ARTful Concurrency Control à la FoundationDB

Andrew Noyes [*]

Version 0.0.7

Abstract

FoundationDB [18] provides serializability using a specialized data structure called *lastCommit* [1] to implement optimistic concurrency control [8]. This data structure encodes the write sets for recent transactions as a map from key ranges (represented as bitwise-lexicographically-ordered half-open intervals) to most recent write versions. Before a transaction is allowed to commit, its read set is checked for writes that happened after its read version. FoundationDB implements *lastCommit* as a version-augmented probabilistic skip list [12]. In this paper, we propose an alternative implementation as a version-augmented Adaptive Radix Tree (ART) [9], and evaluate its performance. This implementation is available at `https://git.weaselab.dev/weaselab/conflict-set/releases` as a C or C++ library. *lastCommit* operates on logical keys and is agnostic to the physical structure of the data store, so it should be straightforward to use outside of FoundationDB.

1 Introduction

Let's begin by considering design options for *lastCommit*. In order to manage half-open intervals we need an ordered data structure, so hash tables are out of consideration. For any ordered data structure we can implement *lastCommit* using a representation where a logical key range (figure 2) is mapped so that the value of a key is the value of the last physical key (figure 1) less than or equal to the key. This is a standard technique used throughout FoundationDB called a *range map*.

Figure 1: Physical structure of range map

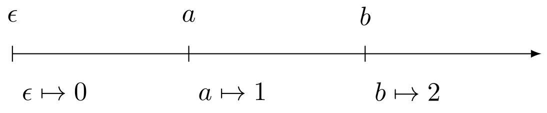
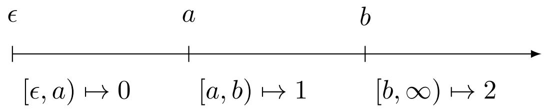


Figure 2: Logical structure of range map



The problem with applying this to an off-the-shelf ordered data structure is that checking a read range is linear in the number of intersecting physical keys. Scanning through every recent point write intersecting a large range read would make conflict checking unacceptably slow for high-write-throughput workloads.

This suggests we consider augmenting [5] an ordered data structure to make checking the max version of a range sublinear. Since finding the maximum of a set of elements is a decomposable search problem [2], we could apply the general technique using `std::max` as our binary operation, and `MIN_INT` as our identity. Algorithmically, this describes FoundationDB's skip list. We can also consider any other ordered data structure to augment, such as any variant of a balanced search tree [1, 6, 13, 4], or a radix tree [9, 3].

Let's compare the relevant properties of our candidate data structures for insertion/update and read

[*]andrew@weaselab.dev

[1]See Algorithm 1 referenced in [18].

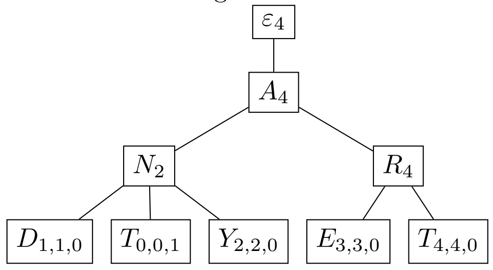
operations. After insertion, the max version along the search path must reflect the update. For self-balancing comparison-based trees, updating max version along the search path cannot be done during top-down search, because *insertion will change the search path*, and we do not know whether or not this is an insert or an update until we complete the top-down search. We have no choice but to do a second, bottom-up pass to propagate max version changes. Furthermore, the change will always propagate all the way to the root, since inserts always use the highest-yet version. For a radix tree, insertion does not affect the search path, and so max version can be updated on the top-down pass. There's minimal overhead compared to the radix tree unaugmented.

For "last less than or equal to" queries (which make up the core of our read workload), skip lists have the convenient property that no backtracking is necessary, since the bottommost level is a sorted linked list. Binary search trees and radix trees both require backtracking up the search path when an equal element is not found. It's possible to trade off the backtracking for the increased overhead of maintaining the elements in an auxiliary sorted linked list during insertion.

Our options also have various tradeoffs inherited from their unaugmented versions such as different worst-case and expected bounds on the length of search paths and the number of rotations performed upon insert. ART has been shown [9] to offer superior performance to comparison-based data structures on modern hardware, which is on its own a compelling reason to consider it. The Height Optimized Trie (HOT) [3] outperforms ART, but has a few practical disadvantages [2] and will not be considered in this paper.

[2]Implementing HOT is more complex than the already-daunting ART, and requires AVX2 and BMI2 instructions. HOT also involves rebalancing operations during insertion. Even so, it's likely that a HOT-based *lastCommit* implementation would be superior.

Figure 3:



2 Augmented radix tree

We now propose our design for an augmented radix tree implementation of *lastCommit*. The design is the same as the Adaptive Radix Tree [9], but each node in the tree is annotated with either one field *max*, or three fields: *max*, *point*, and *range*. *max* represents the maximum version among all keys starting with the prefix associated with the node's place in the tree (i.e. the search path from the root to this node). *point* represents the version of the exact key matching this node's prefix. *range* represents the version of all keys k such that there is no node matching k and this is the first node greater than k with all three fields set. See figure 3 for an example tree after inserting $[AND, ANT) \rightarrow 1$, $\{ANY\} \rightarrow 2$, $\{ARE\} \rightarrow 3$, and $\{ART\} \rightarrow 4$. Each node shows its partial prefix annotated with *max* or *max*, *point*, *range*.

2.1 Checking point reads

The algorithm for checking point reads follows directly from the definitions of the *point* and *range* fields. Our input is a key k and a read version r, and we must report whether or not the write version v_k of k is less than or equal to r. In order to find v_k, we search for the node whose prefix matches k. If such a node exists and has *point* set, then v_k is its *point* field. Otherwise, we scan forward to find the first node greater than k with *range* set. If such a node exists, then v_k is its *range* field. Otherwise k logically has no write version, and so the read does not conflict.

As an optimization, during the search phase for a point read we can inspect the *max* at each node that's a prefix of k. If the max version among all keys starting with a prefix of k is less than or equal to r, then $v_k \leq r$.

2.2 Checking range reads

Checking range reads is more involved. Logically the idea is to partition the range read so that each subrange in the partition is a single point or coincides with the set of keys beginning with a prefix (a *prefix range*). The max version of a single point is v as described in 2.1. The max version of a prefix range is the *max* of the node associated with the prefix if such a node exists, and *range* of the next node with a *range* field otherwise. If there is no next node with a range field, then we ignore that subrange in our max version calculation. The max version among all max versions of subranges in this partition is the max version of the whole range, which we compare to r.

Let's start with partitioning the range in the case where the beginning of the range is a prefix of the end of the range. We'll be able to use this as a subroutine in the general case. Suppose our range is $[a_0 \ldots a_k, a_0 \ldots a_n)$ where $k < n$, and $a_i \in [0, 256)$. The partition starts with the singleton set containing the first key in the range.

$$\{a_0 \ldots a_k\}$$

or equivalently

$$[a_0 \ldots a_k, a_0 \ldots a_k 0)$$

and continues with a sequence of prefix ranges ending in each digit up until a_{k+1}.

$$\begin{aligned} \ldots \quad \cup \quad & [a_0 \ldots a_k 0, a_0 \ldots a_k 1) \quad \cup \\ & [a_0 \ldots a_k 1, a_0 \ldots a_k 2) \quad \cup \\ & \ldots \quad \cup \\ & [a_0 \ldots a_k (a_{k+1} - 1), a_0 \ldots a_{k+1}) \end{aligned}$$

Recall that the range $[a_0 \ldots a_k 0, a_0 \ldots a_k 1)$ is the set of keys starting with $a_0 \ldots a_k 0$. The remainder of the partition begins with the singleton set

$$\ldots \quad \cup \quad [a_0 \ldots a_{k+1}, a_0 \ldots a_{k+1} 0) \quad \cup \quad \ldots$$

and proceeds as above until a range ending at $a_0 \ldots a_n$.

Let's now consider a range where begin is not a prefix of end.

$$[a_0 \ldots a_m, b_0 \ldots b_n)$$

Let i be the lowest index such that $a_i \neq b_i$. For brevity we will elide the common prefix up until i in the following discussion so that our range is denoted as $[a_i \ldots a_m, b_i \ldots b_n)$. We'll start with partitioning this range coarsely:

$$\begin{aligned} & [a_i \ldots a_m, a_i + 1) \quad \cup \\ & [a_i + 1, a_i + 2) \quad \cup \\ & \ldots \\ & [b_i - 1, b_i) \quad \cup \\ & [b_i, b_i \ldots b_n) \end{aligned}$$

The last range has a begin that's a prefix of end, and so we'll partition that as before. The inner ranges are already prefix ranges. This leaves only $[a_i \ldots a_m, a_i + 1)$.

If $m = i$, then this range is adjacent to the first inner range above, and we're done. Otherwise we'll partition this into

$$\begin{aligned} & [a_i \ldots a_m, a_i \ldots (a_m + 1)) \quad \cup \\ & [a_i \ldots (a_m + 1), a_i \ldots (a_m + 2)) \quad \cup \\ & \ldots \\ & [a_i \ldots 254, a_i \ldots 255) \quad \cup \\ & [a_i \ldots 255, a_i \ldots (a_{m-1} + 1)) \end{aligned}$$

and repeat [3] starting at

$$\ldots \quad \cup \quad [a_i \ldots (a_{m-1} + 1), a_i \ldots (a_{m-1} + 2))$$

[3] This doesn't explicitly describe how to handle the case where $a_{m-1} = 255$. In this case we would skip to the largest $j < m$ such that $a_j \neq 255$. We know $j \geq i$ since if $a_i = 255$ then the range is inverted.

until we end at $a_i + 1$, adjacent to the first inner range.

A few notes on implementation:

- For clarity, the above algorithm decouples the logical partitioning from the physical structure of the tree. An optimized implementation would merge adjacent prefix ranges that don't correspond to nodes in the tree as it scans, so that it only calculates the version of such merged ranges once. Additionally, our implementation stores an index of which child pointers are valid as a bitset for Node48 and Node256 to speed up this scan using techniques inspired by [10].

- In order to avoid many costly pointer indirections, we can store the max version not in each node itself but next to each node's parent pointer. Without this, the range read performance is not competetive with the skip list.

- An optimized implementation would visit the partition of $[a_i \dots a_m, a_i + 1)$ in reverse order, as it descends along the search path to $a_i \dots a_m$

- An optimized implementation would search for the common prefix first, and return early if any prefix of the common prefix has a $max \leq r$.

2.3 Reclaiming old entries

In order to bound memory usage, we track an *oldest version*, reject transactions with read versions before *oldest version*, and reclaim nodes made redundant by *oldest version*. We track the rate of insertions of new nodes and make sure that our incremental reclaiming of old nodes according to *oldest version* outpaces inserts.

2.4 Adding point writes

A point write of k at version v simply sets $max \leftarrow v$ [4] for every node along k's search path, and sets *range* for k's node to the *range* of the first node greater than k, or *oldest version* if none exists.

[4] Write versions are non-decreasing.

2.5 Adding range writes

A range write of $[b, e)$ at version v performs a point write of b at v, and then inserts a node at e with *range* set to v, and *point* set such that the result of checking a read of e is unaffected. Nodes along the search path to e that are a strict prefix of e get *max* set to v, and all nodes between b and e are removed.

3 Evaluation

4 Testing

The correctness of *lastCommit* is critically important, as a bug would likely result in data corruption, and so we use a variety of testing techniques. The main technique is to let libfuzzer [11] generate sequences of arbitrary operations, and apply each sequence to both the optimized radix tree and a naive implementation based on an unaugmented ordered map that serves as the specification of the intended behavior. After libfuzzer generates inputs with broad code coverage, we use libfuzzer's "corpus minimization" feature to pare down the test inputs without losing coverage (as measured by libfuzzer) into a fixed set of tests short enough that it's feasible to run interactively during development. In order to keep these test inputs short, we constrain the size of keys at the loss of some generality. We believe there isn't anything in the implementation particularly sensitive to the exact length of keys [5]. Libfuzzer's minimized corpus achieves 98% line coverage on its own. We regenerate the corpus on an ad hoc basis by running libfuzzer for a few cpu-hours, during which it tests millions of unique inputs.

In addition to asserting correct externally-visible behavior, in each of these tests we assert that internal invariants hold between operations. We also use address sanitizer [15] to detect memory errors, undefined behavior sanitizer [17] to detect invocations of undefined behavior, and thread sanitizer [14] (while exercising concurrent access as allowed by the con-

[5] `longestCommonPrefix` (a routine in the implementation) is a possible exception, but its length sensitivity is well encapsulated

tract documented in the c++ header file) to detect data-race-related undefined behavior.

Each of these sanitizers is implemented using compiler instrumentation, which means that they are not testing the final binary artifact that will be run in production. Therefore we also run the test inputs linking directly to the final release artifact, both standalone and under valgrind [16]. When testing the final artifacts, we do not assert internal invariants as we lack convenient access to the internals. As a defense against possible bugs in compilers' sanitizer and optimizer passes [7], we also test with sanitizers enabled and optimizations disabled, and test with both clang and gcc.

We audited the 2% of lines that were not covered by libfuzzer [6] and found the following:

- Three occurrences which can be reached from an input that libfuzzer could theoretically generate. In each case the uncovered code is straightforward, and is exercised from an entry point by a manually written test.

- One occurrence which requires a large number of operations, and cannot be reached from an input satisfying the size constraints we impose on libfuzzer. This code is also straightforward, and is exercised from an entry point by a manually written test. The purpose of this code is to keep memory usage in check, and so it's expected that it cannot be reached without a large number of operations.

- One occurrence which is not reachable from any entry point. This line is now suppressed with an explanatory comment.

We assert 100% line coverage in continuous integration, which is achieved with a few caveats. 2% of the code is only covered by a few manually written tests. We suppress lines manually checked to be unreachable from an entry point. There is also a significant amount of test-only code which is suppressed from coverage measurements. There's a small difference in the behavior between debug and release builds: the code which scans for old entries gets run more frequently when assertions are enabled. This code is not straightforward, so exercising it from only a manually written test seems insufficient.

[6] In order to see the uncovered lines for yourself, exclude all tests containing the word "script" with `ctest -E script`. Look in `Jenkinsfile` in the root of the source tree for an example of how to measure coverage.

5 Conclusion

References

[1] Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. "An algorithm for organization of information". In: *Doklady Akademii Nauk.* Vol. 146. 2. Russian Academy of Sciences. 1962, pp. 263–266.

[2] Jon Louis Bentley et al. "Decomposable searching problems". In: *Inf. Process. Lett.* 8.5 (1979), pp. 244–251.

[3] Robert Binna et al. "HOT: A height optimized trie index for main-memory database systems". In: *Proceedings of the 2018 International Conference on Management of Data.* 2018, pp. 521–534.

[4] Douglas Comer. "Ubiquitous B-tree". In: *ACM Computing Surveys (CSUR)* 11.2 (1979), pp. 121–137.

[5] Thomas H Cormen et al. *Introduction to algorithms.* MIT press, 2022. Chap. 17 Augmenting Data Structures.

[6] Leo J Guibas and Robert Sedgewick. "A dichromatic framework for balanced trees". In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978).* IEEE. 1978, pp. 8–21.

[7] Raphael Isemann et al. "Don't Look UB: Exposing Sanitizer-Eliding Compiler Optimizations". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: `10.1145/3591257`. URL: `https://doi.org/10.1145/3591257`.

[8] Hsiang-Tsung Kung and John T Robinson. "On optimistic methods for concurrency control". In: *ACM Transactions on Database Systems (TODS)* 6.2 (1981), pp. 213–226.

[9] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, 2013, pp. 38–49. DOI: 10.1109/ICDE.2013.6544812. URL: https://doi.org/10.1109/ICDE.2013.6544812.

[10] Daniel Lemire et al. "Roaring bitmaps: Implementation of an optimized software library". In: *Software: Practice and Experience* 48.4 (Jan. 2018), pp. 867–895. ISSN: 1097-024X. DOI: 10.1002/spe.2560. URL: http://dx.doi.org/10.1002/spe.2560.

[11] *libFuzzer – a library for coverage-guided fuzz testing.* https://llvm.org/docs/LibFuzzer.html. Accessed: 2024-04-19.

[12] William Pugh. "Skip lists: a probabilistic alternative to balanced trees". In: *Commun. ACM* 33.6 (June 1990), pp. 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: https://doi.org/10.1145/78973.78977.

[13] Raimund Seidel and Cecilia R Aragon. "Randomized search trees". In: *Algorithmica* 16.4-5 (1996), pp. 464–497.

[14] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice". In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA '09. New York, New York, USA: Association for Computing Machinery, 2009, pp. 62–71. ISBN: 9781605587936. DOI: 10.1145/1791194.1791203. URL: https://doi.org/10.1145/1791194.1791203.

[15] Konstantin Serebryany et al. "AddressSanitizer: a fast address sanity checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 28.

[16] Julian Seward and Nicholas Nethercote. "Using Valgrind to detect undefined value errors with bit-precision". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 2.

[17] *UndefinedBehaviorSanitizer — Clang 19.0.0git documentation.* https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed: 2024-04-19.

[18] Jingyu Zhou et al. "FoundationDB: A Distributed Unbundled Transactional Key Value Store". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li et al. ACM, 2021, pp. 2653–2666. DOI: 10.1145/3448016.3457559. URL: https://doi.org/10.1145/3448016.3457559.