PNA Rust Project 2: Log-structured file I/O

Task: Create a *persistent* key/value store that *can be accessed from the command line*.

Goals:

- Handle and report errors robustly
- Use serde for serialization
- Write data to disk as a log using standard file APIs
- Read the state of the key/value store from disk
- Map in-memory key-indexes to on-disk values
- Periodically compact the log to remove stale data

Topics: log-structured file I/O, bitcask, the `failure` crate, `Read` / `Write` traits, the `serde` crate.

Introduction

In this project you will create a simple on-disk key/value store that can be modified and queried from the command line. It will use a simplification of the storage algorithm used by [bitcask], chosen for its combination of simplicity and effectiveness. You will start by maintaining a *log* (sometimes called a "write-ahead log" or "WAL") on disk of previous write commands that is evaluated on startup to re-create the state of the database in memory. Then you will extend that by storing only the keys in memory, along with offsets into the on-disk log. Finally, you will introduce log compaction so that it does not grow indefinitely. At the end of this project you will have built a simple, but well-architected database using Rust file APIs.

<!--

Basic database architecture

TODO

- Basic description and terminology of log, memtable, blocks, etc
- good opportunity for a diagram
- find a good background reading
- using the os page cache for caching -->

Terminology

Some terminology we will use in this course. It is the same as or inspired by [bitcask]. Different databases will have slightly different terminology.

- *command* - A request or the representation of a request made to the database. These are issued on the command line or over the network. They have an in-memory representation, a textual representation, and a machine-readable serialized representation.
- *log* - An on-disk sequence of commands, in the order originally received and executed. Our database's on-disk format is almost entirely made up of logs. It will be simple, but also surprisingly efficient.
- *log pointer* - A file offset into the log. Sometimes we'll just call this a "file offset".
- *log compaction* - As writes are issued to the database they sometimes invalidate old log entries. For example, writing key/value `a = 0` then writing `a = 1`, makes the first log entry for "a" useless. Compaction — in our database at least — is the process of reducing the size of the database by remove stale commands from the log.
- *in-memory index* (or *index*) - A map of keys to log pointers. When a read request is issued, the in-memory index is searched for the appropriate log pointer, and when it is found the value is retrieved from the on-disk log. In our key/value store, like in bitcask, the index for the *entire database* is stored in memory.
- *index file* - The on-disk representation of the in-memory index. Without this the log would need to be completely replayed to restore the state of the in-memory index each time the database is started.

Project spec

The cargo project, `kvs`, builds a command-line key-value store client called `kvs`, which in turn calls into a library called `kvs`.

The `kvs` executable supports the following command line arguments:

- `kvs set <KEY> <VALUE>`

  Set the value of a string key to a string. Print an error and return a non-zero exit code on failure.

- `kvs get <KEY>`

Get the string value of a given string key. Print an error and return a non-zero exit code on failure.

- `kvs rm <KEY>`

  Remove a given key. Print an error and return a non-zero exit code on failure.

- `kvs -V`

  Print the version

The `kvs` library contains a type, `KvStore`, that supports the following methods:

- `KvStore::set(&mut self, key: String, value: String) -> Result<()>`

  Set the value of a string key to a string. Return an error if the value is not written successfully.

- `KvStore::get(&mut self, key: String) -> Result<Option<String>>`

  Get the string value of a string key. If the key does not exist, return `None`. Return an error if the value is not read successfully.

- `KvStore::remove(&mut self, key: String) -> Result<()>`

  Remove a given key. Return an error if the key does not exist or is not removed successfully.

When setting a key to a value, `kvs` writes the `set` command to disk in a sequential log, then stores the log pointer (file offset) of that command in the in-memory index from key to pointer. When removing a key, similarly, `kvs` writes the `rm` command in the log, then removes the key from the in-memory index. When retrieving a value for a key with the `get` command, it searches the index, and if found then loads from the log the command at the corresponding log pointer, evaluates the command and returns the result.

On startup, the commands in the log are traversed from oldest to newest, and the in-memory index rebuilt.

When the size of the uncompacted log entries reach a given threshold, `kvs` compacts it into a new log, removing redundent entries to reclaim disk space.

Note that our `kvs` project is both a stateless command-line program, and a library containing a stateful `KvStore` type: for CLI use the `KvStore` type will load the index, execute the command, then exit; for library use it will load the index, then execute multiple commands, maintaining the index state, until it is dropped.

Project setup

Continuing from your previous project, delete your previous `tests` directory and copy this project's `tests` directory into its place. Like the previous project, this project should contain a library and an executable, both named `kvs`.

You need the following dev-dependencies in your `Cargo.toml`:

```
[dev-dependencies]
assert_cmd = "0.11.0"
predicates = "1.0.0"
tempfile = "3.0.7"
walkdir = "2.2.7"
```

As with the previous project, go ahead and write enough empty or panicking definitions to make the test cases build.

Do that now.

Part 1: Error handling

In this project it will be possible for the code to fail due to I/O errors. So before we get started implementing a database we need to do one more thing that is crucial to Rust projects: decide on an error handling strategy.

Rust's error handling is powerful, but involves a lot of boilerplate to use correctly. For this project the [`failure`] crate will provide the tools to easily handle errors of all kinds.

The failure guide describes [several] error handling patterns.

Pick one of those strategies and, in your library, either define your own error type or import `failure`s `Error`. This is the error type you will use in all of your `Result`s, converting error types from other crates to your own with the `?` operator.

After that, define a type alias for `Result` that includes your concrete error type, so that you don't need to type `Result<T, YourErrorType>` everywhere, but can simply type `Result<T>`. This is a common Rust pattern.

Finally, import those types into your executable with `use` statements, and change `main`s function signature to return `Result<()>`. All functions in your library that may fail will pass these `Result`s back down the stack all the way to `main`, and then to the Rust runtime, which will print an error.

Run `cargo check` to look for compiler errors, then fix them. For now it's ok to end `main` with `panic!()` to make the project build.

Set up your error handling strategy before continuing.

As with the previous project, you'll want to create placeholder data structures and methods so that the tests compile. Now that you have defined an error type this should be straightforward. Add panics anywhere necessary to get the test suite to compile (`cargo test --no-run`).

<!--

Aside: The history of Rust error handling

-->

Note: Error-handling practices in Rust are still evolving. This course currently uses the [`failure`] crate to make defining error types easier. While `failure` has a good design, it's use is arguably not a best practice. It may not continue to be viewed favorably by Rust experts. Future iterations of the course will likely not use `failure`. In the meantime, it is fine, and presents an opportunity to learn more of the history and nuance of Rust error handling.

<!-- Rust error handling has a long and winding history. Expert Rust programmers will be aware of it, as that history informs and explains modern Rust error handling.

TODO -->

Part 2: How the log behaves

Now we are finally going to begin implementing the beginnings of a real database by reading and writing from disk. You will use [`serde`] to serialize the "set" and "rm" commands to a string, and the standard file I/O APIs to write it to disk.

This is the basic behavior of `kvs` with a log:

- "set"
  - The user invokes `kvs set mykey myvalue`
  - `kvs` creates a value representing the "set" command, containing its key and value
  - It then serializes that command to a `String`
  - It then appends the serialized command to a file containing the log
  - If that succeeds, it exits silently with error code 0
  - If it fails, it exits by printing the error and returning a non-zero error code
- "get"
  - The user invokes `kvs get mykey`
  - `kvs` reads the entire log, one command at a time, recording the affected key and file offset of the command to an in-memory *key -> log pointer* map
  - It then checks the map for the log pointer
  - If it fails, it prints "Key not found", and exits with exit code 0
  - If it succeeds
    - It deserializes the command to get the last recorded value of the key
    - It prints the value to stdout and exits with exit code 0
- "rm"
  - The user invokes `kvs rm mykey`
  - Same as the "get" command, `kvs` reads the entire log to build the in-memory index
  - It then checks the map if the given key exists
  - If the key does not exist, it prints "Key not found", and exits with a non-zero error code
  - If it succeeds
    - It creates a value representing the "rm" command, containing its key
    - It then appends the serialized command to the log
    - If that succeeds, it exits silently with error code 0

The log is a record of the transactions committed to the database. By "replaying" the records in the log on startup we reconstruct the previous state of the database.

In this iteration you may store the value of the keys directly in memory (and thus never read from the log after initial startup and log replay). In a future iteration you will store only "log pointers" (file offsets) into the log.

Part 3: Writing to the log

You will start by implementing the "set" flow. There are a number of steps here. Most of them are straightforward to implement and you can verify you've done so by running the appropriate `cli_*` test cases.

`serde` is a large library, with many options, and supporting many serialization formats. Basic serialization and deserialization only requires annotating your data structure correctly, and calling a function to write it either to a `String` or a stream implementing `Write`.

You need to pick a serialization format. Think about the properties you want in your serialization format — do you want to prioritize performance? Do you want to be able to read the content of the log in plain text? It's your choice, but maybe you should include a comment in the code explaining it.

Other things to consider include: where is the system performing buffering and where do you need buffering? What is the impact of buffering on subsequent reads? When should you open and close file handles? For each command? For the lifetime of the `KvStore`?

Some of the APIs you will call may fail, and return a `Result` of some error type. Make sure that your calling functions return a `Result` of *your own* error type, and that you convert between the two with `?`.

It is similar to implement the "rm" command, but you should additionally check if the key exists before writing the command to the log. As we have two different commands that must be distinguished, you may use variants of a single enum type to represent each command. `serde` just works perfectly with enums.

You may implement the "set" and "rm" commands now, focusing on the `set` / `rm` test cases, or you can proceed to the next section to read about the "get" command. It may help to keep both in mind, or to implement them both simultaneously. It is your choice.

Part 4: Reading from the log

Now it's time to implement "get". In this part, you don't need to store log pointers in the index, we will leave the work to the next part. Instead, just read each command in the log on startup, executing them to save every key and value in the memory. Then read from the memory.

Should you read all records in the log into memory at once and then replay them into your map type; or should you read them one at a time while replaying them into your map? Should you read into a buffer before deserializing or deserialize from a file stream? Think about the memory usage of your approach. Think about the way reading from I/O streams interacts with the kernel.

Remember that "get" may not find a value and that case has to be handled specially. Here, our API returns `None` and our command line client prints a particular message and exits with a zero exit code.

There's one complication to reading the log, and you may have already considered it while writing the "set" code: how do you distinguish between each record in the log? That is, how do you know when to stop reading one record, and start reading the next? Do you even need to? Maybe serde will deserialize a record directly from an I/O stream and stop reading when it's done, leaving the file cursor in the correct place to read subsequent records. Maybe serde will report an error when it sees two records back-to-back. Maybe you need to insert additional information to distinguish the length of each record. Maybe not.

Implement "get" now.

Part 5: Storing log pointers in the index

At this point most, if not all (besides the compaction test), other test suite should all pass. The changes introduced in the next steps are simple optimizations, necessary for fast performance and reduced storage. As you implement them, pay attention to what exactly they are optimizing for.

As we've described, the database you are building maintains an in-memory index of all keys in the database. That index maps from string keys to log pointers, not the values themselves.

This change introduces the need to perform reads from the log at arbitrary offsets. Consider how that might impact the way you manage file handles.

If, in the previous steps, you elected to store the string values directly in memory, now is the time to update your code to store log pointers instead, loading from disk on demand.

Part 6: Stateless vs. stateful `KvStore`

Remember that our project is both a library and a command-line program. They have sligtly different requirements: the `kvs` CLI commits a single change to disk, then exits (it is stateless); the `KvStore` type commits changes to disk, then stays resident in memory to service future queries (it is stateful).

Is your `KvStore` stateful or stateless?

Make your `KvStore` retain the index in memory so it doesn't need to re-evaluate it for every call to `get`.

Part 7: Compacting the log

At this point the database works just fine, but the log grows indefinitely. That is appropriate for some databases, but not the one we're building — we want to minimize disk usage as much as we can.

So the final step in creating your database is to compact the log. Consider that as the log grows that multiple entries may set the value of a given key. Consider also that only the most recent command that modified a given key has any effect on the current value of that key:

idx	command
0	~Command::Set("key-1", "value-1a")~
20	Command::Set("key-2", "value-2")
	...
100	Command::Set("key-1", "value-1b")

In this example obviously the command at index 0 is redundant, so it doesn't need to be stored. Log compaction then is about rebuilding the log to remove redundancy:

idx	command
0	Command::Set("key-2", "value-2")
	...
99	Command::Set("key-1", "value-1b")

Here's the basic algorithm you will use:

How you re-build the log is up to you. Consider questions like: what is the naive solution? How much memory do you need? What is the minimum amount of copying necessary to compact the log? Can the compaction be done in-place? How do you maintain data-integrity if compaction fails?

So far we've been refering to "the log", but in actuallity it is common for a database to store many logs, in different files. You may find it easier to compact the log if you split your log across files.

Implement log compaction for your database.

Congratulations! You have written a fully-functional database.

If you are curious, now is a good time to start comparing the performance of your key/value store to others, like [sled], [bitcask], [badger], or [RocksDB]. You might enjoy investigating their architectures, thinking about how theirs compare to yours, and how architecture affects performance. The next few projects will give you opportunities to optimize.

Nice coding, friend. Enjoy a nice break.

<!--

- Potential readings
  - error handling https://github.com/rust-lang-nursery/rust-cookbook/issues/502#issue-387418261

TODOs

- should flushing the log be part of the main project or an extension?

- check terminology
  - what's the correct term for the in-memory representation of the executed log?
- is there a term for converting a log to it's permanent format?
- custom main error handling
- limits on k/v size?
- maintaining data integrity on failure
  - *must* call flush at least
- todo: `Result<Option<String>>` vs `Result<String>`
  - is "not found" exit code 0 or !0?
- error context
- serialize directly to file stream
- need readings for WAL
- add code samples and digarams to illustrate the text and be less monotonous
- maintain the index file!
- specify where data should be stored
- caching the index

-->