November 2021

Filesystem API based on Log Structured Merge (LSM) Trees

Chris Suter

James Sullivan

Filesystem API based on Log Structured Merge (LSM) Trees

ABSTRACT

This disclosure describes the application of LSM trees as a primitive data structure to provide a file-like Application Programming Interface (API). The LSM tree data structure is used to implement an object store. Namespace features, such as support for directories, can be layered on top of objects to form a hierarchy of object stores, with a root object store connected to an arbitrary number of child object stores that store user directories and files. Since the tree stores deltas on objects, it can contain multiple entries per object ID. However, within a given layer of the tree, there can be at most one entry for a given key within the object. The filesystem achieves performance by writing mutations to a fast in-memory layer, which is flushed at appropriate times to minimize latency caused by disk write operations. Moreover, the use of a journal can help minimize data loss from unflushed write operations in the event of sudden loss of power.

KEYWORDS

- Log Structured Merge (LSM) tree
- Filesystem
- Object store
- Journal
- Tree compaction
- Mutable tree layer
- Skip list

BACKGROUND

A Log Structured Merge (LSM) tree is a data structure in which each successive layer records the differences (also referred to as deltas) over the previous layer in the tree.

Periodically, the layers are merged to generate new layers to replace the old ones. The process of merging layers to generate new ones is termed as compaction. The precise format of the persistent layers within an LSM tree can be easily evolved, which is one of the compelling reasons for using LSM trees as a primitive data structure to support operations within applications.

The LSM tree can serve as an independent library for storing arbitrary key-value pairs. However, it needs appropriate interfaces to be used as the backing for a filesystem.

DESCRIPTION

This disclosure describes the application of LSM trees as a primitive data structure to provide a file-like Application Programming Interface (API) system. The layer files within the LSM tree store deltas of various objects, such as allocation records or directory entries. The in-memory mutable layer can be implemented as a skip list. These LSM trees are used to implement an object store, which is a key-value interface with basic namespace features, such as support for directories. A hierarchy of object stores are constructed:

1. **Root parent object store**: This object store provides only a memory layer and no persistent layers, thus avoiding the need for it to have a parent object store. The root parent object store exists only for the purposes of providing backing for the root object store and the journal file.
2. **Root object store**: The root object store contains all other filesystem metadata (e.g., allocation information) and serves as the backing for all other object stores. No user data is stored as part of the root object store.

3. **Child object stores**: A root object store can be connected to an arbitrary number of child object stores, each of which represents a filesystem volume. Child object stores can store user data in directories and files.

```
struct ObjectKey {
    object_id: u64,
    data: ObjectKeyData,
}

enum ObjectKeyData {
    Object,
    Attribute { attribute_id: u64 },
    Extent(ExtentKey),
}

struct ExtentKey {
    attribute_id: u64,
    range: std::ops::Range<u64>,
}

enum ObjectValue {
    Object { object_type: ObjectType },
    Attribute { size: u64 },
    Extent(ExtentValue),
}

enum ObjectType {
    File,
    Directory,
    Volume,
}

struct ExtentValue {
    device_offset: u64,
}
```

Fig. 1: Example pseudocode describing contents of the data structure for filesystem objects

As illustrated in the pseudocode in Fig. 1, objects contain a collection of attributes in the form of key-value pairs, with Attribute ID 0 as the default attribute where a file's data is contained. For instance, attributes can be used to store POSIX-style extended attributes.

Attributes contain extents. Objects are indexed using 64-bit unsigned integers that serve as object identifiers (ID). All offsets in Fig. 1 are byte offsets. Since the tree stores deltas on objects, it can contain multiple entries per object ID. However, within a given layer of the tree, there can be at most one entry for a given key within the object. Namespace features, such as support for directories, can be layered on top of objects.

The object store can support two write modes: copy-on-write and overwrite. While the overwrite mode provides a performance boost for some applications, it can interfere with features that involve snapshots or cloning. Importantly, the overwrite mode can support a journal implemented as an ever-growing file. The journal cannot use copy-on-write because the extents for the journal must be known when replaying the journal at mount time. Therefore, all extents for the journal are preallocated, and writing to the journal is done so as to ensure that enough extents are preallocated for the next range of the journal file before the current range is finished.

The journal can support transactions - the ability to stream and apply atomic mutations to multiple filesystem objects. Mutations can be grouped together into a single transaction such that only complete transactions are applied at replay time. When the LSM tree is compacted, the extents at the beginning of the journal file can be freed up. Having object IDs that are monotonically increasing 64-bit numbers can avoid the need for wrapping. In addition, the journal can provide fast persistence of changes that may otherwise be lost in the event of a power loss. Fig. 2 shows an illustrative pseudocode for a number of mutations that can be supported.

```
enum Mutation {
    // Inserts a record.
    Insert {
        item: ObjectItem,
    },

    // Inserts or replaces a record.
    ReplaceOrInsert {
        item: ObjectItem,
    },

    // Inserts or replaces an extent.
    ReplaceExtent {
        item: ObjectItem,
    },

    Allocate(AllocatorItem),

    Deallocate(AllocatorItem),

    // Seal the mutable layer and create a new one.
    TreeSeal,

    // Indicates a compaction took place and the immutable layers have changed.
    TreeCompact,
}
```

Fig. 2: Example pseudocode illustrating mutations that can be applied to filesystem objects

The `Insert`, `ReplaceOrInsert` and `ReplaceExtent` mutations shown in Fig. 2 apply to object stores. The operational semantics of the mutation may differ depending on the implementation. For example, `Insert` can assume there is no existing entry, `ReplaceExtent` can perform a range-based replace in which a single extent overwrites multiple existing records. The precise semantics of the operations can be determined based on the specifics of the use case and performance requirements.

The `Allocate` and `Deallocate` mutations are applied to an LSM tree which stores allocation information, and increment or decrement reference-counts of extents which are stored in this tree. Initially, all extents have only a reference count of 1. However, if cloning is supported, extents can have counts greater than 1. Deallocations are represented with a delta of -1. These operations can be performed simply by iterating over the tree and looking for holes.

The `TreeSeal` and `TreeCompact` mutations shown in Fig. 2 are tree operations that support compactions. Sealing involves making the current mutable layer immutable and then creating a new mutable layer. Once sealing is done, a background compaction process takes place. When compaction is complete, a compact operation is recorded in the journal to indicate that the immutable layers of a tree have changed. The background compaction process has no impact on other filesystem activity (e.g., other reads or writes) since it operates only on immutable layers that do not require locking to prevent access by other processes.

Since there are no explicit flush calls involved, the underlying device can reorder writes. Therefore, all writes to the journal include a checksum (e.g., a Fletcher-64 checksum) for each block of configurable size. For example, if the block size is 4 kB, each block in the journal file will consist of 4088 bytes of serialized data, followed by an 8-byte Fletcher-64 checksum. The checksum can be salted with the checksum value for the previous block. The salt for the first block of the journal can be chosen at random at mount time to make it unlikely for the journal to be confused for one from a previous incarnation.

At any point in time, there can be at least a block of data waiting to be flushed to the journal. For enhanced performance and write amplification, it may be useful to delay flushing as much as possible within memory and other operational constraints. However, if a client forces a flush before a block is fully filled, a special record can be used to pad empty space within the block prior to writing it along with its checksum. At that point, no more data would need to be written to fully recover all outstanding data. However, it may be necessary to flush the underlying devices to ensure persistence of the data cached at their levels.

If replaying the journal encounters a block with mismatched checksum, the writing can be continued from the previous point if the shutdown prior to the replay is known to be graceful. Otherwise, it is possible that the previous block left an uncommitted transaction and, quite probably, a record that cannot be deserialized. Such situations can be addressed with a checksum marker for resetting the journal stream. For instance, if the stored checksum is the expected checksum ^ 0xffffffffffffffff, then the journal stream is considered reset, and all half complete transactions are discarded.

Since different LSM trees can flush their mutable layers at different times, the location of synchronization of each tree within the journal stream must be marked. Since the journal file is ever growing, the file offset within the journal file for all trees that use the journal can be recorded for these purposes.

A superblock can contain version information, an instance identifier, e.g., Universally Unique Identifier (UUID), etc. In addition, a superblock contains the information shown in Fig. 3.

```
struct SuperBlock {
    // The object ID of the root store (which is stored in the root parent store).
    root_store_object_id: u64,

    // The object ID of the allocator object (which is stored in the root store).
    allocator_object_id: u64,

    // The object ID of the journal (which is stored in the root parent store).
    journal_object_id: u64,

    // The point to start reading the journal.
    journal_checkpoint: JournalCheckpoint,

    // Offset of the journal file when the superblock was written.
    super_block_journal_file_offset: u64,

    // object id -> journal file offset. Indicates where each object has been
    // flushed to at the time the superblock was written.
    journal_file_offsets: HashMap<u64, u64>,
}
```

Fig. 3: Example pseudocode listing various pieces of information contained in a superblock

A superblock can be immediately followed with a copy of all records present in the root parent store from its mutable layer. A superblock exists as A/B copies that are two corresponding files in the root object store. The beginning of each of the two files is at known locations on the disk. The only reserved regions on the block device are for the initial extents for the A/B superblocks. All other regions are free to be used for metadata or data as required. Records can be stored in the superblock in the same way as in the journal as described above, and the same reader as the journal can be used.

When writing a new superblock, first, the minimum referenced offset within the journal is located. Then, all extents preceding that offset can be trimmed, thus freeing the space used by the journal. Further compactions can be initiated in case free space is running low. In addition, a limit can be imposed on the amount of data that needs to be replayed from the journal and trigger compactions when the limit is exceeded.

As Fig. 4 shows, records for directory entries can be stored in the same tree that stores other object store information.

```
enum ObjectKeyData {
    …
    Child { name: String },
    …
}

enum ObjectValue {
    …
    Child { object_id: u64, object_type: ObjectType },
    …
}
```

Fig. 4: Example pseudocode showing a record for a directory entry stored in an LSM tree

Support for objects that must be purged upon mount can be provided by storing such objects in a special directory that is purged upon replaying the journal after mounting. The techniques described in this disclosure can support case-sensitive filesystems, including Unicode filenames. If needed, filenames can be normalized using techniques similar to those typical in mainstream device operating systems.

The format of the persistent layer files is flexible and can be evolved as needed. For example, simply serializing records in sorted order might suffice initially, but there is room to optimize these layer files perhaps using B-Trees, Bloom filters and compression (since the data will likely compress well).

The techniques for implementing an existing filesystem API in a manner described in this disclosure can be invoked by any device, operating system, or application. The API can boost performance and speed of file operations by augmenting the on-device filesystem with fast in-memory file write operations that are flushed at appropriate times to minimize latency caused by disk write operations. Use of the filesystem API can help reduce data loss from unflushed file write operations in the event of sudden loss of power.

CONCLUSION

This disclosure describes the application of LSM trees as a primitive data structure to provide a file-like Application Programming Interface (API). The LSM tree data structure is used to implement an object store. Namespace features, such as support for directories, can be layered on top of objects to form a hierarchy of object stores, with a root object store connected to an arbitrary number of child object stores that store user directories and files. Since the tree stores deltas on objects, it can contain multiple entries per object ID. However, within a given layer of the tree, there can be at most one entry for a given key within the object. The filesystem achieves performance by writing mutations to a fast in-memory layer, which is flushed at appropriate times to minimize latency caused by disk write operations. Moreover, the use of a journal can help minimize data loss from unflushed write operations in the event of sudden loss of power.

REFERENCES

1. Messmer, Sebastian. "CryFS: Design and implementation of a provably secure encrypted cloud filesystem." Institute of Theoretical Informatics, Karlsruhe Institute of Technology (2015).