Cloud Native LevelDB

Aditi Singh (aditi@cs.wisc.edu)
John Shawger (shawgerj@cs.wisc.edu)
Karan Grover (kgrover2@wisc.edu)
Shubhankit Rathore (shubhankit@cs.wisc.edu)

12th May, 2022

1 Introduction

LevelDB is a key-value store based on LSM Trees. It supports snapshots, range-queries and other operations that are useful for modern applications. In this project, we try to make LevelDB cloud native by leveraging the elastic object storage provided by Amazon S3. This also allows us to decouple the compaction process from the process serving the requests to LevelDB. In this report, we describe the design, implementation and evaluation of our system.

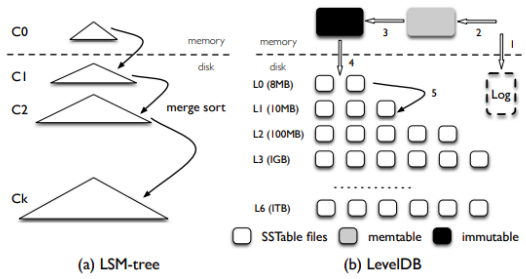

Figure 1: Architecture of LevelDB

2 Background

2.1 LevelDB

LevelDB is a key-value store based on LSM Trees[3]. The architecture is inspired from BigTable. When LevelDB receives a write request, it stores the key-value pair in a log file and an in-memory memtable. Once the memtable is filled to a pre-fixed threshold size, LevelDB switches to a new memtable and log file. In the background the old memtable is converted to an immutable memtable and a different thread flushes it to disk. This generates a new SSTable file at level 0. The corresponding log file is discarded. The memtable is implemented as an in-memory sorted skiplist. The size of files in each level has a fixed limit and increases by 10 as the level increases. The number of levels are limited to seven. Once a level is full, the compaction thread takes one file from that level, performs merge-sort of the file with all files of the higher-numbered level and writes new file on that level. This is done until all levels are within their size limit.

2.2 Amazon S3

Amazon S3 is an object storage service provided by Amazon that provides scalability, availability and performance. The basic storage unit in Amazon S3 are objects that are stored in buckets. Each object is identified by a unique user-specified key.

2.3 Related Work

WiscKey[2] improves the performance of LevelDB by storing keys and values separately, thus reducing the IO amplification of the LSM tree. Bourbon[1] uses a learned index to improve the lookup time of keys in WiscKey. In our system, storing SSTables on S3 and performing compaction in the background will not lead to lower write latencies, but it could allow the system to better respond to incoming read requests while compaction is being performed.

3 Motivation

An instance of LevelDB stores the SSTables on the filesystem of the same node. The storage and compute of the system is tightly coupled together. This can lead to scenarios where the underlying storage is over or under-provisioned. Decoupling storage with compute will allow us to better utilize the storage and using S3 would allow us to pay only for what we use and as we use it. Another opportunity of improvement lies in performing the compaction process on a different node, freeing up resources on the node that's serving requests on the client.

Hence, we propose two improvements over the current levelDB architecture:

- Using Amazon S3 for storing SSTables
- Performing compaction on a separate node

4 Design

4.1 Amazon S3

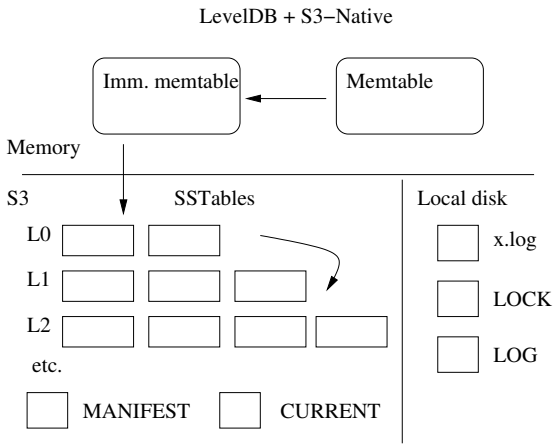

Figure 2: LevelDB, Local storage, and S3

LevelDB uses a write-ahead log to persist write operations before storing them in the in-memory memtable. Without the write-ahead log, writes stored only in the memtable would be lost if the system crashed or was shutdown. Since S3 objects are not mutable, maintaining the write-ahead log on S3 would be very expensive – an entire new object would have to be written every time the log was appended to. Thus, we chose to keep the log locally, which gives us the same durability guarantees as the original LevelDB, and has no impact on our ability to perform compaction on a separate thread.

We explored two methods of storing LevelDB SSTables on S3. The first was to use a FUSE filesystem, S3FS, and mounting our S3 bucket to a local directory. This had the advantage, of being easier to use, but later experiments revealed that S3FS did not provide adequate performance. We were able to improve performance considerably by using the S3 API directly.

S3 is an object store, so it primarily supports two operations – Get and Put. Whenever LevelDB opens a file for reading or writing, we Get the file from S3, perform read and write operations locally, and Put it back if it was modified. This simple scheme yielded better performance than S3FS, although one can imagine further improvements such as an LRU cache, with entries invalidated if the file on S3 is updated by another node.

4.2 Background Compaction

One of our contributions was to perform compaction, an expensive process, on a background node to free the primary node in order to respond to other requests. In the course of designing the system, we considered two strategies of communication between our two nodes.

4.2.1 Design Attempt 1

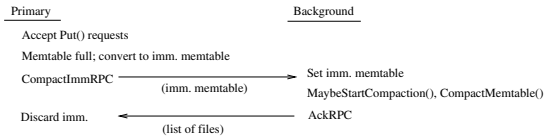

Figure 3: Design Attempt 1 – Discarded Protocol

After LevelDB has accepted a certain amount of write requests, its in-memory memtable fills up, is converted to an immutable memtable, and a new memtable is created. The immutable memtable is then written to Level 0 of the database as an SSTable. Our first design centered around sending the immutable memtable to the background node, and having the background node begin compaction. In the simplest case, compaction would simply result in the immutable memtable being written to disk, but it could also trigger a longer compaction if multiple files in level 0 needed to be merged. While waiting for an acknowledgement from the background node, the primary node could continue to serve read requests using the immutable memtable, and would also be free to accept incoming write requests in a new, empty, memtable. Upon completion of the compaction operation, the background node would respond to the RPC, and primary node could discard the immutable memtable, which has now been written to disk. Advantages of the solution are that the primary node never needs to write to S3, and it can serve read requests for key recently written directly from memory.

Unfortunately this solution has a few drawbacks, mostly involving communication. The memtable is a complex data structure, would would need to be serialized before being sent in an RPC. For a data structure several MB in size, this is non-trivial, both from an implementation perspective and also computationally. Additionally, the background node would need to send the primary an updated list of SSTable files when compaction has completed, because the files on disk would have changed. In a large database, there could be potentially hundreds or thousands of SSTable files in S3, so serializing and processing this list would be another expensive operation. For these reasons, we discarded this approach.

4.2.2 Design Attempt 2

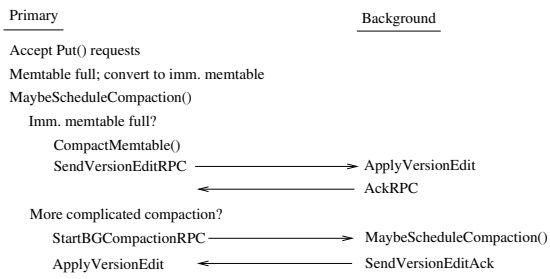

Figure 4: Design Attempt 2 – Accepted Protocol

Some of our experiments had shown that compacting the immutable memtable (that is, writing it to S3, without any further compaction operations) was a relatively cheap operation. We decided that this could best be performed by the primary node, and the background node should be used for longer compaction operations.

LevelDB maintains the state of the system, such as lists of files in each level, in a data structure called a `VersionSet`. The most recent of these versions is called `current`. When a new current version is created, LevelDB first populated a data structure called a `VersionEdit`, which contains all the updates to the current version. The Edit is then applied and a new current version is created. Fortunately, LevelDB provides `EncodeTo` and `DecodeFrom` methods which encode and decode `VersionEdit`s using strings, which are easily serializable. Thus, we can use RPCs and `VersionEdit`s to synchronize the current versions of both of our nodes. Whenever a node modifies the current version in a compaction operation, it generates a `VersionEdit` and sends it via RPC to the other node.

5 Implementation

To implement our system we modified existing LevelDB codebase. We run two EC2 instances as described in Section 2. The EC2 instances have 4 CPUs, 16GB memory, 30GB of EBS storage and 5GBps bandwidth between them.

The communication between the two nodes takes place using gRPC. We used gRPC instead of other communication methods like REST API or Pub-Sub framework as gRPC allows us to serialize objects efficiently. The RPC specification is given below.

```
message VersionEditMsg {
        bytes encversionedit   = 1;
        uint64 lastsequence    = 2;
        uint64 nextfilenumber = 3;
}

message StartCompaction {
        uint64 nextfilenumber = 1;
}
```

Both instances have access to the S3 instance where the SSTables are stored. To communicate with S3 we use the native S3 C++ API.

5.1 Interesting Bugs

There were two bugs which cost us considerable time in implemented background compaction. Once compaction has completed on the background node, it must set a VersionEdit to the primary node, which is encoded into a string. Since the VersionEdit is generated in a separate thread, the easiest solution was to write into a global string, so the RPC could access it before responding to the primary node. Somehow, the value in the global string was being changed between the time that it was created and it was being accessed to store in the RPC. We verified that no other functions that modified the string were being called, and that there was no ordering problem due to the separate threads. Saving the VersionEdit to a local string and immediately copying to the global string resolved the issue.

The second bug involved some state that we assumed was send in the VersionEdits but is maintained separately. Every file created in LevelDB is assigned a new number – this is maintained by a counter that is incremented whenever a log or SSTable file is created. We were getting mysterious bus errors while performing compaction on the background node because it was overwriting files with the wrong numbers (files it was supposed to be reading). Sending the `next_file` counter along with our RPC to begin background compaction resolved the issue.

6 Evaluation

6.1 S3FS vs Native-API

We performed latency and throughput analysis by running LevelDB on S3 mounted as File System and by integrating the native APIs of S3 with the LevelDB. We had expected a higher latency and a lower throughput on S3FS due to the fact that it has to capture all the POSIX commands on a mounted directory in order to perform operations on S3. Our measurement results are shown in Fig.5 for get() operations and in Fig.6 for put() operations. We can clearly observe that the throughput is higher for the native S3 APIs for majority of file sizes, but eventually the performance of S3FS passes at large

file sizes. It could be due to some of the optimizations performed by S3FS for larger files. It is more evident in the case of get() probably due to some caching and prefetching done by S3FS in the background, while it is not the case for put() when file has to be written back to S3.

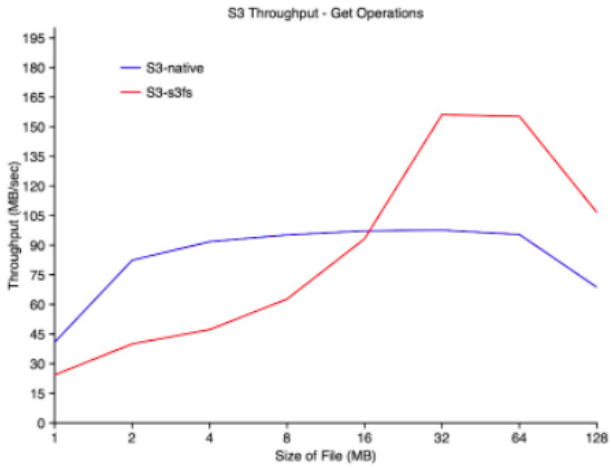

Figure 5: Throughput comparison for get() on s3

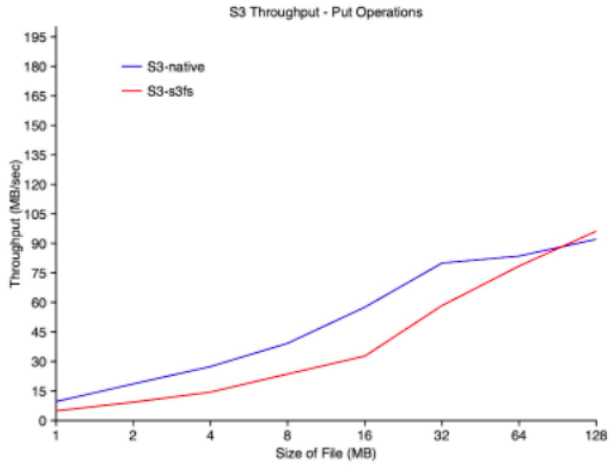

Figure 6: Throughput comparison for put() on s3

6.2 Latency

We measure the write latency of two different workloads - Sequential and Random. As the name suggests, the Sequential Workload writes sequential keys to LevelDB, and Random Workload writes random keys to LevelDB. We measure the write latency for both of these workload in a variety of settings. The settings are as off-the-shelf leveldb, an implementation where all files are written to S3 using S3FS, an implementation where some files are written to local and some to S3 using S3FS, the same implementation with the native S3 API, and the same implementation with a different node doing background compaction. As seen in Figure 7, we were able to continuously improve latency performance in implementations involving writing to S3 with a small degradation in our final implementation. This was because the extra RPC calls that we make during compaction and the overhead of marshalling and unmarshalling data.

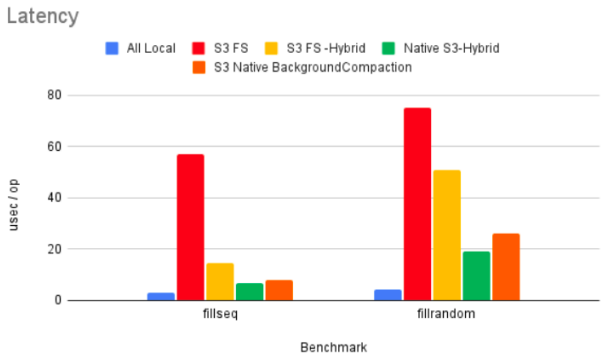

Figure 7: Latency measurements

6.3 Throughput

We analyzed the write throughput for the two workloads described above - Sequential and Random in Figure 9 and Figure 8. It is quite clear from the graph that the throughput of off-the-shelf LevelDB is the highest. This is because it writes everything to local and disk and does not incur the extra overhead of communicating with a separate node. We see a performance degradation when we write the SSTables to S3 and we notice and extra degradation due to the overhead of communicating with a separate node. While we see a uniform trend for the Sequential Workload, the Random Workload shows mixed results. We assume this is because of a variation in times and frequency of background compaction when adding random nodes. As future work, we would like to run the workload with fixed seed to be able to see a more uniform trend.

6.4 Trend of Write Latency

We have plotted the write latencies in a sequential write workload over the course of several compactions. Our goal was to visualize how Put() latencies vary when the system is compacting and when it is not. In 10, we present the trend for vanilla leveldb. This is our baseline. The red bars represent compaction in progress. We see that write latencies are slightly higher during compaction stage. This is because writes are delayed

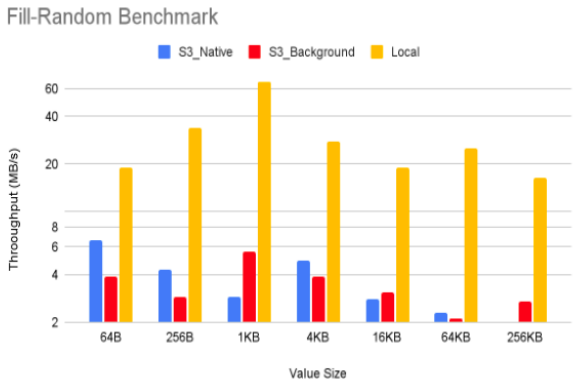

Figure 8: Throughput for Fill Random Benchmark

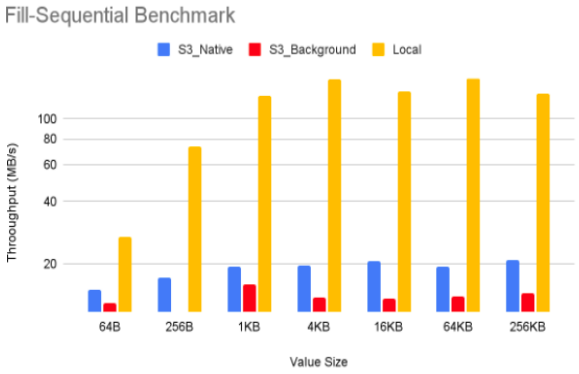

Figure 9: Throughput for Fill Sequential Benchmark

by 1ms by the system automatically to prevent the next memtable from filling up too soon.

Comparing this baseline with the trend in 11, we see that there is a stretch of time around the 24 sec mark on the x-axis when there are no writes done. This is because running the compaction naively on s3fs is so slow that another memtable is filled up while an older memtable is being compacted. Thus the writes are waiting. Also, write latencies spike more frequently and to higher numbers in this case.

Comparing the baseline with the trend in 12, the compaction takes longer and latencies spike much higher during it.

Comparing the baseline with the trend in 13, we see that the write latencies are the highest of all cases. The compaction requires RPC calls to trigger it. RPCs incur extra communication cost and computation cost for marshalling and unmarshalling of data. Due to this slow down we observe further spikes in write latency and multiple intervals of time where no write is happening.

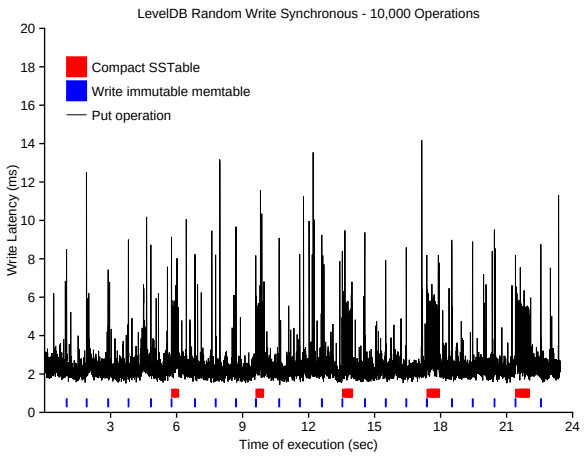

Figure 10: Latency Timeline plot for vanilla leveldb

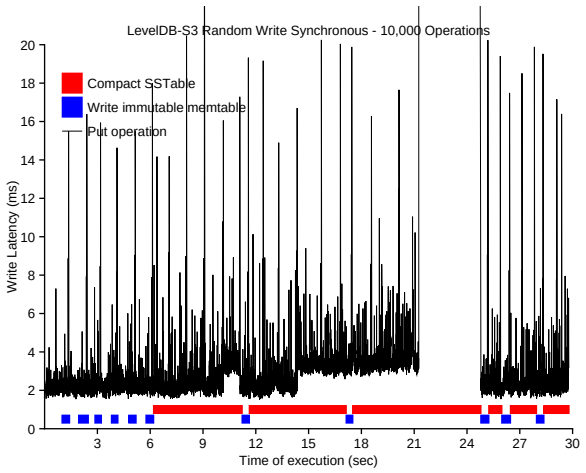

Figure 11: Latency Timeline plot for leveldb with S3FS

7 Conclusion

Through our experiments, we have arrived at the following conclusions:

- Using the Native S3 API is much more efficient than S3FS. While S3FS can be used to implement a proof-of-concept quickly, using the Native API brings huge performance gains.
- It is possible to redesign LevelDB in a way where we can take advantage of both local and remote storage.
- A Cloud Native LevelDB will always be slower than a local LevelDB. The gains lie in being able to achieve lower latency for writes and reads which happen concurrently with compaction. We would

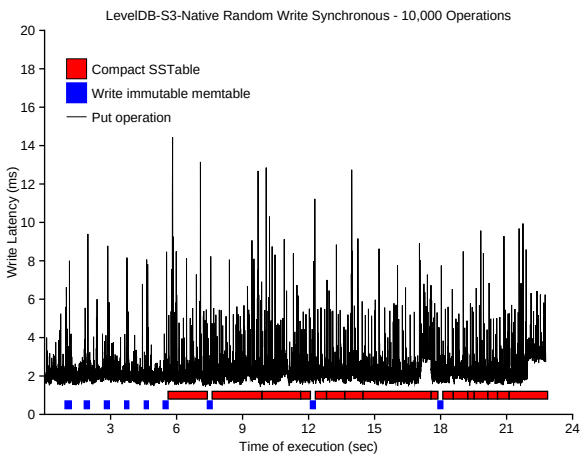

Figure 12: Latency Timeline plot for leveldb with S3 Native API integration

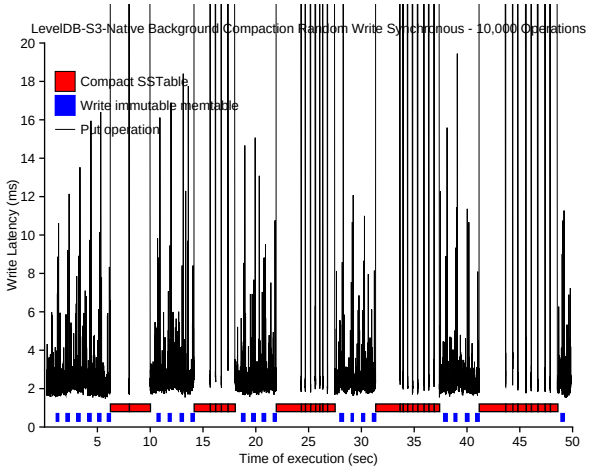

Figure 13: Latency Timeline plot for leveldb with background compaction

like to find a workload which best suits this use-case.

References

[1] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. From {WiscKey} to bourbon: A learned index for {Log-Structured} merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171, 2020.

[2] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[3] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.