A B-tree library for OCaml

Tom Ridge *University of Leicester*

This proposal describes a presentation to be given at the OCaml 2017 workshop. The presentation will cover B-trees as a data-structure (including motivating why variants of B-trees are often the data-structure of choice for filesystems and databases), and the particular variant that is provided by the `tjr_btree` library available from GitHub[1]*. This library uses code generated from an Isabelle/HOL theory, which is also available from GitHub*[2].

1 Introduction

In 1979, Comer authored a paper titled "The Ubiquitous B-tree" (Comer, 1979). Almost 40 years later, B-trees are still a topic of interest and research: 10 years ago Rodeh authored a paper "B-trees, shadowing and clones" (Rodeh, 2007) which inspired the creation of BtrFS, a new filesystem for Linux. BtrFS is currently under active development, and looks likely to become the default Linux filesystem in the near future.

This presentation will cover the basics of B-trees, the particular design that we have chosen, the core definitions in Isabelle/HOL (whose syntax will be familiar to OCaml users) and the OCaml library itself. We will conclude with an example application: a continuously snapshotting, network-replicated block device. This library is part of the "ImpFS" project which aims to build a formally-verified, modern filesystem with BtrFS-like features.

2 B-trees

The B-tree data-structure is particularly suited for implementing key-value maps on block-based storage, such as that provided by hard drives and SSDs. In-memory B-trees can also perform well, even outperforming data-structures such as Red-Black trees, due to their favourable interaction with page-based caches. Keys and values must be "small" compared to the block size so that multiple key-value pairs can fit into each block, but lifting this restriction is trivial for values (the value becomes a pointer to data in other blocks). For large keys, one can often use relatively small representations of the key rather than the key itself. For example, for keys which are arbitrarily long strings, one can use the relatively small hash of the string as the key, rather than the string itself.

B-trees come in many variations, and the nomenclature is not standardized. We develop a variant which is closest to Rodeh's B-trees.

A B-tree is a balanced tree where leaf nodes store lists of key-value pairs, and each internal node stores a (variable) number of keys $k_0 \ldots k_{n-1}$ in increasing order (and no values), and has children $c_0 \ldots c_n$. The value of n varies for each node, but there are typically global minimum and maximum bounds $min_n \leq n \leq max_n$ for the entire tree. The key-value pairs in the leaves correspond to the entries in the map[3]. Given a search key k, the primary purpose of the internal keys is to guide search to the leaf that possibly contains k. If $map(t)$ denotes the map corresponding to the B-tree t, and $kvs(t)$ denotes this key-value map considered as a set of (k, v) pairs, then the B-tree guarantees this important property: Given an internal node $t = (k_0 \ldots, c_0 \ldots)$ and a pair $(k, v) \in kvs(t)$, then $k_i \leq k < k_{i+1}$ iff (k, v) occurs in c_{i+1}.

This property holds for *any* node in the tree. Given a particular key k, one can locate the pair (k, v) by starting from the root node, and repeatedly descending to a child of the node (chosen as above), eventually reaching a leaf[4].

In practice, nodes in a B-tree are recorded in blocks on disk. Typical block sizes are 1024 bytes, and 4096 bytes. Children of a node are represented by pointers

[1] `github.com/tomjridge/tjr_btree`

[2] `github.com/tomjridge/isa_btree`

[3] The list of key-value pairs contains a key k at most once.

[4] We omit technical complications that arise for left-most and right-most branches, where lower and upper bounds k_i, k_{i+1} may not exist.

(typically 32 or 64 bit ints). Assuming keys are 8 bytes and child pointers are 8 bytes, a 4096-byte block can store around 256 keys together with the associated child pointers. Thus, B-trees have large fan-out. It is possible to store over 16 million key-value pairs in a B-tree with a root, one layer of internal nodes, and one layer of leaves. If all blocks are on disk, 3 block reads suffice to locate a particular key-value. Thus, B-trees require relatively few disk reads, and moreover have very low memory overhead, since only one block need be held in memory at any point when searching for a key. Usually the root and first few layers of internal nodes are cached in memory. In our example, caching the root block and the single layer of internal nodes involves 257 blocks, or approximately 1MB of memory. Then any key-value pair can be retrieved with at most one block lookup, and with a memory overhead of about 1MB.

B-trees also excel when data is updated. For example, deleting a key-value pair in a B-tree can be as simple as updating the block corresponding to the relevant leaf. For copy-on-write B-trees, a new version of the tree can share almost all blocks with the previous version, so that features such as "persistent on-disk snapshots" execute almost instantaneously.

3 Mechanization in Isabelle

Although B-trees are well-known and have been around for a long time, most presentations are relatively informal. The standard algorithms text by Cormen et al. (Cormen et al., 2009) is a typical example. It includes imperative code for the standard map operations *find*(k) (reasonably simple) and *insert*(k, v) (significantly more complicated, involving splitting and rebalancing), but code for *delete*(k) (which is harder still, involving coalescing and rebalancing) is not presented. The paper by Sexton and Thielecke (Sexton and Thielecke, 2008) attempts to formalize the (considerable) details, however, this work was not machine-checked, and contains various errors as an almost inevitable consequence. They also do not treat concurrency.

Our definitions are formalized in Isabelle/HOL. Our formalization is in small-step state-passing style, with each disk access modelled as a separate step, which supports reasoning about the interleaving when multiple processes access the disk concurrently. In addition to the definitions of the B-tree operations, we also formalize the correctness conditions, taking care to ensure that they are executable as code.

Isabelle code extraction is used to produce OCaml definitions which are then lightly patched to remove various unfortunate artefacts (e.g., the dictionary-passing implementation of equality, which is not needed for the B-tree operations, and can be replaced with OCaml's built-in equality for the correctness conditions).

4 OCaml library

The OCaml library builds on the Isabelle definitions to provide OCaml-friendly interfaces. For example, we expose the B-tree functionality via standard map operations, whereas the Isabelle definitions use an explicit state-passing style. The OCaml library is heavily parameterized[5]. For example, the B-tree map operations are parameterized by key type $'k$, values $'v$, block pointers $'r$ and the global state $'t$. Further parameterization covers the block device, block size, and on-disk persistence/marshalling strategy. The library also includes some examples, such as an on-disk key-value map from strings to strings.

The main features of the library are:

- provides a copy-on-write, persistent (on-disk) B-tree, enabling fast snapshot of the whole map state, and instant copy-on-write access to any snapshot;
- provides a generic LRU cache which can be used at various levels (above the disk and above the store, and above the map in order to batch high-level map operations);
- the code is purely functional, with mutable state (and errors) modelled via a state-passing monad;
- the library has preliminary support for "write omission", when it can be determined that cached writes would not be reachable from an on-disk synced state (this helps ameliorate the cost of copy-on-write B-tree operations);
- support for an *insert_many* operation, to insert multiple key-value pairs at once;
- support for a *bindings*[6] operation, to return a stream of all the entries in the map; importantly, regardless of the number of entries, this requires only constant memory overhead, since (chunks of) entries are loaded as needed from disk (if they are not already cached);
- extensive parameterization, including: types (as above); minimum and maximum sizes for nodes and leaves; on-disk layout; and pickling/marshalling functions;
- "B-tree kit" functionality is supported: many maps can use the same disk, or a single map can be spread over multiple disks; caches can be shared or not, and layers can be cached or not, and so on.

The library has been extensively tested, including exhaustive state-space exploration for small parameter values, however, the formal Isabelle proof remains future work.

[5]We have used type variables, and records of functions, rather than functor parameterization.

[6]Called *ls_kvs* in the library.

5 Example application: virtual block-device

As well as standard applications, such as a key-value store, we have developed a continuously snapshotting, network-replicated, virtual block-device. We use the B-tree to map keys that are block ids (natural numbers) to values which are blocks. With this we can use the B-tree to expose a "virtual" block device via the loop-back device, ultimately backed by a "normal" file. We then use the virtual block device to host a traditional filesystem such as ext4. Ext4 updates to blocks result in new versions of the B-tree data stored in the backing file. Each version of the B-tree can be accessed independently, providing a continuously snapshotted block device. The backing file, which stores all data for all versions of the block device, is append-only and so can be synced trivially over the network. This approach can be made reasonably fast, and for typical developer workloads, caching can limit the space overhead (at the cost of providing less frequent snapshots). All this can be accomplished with a very small amount of code (less than 50 lines) above the B-tree library, which provides evidence that the library is flexible enough to support many other applications. Indeed, we plan to use the library as the basis for "ImpFS", a modern, verified filesystem.

Bibliography

Comer, Douglas (1979). "The Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2, pp. 121–137. DOI: `10.1145/356770.356776`.

Cormen, Thomas H. et al. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press. ISBN: 978-0-262-03384-8. URL: `http://mitpress.mit.edu/books/introduction-algorithms`.

Rodeh, Ohad (2007). "B-trees, Shadowing, and Clones". In: *2007 Linux Storage & Filesystem Workshop, LSF 2007, San Jose, CA, USA, February 12-13, 2007*. Ed. by Ric Wheeler. USENIX Association. URL: `https://www.usenix.org/conference/2007-linux-storage-filesystem-workshop/b-trees-shadowing-and-clones`.

Sexton, Alan P. and Hayo Thielecke (2008). "Reasoning about B+ Trees with Operational Semantics and Separation Logic". In: *Electr. Notes Theor. Comput. Sci.* 218, pp. 355–369. DOI: `10.1016/j.entcs.2008.10.021`.