École Normale Supérieure d'Ulm

Internship report

Designing and implementing a new index data structure for the Irmin backend

Gabriel Belouze

under the direction of
Ioana Cristescu, Craig Ferguson, Clément Pascutto

Tarides
75005, Paris

February-July 2021

Contents

This document is both a report of my internship at Tarides, and a guide to use or follow up my work there. As such, it contains both theoretical discussions and low level, implementation-directed information.

Acknowledgements

1 Introduction

`irmin-pack` is a storage backend written in OCaml, developed for and used by the Tezos blockchain. `irmin-pack` maintains a component for pure data storage, called the `pack`, and an index component containing metadata to access content in the pack, logically called the `index`. The index maps hashes of content to location information in the pack ; in short it is no more than a key value data structure.
The current implementation of `index` provides fast look-ups but quadratic insertions. The goal of the internship was to design and implement a more scalable data structure, fine tuned to the `irmin-pack` use case. In particular, it would need to provide data integrity and partial support for concurrency, while solving the performance issues.

We considered two potential data structures, PGM-index and B-trees. Many well-known database systems are B-tree based (MySQL, InnoDB, SQLite), and B-trees were our ultimate choice. We implemented `cactus`, a disk-resident alternative to `index` using B-trees as its underlying datastructure, and successfully integrated it to `irmin-pack`. Benchmarks showed promising results, gaining several orders of magnitude on insertions over `index`, and maintaining slightly worse lookups. Concurrency support was designed but remains to be implemented.

We first present in section 2 an overview of the theoretical works that laid the ground for our design. In sections 3 and 4, we give a more in-depth description of the core btree algorithms, then discuss our implementation – its design and how it performs. Those two sections are closely related: it is necessary to be familiar with btree concepts to understand the rational for our design, but the precise algorithms also depend on structural design choices. Finally, we discuss in section 5 perspectives for future improvements on the project.

2 State of the art

2.1 The external memory model

The external memory (EM) model is a model of computation most relevant to evaluate algorithms that work with disk-resident data. When data is too big to fit into main memory and is stored on a disk drive, I/O is typically the bottleneck as its latency is several order of magnitude greater than any other elementary operation. To read data at address *addr*, a hard drive first moves its magnetic head to the correct location, then retrieve a *page* of contiguous data. Typically, a page is a few kilobytes.
The external memory model capture a two-level memory hierarchy, where the CPU is connected to a fast cache of bounded size M, itself connected to an unbounded, *slow*, disk. The CPU can transfer blocks of B bytes of data to and from the disk (I/O calls). The complexity of an algorithm is then evaluated with regards to its I/O cost.

Binary Search Trees in the EM model To understand the need for EM-specific algorithms, it is useful to understand where a classical data structure such as a BST falls short. BST's nodes contain very few information – only a single binding and two pointers. Because BSTs are constantly rebalanced, there is no concept of locality: a node may very well be stored on a different block than its parent. This means that inspecting a node is highly unoptimal, fetching

B bytes of data and only reading a few. To look up a value in a BST with n bindings, $\log_2(n)$ I/O calls may be necessary (a whole $\log_2$!). We can improve the base of the logarithm considerably by storing B bytes of actual data inside each node (subsection 2.3) .

2.2 PGM-index

The PGM-index ([3]) is a learnt index with provable worst-case bounds similar to canonical indexes. It recursively builds a tree-like data structure on top of the data, to find and exploit local regularities. Compared to other traditional indexes, it considerably reduces the space overhead, while maintaining queries and updates fast in theory, and very fast in practice.

The rational behind the PGM-index can be derived from interpolation search. Interpolation search improves upon binary search when the values are uniformy distributed. Indeed in that case, rather than inspecting blindly the middle slot, one can predict a linear model for the function $rank$ from the extreme values, i.e. $rank(val) = a \cdot val + b$, and look up the $rank(v)$-th slot. This provides an almost constant $O(\log \log n)$ query complexity. Of course in general, one cannot assume uniform distributions, but this bound makes it desirable to find local linearities in the data, where interpolatoin search can be used.

Building on this, the PGM-index predicts the $rank$ function to be piecewise linear. More precisely, it sets a maximal error ε (usually $\varepsilon = 2^k$ for some small integer k), and optimally splits the data into contiguous subarrays, where linear interpolation approximates $rank$ up to ε. Thus a new array is built on top of the data, where each element describes one subarray, i.e. its linear interpolation and the range of values it covers. To look up a key k, we first find the sub-array which range k falls into. This gives a linear interpolating function ϕ, and we can predict the rank of k to be approximately $\phi(k)$. Thus we can finally find k with a binary search between $\phi(k) - \varepsilon$ and $\phi(k) + \varepsilon$. Still, how can we efficiently find the correct sub-array ? Recursively, of course. That is, we build more levels of sub-arrays on top of the one we constructed, until we get an array of length one. Look ups can proceed from one level to the one below with the strategy described above.

The PGM-index suffers from a few limitations which made it unsuitable as an `index` replacement. Mainly, the issue lies in the fact that PGM-index assumes given the sorted array of values (without which the rank function is irrelevant). When the data does not fit in memory, maintining such array becomes a challenge. This is the very task that `index` tackles - and does with quadratic (read problematic) complexity. In fact, `index` uses hashes as keys, which *are* uniformly distributed ; there is no improvement to gain from finding local regularities.
Note that the PGM-index does not find the same pitfall as binary trees. Indeed, even though a tree traversal (e.g. during a lookup) must access the array of values several times, all tree levels but the last one are typically small enough to be cached. It appears that the PGM-index would be an extremely well suited data structure when the keys are by nature monotonous (for instance, timestamps), as insertion becomes trivial.

2.3 B-trees

Introduced in 1971 by Bayer and McCreight, and described as "ubiquitous" less than 10 years laters ([2]), B-trees still are the canonical data structure for indexing. They provide $O(\log(n))$ lookups, insertions and removal, support several techniques to provide data integrity and concurrency access, and are backed-up by a wide use in the industry (see [4, 5] for an in-depth survey on btree techniques). Importantly, the complexity for btree operations in the EM-model is $O(\log_B(n))$ where B is the size of a disk page (typically, 4096 octets). They were the retained data structure for `index` .

Just like binary trees, B-trees are search trees, where keys in interior nodes act as separators between subtrees. Unlike binary trees, the number of such separators can vary and is much greater than 2. More precisely, btrees are search trees that follow the three invariants: *sortedness*, *density*, *balance*. B-trees are sorted because any key from a left subtree is smaller than any key from a right subtree. They are dense because every node contains at least f keys and at most $2f$ keys, where f is a hyperparameter called the tree **fanout**. Finally, they are balanced because all root-to-leaf paths have the same length.
Typically, we want a node of a btree to fill exactly one disk page, and we choose accordingly a fanout $f = \Theta(B)$. It follows that the height of the tree is always $O(\log_B(n))$, which consequently means that LOOKUP is $O(\log_B(n))$ – which is optimal in the EM model. As we will see later (section 3), this bound also holds for INSERT and DELETE.

3 Detailed B-tree algorithms

In this section, we detail the core algorithms of our btrees: LOOKUP, INSERT, REMOVE, BATCH-INIT, RECOVER. Even though LOOKUP, INSERT and REMOVE are well-known in the literature, their exact specifications depend on some design choices about the structure of btrees and their representation. The reader may want to refer to subsubsection 4.2.1 (and in particular 4.2.1, §Localisation of values) to fully appreciate this section. This is even more true of the last two algorithms BATCH-INIT and RECOVER which were developed *ad hoc* (see respectively section 4.3.2, §Batch initialisation and subsection 4.4).

Notations We let f be the fanout of the btree, i.e. a set hyperparameter. Btrees are composed *vertices*, we can be either leaves or nodes (sometimes refered to as *interior nodes*). We call *canopy* the non-leaf part of the btree.

Reminders Three B-tree invariants drive the design of the subsequent algorithms.

`Sortedness` In a node N, if the key k separates the left subtree T_1 from the right subtree T_2, then $\forall k_1 \in keys(T_1)$, $\forall k_2 \in keys(T_2)$, $k_1 \leq k < k_2$. Moreover, in any vertices, the keys are sorted.

`Density` In a vertex V, $f \leq |keys(V)| \leq 2f$. The root is an exception and may contain less than f keys.

`Balance` There exists h such that the (shortest) path from the root to any leaf uses exactly h edges. h is called the height of the tree.

Premilinary results Note that `density` and `balance` together ensure that $h = \theta(\log_f(n))$.

3.1 Lookup

LOOKUP(k) is straightforward. It starts at the root, and recursively descends the tree to the only leaf L that would be allowed to contain k according to `sortedness`. Then it is simply a matter of looking up among the keys in L, which a classical binary search does well (keys in L are sorted).
To descend from a node N containing separators $k_1 < \ldots < k_p$, LOOKUP finds the greatest i such that $k_i \leq k < k_{i+1}$. Again, a binary search can achieve this efficiently.

Performance Each node inspection may require a single I/O call, and there are no more such inspections that the height of the tree. Hence, the algorithm achieves a $O(\log_f(n))$ complexity in the EM model. The number of comparisons is $O(\log_2(f)\log_f(n)) = O(\log_2(n))$.

3.2 Insertion

The insertions of key k can be described in 3 steps.

`Descent` Find the only leaf L where k is allowed to go as per `sortedness`.

`Insertion` Add k to L in its correct slot so as to leave $keys(L)$ sorted.

`Rebalance` If necessary, operate the necessary balancing operations to maintain `density`

The `descent` step is similar to that of LOOKUP. At step `rebalance`, two situations may arise.

No split If L still has less than $2f$ keys, there is nothing to do (Figure 1).

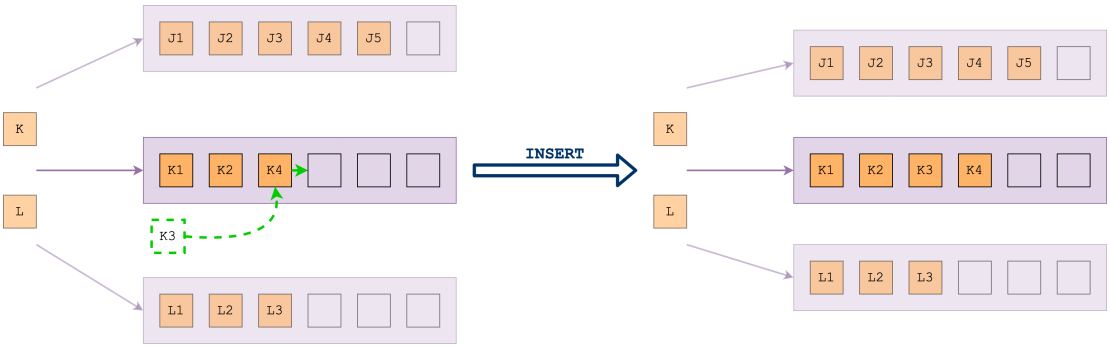


Figure 1: No split is needed.

Split If L was already maxed out at $2f$ keys before the insertion, it **overflows**. A new leaf L' is created and half the bindings from L are moved there, leaving L with $f+1$ bindings and L' with f bindings – this is called a `split`. A separator must be inserted in the parent node to separate L and L'. This of course is done recursively, by following steps `insertion` and `rebalance` (Figure 2).

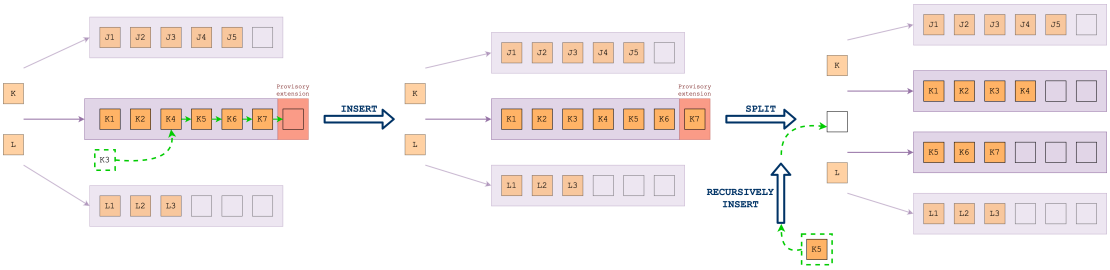


Figure 2: Leaf L overflows and splits into 2 leaves. A separator is added in the parent node.

Updating the separators In the case of a split, we need a strategy to choose a separator for L and L'. First, we can notice that all other separators can be left unchanged without breaking `sortedness`, so we indeed should only be concerned with the new separator. What key to use ? There is always a valid separator between two neighbouring subtrees: the lowest key of the higher subtree. In the drawing for instance, $K5$ can be chosen as a valid separator.

When the root splits It may happen eventually that INSERT recursively reaches the root. If the root itself is maximally filled, then it splits into two nodes N_1 and N_2, and a new root is created with them as its only children (which is indeed allowed for a root). This is the only situation where the height of the tree may increase, as all other splits grow the tree in width.

Performance INSERT reads at most h nodes, and creates at most another h nodes. In the end, the algorithm runs, like LOOKUP, with $O(\log_f(n))$ I/O calls (external memory model).

3.3 Deletion

As for INSERT, the deletion of a key k follows three steps

`Descent` Find the only leaf L where k could as per `sortedness`.

`Deletion` Delete k from L.

`Rebalance` If necessary, operate the necessary balancing operations to maintain `density`

Again, only the third step `rebalance` needs precision. 3 situations may arise.

No merge If leaf L still maintains `density` after the deletion, there is nothing more to do.

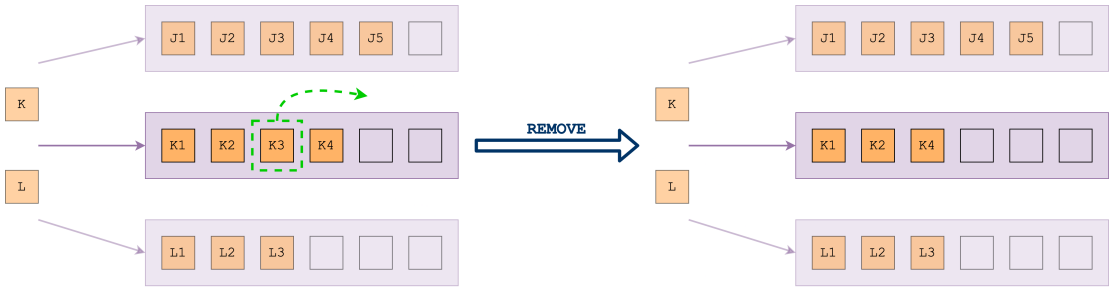
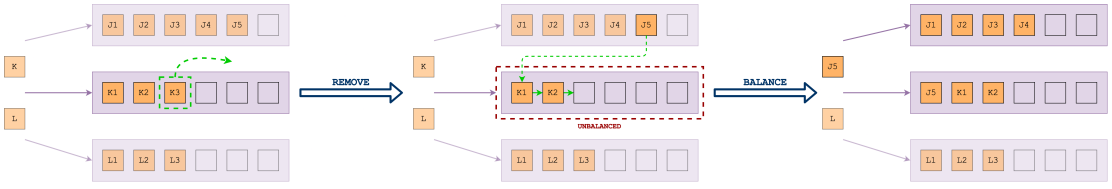
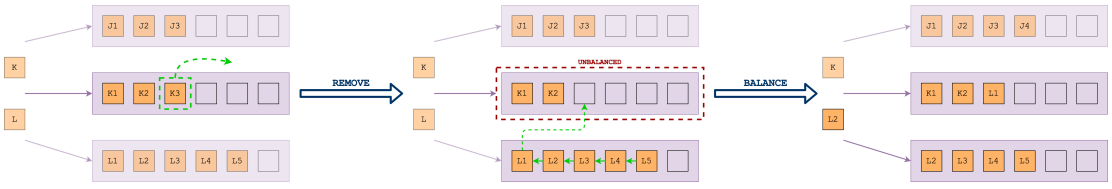


Figure 3: No merge is needed.

Partial merge If leaf L had exactly f keys at the start of the algorithm, it now **underflows**. L's neighbours are inspected. If one neighbour L' has some keys to spare, we redistribute $keys(L) \cup keys(L')$ evenly among L and L'. This is called a **partial merge**, and restores `density` but breaks `sortedness`. The separator between L and L' is thus updated to restore `sortedness`. The details of this last step vary whether L' was L's lower or upper neighbour (see Figure 4).



(a) Low partial merge.



(b) High partial merge.

Figure 4: L underflows, but there are keys to spare in L's neighbours. The low and high partial merges correspond to merging with the higher or lower neighbour.

Total merge Lastly, it may happen that L underflows while both its neighbour have exactly f keys. In this case, redistributing won't restore `density`. Instead L is **totally merged** with one of its neighbour L': the bindings from L move to L', and L is deleted. In doing so, we must also delete the separator in the parent node between L and L'. This of course is done recursively, by following steps `deletion` and `rebalance` (see Figure 5).

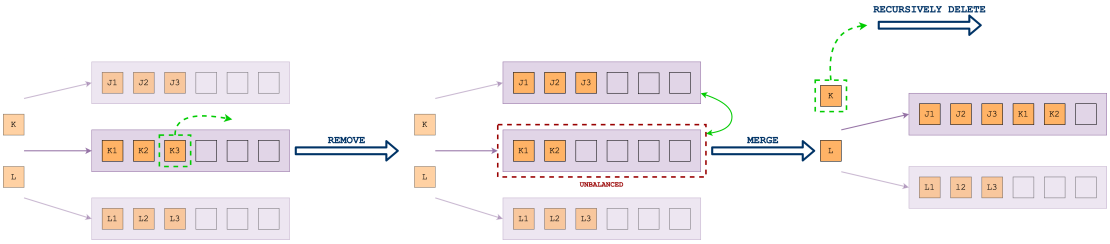


Figure 5: Total merge.

Deleting the root When the tree is shallow, it may happen than deletions recursively procede to the root. The root has no neighbour to rebalance with, but is allowed to underflow, so the no-merge case occurs. If however the root had only a single key (and hence a single child) to begin with, then it is deleted and its only child becomes the new root. This is the only time the height of the tree may decrease, as indeed other merges only decrease the width of the tree.

Performance REMOVE inspects at most 3 vertex per level of the tree. Thus it runs, as the other first two algorithms, with $O(\log_f(n))$ I/O calls (external memory model.)

3.4 Batch initialisation

Btrees are known to be relatively (although consistently) slow, notably because of the need of balancing operations. Often, we need to initialise a tree with a batch of bindings first. In our use-case, this will come especially handy to support migration from an `index` store (see also section 4.3.2). In such situation, it is much more efficient to have a dedicated batch initialisation algorithm, rather than to insert values of the batch one by one.

Our BATCH-INIT algorithm works in two phases. First, sort the batched bindings by key order, and second, recursively build the btree levels, starting from the leaves.

Sorting We cannot assume that the batch of initial values fits in memory. In fact, in the case of a migration from an `index` store, it is likely not to be the case. Thus, an external sorting algorithm must be called – we chose to use EXTERNAL-MERGE-SORT, an instance of MERGE-SORT adapted to work with disk-resident data.

Building the tree Given the array of sorted bindings, the btree can be recursively built, level by level, starting from the leaves. Simply put, the array of bindings is partitioned into groups of size f. Each group forms a vertex and produces a new binding for the next level. The new binding is the tuple composed of the group's leftmost key and of a pointer to the vertex formed by the group. When all groups have been processed, a sorted array of new bindings (f times smaller than the initial array) has been constructed. It is recursively fed to the algorithm to produce the next btree level, as shown in figure 6.

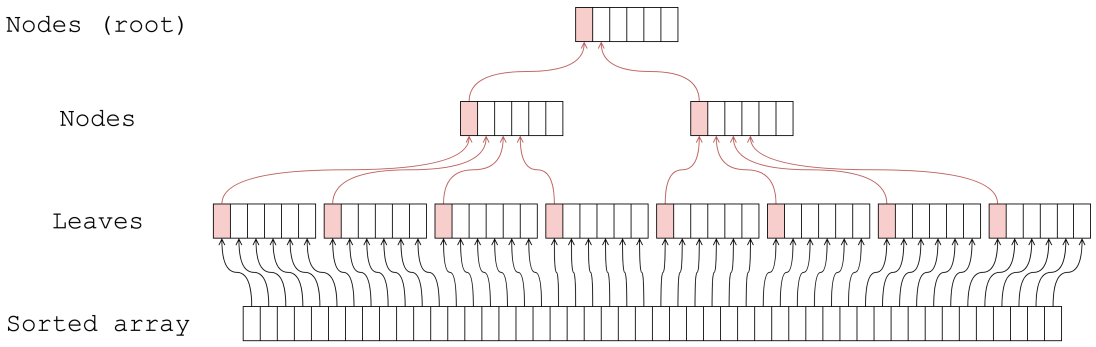


Figure 6: Schematic for the functionning of BATCH-INIT

3.5 Recovery

A close variant of BATCH-INIT is RECOVER that we developed to ensure data integrity (see subsection 4.4). The recover task asks to build the btree knowing only its leaves. This is in fact a task we can readily perform, as it is exactly the tree building step of BATCH-INIT.

4 Implementation

4.1 Specifications

The most notable use-case of `irmin` is as the backend for Tezos. As such, `cactus` inherits the same constraints as `index`, namely

`Memory` The maximum allowed memory usage is 1GB.

`Latency` Any operation should complete under 1s.

`Integrity` The system may crash at any point and `cactus` must be able to recover with minimal data loss.

`Concurrency` `cactus` must at least be able to handle concurrent access from 1 writer and 3 readers. Crucially, those instances may not share memory.

We were able to meet all constraints but concurrency. Concurrency is specifically challenging because instances cannot share a cache – yet our implementation relies on an efficient caching framework. A roadmap was designed to provide concurrent access, though not implemented (see 4.5).

Additionally, `cactus` may assume all keys and values have the same fixed size (indeed in the Tezos use-case, keys are hashes). This will come in handy for serialisation.

4.2 Design

4.2.1 Implemented variant

Localisation of values We chose to implement B^+-trees, a common variant of btrees. In B^+-trees, all bindings are stored in the leaves. Keys in interior nodes act as separators but do not hold any additional information. This design presents several convenient properties. Firstly, this makes the implementation more straightforward, as it separates nicely tree traversal from the retrieval of values. Secondly, separators need not anymore be keys that are bound in the tree. This offers opportunities for optimisations (see 5.2.1 and 5.2.2). Finally, because all of the information lives in the leaves, data integrity can only be concerned with preserving the leaves. This fits well into our caching protocol and makes ensuring data integrity almost trivial (see 4.4).

Adding a min key An internal node with n key separators $k_1, \ldots, k_n$ points to $n+1$ children $c_0, \ldots, c_n$. If $\mathcal{K}$ is the space of keys, we can notice there are two implicit bound keys $k_0 = \inf \mathcal{K}$ and $k_{n+1} = \sup \mathcal{K}$, such that any key k from child c_i verifies $k_i \leq k < k_{i+1}$. It is convenient in terms of implementation to think as internal nodes as maps from key separators to other nodes ; unfortunately there is one too many children. This was solved by explicitly storing the low bound key $k_0 = \inf \mathcal{K}$ in the node, and mapping separator k_i to child c_i.

Mutable v immutable Two kinds of mutability are at stake: the data structure's and its physical representation's. We want an interface similar to that of `index` :

```
val replace: t -> key -> value -> unit
val find: t -> key -> value
val mem: t -> key -> bool
```

Hence the data structure must be mutable. We chose to make the underlying representation on disk of the tree mutable as well. This is a trade-off: on the one hand, immutability provides easier guarantee of correction, easier concurrent access, and easier data integrity ; on the other hand, mutability decreases the write amplification, the overall number of syscalls, and allows for efficient caching (section 4.2.2, §Caching).

4.2.2 Representation on disk, representation in memory

Representation on disk We represent a btree in a `b.tree` file. A `b.tree` file is composed of a header followed by a succession of fix-sized sections called *pages*. One page can encode an internal node or a leaf. A page is composed of a header, which among other things encode if the page is a leaf or a node, followed by a succession of fix-sized sections called *bindings*. Each binding contains -at least- a key and a *bound* (understand, the object that is bound in the page). A bound is an actual value in leaves, and an address to another page in nodes.

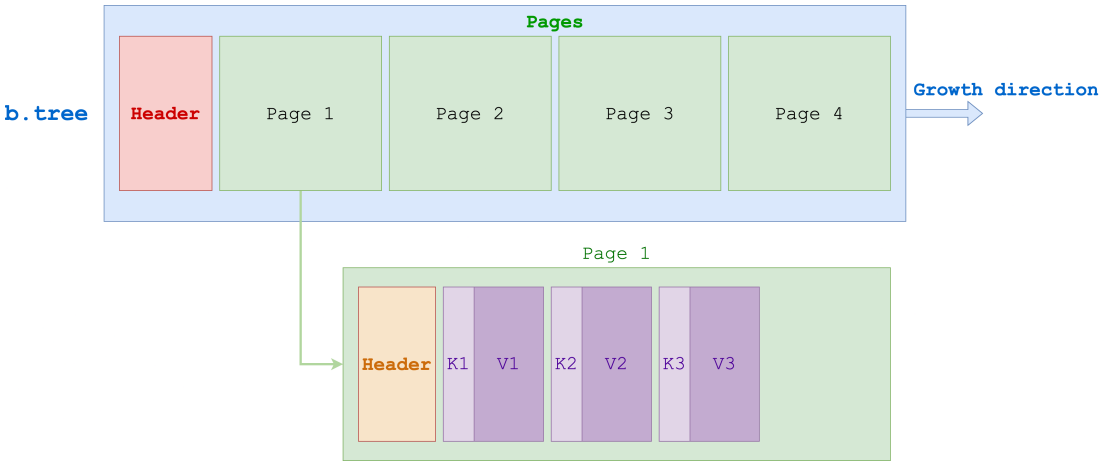


Figure 7: `b.tree` file format

To know how to decode a page, one must know whether it represents a node or a leaf. Hence it is important that node headers and leaf headers share a part which, at least, encodes the underlying kind of the page.

Represention in memory The granularity of access to `b.tree` is the scale of a page. Usually, a btree transaction only inspects a few bindings from the nodes and leaves it traverses. To avoid unecessary page deserialisation overhead, we directly manipulate raw `bytes` . Because bindings have fixed-length, this representation is equivalent to working with an array of bindings. In particular, it has constant random binding access, and thus allows for $O(\log n)$ binary search.

Caching A single btree lookup must traverse the tree from the root to the correct leaf. This yields `tree_height` access to pages, and as many syscalls if those pages are not already buffered. There is then much to gain from an efficient cache, reducing several syscalls to potentially only one.
A crucial consideration is that no vertex can be accessed without first inspecting its parent. Hence any optimal cache should be "stable by predecessor". We chose to implement a general-purpose caching protocol, that is optimal in a random access setting, i.e. when the keys bound in the btree are all equally likely to be looked up. In such cases, no branch needs to be prefered over another, and the caching hierarchy should be breadth-first.
We implemented a cache that is parametrised by a hyperparameter `cache_size`, expressing an upper bound on the in-memory size of the cache. Our cache is structured in 3 ordered levels:

`California` *you can never leave* The california level permanently buffers the first l levels of the tree, where l is the largest possible so that CALIFORNIA cannot overgrow the allowed cached size.
l is in fact directly a function over f the fanout and c the scaled cache size ($c = \frac{cache\_size}{page\_size}$ is the upper bound on the number of cached pages). The number of pages at depth h is at most $2f$ that at depth $h-1$, hence the first x levels can contain at most $\sum_{k=0}^{x-1}(2f)^k = \frac{(2f)^x-1}{2f-1}$ pages. This gives $l = \lfloor \log_{2f}(1 + c \cdot (2f-1)) \rfloor$.

`Lru` The exponential nature of a btree means that, even though the level right below CALIFORNIA cannot be fully buffered, there may be most of the allowed cache size left to use. We leverage this by caching as much of this level as allowed in an LRU cache.

`Volatile` The last cache level is dedicated to the pages in all other levels, which are only buffered through the length of one transaction, and immediately flushed away afterwards. We must only be careful not to flush a page that is still in use by a another module of the application. In practice, this means that the top level module of the application (the transaction manager) explicitly advocates when it is safe to flush out VOLATILE entirely, e.g. at the end of a transaction.

Note that the cache level a page must be loaded to is only dependent on the depth of the underlying vertex it represents.
Importantly, the exponential nature of btrees means that most of the tree can be cached in practice. For instance, if a btree holds 50GB of data, and has a fanout of 50, then only a little more than 1GB of memory are necessary to hold all but the last level of the tree in the cache.
When lookups are not uniform, but instead are skewed to, for instance, a specific key range, a different caching protocol may yield better performance. For instance, the whole cache could be fitted inside a single LRU data structure. It must only made sure that the benefits exceed the cost of maintaining the LRU ordering.

4.2.3 Leveraging the OCaml module system for safer serialisation

Manipulating raw `bytes` is prone to serialisation and deserialisation errors. Each field must be read and written at the correct offset, with the correct serialiser. The underlying value type is of course not enough to catch a serialiser being wrongly used, as several fields may hold data of the same type, but yet require different serialisers. For instance, node headers contain a field `n_entry: int` counting the number of bindings present, but also contain many `address: int`, i.e. pointers to other pages. A `n_entry` field need only enough bytes to encode $2 \cdot fanout$ while an `address` may need to encode a much larger `int`, thus different serialisers may be desired.

I was keen on setting up a serialisation framework with both decent type safety, enough flexibility that the file format could be easily modified, and granularity precise enough to access or modify part of a vertex without recomputing its serialisation.
Let's illustrate the implemented framework with a toy file format `.toy`. A `.toy` file encode

a `type toy = [`Dead of int | `Alive of int] array`. It contains a header, with a field `n_alive: int` and a field `n_dead: int`, followed with a succession of elements. Each element has a field `alive: bool` and a field `value: int`. We want to write two functions

```
val serialise: toy -> bytes
val deserialise: bytes -> toy
```

`Encapsulate` First, each field -representing the encoding of some object with type `underlying`- is dedicated its own module, with signature

```
sig
  type t
  type convert

  val set: bytes -> off:int -> t -> unit
  val get: bytes -> off:int -> t
  val size: int
  val to_t: convert -> t
  val from_t: t -> convert
end
with type convert := underlying
```

The two functions `set` and `get` are the serialiser and deserialiser of the field, respectively. Note that we fix in advance the length of the serialisation to `size`. This may sometimes waste a few bytes, but let us access any field without traversing the ones preceeding.

`Offsets and sizes` Second, each "basic brick" of the file format, i.e. the *header* and the *element* yields an offset record and a size variable.

```
type header_off = { n_alive: int ; n_dead: int }
type elem_off   = { alive  : int ; value : int }

let header_sizes = [ Nalive.size ; Ndead.size ]
let header_size = List.fold_left (+) 0 header_sizes
let header_offs = match sizes_to_offsets header_sizes with
  | [ n_alive ; n_dead ] -> {n_alive ; n_dead}
  | _ -> assert false

let elem_sizes = [ Alive.size; Value.size ]
let elem_size = List.fold_left (+) 0 header_sizes
let elem_offs = match sizes_to_offsets elem_sizes with
  | [ alive ; value ] -> {alive ; value}
  | _ -> assert false
```

`Serialise` Finally, it is now straightforward to implement our desired serialisation functions (here only `deserialise` is illustrated).

```
let deserialise buff =
  let n_alive = Nalive.get buff ~off:header_offs.n_alive in
  let n_dead  = Ndead.get  buff ~off:header_offs.n_dead in
  let n_elem = (Nalive.from_t n_alive) + (Ndead.from_t n_dead) in
  Array.init n_elem @@ fun i ->
    let off = header_size + i * elem_size in
```

```
let alive = Alive.get buff ~off:(off + elem_offs.alive) in
let value = Value.get buff ~off:(off + elem_offs.value) in
if Alive.from_t alive then
  `Alive (Value.from_t value)
else
  `Dead (Value.from_t value)
```

This framework has the granularity and flexibility desired. It catches *some* errors at compilation time, but is still liable to - for instance - mixing up `header_offs.n_alive` and `header_offs.n_dead`. However, the consistent namespacing should make most mistakes apparent - and in my experience it has.
Note that if our goal was only to implement `serialise` and `deserialise`, a solution based on the OCaml `Repr` module should be prefered. Our solution has the desired property that we can modify part of the object with minimal serialisation (e.g. if we want to change an element from `` `Alive `` to `` `Dead `` in our toy example).

4.2.4 Code structure

The top level module is `Btree`. It is the only module to have knowledge of the btree structure, and logically, is in charge of monitoring the tree traversal and balancing operations. The `Vertex` module handle leaf-local and node-local operations. In particular, this is where the page format and the matching serialisation protocol live. Finally, the `Store` module is the storage manager; it abstracts the cache and syscalls away. This is also where the file format specification is implemented.

4.3 Performance

4.3.1 Benchmarking

Replacing the data structure underlying `index` was primarily motivated by performance concerns. `index` performs lookups very efficiently, with in practice 1 syscall only, but has quadratic complexity for insertions. As the Tezos blockchain grows, it should become infeasible to keep it as is. Therefore it is crucial that `cactus` performs and scales better comparatively – and a great deal of time was spent monitoring its performance.

Profiling tools We used the `perf` analysis tool to profile time performance, and the `memtrace` ocaml package to profile memory allocations. Both tools produce flamegraphs, which represent function calls in layers, ordered with respect to the call stack and with size proportional to the total time taken by the function (or in the case of `memtrace`, the total memory allocations entailed). They often make apparent the bottlenecks of the application.
In addition, we implemented our own btree-local timers, to be wrapped around function calls (see module `Stats`). They make it possible to track the distribution of a few handpicked functions' call latency, where flamegraphs average over all calls to the function. This is especially relevant to see the evolution in time of such distributions, which shows the impact the size of the tree has on performance.

Benchmarks We tested `cactus` in several contexts. For each of those, several setups, with different hardware, were used. Namely, benchmarks were run on

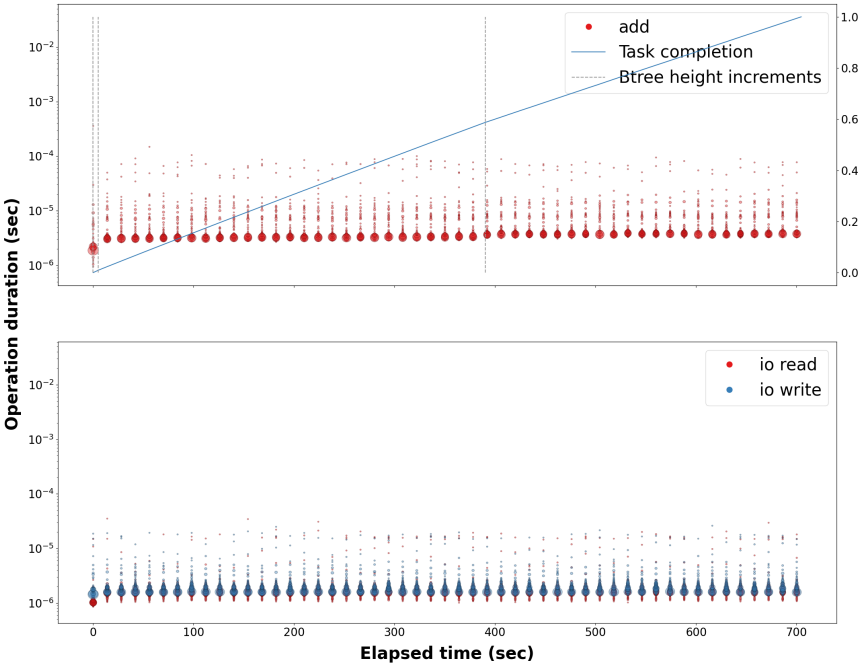
`rasp` a Raspberry 4 model B with 4GB RAM.

`packet1` an equinix t1.small.x86, with 8GB RAM and one 80GB SSD.

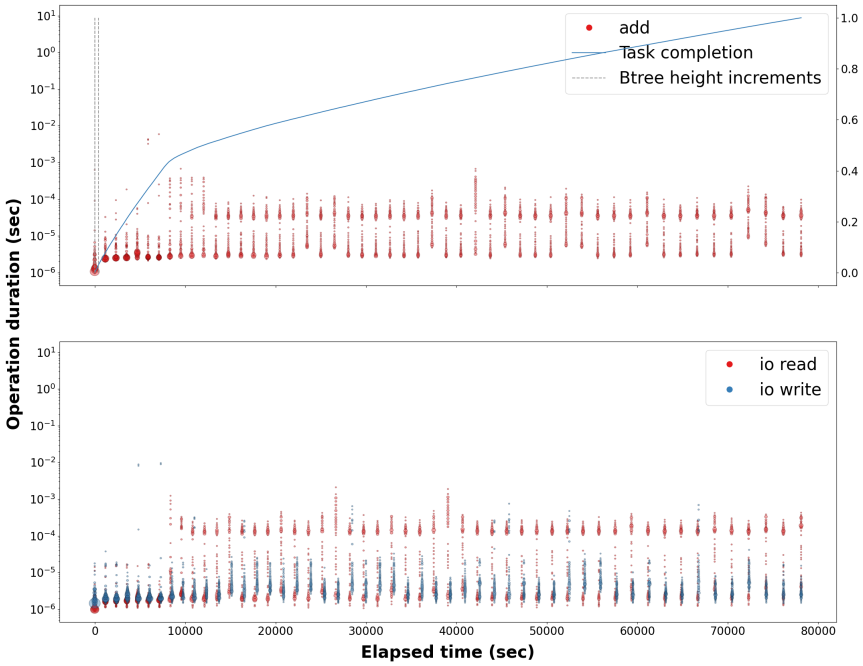
`packet2` an equinix c1.small.x86, with 32GB RAM and two 120GB SSD.

We set the fanout to be 50 and allowed a maximal cache size of 1GB.

`High-insert` The first benchmarking context continuously adds new bindings to the tree. This tests the maximal supported insertion throughput and its relation to the tree size.



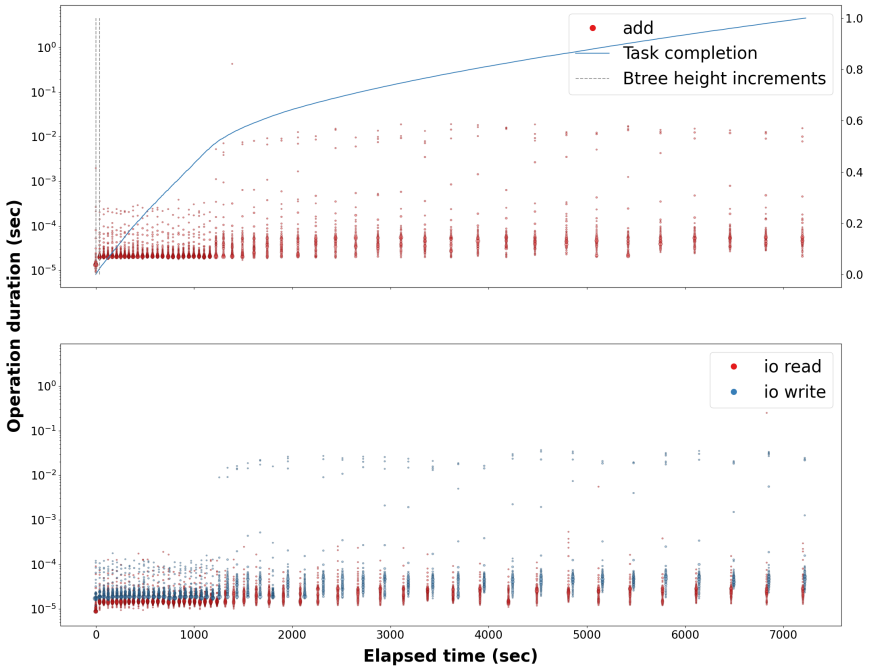
(a) `packet2` 50M insertions in an initially empty store. I/O is steady as the `b.tree` file grows. The completion curve is basically linear, indicating little variance in the rate of insertions 71M ops/sec.



(b) `packet2` 1G insertions in an initially empty store. Once the store is big enough, I/O gets costlier. Insertion rates subsequently drops from 52M ops/sec to 8.2M ops/sec.

Figure 8: Violin graph of different runs of the `high-insert` benchmark.

`Bootstrap trace` The second benchmarking context runs `cactus` against realistic user com-



(c) `rasp` 20M insertions in an intially empty store. Once the store is big enough, I/O gets costlier. Insertion rate subsequently drops from 8M ops/sec to 1.6M ops/sec.

Figure 8: Violin graph of different runs of the `high-insert` benchmark.

mands. The commands replay the end of the bootstrap of a Tezos node. The initial store already contains 291M bindings, the trace contains 100M operations – 10M `add`, 69M `find` and 20M `mem`.

`Irmin-btree` The last benchmarking context plugs `cactus` into `irmin-pack` and runs the bootstrap of a tezos node.

Difference between `cactus` and `index` The quadratic cost of insertion in `index` is only problematic in so far as there are realistic situations where the store can reach a problematic size. In particular, the $O(n^2)$ complexity hides a small constant, while for btrees the $O(\log n)$ hides a big constant. It was therefore crucial to compare them against one another in a Tezos production setting.
Additionally, it must be noted that the pressure on hardware is of a different kind between the two data structures. `index` writes a lot of data in append only mode ; `cactus` writes small bits of data in random locations. This questions almost orthogonal properties of the hardware: the maximal byte throughput and the random disk access overhead.

4.3.2 Optimisations

Flushing dirty pages only Pages are regularly flushed in and out of the cache to maintain memory constraints. Flushing out a page yields one syscall, which are the bottleneck of the data structure. Sometimes, the flushed-out page does not need to be written on disk, as it has not been modified – it is said to be *clean*. This happens when a page is loaded in response to a read-only transaction, such as a lookup.
We can and should keep track of dirty pages: we can leverage our convenient serialisation protocol which ensures that any page modification passes through a `set`-like function. To

avoid mistakenly modifying a page without marking it as dirty, it is enough to modify slighty the interface of a field module (section 4.2.3, §) and add a "mark-as-dirty" argument.

```
val set: marker:(unit -> unit) -> bytes -> off:int -> t -> unit
```

Minimising buffer allocations The application can be intensive in memory allocations. Whenever a page is loaded into memory, a new buffer (several Ko long) must be allocated. On the other hand, when a page is flushed out, the buffer holding its content becomes free to use. We subsequently reuse leftover buffers to load new pages in the cache.

Phantom keys A common btree trick is to associate a "dead" flag to every key. Deleting a key inside a node becomes a matter of simply flipping a bit, balancing operations notwithstanding. This conveniently creates holes in the node that can later on be filled during the insertion of another key, which otherwise would need to `shift` keys to create an opening. Some authors (see for instance [4]) also recommend to allow nodes to underflow, predicting subsequent insertions which could bypass calls to `merge` altogether – although we chose not to implement this feature. In fact, we found a caveat that came with phantom keys (see subsubsection 4.6.2), while it was not clear that they improved performance.

Batch initialisation Replacing `index` with `cactus` likely will need to migrate existing stores to the new data structure. To be more efficient than adding the migrating bindings one by one, which is quite slow, we implemented a batch initialisation utility. The algorithm (see 4.3.2) first calls an external merge-sort on the keys, then creates the btree in a bottom up manner. To this end, we developed `oracle`, a small external sorting OCaml library.

4.4 Integrity

A real-world data base system needs to be able to recover gracefully from abrupt interruption, with minimal data loss. An interruption can have many sources: a system crash, a power cut or simply an interrupt signal sent by the user. Two invariants express this integrity constraint.

`Recoverability` The disk content should be consistent at any point of the program, that is the application should be able to read and make sense of the information persisted on disk.

`No-loss` When the application yields control back to the user at the end of a transaction, enough information should be persisted that the modifications incured by the transaction can be retrieved, e.g. after INSERT X yields back control, x can be considered to be part of the data base even in the event of a crash.

Data integrity is not hard to achieve naively. For instance, one could maintain an ordered record of all transactions from the creation of the data base. Recovering from a crash then is a simple matter of replaying back the full history. Of course, this is not feasible: first, the history log will grow indefinitely, and second, recovery might take as long as the database is old. Worst, if the underlying system is unstable, say because the system works over a network which often breaks, the recovery process could crash before termination and need to start over. If recovery takes too long, it might start over and over and never be able to terminate.
On the other hand, it is not satisfying either to bypass the application cache altogether and persist everything all the time. NO-LOSS would be given for free, but the application latency would be greatly increased overall.

Our solution to integrity comes naturally from our caching hierarchy applied to B^+-trees. For large enough trees, only the top part of the tree can be cached and leaves are continuously flushed to the disk. Now, in our design, all information is leaf-resident; nodes only hold meta-information to navigate the tree. This means that `recoverability` is already almost free: a

recovery process may inspect every page, ignoring all non-leaves, and writing a new, equally valid, canopy over the existing one (see subsection 3.5 for details on the recovery process).
`no-loss` is almost given since leaves are flushed continuously. The only troublesome situation is when several leaves are modified as part of a single transaction. What happens if the system crashes while some but not all leaves have been written to disk ? There are two operations where such issue could arise, we deal with them separately.

crash during `split` An insertion in a leaf L_1 may trigger a split if the leaf was already full. After the split, a new leaf L_2 has been created and half the bindings from L_1 have been moved to L_2. Sadly, we cannot atomically flush L_1 and L_2 at the same time.

1. If we first flush L_1 and the system crashes, we'll have lost half the bindings and broken the `no-loss` invariant.
2. If we first flush L_2 and the system crashes, we'll have duplicata among the leaves and the `recoverability` invariant will be broken (leaves with duplicata do not describe a consistent btree).

We chose a simple solution which consist in detecting duplicata during recovery. However, we want to avoid inspecting every key from every leaf. Instead, it is sufficient to detect when a leaf's covered key range is a subset of one of its neighbours'. This can be done simply by comparing the leftmost and rightmost keys of neighbouring leaves.

crash during `merge` A deletion of a key k in a leaf L_1 may trigger a merge if the leaf was already at bottom capacity. After the merge, a neighbouring leaf L_2 will have received the bindings from L_1, and L_1 will be deleted.

1. If we wirst flush the deletion of L_1 ans the system crashes, we'll have lost all its bindings and broken the **no-loss** invariant.
2. If we first flush L_2 and the system crashes, we'll have duplicata among the leaves and the `recoverability` invariant will be broken

It seems we can pick option 2 since the modified recovery algorithm described above can already handle duplicata in the leaves. However, it is possible that the range of keys from pre-merge L_1 is not a subset of the range of keys of L_2 post-merge ! This is the case when k is L_1's smallest key (and L_2 is the right neighbour) or its largest (and L_2 is the left neighbour). The solution is a 3-step flushing process :

1. Delete k from L_1. Flush L_1.
2. Merge L_1 into L_2. Flush L_2.
3. Flush the deletion of L_1.

Caveat We did not precisely benchmark this recovery process, but it is somewhat slow. It should still be sufficient for episodic post-crash recovery. Moreover, the recovery process is not yet duplicata-aware as it was described above.

4.5 Concurrency

The concurrency specifications that `cactus` should support differ from the usual concurrency requirements quite a bit. They are in general more relieving, as a single concurrent process may have write rights, which in particular suppresses the need for any latching and locking mechanism. The trouble, because trouble there is, lies in the fact that the processes may not share memory. Hence the question: how should readers keep up with the latest updates given to the writer ?

Naively, of course, it would be enough to flush all dirty pages on disk every time a reader needs to sync up. This adds latency to the writer, because it reduces the benefits of efficient caching,

and to the reader, which needs to get rid of its cache altogether at each `sync`. Our goal is to do better than the naive approach, that is, we still want to leverage the efficient caching of both the writer and the readers.

The first key idea is to realise that pages that are neither in the reader nor the writer's cache don't need to be synced. The second idea is that, provided that `sync` is called often enough, pages in the reader's cache are *almost* in sync. Finally, the third realisation is that the writer's California cache and the reader's California cache contain the same pages. This provides the bases for our roadmap to achieve concurrency.

TODO

4.6 Caveats

This subsection targets specifically readers interested in working on `cactus`. It lists out miscellaneous recommendations and issues we faced during implementation, that we think are relevant again for further implementation.

4.6.1 `bytes` v `string`

A reader keen on the functional programming paradigm might regret that the use of mutable representations at the core of our design does not fully leverage the assets of OCaml, and they would be right. Using `bytes` as the internal representation of vertices - and worst, reusing them to buffer new pages - was a source of bugs and might still be. In fact `strings`, the immutable counterpart of `bytes` in OCaml, were at first the data structure of choice.

We decided to migrate to `bytes` for performance consideration. In a typical btree insertion, only a few bytes per page are modified. With immutable buffers, each of those modifications entails an allocation of a new page-sized string. In comparison, modifying cached buffers in place incur no memory allocation.

4.6.2 Phantom keys and deletion

Recall from section 4.3.2, §Phantom keys that we introduced a "phantom bit" to facilitate deletion. Without phantom keys, traversing a node (for instance during LOOKUP(k)) is a matter of finding the greatest i such that $k_i \leq k$ (where $k_0 < \ldots < k_p$ are the node separators). With phantom keys, we need to find the greatest i such that $k_i \leq k$ *and such that* k_i *is not a phantom key*. For instance we can get this with a regular binary search, then looking left to the first non-phantom key.

Here comes the issue: during a merge in a run of DELETE (see 3.3), we look for neighbours to take keys from, *glancing over phantom neighbours*. This is actually a source of bug that can break the `sortedness` invariant, as illustrated in figure 9. Our solution was to detect when such cases occur and locally remove phantom keys. It is not obvious that this cannot hide another bug, and we think that phantom keys may need to be removed from the design in the future.

4.6.3 Performance drop

We observed in some situations a significant performance drop with the rate of insertions decreasing by an order of magnitude. This happened on medium to big stores -with `b.tree` being several gigabytes large- and, interestingly, only on machine running MacOS, although we are not sure if and why it is OS related.

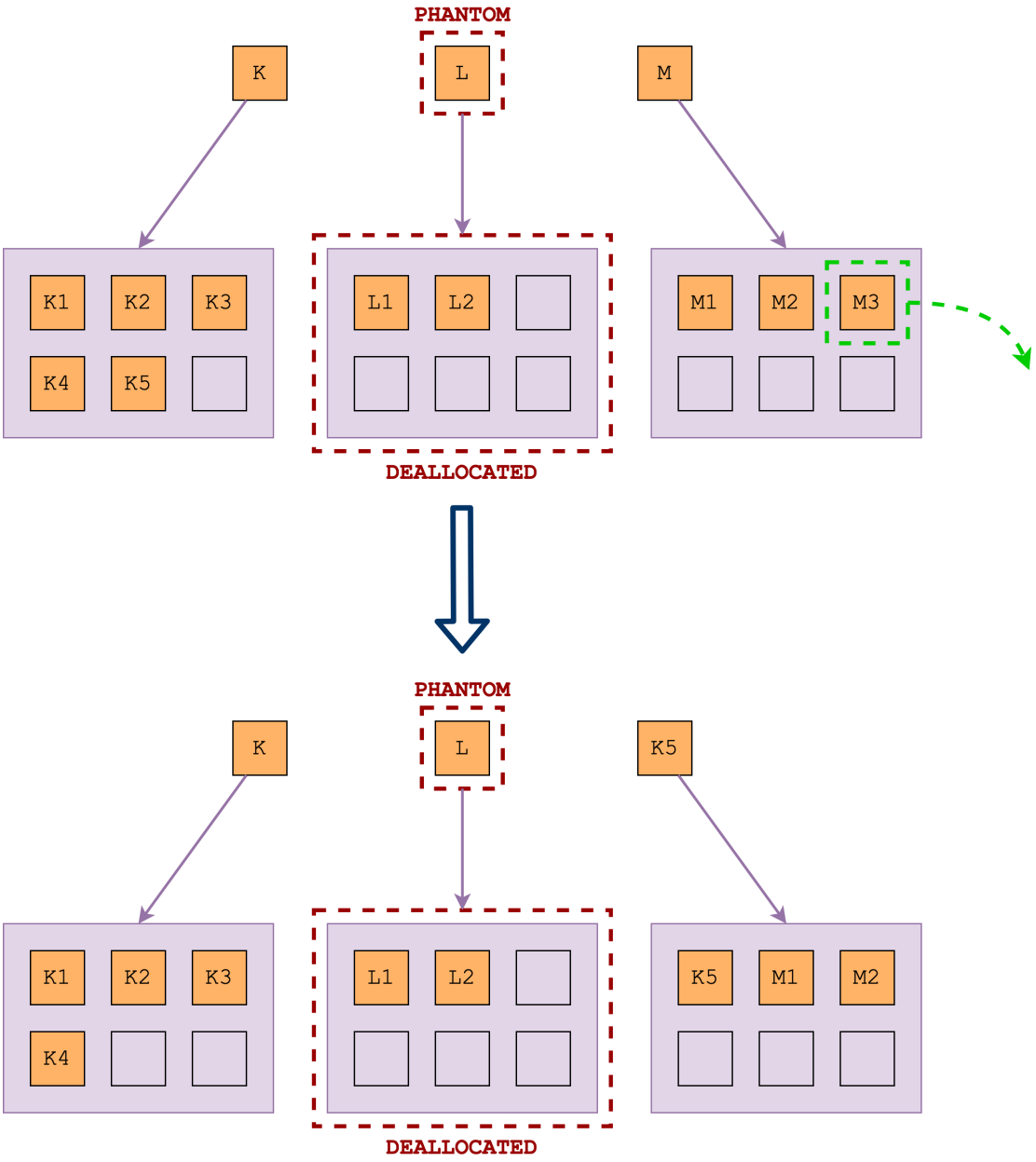


Figure 9: A partial merge can break `sortedness` when phantom keys are present. In the final, keys in the parent node are not sorted anymore.

Figures 10a and 10b show the run of 50M insertions on an initially empty btree on different machines. The violin graphs plot the latency of calls to `Btree.add` and `Syscalls.write`. The task completion plot shows the percentage of insertions done so far. On the MacOS machine, after about 25M bindings are inserted, some I/O calls become expensive, insertions subsequently slow down and the completion curve flattens. Comparing the latency of insertions and I/O calls, clearly I/O is responsible for the bad performance.

An additional observation that is not visible in the graphs is that concomitant to the performance drop are punctual high-latency I/O calls which take more than a second. Those happened once every 30 seconds exactly, and were not disturbed by the addition of random sleeps – mean-

ing that the 30 seconds are a manifestation of some sort of clock external to the application. We could not pinpoint the cause of this behaviour, although we suspect that the OS I/O cache might be at stake. Fortunately, the Tezos use-case will run `cactus` on linux, which behaved fine in our benchmarks. Additional details can be found at [1].

4.6.4 Memory blow ups

The amount of RAM allowed to the cache is a user-defined btree parameter. Yet, when btrees were plugged to `irmin`, we observed inconsistent blow-ups of peak memory usage. We know that this does not come from btrees but rather from memory allocations in `irmin` not yet collected by the gc, as these blow-ups did not occur when replaying the same operations to btrees as a standalone module. However, these blow-ups also do not occur when index is plugged into `irmin`. We are not sure where the peak memory usage came from, but it also seems that the issue eventually fixed itself out, which was both a good and a bad sign.

4.6.5 Impact of `Stats` on performance

A `Stats` module can be used in btrees to monitor the performance of the main functions. The functions are by default not tracked, and tracking can be independently set for each of them. While this provides convenient information, we warn any future developer that the cost on performance is noticeable, enough that only a handful functions should be tracked during benchmarking, and none in production.

5 Future work

5.1 Milestones

Integrity checks While we are able to recover from a system crash, btrees are currently reliable as far as the stable storage hardware is. This is too optimistic, it happens that individual bits randomly flip, and we'd like to be able to detect and recover from those as well. The implementation of an integrity-checking background process should be a priority before `cactus` can go into production.

Concurrency Obviously, support for concurrency should be implemented, all the more that a design to achieve it was already sketched.

5.2 Optimisations to explore

5.2.1 Prefix B-trees

As a *btree* grows, bottom nodes (and leaves especially) cover a narrower range of keys. In particular, we can know in advance the first few bytes that a key must have in order to appear in a given node. This leaves room for an optimisation: we may only store the unknown suffix part of the keys. In particular, this allows us to pack more bindings into a single page. This implements the well-known prefix btree (see for instance [4], chapter TODO).

5.2.2 Suffix truncation

Suffix truncation is another technique that allows more bindings to be packed into a single page. In internal nodes, separators need not be actual keys from the tree, they only need to correctly split the key ranges. We can then choose the separator with minimal length that still has the correct logical semantic. For instance, to separate key $abcdefg$ and key $accdefg$, it is enough to store the separator ac.
There is a caveat to this, in that we cannot use this technique to separate between already truncated separators (which we may be tempted to do two levels above the leaves and higher).

Indeed consider figure 11 where the root separates between I and INT instead of INDEX and INODE. The transaction LOOKUP(INDEX) willincorrectly look in the right subtree.

5.2.3 Offline insertions

Suppose the user wants to run both INSERT(X) and INSERT(Y) on a btree store. If x and y are "very different", those transactions are almost orthogonal, in that x will be inserted in, e.g., some left branch of the tree, and y in some right branch. Provided that those transactions do not provoke a root-split, they can happen concurrently. Implementing such application-local concurrency may improve performance. It is unlikely to increase the maximal insertions throughput, since I/O calls are the bottleneck. However in a real world application, where insertions are interwoven with random waits and other transactions, it might provide almost instantaneous insertions.
Note that this framework has other implications - such as on data integrity - and likely is not trivial to implement.

5.3 Miscellaneous improvements

Unix dependency Currently, btree uses the `Unix` module to interact with the file system. Mostly, this dependency comes from I/O calls which already are abstracted away in `Syscalls`. It should be desirable and relatively straightforward to be platform-agnostic. The first for this will be to functorise the storage manager over `Syscalls`.

A more versatile release The application explicitly advocates to the storage manager when it is safe to clear the Volatile cache, through a `release` call. Currently, the semantic of a release is that no buffer is held anywhere in the application anymore. This is quite coarse, there are situations where we wish to release a single page only. In particular, this will be necessary to have control over the order of calls to `flush`, as needed for data integrity (see 4.4).

References

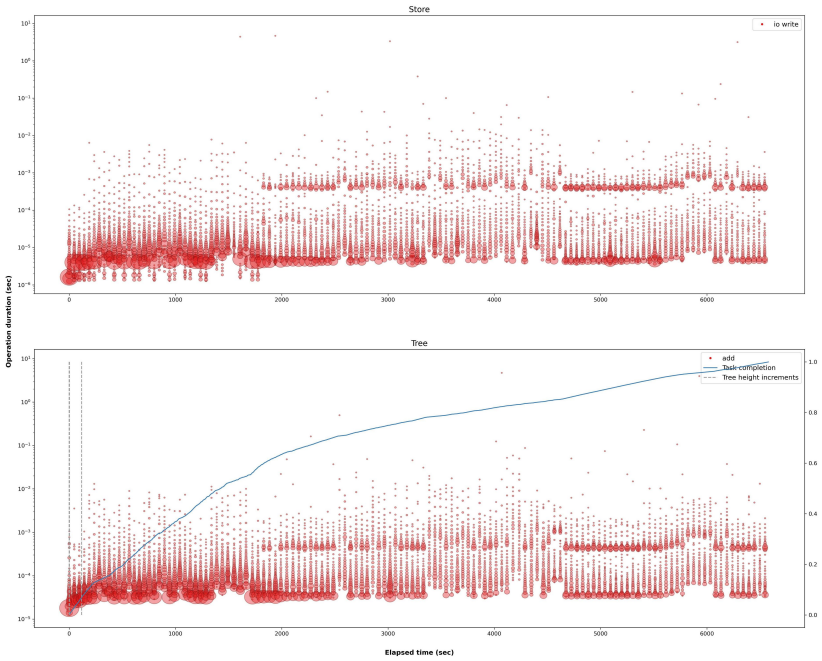
[1] Exploring a performance drop in btrees. `https://hackmd.io/S017exncRju-P9WvMtz6Ow`. Last updated: 2021-05-01.

[2] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

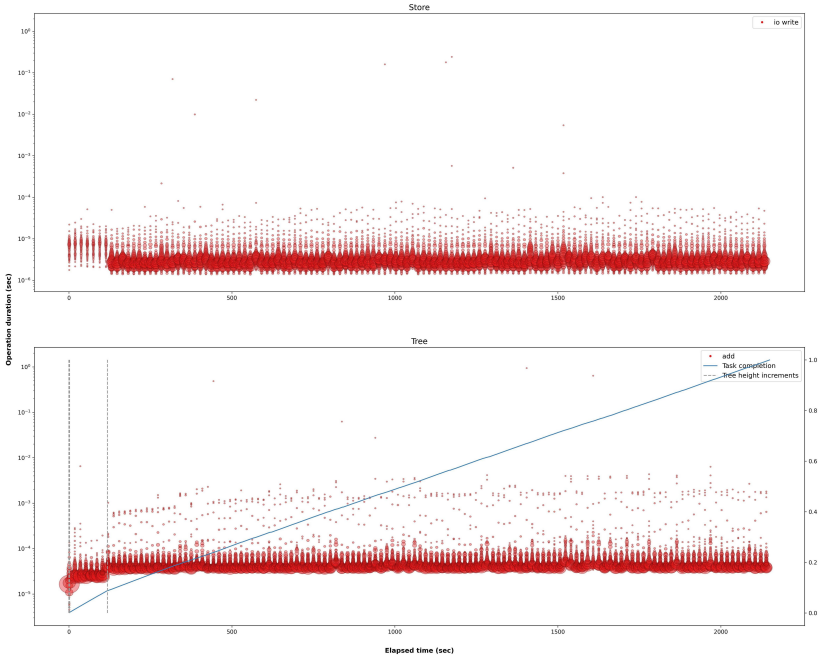
[3] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(10):1162–1175, April 2020.

[4] Goetz Graefe. Modern b-tree techniques. *Found. Trends Databases*, 3(4):203–402, April 2011.

[5] Alex Petrov. *Database Internals*. O'Reilly Media, Inc., October 2019.



(a) MacOS, 16GB RAM, 2.3GHz CPU



(b) Ubuntu, 32GB RAM, 2GHz CPU

Figure 10: Performance drop appears on MacOS but not Ubuntu, for roughly equivalent hardware.

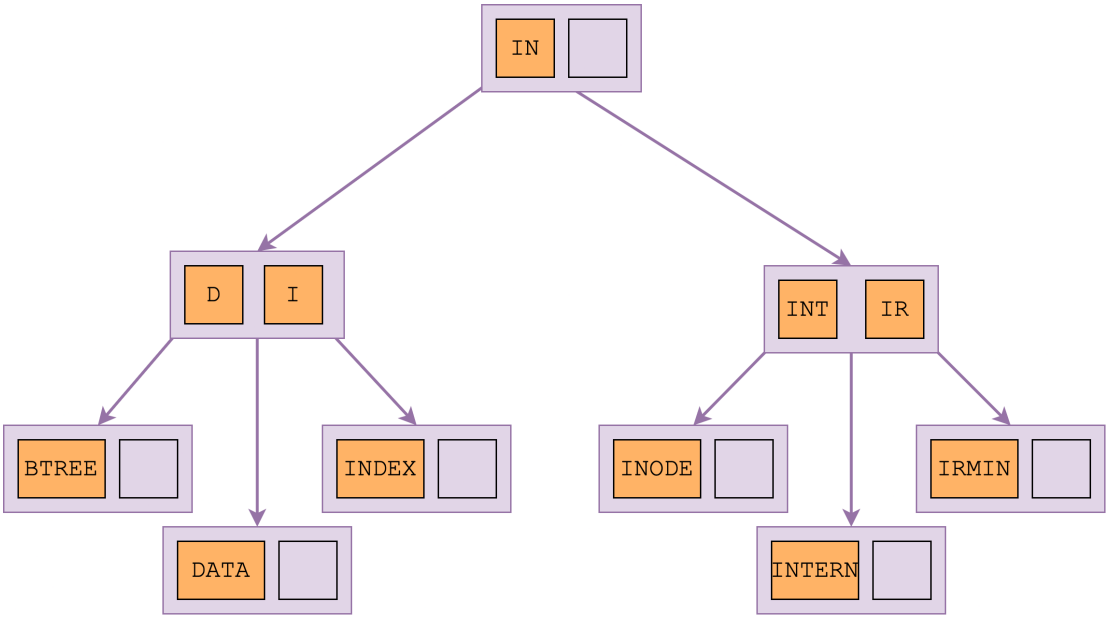


Figure 11: Incorrect suffix truncation : the root key IN separates between I and INT instead of INDEX and INODE.