and windows server 2003. http://www.raxco.com/products/perfectdisk2k/whitepapers/defrag_tutorial%.pdf, May 2003.
- [6] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Symposium on Operating Systems Principles*, pages 117–130, 2001.
- [7] Marshall K. McKusick et al. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [8] J. Menon and L. Stockmeyer. An age-threshold algorithm for garbage collection in log-structured arrays and file systems. IBM Research Report RJ 10120, 1998.
- [9] J. Ousterhout and F. Douglass. Beating the I/O bottleneck: A case for log-structured file systems. *ACM Operating Systems Review*, 23(1):11–28, January 1989.
- [10] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, December 1995.
- [11] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [12] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.
- [13] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the Winter 1993 USENIX Conference*, pages 307–326, January 1993.
- [14] Jun Wang and Yiming Hu. Wolf – a novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, Monterey, CA, 2002.
- [15] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP autoraid hierarchical storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.
- [16] Stan Zdonik et al. The aurora and medusa projects. *Bulletin of the Technical Committee on Data Engineering*, pages 3–10, March 2003.