CS5600 project: s3logfs

Team: Rene Adaimi, Brian Hayes and Christopher Stadler
Repository: https://github.com/CJStadler/s3logfs
4/25/18

Introduction

Our goal in this project was to build a log-structured filesystem backed by Amazon's S3 cloud storage. This allowed us to take many of the filesystem concepts from the course, in particular from "The Design and Implementation of a Log-Structured File System" (Rosenblum and Ousterhout), and apply them to a novel domain.

Building a filesystem on top of S3 allows the user to take advantage of the reliability and scalability of S3 without needing to use S3-specific tools. There are already many such filesystem's available, but we have not found any open-source examples that are log-structured[1]. Aside from novelty though, is there any advantage to be had from using a log-structure on top of S3?

The primary justification for the log-structure given by Rosenblum and Ousterhout is that "this approach increases write performance dramatically by eliminating almost all seeks"[2]. S3 is an object store where each object is identified by a key. There is no concept of sequential keys, and so no "seek penalty.", though we trade this for latency. Most S3-backed filesystem's either store each file as an object (e.g. s3fs), or break each file into blocks which are each stored as an object (e.g. s3ql[3]). Both of these have performance disadvantages: the one object per file approach means that "random writes or appends to files require rewriting the entire file"[4]; the one object per block approach results in multiple requests for a single read or write spanning multiple blocks. Using a log structure eliminates both of these issues: we only need to re-write the blocks of a file that have been changed, but we can also make only one request to write multiple blocks (even if those blocks are from multiple files). Instead of reducing the *seek* penalty, the log structure amortizes the *request* penalty.

Another potential issue with implementing a filesystem backed by S3 is that S3 provides only "eventual consistency" for object updates[5]. For example, because it stores each file as a single object, keyed by name, read from s3fs "can temporarily yield stale data"[6]. The log-structure avoids this issue because each segment is never updated — new segments are

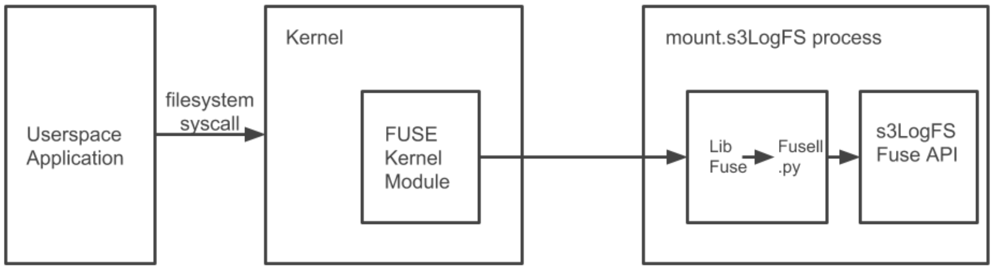
[1] ObjectiveFS is a proprietary S3-backed filesystem which is described as "log-structured" (https://objectivefs.com/features#technical-specification), but since the source code and detailed documentation are not available we cannot discuss its implementation.
[2] Rosenblum and Ousterhout 1999, pp. 1
[3] http://www.rath.org/s3ql-docs/impl_details.html#data-storage
[4] https://github.com/s3fs-fuse/s3fs-fuse#limitations
[5] https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel
[6] https://github.com/s3fs-fuse/s3fs-fuse#limitations

only appended to the log. S3 provides "read-after-write consistency for PUTS of new objects", so segments are guaranteed to be readable after they have been written.

Using S3 as the backend for a log-structured filesystem also has one advantage over the traditional disk based log-structure. Rosenblum and Ousterhout note that "for a log-structured file system to operate efficiently, it must ensure that there are always large extents of free space available for writing new data. This is the most difficult challenge in the design of a log-structured file system"[7]. By using S3 we do not need to worry about free space as it provides unlimited storage[8]. The only limitation is the segment "key space" — segments are keyed by an incrementing counter, so a filesystem could eventually run out of keys. However, in practice this is unlikely as S3 keys are limited to 1024 utf-8 encoded bytes[9].

Implementation

To implement our modified log-structured filesystem, we decided to make use of the FUSE (Filesystem in Userspace) module and an python wrapper called fusepy. This architecture allowed us to spend our time on the file system implementation and how it interacts with S3 considering the project deadline of only a few weeks. With this implementation user requests (read/write/etc) would be handled by FUSE and translated through the fusepy library to our code which defined the structure of the storage.

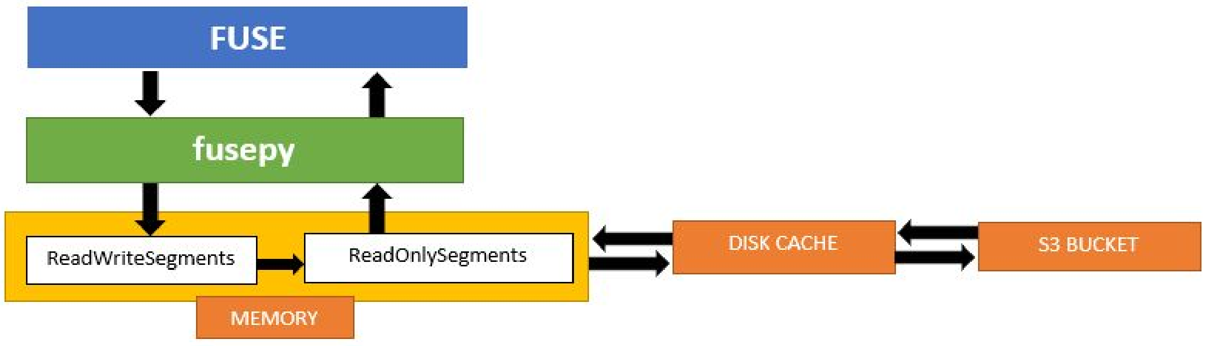


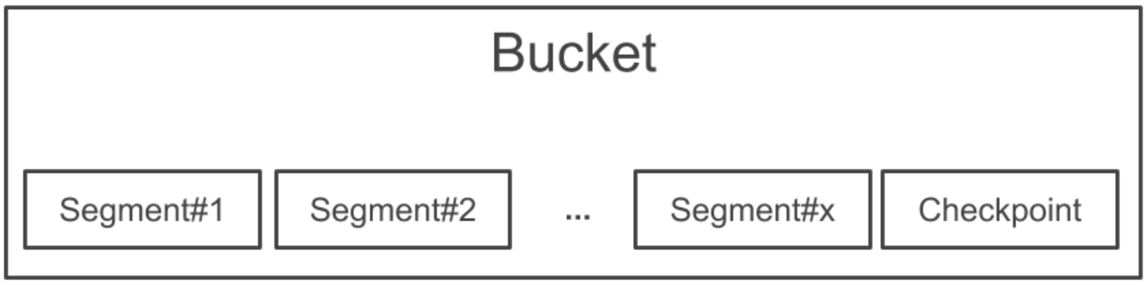
We have a primary API class called FuseApi which contains our code for each of the fusepy methods for handling files, in which we translate this interaction into reading and writing segment data. A segment being a grouping of writes, each representing different changes to the filesystem, which would then all be written together in one operation to S3. In each situation where a change happens to the file system, data is layered into the segment in a data first, then structure order, then finally updating any references that would connect these changes into the overall system. In this way, pointers and status would only take effect once the data that made it relevant was already written.

For performance and crash recovery reasons we implemented our data structures and classes in a way where we would maintain a number of segments in memory, using it as a

[7] Rosenblum and Ousterhout 1999, pp. 1
[8] https://aws.amazon.com/s3/features/
[9] https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html#object-keys

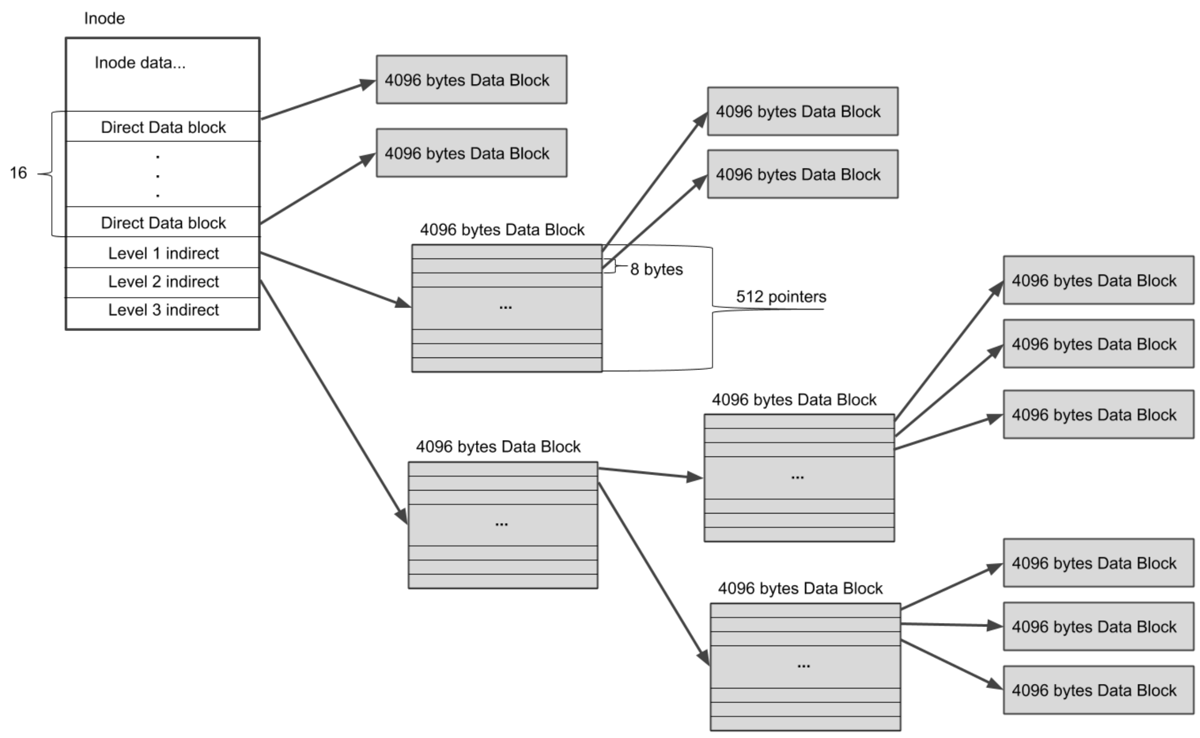
cache to speed up reads over requesting all data from S3. To add to this we had a secondary disk cache that would hold a larger number of segments in memory improving read performance for data that was not as recently accessed. Each segment write would be handed off to another thread so that the current I/O operations could continue without waiting. These threads would move the data down through the disk cache and then out to S3 storage. Read's would check at each cache layer before moving out to S3, and then would write the data upwards through the cache layers so that related and spatial data would be available in cache for future reads.





The organization of our file system is managed through our Inode, Segment and Log classes. Inode's maintain the relational information for the data, be it a file, directory, link, etc. Inodes are written into a segment (or segments) via the Log class which manages how segments are moved around and pushed out to S3 with a set of backend helper classes for each layer of storage. There is a Checkpoint class which is maintained in memory that maps the Inode's unique id to its address in the log. We represent addresses with a BlockAddress class, which stores a 6 byte unsigned int for the segment, and a 2 byte integer for the offset in the segment, totaling an 8 byte address. The Checkpoint tracks the root inode for the file system, as well as all related inode to address mappings which is our imap. The Checkpoint is written to S3 on a schedule, which would define our worst case crash failure data loss period. When a file system remounts, it loads the last Checkpoint, and then reads forward through the available segments that were written after the checkpoint, rebuilding its consistency. This generally leads to a fairly quick recovery.

Large file support is implemented via a common method of a set of direct addresses which point at data, as well as a single, double and triple layered indirect address chain leading to larger sets of data. The direct addresses requiring only a single read to reach their destination, whereas the indirect blocks read to obtain a block of addresses, which chain to other blocks of addresses and/or data mattering on the number of layers involved. Our implementation we generally used 16 direct blocks which could address approx 65K of data. We also implemented a single indirect, a double indirect and a triple indirect addresses. The combined storage for all mapping would allow for files up to 550GB in size, but this could be easily extended by increasing our block size which we tested at 4096 bytes, or by adding more indirect pointers at each level, or higher levels.



Known Issues and Missing Features

The following are known issues and/or features that we did not have time to implement with some notes as to how we would approach / resolve them if we had more time.

1. <u>Garbage collection</u>
   We started the implementation of a summary segment which would be written into each segment in our storage. This summary segment would have contained pointers to each block of data in our segment, the inode it was related too, and the version of the inode at the time it was written. We could then process segments in our Log, and test to see if when we check the inode, if the same data is still at the specified address

(segment/offset) we are checking. Additionally we can test if the inode version has increased since the block was written, indicating the file had been truncated and/or deleted. In either case we could mark the data as junk and then copy good data from the segment into a new segment and update the related pointers, then delete the old segment.

2. Remount Inconsistency
   File systems sometimes can not be mounted after being unmounted. We were able to identify an issue with how we were tracking hard links in relation to how FUSE tracked files internally. This has made the remount process more stable, but there are some situations where it has issues. We believe this to be caused by the timing of our Checkpoint write.

3. Reading files larger than 1GB
   Currently we can handle writing/reading files up to 1GB in size, the recursive logic we are using for indirect blocks when reading needs some adjustment to handle the 3rd indirect layer. I believe it has to do with determining the number of blocks it needs to load before handing off to the next step.

4. Directory data larger than 1.5MB
   As it stands right now, the data size for directories is limited to our direct addresses. We did not have time to adjust our code logic to take advantage of the larger file sizes since we were still working on how that functioned. Our short term workaround was increasing the number of direct addresses from 16 to 375, which increased our limitation from 65K to 1.5MB. Once our extended file support is completed, we would be able to use these methods with some adjustment to support expansive directory size.

Benchmarking

To test the performance of our implementation we ran several benchmarks against it and two other projects: s3fs and S3QL. We chose these because they seem to be two of the more popular and mature filesystems for S3, and they use file-based and block-based objects structures, respectively.

We created several benchmarks to test various access patterns: writing 32K (sequentially) to 100 different files, creating 100 4K files, and 1M of sequential and random reads and writes to a single file. Of course these are much simpler than real-world access patterns, but we hoped they would allow us to isolate the strengths and weaknesses of each filesystem. The benchmarks were run using the fio (Flexible I/O Tester) tool[10] and their configuration files are included in the "benchmarks" directory of the s3logfs repository. The results are reported in the following table, in I/O operations per second:

[10] https://github.com/axboe/fio

	Write 32K to 100 Files	Create 100 4K Files	Single File Rand. Reads	Single File Rand. Writes	Single File Seq. Reads	Single File Seq. Writes
s3fs (cached)	550	6	3047	1412	5059	2293
S3QL (cached)	5095	1041	13913	6994	14065	7111
s3logfs (cached)	2285	740	5638	2237	5311	2249
S3QL (uncached)	11	19	18	2	17	3
s3logfs (uncached)	29	248	28	731	36	647

The results are divided into two groups: first we used the default configuration for each filesystem, and then we re-ran the benchmarks on S3QL and s3logfs while disabling their caching (we were not able to disable caching for s3fs). The goal of this was to isolate the differences in filesystem structure from the caching behavior.

In the cached group we can see that s3logfs outperformed or was competitive with s3fs for all benchmarks. This is not very surprising as s3fs's object-per-file approach favors simplicity over performance. The performance difference is most significant when writing to and creating multiple files, which is consistent with our expectation that s3fs would need to make 1 request per file, while s3logfs can include changes to multiple files in the same segment.

S3QL had the highest performance across all benchmarks in the cached group, although the gap was closest for creating and writing multiple files. S3QL was clearly slower at writing to multiple files than to a single file, while all write benchmarks were approximately equal for s3logfs. This suggests that the log-structure may be providing a benefit for writes to multiple files, but it is overshadowed by S3QL's aggressive use of caching and asynchronous requests.

In the uncached group S3QL's performance greatly degrades. Of course this is not very realistic, and a more nuanced set of benchmarks would compare performance across a range of cache sizes, but it does suggest that the block-based structure is highly dependent on caching. S3logfs also has worse performance, performance decreased much less for writes than for reads. Without caching each read requires a request, while when writing a request is only necessary when the current segment has been filled.

Conclusion

In conclusion, we showed that even a somewhat incomplete implementation of a log-structured file system built on S3 can work. It has its strengths and weaknesses in relation to performance, being fast with writes and file creation, and mattering on cache hit/miss and spatiality of data, result in slow reads. That said it could still be well suited to specific use cases such as server environments where data needs to be written often, and offloading such data to inexpensive S3 storage aligns with a business organizational goals. Additionally there are a number of legacy systems which have file based backup technologies which can not talk to systems like S3 via API calls, but could use this method instead. Lastly, if we had more time, we feel we could expand this system's file size support, implement versioning, and likely improve its performance as the current system is a proof of concept rather than a production-ready product.