Simple Log-structured Key-Value Store

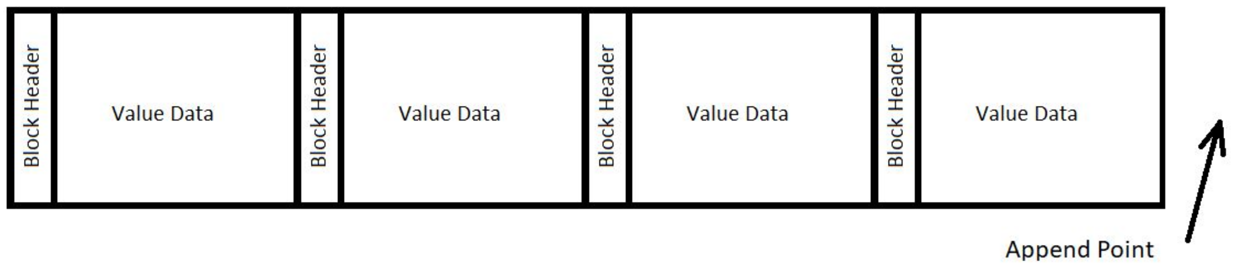
Jonathan Ludwig

Overview

This simple key-value store will allow storing key-value pairs to a log-structured file. All keys will be unique 64-bit numbers and all values will be variable length byte arrays from 1 to 4096 bytes in size. The key-value store will hold up to 100,000 values.

On-disk Layout

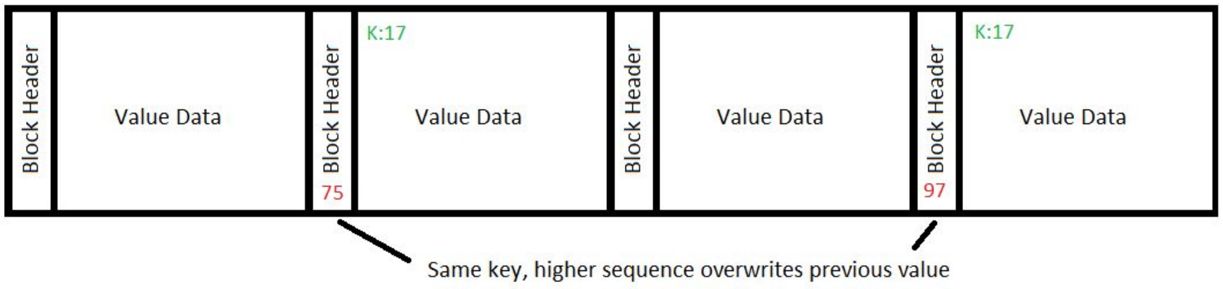
The on-disk layout is simple. Because the keys are a fixed size and because values can only be up to 4096 bytes in size, I have decided to store all key-value pairs in a fix-sized 4128 byte block in the file. 4096 bytes for the value and 32 bytes for a key-value block header. The fixed key-value block size was chosen to simplify the implementation. With a single fixed size block for each value, garbage collection is not necessary, because no free space is lost due to fragmentation: all allocations for a key-value pair are like sized.



The key-value block header contains the following values:

- A 64-bit signature identifying the block as a valid log block
- The 64-bit key
- A 64-bit log sequence number for the block
- The size of the value data stored in the block
    - A valid value size is 1-4096 bytes
    - A 0 byte size means the block is not allocated and is used to delete a prior version of that key.

Block allocation starts with the first block of a file and continues to the next block. Once the end of the file is reached, allocation starts back at the beginning of the log at the first unallocated block. This means the log will always move forward, but it will sometimes skip blocks for keys that are already allocated and were stored in a previous scan of the log. This is necessary, because the log does not support garbage collection.



Key-value Store operations

The key-value store maintains a 64-bit sequence that is incremented each time a key-value pair is stored or deleted. This is a global sequence and is not a per-key sequence. An in-memory directory object is used to map each key to its latest block or not at all, if the key has been deleted from the store. An in-memory bitmap is maintained of all blocks used by valid keys.

Set

For a set operation, a new key-value block is written to the log for this key id. The value size must be non-zero. The latest sequence is stored in the block. This block is marked in memory as allocated and the mapping from this key to the block is stored in the in-memory directory.

Get

If the key is not stored in the directory, a not-found error is returned. Otherwise, the directory returns the block where this value is stored and its size in bytes. The key-value store reads the value from the block.

Delete

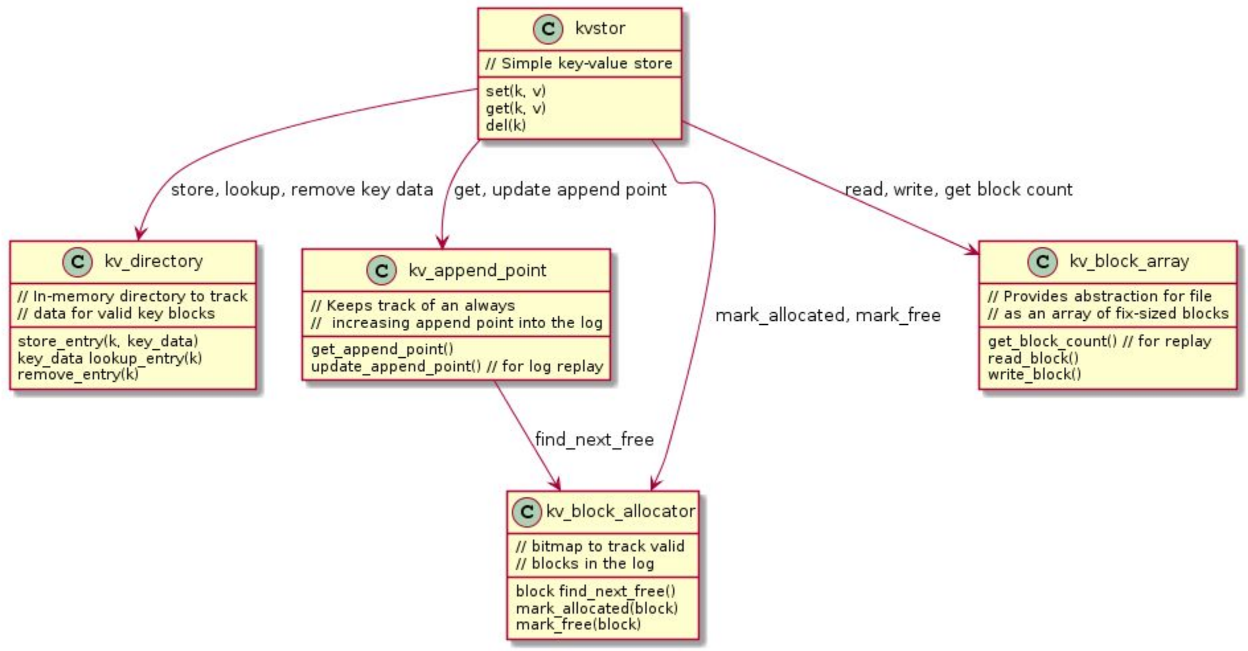
For delete operations a key-value block is written to the log with the key of the key-value entry being deleted and a size of 0. The latest sequence number is used so that at replay the key will be marked as deleted. The key is deleted from the in-memory directory and the block is marked allocated in memory. The old block is marked as not allocated so it can be reused. This has the disadvantage that the entry for the deleted block must be marked as allocated and be maintained in the log until a new value is written for that key. This wastes a lot of space in a workload with a lot of key deletes, but was done for simplicity. To solve this we would have to maintain a validity map in the log and write it to the log for every delete key operation.

Log Replay

For simplicity, the log does not use a header at the beginning of the file. The key-value store is replayed each time it is opened. The log replay starts at the beginning of the file and checks each block. It keeps the value with the latest sequence for each key. If it encounters a valid entry for the key with a zero byte size and that entry has the highest known sequence for that key, the key is marked as deleted. If the block has the highest sequence number seen for that block, it is marked as allocated in memory, even if it was for a deleted key. If the key overwrites a previous block that is tracked by the directory, the previous block is marked as free.

Code Design

The code to implement the key-value store has been broken into separate classes and modules. This has been done isolate different parts of the code and facilitate unit testing to test different parts of the code more easily. The following diagram shows the classes and their relationships.



kvstor
// Simple key-value store
set(k, v)
get(k, v)
del(k)
store, lookup, remove key data
get, update append point
mark_allocated, mark_free
read, write, get block count
kv_directory
// In-memory directory to track
// data for valid key blocks
store_entry(k, key_data)
key_data lookup_entry(k)
remove_entry(k)
kv_append_point
// Keeps track of an always
// increasing append point into the log
get_append_point()
update_append_point() // for log replay
kv_block_array
// Provides abstraction for file
// as an array of fix-sized blocks
get_block_count() // for replay
read_block()
write_block()
find_next_free
kv_block_allocator
// bitmap to track valid
// blocks in the log
block find_next_free()
mark_allocated(block)
mark_free(block)