A Key-Value store for OCaml

Tom Ridge *University of Leicester*

This proposal describes a presentation to be given at the OCaml'19 workshop. The presentation will cover a new OCaml persistent-on-disk key-value store library. The library uses a B-tree library introduced at OCaml'17. The B-tree code has been extensively reworked to make it fast, and to support both copy-on-write and mutate-in-place operations. On top of this core, we include a write-ahead log, session initialization and termination, and extensive LRU-style caching. The API supports various "sync" operations, analogous to those found in filesystems.

1 Introduction

Key-value stores have become very popular in the last decade. Notable examples include dbm, Redis[1], BDB[2], LMDB[3] and many others (the Wikipedia page[4] currently lists over 40).

At the OCaml'17 workshop, I introduced a Copy-on-Write (CoW) B-tree library for OCaml[5]. In the intervening time, this library has been progressively refined and extended. It now supports both CoW and mutate-in-place (which turns out to be necessary to compete with existing implementations such as LMDB), as well as efficient in-memory datastructures for (in-memory) nodes and leaves. However, a raw B-tree interface is not what programmers usually want to work with. Instead, they would like to use a key-value store interface.

In addition to the usual find-insert-delete map interface, a key-value store should provide:

- synchronization operations such as the `fsync`-like "sync this key (and associated value) to disk", "sync this set of keys to disk" and "sync the entire store to disk";
- good performance (e.g. through extensive in-memory caching)[6];
- concurrency control; and
- session initialization (initializing a store from file or block device) and termination.

2 System architecture

A simple in-memory cache linked to an on-disk B-tree gives reasonable performance. However, for even better performance, a "write-ahead" log is needed. The log allows operations to be made persistent-on-disk without the overhead of modifying the B-tree. Periodically entries in the log are asynchronously flushed to the on-disk B-tree.

Following this approach, our system is composed of four components (see Figure 1):

1. The syncable key-value map interface, backed by a Least Recently Used (LRU) in-memory cache.
2. The write-ahead log (here also called a "detachable log").
3. The B-tree itself.
4. The "root manager" which takes care of tracking the B-tree root block and other session information.

3 Detachable log

Although we have used the term "write-ahead log" (WAL) to link our log conceptually with the logs employed by existing databases and key-value stores, our design differs considerably from a traditional WAL and so we prefer the term "detachable log".

The detachable log is a persistent-on-disk log that caches updates to the B-tree[7]. New updates go on the

[1] `https://redis.io/`

[2] `https://www.oracle.com/database/berkeley-db/db.html`

[3] `https://symas.com/lmdb/`

[4] `https://en.wikipedia.org/wiki/Key-value_database`

[5] Slides at `http://www.tom-ridge.com/resources/ocaml_2017_slides.pdf`

[6] Caching parameters should be adjustable to limit (for example) the maximum amount of memory used.

[7] Of course, there is also an in-memory cache of the log.

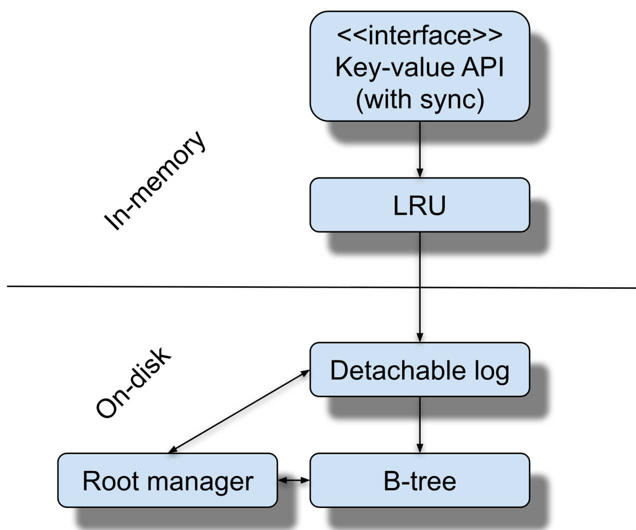

Figure 1: *System architecture*

end of the log. Periodically, the updates that form some prefix of the log are applied to the B-tree. This happens asynchronously so that the log remains live, and can still have updates added at the end.

The entire persistent state consists of the B-tree state (identified via a "root" block on disk), and the log state (again, identified by a root block on disk).

When the prefix updates have executed against the B-tree, the prefix is dropped (detached) from the log. This is done using an atomic disk operation that records the new root block for the log, together with the new B-tree root block.

With this approach, synchronization calls need only persist operations to the on-disk log. Reads from the log are served from the in-memory cache of the log, or (if the relevant entry is not present) from the on-disk B-tree.

4 Concurrency

We support fine-grained concurrent operations at the user-facing syncable-map interface[8]. This necessarily involves some form of concurrency control.

Ideally threads should, as far as possible, operate independently so that one thread does not delay another. For example, we want to avoid the situation where one thread executes a sync operation which delays other threads. Instead, we want threads that execute operations on different subsets of the key space to avoid delaying each other as far as possible.

For a single thread, we also want the system to operate with minimal per-thread delay.

Since all operations may involve the block layer, and hence may take some time, we introduce three active threads to manage different components: one thread manages the LRU cache; one manages updates to the log; and one manages both updates to the B-tree, and the recording of new roots (for the B-tree and the log).

Internally, user API calls are converted to messages passed between the three threads. Messages may themselves contain callbacks which are invoked when lengthy operations, such as disk accesses, complete.

The usual concurrency issues are also present. For example, if one thread accesses a key, necessitating a lengthy call to disk, we need to ensure that another thread that updates that key with a more recent value does not have that value overwritten when the long-running disk operation returns.

5 Rough performance measurements

To give a rough idea of performance, our library can currently insert an ordered (by key) sequence of 10^7 key-value pairs into an empty store in about 20 seconds, and 10^8 pairs in about 190 s (indicating that the system scales smoothly to huge numbers of entries). Random write performance is of the order of 10^5 writes/s on an old SSD, and (as expected given the B-tree technology) this number decreases only very slowly as the store grows.

Using mutate-in-place (except where consistency requires copying... typically when B-tree leaves are split or merged), inserting 10^8 entries results in 3.4GB of space being used, which is roughly a factor of two overhead compared to storing the entries directly. The overhead consists of a small number of blocks for the B-tree index (the non-leaf nodes), and further garbage blocks introduced during the insertion that are no-longer needed and should be reclaimed[9].

6 Workshop presentation

The plan for the workshop talk is as follows:

- I will start with a brief overview of key-value stores and the key-value store interface;
- then briefly recall the concept of a B-tree and the previous work, whilst motivating the differences with this new library;
- then discuss the architecture of the library itself, including the main interfaces and the features that the architecture supports;
- then conclude with a performance comparison with similar key-value stores that support on-disk persistence and good performance (in particular, LMDB).

[8]Currently we support Lwt threads (see `http://ocsigen.org/lwt/4.1.0/manual/manual`), but our code is generic and should be usable with other concurrency libraries.

[9]We do not yet implement garbage collection at the block layer. However, the detachable log batches and merges updates to the same key, and so many updates avoid hitting the B-tree altogether.

The code is on GitHub[10]. It is currently being refactored, prior to initial release, to make use of Jane Street Core_kernel libraries.

[10] https://github.com/tomjridge/tjr_btree and https://github.com/tomjridge/tjr_kv