Optimizing Collaborative Plain Text Editing Algorithms

for Decentralized Non-Realtime Text Editing

Bachelor thesis by Moritz Hedtke

Date of submission: August 5, 2024

1. Review: Prof. Dr.-Ing. Mira Mezini
2. Review: Dr.-Ing. Ragnar Mogk

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

TU Darmstadt

Software Technology Group

Optimizing Collaborative Plain Text Editing Algorithms
for Decentralized Non-Realtime Text Editing

Bachelor thesis by Moritz Hedtke

Date of submission: August 5, 2024

Darmstadt

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-278347
URL: https://tuprints.ulb.tu-darmstadt.de/27834
Jahr der Veröffentlichung auf TUprints: 2024

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
https://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Moritz Hedtke, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 5. August 2024

Moritz Hedtke

Abstract

Text editing is ubiquitous, as it occurs on almost every website, mobile app, and desktop application. Collaborative text editing avoids manual synchronization when working together with others on text. This requires algorithms that can efficiently combine the concurrent edit operations in an intent-preserving way. Additionally, supporting a wide range of network scenarios enables offline work in a decentralized manner with better availability and reliability than with central servers. In this thesis, we first look at prior solutions for plain text editing and their ability to preserve user intentions, as users should not experience unexpected behavior when concurrently editing text. Then, we improve the benchmarking approach of prior research to estimate asymptotic complexity and to measure performance of algorithmic edge cases. Based on that, we propose optimizations for a prior collaborative text editing algorithm called Fugue. Our optimized algorithm can handle character insertion and deletion in logarithmic runtime in relation to the text length and with constant memory usage per character operation. It uses 25 bytes and one microsecond per operation on four Intel Xeon Gold vCPUs for a representative text with 25 million operations. We also develop a local web application as a proof of concept for working on plain text collaboratively using WebRTC. Additionally, we show that the maximally non-interleaving property in the Fugue paper can exhibit interleaving when deletions are involved.

Contents

1. Introduction

Nearly all applications require text editing in some form — even if just for text entry into a form element. When we want to make these applications collaborative, those text fields need collaborative text editing. However, such functionality is not yet easily available. In contrast to single user text editing, collaborative text editing creates challenges with merging concurrent edit operations and especially handling conflicting edit operations and performance edge cases. Collaborative text editing algorithms need to handle conflicts in an intent-preserving and converging way.

Prior collaborative solutions require a central server, such as Microsoft 365 and Google Docs, or open source variants such as MediaWiki (which powers Wikipedia), Overleaf, Etherpad, Collabora Online or OnlyOffice. Needing a central server for text editing can be undesired for several reasons. First, when the server is operated by a third party it usually requires sending the text to the third party to handle the edit actions. Second, this creates a dependency on the availability of the server. The availability can be affected by power outages, cyberattacks, software and hardware failures including the network, overloading or natural disasters. Third, this also creates a dependency on the reliability or integrity of the server. Software and hardware failures especially of the storage can destroy the data, cyberattacks and human mistakes can manipulate or destroy the data, natural disasters or fire can destroy the server. Examples for such issues are OVHcloud's burned down data center, CrowdStrike, the Facebook BGP outage, the Google Cloud UniSuper incident, the XZ Utils backdoor, WannaCry, and many others.

Similarly, the client may not be able to reach the server. This may be the case when public infrastructure like cell towers is unavailable, because of natural disasters, sabotage, cyberattacks, failure of infrastructure they depend on such as the power grid or for any other reason. A recent example are the Ahrtal floods.

Decentralized algorithms can adapt to these challenges by functioning in a wide range of network scenarios. For example, peer-to-peer (P2P) networks work without a central server. Furthermore, mobile ad hoc networks (MANETs) and delay tolerant networks

(DTNs) do not require public communication infrastructure at all but instead can utilize Wi-Fi, Bluetooth and other short-range communication technology.

In a decentralized setting there is no guarantee that peers are frequently online. Therefore, the ability to handle *non-realtime* editing with potentially long periods of offline activity is essential. This combination of offline and decentralized software is often called local-first software [8].

The two major ways in research to approach collaborative text editing are operational transformation (OT) and conflict-free replicated data types (CRDTs) [15, page 2]. OT algorithms store edit operations based on the text position and therefore need to transform concurrent edit operations against each other to correct the text positions. Then, the algorithms apply the operations directly to the text. Prior algorithms for OT are, for example, COT [16] and Jupiter [11]. While some of these are *not* able to work in a decentralized network but need a central server to order changes like Jupiter [11], a lot of them *are* able to work in a decentralized network like COT [16] [17, Section 4]. Prior OT algorithms have a runtime complexity per remote operation that is linear in the amount of concurrent edit operations [18, Section 3.1.4]. This makes them really efficient for *near-realtime* editing where only few concurrent edit operations occur. Near-realtime editing means that only short connection interruptions happen [12]. For *non-realtime* text editing this leads to a highly inefficient runtime complexity because the many concurrent edit operations must be transformed against each other [17, Section 1]. Therefore, prior OT algorithms are undesirable for supporting a wide range of network scenarios like DTNs.

In contrast, CRDTs associate parts of the text with identifiers and merge these together on synchronization. Therefore, they need to convert between identifiers and text positions to handle text edit operations. Prior algorithms for CRDTs are, for example, WithOut Operational Transforms (WOOT) [13], Logoot [26], Replicated Growable Arrays (RGAs) [14] and Fugue [25]. CRDTs work in decentralized networks, but each prior algorithm has shortcomings that make it undesirable for a general solution. For example, Logoot [26] has quadratic memory use in some cases. Also, for handling text of some length their runtime complexity is often quadratic or worse in relation to the text length, as with WOOT [13], RGA [14] and Fugue [25] [15, Section 5.3].

While Fugue [25] avoids interleaving issues of prior solutions and works in an offline setting, the current implementation for handling text of some length has quadratic runtime complexity in relation to the text length.

In this thesis, we first investigate suitable algorithms for local-first plain text editing to integrate into our Scala based applications, see Chapter 2. Based on the evaluation of prior solutions in Fugue [25], we consider interleaving the major issue apart from performance issues, see Section 2.1. Therefore, we extensively investigate how the Fugue algorithm avoids interleaving by looking at the algorithm, the examples and the proofs in the Fugue paper [25], see Section 2.2. Additionally, we show that the property of *maximally non-interleaving* in the Fugue paper [25] still allows interleaving when deletions are involved. Section 2.3 gives an insight into CRDTs and OT and their advantages and disadvantages.

Then, Chapter 3 describes the Fugue algorithm [25] in depth. Chapter 4 discusses our base implementation of Fugue in Scala to be able to experiment with the algorithm and proposes using property tests to ensure the convergence of our implementation. For easier experimentation and as a showcase we create a local web application to collaboratively edit a text using WebRTC.

The benchmarks in Chapter 5 show that the base implementation has severe performance issues. Therefore, we optimize our implementation based on our benchmarks and propose optimizations of the Fugue algorithm. Through the use of binary search trees at relevant places with some use-case specific customizations we achieve amortized logarithmic runtime per character insertion or deletion and thus an amortized runtime of $O(n \log(n))$ for handling n character operations. Additionally, we implement batching of sequential insertions to reduce memory usage, which was already roughly mentioned in the Fugue paper without details on the exact implementation [25]. Furthermore, we contribute a benchmark that in comparison to prior work shows the asymptotic runtime and focuses on edge cases in the algorithm that may have performance characteristics different from those of the common execution path. Specifically, we focus on ensuring that the algorithm also has an acceptable runtime complexity when considering malicious or unexpected behavior of peers.

We evaluate our optimized implementation in Chapter 6 and show that we achieve the targeted $O(n \log(n))$ runtime complexity with a runtime of one microsecond per character operation and memory use of 25 bytes per operation for a realistic editing session on four Intel Xeon Gold vCPU. Finally, Chapter 7 shows future work such as rich text editing, and Chapter 8 concludes our work.

2. Challenges with Collaborative Text Editing

This chapter first introduces the goal of user intent-preservation by showing the problem of text interleaving in Section 2.1. Then, Section 2.2 introduces the solution proposed by Fugue [25] to solve text interleaving. Finally, Section 2.3 compares CRDTs and OT and shows that the current CRDTs runtime complexity is quadratic and current OT algorithms are unsuitable for *non-realtime* editing.

2.1. Text Interleaving

When users write text in a collaborative text editor, they expect that their text is not modified in an unexpected way by concurrent edits from other users. One example are insertions at *different* positions. Starting with the text `"Alice plays Minecraft"`, Alice changes the text to `"Alice happily plays Minecraft"`. Concurrently, Bob changes the text to `"Alice plays Minecraft with Bob"`. Then, the expected result after synchronizing is `"Alice happily plays Minecraft with Bob"`. As the insertions are at different positions in the text, the expected outcome is unambiguous, and all characters should stay at their relative position to the surrounding characters. Users also expect that text they wrote in one go is not interleaved by text that another user wrote concurrently. An example with insertions at the *same* position is the following. Starting with the text `"milk, chocolate"`, Alice changes the text to `"milk, eggs, chocolate"` and Bob concurrently changes the text to `"milk, bread, chocolate"`. The expected result after synchronizing is either `"milk, eggs, bread, chocolate"` or `"milk, bread, eggs, chocolate"`. While there are two possibilities in this case, no interleaving occurs in either case.

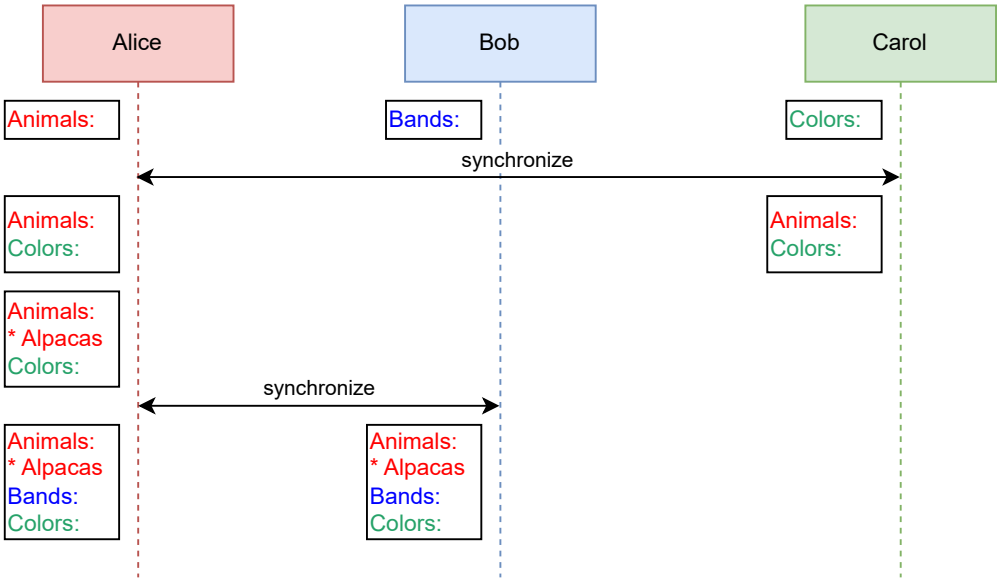


Figure 2.1.: Example for prioritizing forward insertions inspired by Figure 6 in Fugue [25]

For an insertion in the middle of a text, current editing behavior does not convey whether the insertion semantically belongs to the left side or the right side. Because most text is written in a forward direction, so for left-to-right script from left to right, it is more likely that an insertion in the middle of some text is appending to the left side of the insertion point instead of prepending to the right side of the insertion point. Figure 2.1 exemplifies this. The three replicas Alice, Bob and Carol independently add three lists to some text. Then, Alice and Carol synchronize. Afterwards, Alice adds `"* Alpacas"` to her list, such that it comes after `"Animals:"` and before `"Colors:"` but inherently there is no information to which part it belongs. Finally, Alice and Bob synchronize. This separates `"* Alpacas"` and `"Colors:"` by the received `"Bands:"`, which may not be wanted. In this example the assumption of the more common forward insertion is correct though. Further improvements to this would need analysis of the language semantics of the text which Bauwens et al. looked into [2]. For the concrete example, a different idea could be to insert `"Bands:"` after `"Colors:"` so `"* Alpacas"` stays in place in relation to the text preceding and following it. Unfortunately this would lead to even more unexpected behavior for example when Bob and Carol synchronized before and would order the entries alphabetically because they do not know about the insertion of `"* Alpacas"`. As soon as Alice would then synchronize with them, the entries would need to be reordered, so that they converge. As the synchronized data is not structured like the example may suggest, but instead consists of arbitrary characters, this reordering could result in sentence reordering or other unwanted results. Another idea could be to prefer the side by the same replica. This has similar issues if concurrent edits are received later and change the effect of that rule.

2.2. Fugues Approach to Avoid Text Interleaving

This section shows the proposed solution by Fugue [25] to solve text interleaving. It also gives an example that the proposed *maximally non-interleaving* property can still interleave text when deletions are involved.

Weidner et al. [25] show that a previous attempt at formalizing a property for non-interleaving by Kleppmann et al. [7] is incorrect [25, Section 2.5]. Therefore, they propose their own property which they refer to as *maximally non-interleaving*. It associates every inserted character with the character to its left and right, which they label left and right origin. The property orders the characters by prioritizing keeping the left origin as the previous character because of the common forward insertions and otherwise ordering

to preserve the right origin as the following character if possible. Only if both origins are the same, the order is arbitrary but deterministically chosen. Therefore, this property creates a unique order aside from tie-breaking [25, Section 4.5].

Fugue refers to an interleaving issue as forward interleaving, when only one character has another character as a left origin, yet the two characters are not consecutive. One example where the Logoot algorithm [26] interleaved characters, which also violates this rule, is concurrently inserting `"bread"` and `"eggs"`, producing `"bergegasd"` [18, Section 4.4.1]. For example the `"r"` from `"bread"` has the `"b"` as its left origin and no other character has the `"b"` as its left origin but in the result they are not consecutive characters.

Weidner et al. refer to another problem that many prior algorithms exhibit as backward interleaving. When two insertions have the same left origin but a different right origin, they should be ordered in a way that they are consecutive with their right origins. Although it may seem this is not a common use case, the following is a plausible example [25, Figure 2]. Starting with the text `"Shopping"`, Alice first appends `"* apples"` after `"Shopping"` and then prepends `"Fruit:"` before `"* apples"`. While semantically she is prepending, both inserted texts have `"Shopping"` as their left origin and different right origins. Concurrently, Bob first appends `"* bread"` after `"Shopping"` and then prepends `"Bakery:"` before `"* bread"`. The category insertions by Alice and Bob both have `"Shopping"` as their left origin but different right origins. Therefore, this should lead to either the outcome of `"ShoppingFruit:* applesBakery:* bread"` or `"ShoppingBakery:* breadFruit:* apples"` which only differ in the order of which users text comes first, which is arbitrary. When algorithms exhibit backward interleaving, `"ShoppingBakery:Fruit:* bread* apples"` can be a possible result. Note that the order of the elements has not changed in relation to each other (e.g. `"Fruit:"` comes before `"* apples"` and after `"Shopping"`) but this still violates the intent of the user.

According to Weidner et al., many popular algorithms they looked into exhibit either forward or backward interleaving [25, Table 1]. A review by Sun [19, 23, 22, 20] that refutes these claims for OT algorithms is addressed in Section 2.3. For Logoot [26] the character-by-character interleaving issue occurs. Further examples are provided in the appendix of the Fugue paper [25]. While the prior CRDT algorithms YjsMod[1] and Sync9[2] do not exhibit interleaving [25, Table 1], those approaches were not considered here due to the lack of documentation and their intrinsic complexity. Weidner et al. propose their own algorithms Fugue and FugueMax to solve these problems. They conjecture that Sync9 is semantically equivalent to Fugue and YjsMod is semantically equivalent to FugueMax [25,

[1] `https://github.com/josephg/reference-crdts`
[2] `https://braid.org/sync9`

Section 6]. They also prove that FugueMax fulfills the *maximally non-interleaving* property [25, Theorem 9], prove that the Fugue algorithm is always forward non-interleaving [25, Lemma 7] and argue that it is also backward non-interleaving when there are not multiple interacting concurrent updates [25, Section 4.3].

A counter example that interleaving can also happen for the *maximally non-interleaving* FugueMax algorithm is the following. Starting with the text `"Shopping"`, Alice appends `"* apples"` after `"Shopping"` and then prepends `"Fruit:"` before `"* apples"`. Concurrently, Bob appends `"* bread"` after `"Shopping"`, then deletes and reinserts the `"g"` of `"Shopping"` and finally prepends `"Bakery:"` before `"* bread"`. The expected result would be `"ShoppingBakery:* breadFruit:* apples"` but the actual result can be `"ShoppingBakery:Fruit:* apples* bread"` when the replicas IDs have a specific order. The code in Appendix A.7 verifies this with the reference implementation[3]. The reason the *maximally non-interleaving* property does not cover this case is that it disregards deletions. This example shows that this simplification is not suitable to ensure non-interleaving.

The basic implementation of Fugue has a linear runtime per character insertion or deletion in relation to the text length (including deleted text) which proved to be too inefficient for larger text given the resulting runtime scales quadratically with the text length. Comparing the results[4] from Weidner et al. for benchmark B1.1 with benchmark B1.3 indicates, that even the optimized variant in the Fugue paper has quadratic runtime for sequential backward insertions.

[3] `https://github.com/mweidner037/fugue`
[4] `https://github.com/mweidner037/fugue/blob/main/results_table.md`

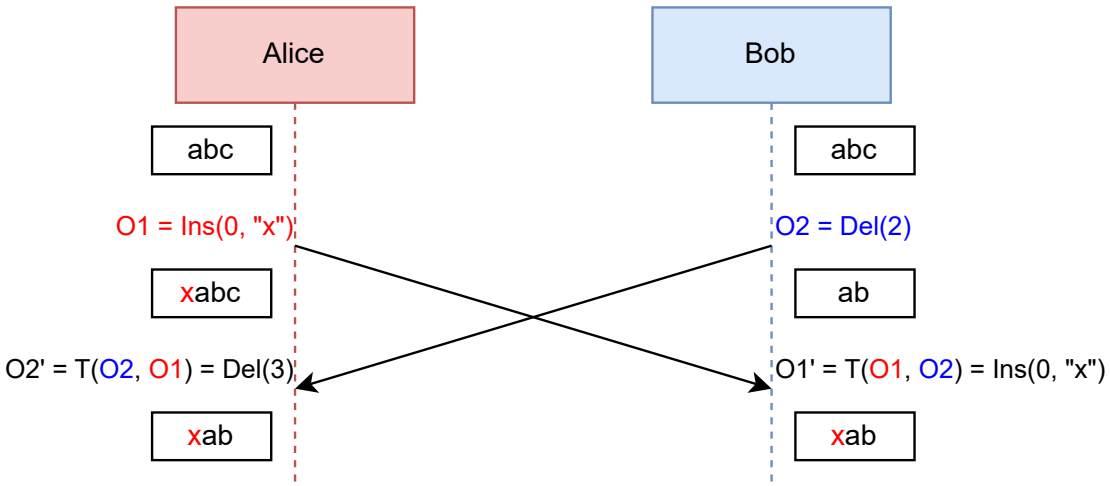


Figure 2.2.: Example for operation transformation with two synchronizing peers based on figure by Sun [21, Section 1.4 Figure 1]

```
Tii(Ins[p1,c1], Ins[p2, c2]) {
  if p1 < p2 or (p1 = p2 and u1 > u2)
    return Ins[p1, c1];
  else
    return Ins[p1+1, c1];
}
```

Listing 2.1.: Example for transformation function from Sun [21, Section 2.15]

2.3. OT in Comparison to CRDTs

This section explains the differences and similarities between OT and CRDTs and shows that the current CRDTs runtime complexity is quadratic and current OT algorithms are unsuitable for *non-realtime* editing.

While CRDT papers often claim CRDTs are superior to OT, CRDTs often miss major relevant parts of the required algorithmic steps which makes them seem potentially simpler and more performant [15, page 2]. For example, CRDTs need to extract the text from their internal state and need to be able to address characters based on their text position as most text editors work that way [15, Section 5.1, Section 5.2]. CRDTs often miss this conversion step which is a major algorithmic complication that also affects their performance a lot [15, page 2]. Note that Fugue also has this issue as it does not describe converting the received operations to character offsets [25, Algorithm 1].

Sun et al. also show that both approaches are more similar than often presented [15, Section 4.1 Table 1]. While OTs have position based operations directly on the character sequence that are then transformed by concurrent operations, CRDTs have identifier based operations on an internal object sequence, that are converted to the position based character sequence after the operations have been applied.

OT based algorithms consist of a control algorithm and a transformation function [21]. The control algorithm is generic, and the transformation function is application specific. For example for plain text editing there could be two operations, Insert(index, character) and Delete(index). The transformation function $T(O_2, O_1)$ transforms O_2 against O_1. This produces the operation that needs to be applied after O_1 if they were concurrent before. Figure 2.2 shows an example where the positions of the concurrent operations are transformed when receiving them and therefore result in the same text at both peers. In that example the transformation function could be defined as shown in Listing 2.1 for transforming two insert operations [21, Section 2.15]. If a concurrent insertion happened at a position after the current insertion it does not need to be transformed. If a concurrent insertion happened at a position before the current insertion it needs to be offset by one. For equal positions, tie breaking using the replica identifier is required.

The control algorithms decide in which order operations need to be transformed to achieve the desired outcome [21, Section 2.2]. Depending on the control algorithm, the transformation function needs to fulfill different properties to ensure correctness [21, Section 2.20]. Also, some control algorithms are able to handle undo, some can undo arbitrary actions out of order, while some cannot [21, Section 2.12].

Transformation functions need to be defined for all possible combinations of operations. This means N^2 such functions are needed for N possible operations. An alternative proposed by Sun et al. is POT+COA (Primitive Operation Transformation plus Complex Operation Adaptation). It consists of having some primitive operations for which transformation functions are defined, and then complex application operations are converted to these primitive operations [17, Section 2.1.3].

OT based algorithms can be integrated into existing editors with little change of the editors source code as OT is operation and concurrency-centric. The algorithm can just apply the received and transformed operations to the local editor and send local operations to other peers. Sun et al. refer to this as Transparent Adaptation (TA) [17, Section 2.1.2].

According to Sun et al., OT uses a concurrency-centric and direct transformation approach and CRDT uses a content-centric and indirect transformation approach [15, Section 1]. This has an important consequence for the time and space complexity. The time and space complexity of OT for *realtime* editing depends on the number of concurrent operations which are usually small in realtime text editing while the time and space complexity of CRDT depends on the length of the text or even the length of the text including all deleted content which are usually a lot larger [15, Section 5.3]. The time complexity for prior OT based algorithms is at least $O(c)$ per remote operation [18, Section 3.1.4]. This means quadratic runtime complexity in relation to the operation count for handling some count of operations, which is unusable for *non-realtime* editing because there can be many concurrent operations. It is important to mention that the time complexity class is relevant. For example, $O(\log(\text{text-length-including-deletions}))$ runtime complexity can be equally acceptable to $O(\text{concurrent-operations})$ runtime complexity because $O(\log(n))$ is growing quite slowly even for extremely large inputs. Prior research of CRDTs mostly managed a linear time complexity or worse except of a paper by Briot et al. which optimizes an RGA adaptation to $O(\log(n))$ per operation similarly to us [18, Table 4]. However, Briot et al. have not gone into the analysis of performance edge cases prohibiting us from drawing a fair comparison. Additionally, it is unclear whether they include the conversion of remote operations to character positions. Furthermore, as the algorithm is based on RGA, it exhibits interleaving [25, Table 1].

While CRDTs often seem to be simple and easy to understand, the fundamental concurrency issues which are inherent to unconstrained co-editing also exist there and mixing content and concurrency creates new difficulties with handling them [18, Section 4].

3. Fugue Algorithm

This chapter explains how the Fugue algorithm works and is heavily based on the Fugue paper [25]. Then, the next chapter discusses our implementation of Fugue. Further chapters discuss our improvements to the Fugue algorithm. The Fugue algorithm handles insertions and deletions for a list data structure and avoids interleaving. For text editing every list element is a character.

3.1. Traversal

Figure 3.1 visualizes the data structure of the algorithm. It is a tree starting with the root node at the top left. The nodes are connected using lines. Lines downwards to the right connect to a right child and lines downwards to the left connect to a left child. A node can have multiple children on each side. For every node except the root node, the first part is the character or whether the character is deleted, followed by a space and the ID of the peer that created that character, a `#` symbol and then a counter for that peer that is increasing for every insertion. For example, `"t A#1"` is the character `"t"` by peer `"A"` with the counter being 1. The ID of the peer combined with the counter that uniquely identifies an element is called a simple ID. The root node is a special node that behaves like a deleted character. To get the current text of the tree, it is traversed starting from the root node by recursively visiting the left children in order, then the value of the node itself and then the right children in order. For the example in Figure 3.1, the traversal starts with the left children of the root node. As there are none, the node itself is visited. As it contains a deleted character, it is ignored. Then the first right child is traversed. Its first left child produces `"small "` by the same rules applied recursively. It itself produces `"t"`. Its right children produces `"rees"`. Therefore, its whole traversal produces `"small trees"`. Then the second right child is traversed in the same way and produces `" grow"`. Combining all that will therefore produce the text `"small trees grow"`.

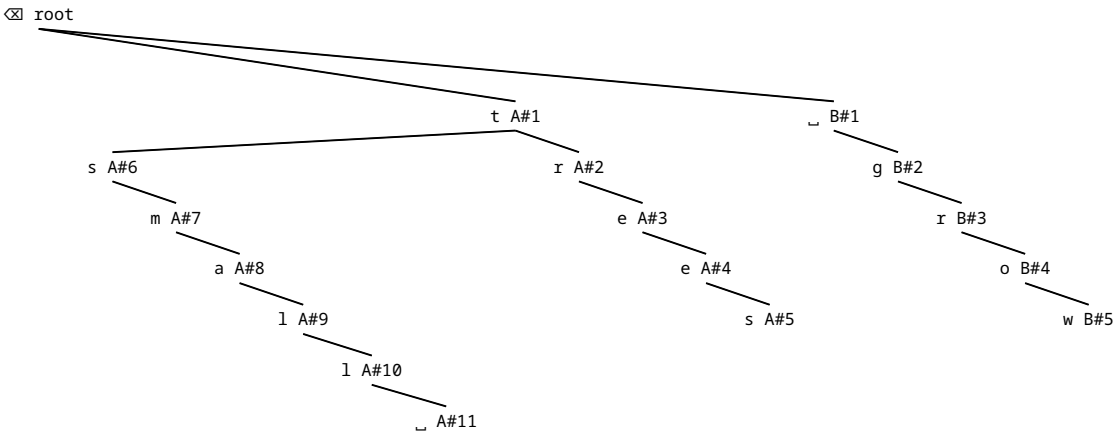


Figure 3.1.: Fugue tree traversal

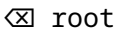


Figure 3.2.: Fugue tree with root node

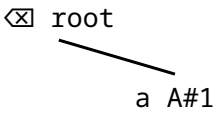


Figure 3.3.: Insertion of `"a"` into Fugue tree at index 0

3.2. Initial State

The initial state consists only of the root node as shown in Figure 3.2. Thus, the tree represents an empty text. The root node for every peer is the same, even though the root node is always created locally at every peer.

3.3. Operations

The chosen operations are insertion and deletion based on an index into the text relative to the start. The reason for choosing that interface is that text editors conform to it. All indices are zero based, so the element at index 0 is the first element.

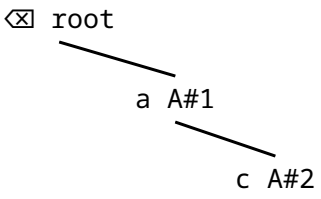


Figure 3.4.: Insertion of `"c"` into Fugue tree at index 1

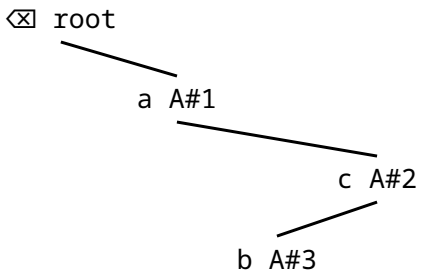


Figure 3.5.: Insertion of `"b"` into Fugue tree at index 1

Insert operation

To insert an element x at a position i, the algorithm first creates a new simple ID. A special case is inserting at position 0. In that case the root node is the left origin of the insertion. To insert the element, it is added as a right child to this left origin. This implies that the root node never has left children as otherwise this would be incorrect based on the tree traversal. Starting with an empty tree, Figure 3.3 shows an insertion at index 0.

Otherwise, the algorithm traverses through the tree, but only counts the visible elements (the ones that are not deleted) until the node at index $i - 1$ is reached. As this is the node before the index for the insertion, the insertion point is to the right of it. Like in the special case for inserting at position 0 this is the left origin. If that node has no right children, it adds the new node as a right child to that left origin. Starting with the previous tree, Figure 3.4 shows an insertion at index 1.

Right children are always deterministically but arbitrarily ordered by their replica IDs. Therefore, if the node already has right children, the new node can not be added to the right while ensuring it is at the correct position. Instead, the algorithm adds it to the left of the right origin to ensure it gets placed at the correct index. The right origin is the next node (visible or not) in the tree traversal after the left origin. This right origin can not already have left children as otherwise one of them would be the right origin as they come earlier in the tree traversal. Starting with the previous tree, Figure 3.5 shows an insertion at index 1.

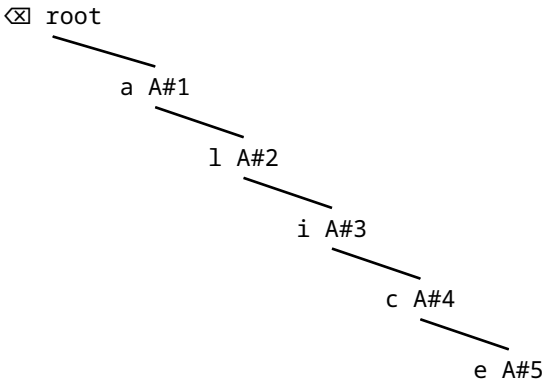


Figure 3.6.: Fugue tree with text insertion at replica A

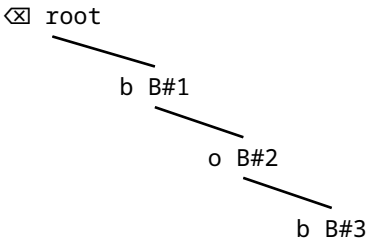


Figure 3.7.: Fugue tree with text insertion at replica B

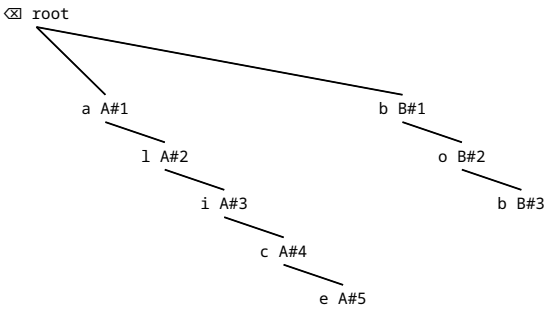


Figure 3.8.: Fugue tree with concurrent insertions after synchronization between replica A and replica B

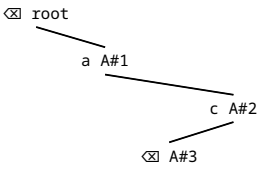


Figure 3.9.: Fugue tree with deletions

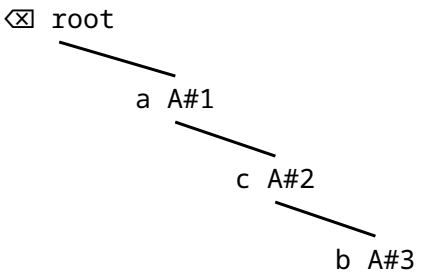


Figure 3.10.: Fugue tree with sequential insertions

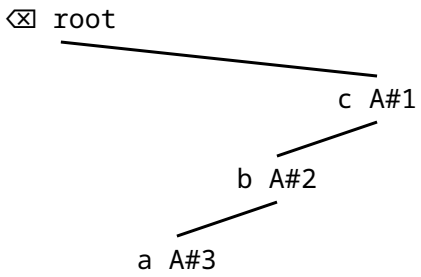


Figure 3.11.: Fugue tree with reverse sequential insertions

Concurrent insert operation

Due to concurrent insertions, it may happen that a node has several right or left children. This proceeds in the following way: To transmit the edits to others, the *node identifier* of the inserted node and its *parent*, the *side* at which it is inserted (left or right) and the *value* that is inserted are transmitted using causal broadcast [25]. Causal broadcast, also referred to as causally-ordered multicasting, ensures that a message with an event, that may have causally happened before another event, is sent before that message [24, 3]. To incorporate remote edits, the received node is added to the own tree based on the parent identifier and side. For example when the edit in Figure 3.6 is concurrent with the edit in Figure 3.7, both clients end up with the tree in Figure 3.8 which has several right children. The order of `"alice"` and `"bob"` is deterministic based on their replica ID but otherwise unspecified. As there is no choice that is inherently better, it is just important that all clients compute the same tree.

Delete operation

Figure 3.9 shows the deletion of a character. The node to delete, which is calculated from the index in the tree traversal of visible nodes, is simply marked as deleted. If it already was deleted by a concurrent user, the operation does nothing.

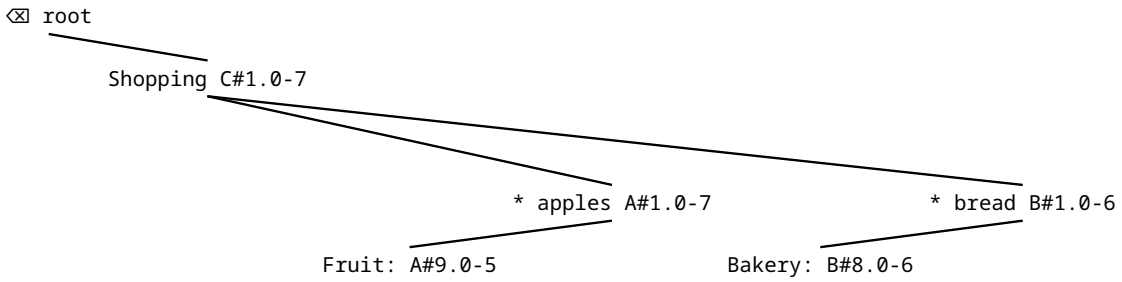


Figure 3.12.: Fugue tree for shopping example

3.4. Intuitive Reason for Avoiding Interleaving

Consecutive insertions as shown in Figure 3.10 and reverse consecutive insertions as shown in Figure 3.11 create a single long branch in the Fugue tree. The intuitive reason this avoids interleaving in these cases is that merging concurrent edits never changes anything within concurrently created trees [25, Section 4].

Another reason is that Fugue prefers linking nodes to their left origin because of forward insertions and if there are already existing nodes it links to the right origin instead to avoid ambiguity. Figure 3.12 shows that this also keeps parts that were inserted together in one subtree. Note that consecutive right children are combined here to make the figure more readable.

4. Implementation of Fugue Algorithm

This chapter first lists the requirements for an implementation of the Fugue algorithm in Section 4.1. Section 4.2 gives insights into our editor implementation in the browser and our P2P functionality and Section 4.3 explains how our synchronization works and is optimized. Section 4.4 introduces our use of property tests to ensure convergence of our implementation and argues that extensive use of assertions for invariants aids in finding the root cause of test failures. Finally, Section 4.5 reveals some small issues in the algorithmic description in the Fugue paper [25].

The source code is available at:
`https://github.com/mohe2015/bachelor-thesis-collaborative-text-editing`

4.1. Required Implementation Functionality

Based on the Fugue paper [25] and our explanation of the Fugue algorithm in the last chapter, an implementation needs to provide the following functionality:

It needs to provide an interface to a tree with left and right children, possibly multiple children on each side but usually only one on one side. It requires fast retrieval of a node based on an index in the non-deleted node traversal and fast retrieval of an index in the non-deleted node traversal based on the node. Additionally, it requires fast retrieval of a node based on its ID. For initial loading it also needs to be able to traverse the whole tree in order. Furthermore, inserting nodes to the right and left of other nodes needs to be efficient with the special case of multiple left or right children.

In practice, trees usually contain many deep right descendants because of consecutive character insertions [25, Figure 5], so this is a case that should be heavily optimized.

```
val schema = Schema(SchemaSpec(orderedmap.from(StringDictionary(
  ("text", NodeSpec()),
  ("doc",
    NodeSpec()
      .setContent("text*")
      .setMarks("")
      .setCode(true)
      .setDefining(true)
      .setParseDOM(
        js.Array(TagParseRule("pre").setPreserveWhitespace(full)))
      .setToDOM(_ => Array("pre", 0)))))))
val hardBreakCommand: Command = (state, dispatch, view) => {
  dispatch.get(state.tr.insertText("\n"))
  true
}
val editorStateConfig = EditorStateConfig().setSchema(schema)
  .setPluginsVarargs(keymap(StringDictionary(("Enter", hardBreakCommand))))
```

Listing 4.1.: Code excerpt of ProseMirror schema setup

4.2. Browser Implementation of Text Editor

To properly use text editing algorithms an editor is required, so we implement an interface to ProseMirror[1] and transpile Scala to JavaScript using Scala.js[2] to be able to use our implementation on the web.

By default, ProseMirror creates newlines using `<br/>` tags and paragraphs using `<p>` tags. This makes it complicated to convert between the ProseMirror document offset and the text offset. Therefore, we configured ProseMirror to only support plaintext and use `\n` for newlines and configured the browser to render `\n` as newlines (which does not work by default) as shown in Listing 4.1.

We also implemented a demo using WebRTC[3] to collaboratively edit a text. It keeps the full history on all connected peers, so it is not possible to permanently delete anything. This is the reason for not implementing persistence, see Chapter 7.

[1] https://prosemirror.net/
[2] https://www.scala-js.org/
[3] https://webrtc.org/

4.3. Synchronization of Changes

The changes are synchronized using causal broadcast as in the Fugue paper [25]. The events are ordered using vector clocks [10, 6]. Only change synchronization updates the vector clock. Therefore, the clock does not need to be updated while working offline, and the changes can be sent in one batch which is more efficient. Instead of creating a message per character insertion or deletion, consecutive deletions and insertions that have the same causality are combined to optimize memory usage.

4.4. Testing Using Property Tests

Property tests are a core part of testing replicated data types (RDTs) as the existence of numerous edge cases make unit testing infeasible. The tests run both on the internal data structure, with an interface for inserting and deleting characters at indices, and on the local web application as a Playwright[4] test.

The property tests run using ScalaCheck[5] and specifically its stateful testing support[6] using `Commands`[7]. ScalaCheck `Commands` store a system under test and a state that is compared to the system under test. Possible actions are defined by implementing the `Command`[7] trait. The trait has several methods for pre conditions, post conditions, running the action and calculating the next state. ScalaCheck generates `Commands` and their contents using Generators, e.g. `Gen.chooseNum(0, Int.MaxValue)` which are then run by ScalaCheck against the system under test and if failures occur it tries to simplify the failure case.

Our property tests randomly create replicas, synchronize replicas, insert text at a replica or delete text at a replica. Then, they check whether replicas have the same text after they synchronized. Unfortunately it is not easily possible to check *what* the expected text would be as that would need more or less a reimplementation of the synchronization logic, see Chapter 7. We also have property tests that check that local operations match the same operations on a `String`.

[4] `https://playwright.dev/java/`
[5] `https://scalacheck.org/`
[6] `https://github.com/typelevel/scalacheck/blob/main/doc/UserGuide.md#stateful-testing`
[7] `https://github.com/typelevel/scalacheck/blob/main/core/shared/src/main/scala/org/scalacheck/commands/Commands.scala`

While trying out new approaches, implementation mistakes are likely, particularly when more complicated approaches have lots of edge cases. It is really laborious to find the root cause for every test failure to fix edge cases, especially for property tests that do not always produce the smallest possible test case. It helps significantly to add lots of assertions into the code that not only check local conditions like traditional uses of assertions but also check global invariants. Some examples of such assertions are ensuring that parent and child references are symmetric to each other and that insertions and deletions correctly update the positions of all characters. These assertions strongly affect the performance, so they need to be disabled for production use.

Ideally, invariant assertions would be automatically checked after every object creation and modification, but that is not easily possible with Scala. Therefore, they were added manually at relevant places. The tests also detect the bugs without these invariant assertions. The failure then happens at a later time in execution, which complicates finding the root cause, but does not decrease the reliability.

4.5. Issues in the Algorithmic Description

While working on our implementation, we found that the algorithmic description [25, Algorithm 1] is, for the most part, satisfactory. However, it contains one large issue. While the Fugue paper includes the conversion from character offsets to their internal representation, it misses the reverse direction [25, Algorithm 1]. Received operations also need to be converted to the index to update the local text editor. Therefore, we extended the algorithmic description with that. This is not just relevant for implementation but also for optimization, which we address in the next chapter. It means that further functionality is required, that can map a node ID to the position in the tree traversal of visible nodes, which is the visible text. The remaining issues were only minor or instances of suboptimal specification.

First, the ID type [25, Algorithm 1] can always be `null`. As this can only be the case for the root node, we moved this case to the places where the root node could potentially be used. There are some places where this could *not* be the case, e.g. remote insertions can not send the root node as the root node is always locally created.

Second, in line 10 of the description [25, Algorithm 1], root is initialized with a value that is invalid according to their specification because the side can only be `L` or `R` but never `null` according to the types. Our implementation arbitrarily chooses the root node to be

on the right side to simplify checks at other places in the code. An alternative would be to use an enumeration for the node and not have an ID, value, side and parent for the root node at all.

Third, each node does not necessarily need to store the ID of its parent and children [25, Algorithm 1]. It could also store a reference directly to them.

Lastly, the node after `leftOrigin` in line 24 [25, Algorithm 1] can be retrieved as the leftmost descendant of the first right child of the `leftOrigin`. The leftmost descendant is the node that is reached by repeatedly descending into the leftmost child until there are no left children. This is logical as the next node must be in the right subtree and there the first node is the leftmost node. Depending on the implementation that may be faster or easier.

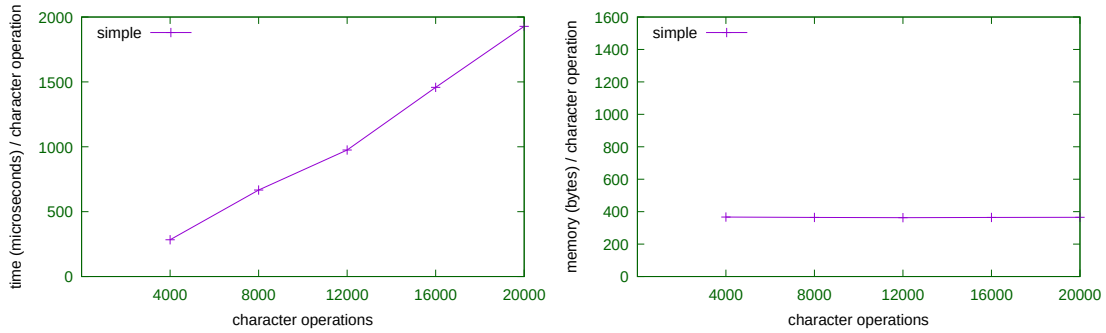
5. Optimizing Common Edit Operations

Based on a theoretical understanding of our base implementation developed from the algorithmic description in the Fugue paper [25, Algorithm 1] we expect quadratic runtime complexity and linear memory usage in relation to the text length. This chapter discusses the implementation of benchmarks to verify the theoretical understanding of the runtime and memory complexity and then proposes optimizations for the implementation based on them.

The Fugue paper already proposes an optimization, but does not go into detail. It proposes to condense sequentially-inserted tree nodes into a single "waypoint" object instead of using one object per node [25, Section 5] but even with their implementation[1] available, their exact approach is unclear. To avoid premature optimization we analyze the performance and optimize based on that.

The benchmarks also indicate a quadratic runtime complexity for our base implementation. In Section 5.1, we start with an optimization that combines consecutively inserted characters as that is how most text is written. This leads to good performance for sequentially written text but still quadratic runtime performance for text with realistic editing behavior like corrections and later additions. In Section 5.2, we create a look-up data structure that can quickly convert between text positions and nodes as that is the main performance bottleneck in the base implementation. This results in $O(log(n))$ runtime complexity per operation. Then, we combine both approaches to reduce memory usage using the batching optimization. This leads to the common case being well optimized, but there are still cases that can be quadratic for text of some length. In Section 5.4 we investigate these performance edge cases, and develop optimizations for them to ensure good performance in all cases. This is important so malicious peers or unusual editing behavior can not lead to unusable runtime performance. Finally, in Section 5.5, we give an overview of the resulting data structure.

[1] `https://github.com/mweidner037/fugue`



(a) time (b) memory

Figure 5.1.: Benchmark results for sequential insertions with the simple algorithm

```
override def atVisibleIndex(i: Int): SimpleTreeNode[V] = {
  factory.nodes().drop(i).iterator.next
}
```

Listing 5.1.: Code excerpt of node search based on index for the simple algorithm

The most basic case is sequential insertion of text which simulates a user that perfectly writes text and never needs to fix any mistakes or add something earlier in the text. Benchmarking our basic implementation called the simple algorithm leads to the result in Figure 5.1.

Note: The graphs show the time and memory *per character operation*, thus the total time to handle the character operations grows quadratically in Figure 5.1. All graphs with the same border color have the same axis scale to make them comparable.

As shown in Appendix A.1 almost all the time is spent in `atVisibleIndex`. This matches the repeated linear search to find the element at which we need to insert based on its index in the original algorithm as shown in Listing 5.1.

```
final case class BatchingTreeNode(
    rid: RID | Null,
    counter: Int,
    var _values: StringBuilder | Null,
    var offset: Int,
    var to: Int,
    side: Side,
    var parent: BatchingTreeNodeSingle | Null,
    var leftChildrenBuffer: mutable.ArrayBuffer[BatchingTreeNode],
    var rightChildrenBuffer: mutable.ArrayBuffer[BatchingTreeNode],
    var allowAppend: Boolean
)
```

Listing 5.2.: Data structure of batching node

5.1. Optimization Using Batching

The optimization that many algorithms already utilize and that the Fugue authors also have hinted at [25, Section 5], is batching sequential insertions by one peer to reduce metadata and memory overhead. In the following section we describe what is needed for that optimization in detail.

The previously used simple ID for tree nodes consists of a replica ID and a counter. To combine sequential tree nodes by the same replica, an offset is added to be able to address single characters for insert and delete operations. This ID that consists of a replica ID, counter and offset is called a batching ID and our algorithm the batching algorithm.

The algorithm intentionally only optimizes consecutive right children or rather forward insertions as that is the most common case. In all cases this is only a best-effort optimization as operations may not be combinable at all, for example if they are from multiple peers.

Listing 5.2 shows the rough data structure of a node. The `replicaId` and `counter` represent the simple ID part of this node. If the `replicaId` is null, then the value of the `counter` is not relevant. This is the case for the root node. The `_values` reference one `ArrayBuffer` per simple ID, so multiple nodes may reference the same `ArrayBuffer`. This happens when a batching node needs to be split. The `offset` and `to` variables represent which subrange of the `ArrayBuffer` this node represents, so which characters of the text it stores. This means the batching IDs for this node then consist of the simple ID part and each value in the range from `offset` until `to` combined with the character at that index in `_values`. In the tree these are always right children of their predecessor as we optimize forward insertions. The `side` stores if this is a left or right child of its parent, except for the root node where this value does not store anything meaningful. `BatchingTreeNodeSingle` stores a reference to the parent `BatchingTreeNode` combined with the offset into that node at which this node is added. The `leftChildrenBuffer` and `rightChildrenBuffer` store the children in an array. `allowAppend` stores whether appending an element to this node is possible by appending an element to `_values`. This is not allowed for the left part of a split because otherwise batching IDs could be duplicated.

Insert operation To insert an element there are the following cases.

Case 1: Insert to the right at the right edge of a non-deleted node with the same replica ID where appending is allowed and which does not already have right children This is the easiest and fastest case. It only consists of adding the value to the array of values.

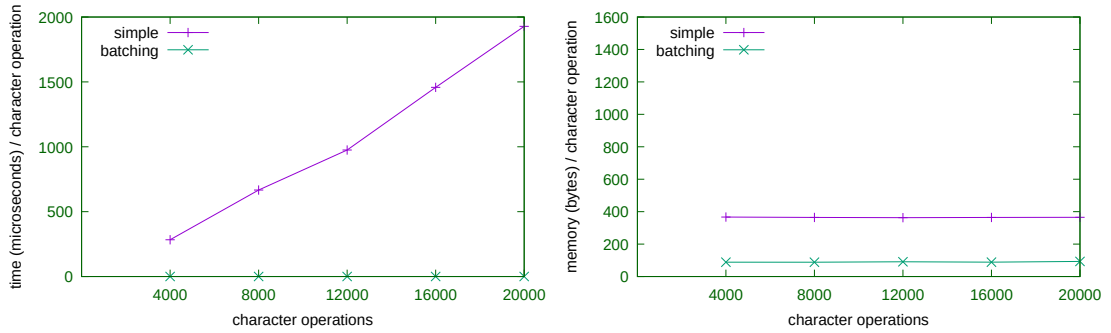
Case 2: Insert to the right at the right edge of a non-deleted node with the same replica ID and counter where appending is allowed but which already has right children Directly appending here is disallowed because otherwise the already existing right children would be at the wrong position. Therefore, add a new right child node that references the existing buffer with correct `offset` and `to` values.

Case 3: Insert to the right at the right edge In this case a new node is added as a right child of the existing node.

Case 4: Insert to the left at the left edge In this case a new node is added as a left child of the existing node.

Otherwise: In the other cases, so ”Insert to the right not at the right edge” and ”Insert to the left not at the left edge” the node needs to be split and inserted at the correct location. Further details about splitting can be found in Section 5.4. As later optimizations combine sequential *deletions*, this also needs to be handled correctly.

Delete operation If an element is already deleted because of concurrent actions, nothing needs to be done. Note that also the editor then does not need any updates. Deletion generally needs to split a node into up to three parts (except if the first or last element is deleted) as there needs to be a node for the part before the deleted element, a node for the deleted element and a node for the part after the deleted element. Later optimizations avoid this for sequential forward and backward deletions by the same replica if both nodes have the same simple ID. Instead, the deleted element is moved to the node containing the other already deleted elements if the parent node has no other right children.



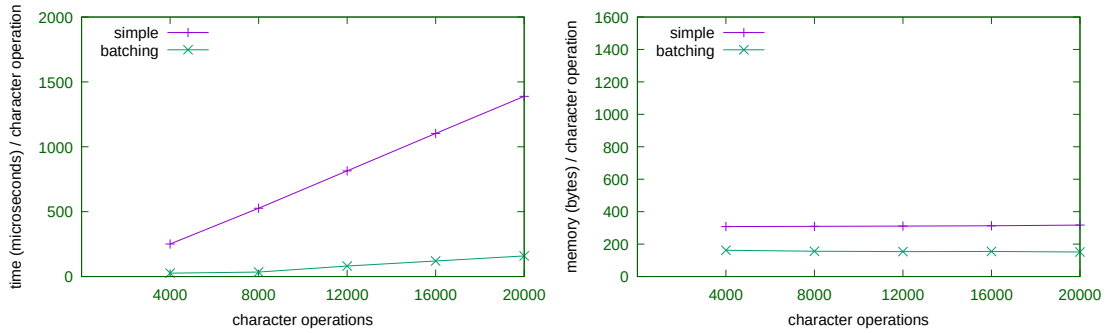
(a) time (b) memory

Figure 5.2.: Benchmark results for sequential insertions comparing the simple algorithm and the batching algorithm

Results for sequential insertions Benchmarking the sequential insertions produces the results in Figure 5.2. The reason the batching algorithm is so fast in comparison to the simple algorithm is that it mainly needs to append to an `ArrayBuffer` for sequential insertions.

Even though every character insertion only needs to append a character to an `ArrayBuffer`, the memory usage per character is about 100 bytes. This is because it also stores the causal history which is required for properly syncing between peers but is only optimized in the final version later.

The CPU profile in Appendix A.2 shows that most time is spent in garbage collection. This indicates that allocating elements for the nodes and messages and resizing ArrayBuffers requires extensive CPU time. The profile shows the CPU time, so this affects the realtime less on a multithreaded system than on a single threaded system. Garbage collection makes it harder to optimize the code as the garbage collector creates a non-local performance bottleneck. It may be helpful to look at the allocation profile in Appendix A.3. There are some things like allocations of temporary values for iterators and views that can be optimized away. In our experience this only leads to limited improvements though. It would be easier to use a programming language that does not use a garbage collector or probably not even a JIT compiler to optimize the algorithm to that depth. Still, Scala, Java and the JVM are well-suited to look at the asymptotic performance because memory allocation or cyclic data structures do not need to be considered in contrast to low level languages like C++ or Rust.

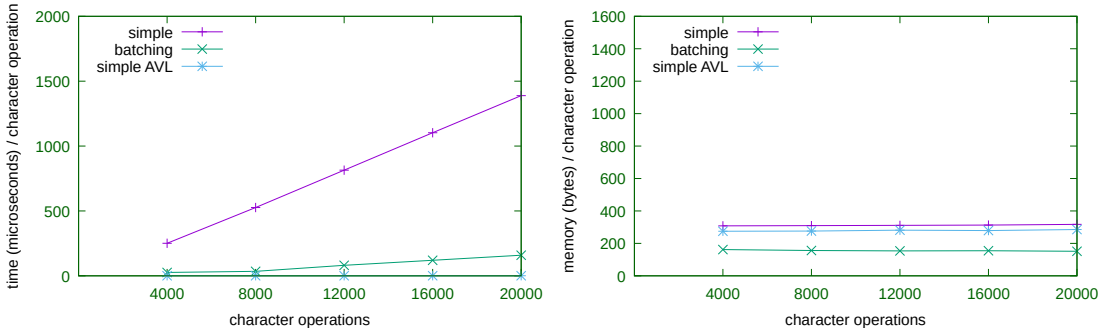


(a) time (b) memory

Figure 5.3.: Benchmark results for real world editing trace comparing the simple algorithm and the batching algorithm

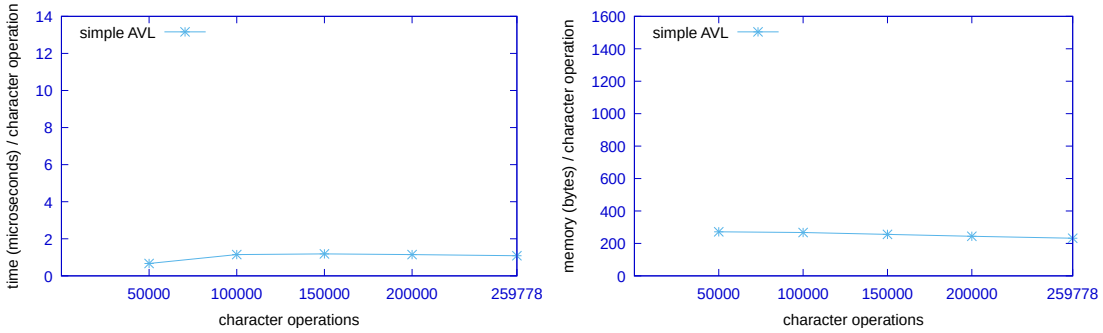
Results for real world editing trace While this results in good performance, it clearly does not cover real world editing behavior. Therefore, we use the dataset from `https://github.com/automerge/automerge-perf` which contains 259,778 insertion and deletion operations that produce a text with 104,852 characters. It is the editing trace from the LaTeX source of `https://arxiv.org/abs/1608.03960`.

Figure 5.3 shows the runtime *per operation* grows linearly and is also extremely slow for only a few tens of thousands of characters. Appendix A.4 shows that most time is spent in `findElementAtIndex` similar to the simple sequential insertions. This is because the batching only helps to improve the performance by some factor that is correlated with the size of consecutive insertions. We therefore looked into an approach that fixes the root cause which is the search of the node in the tree that represents the character at a position in the text.



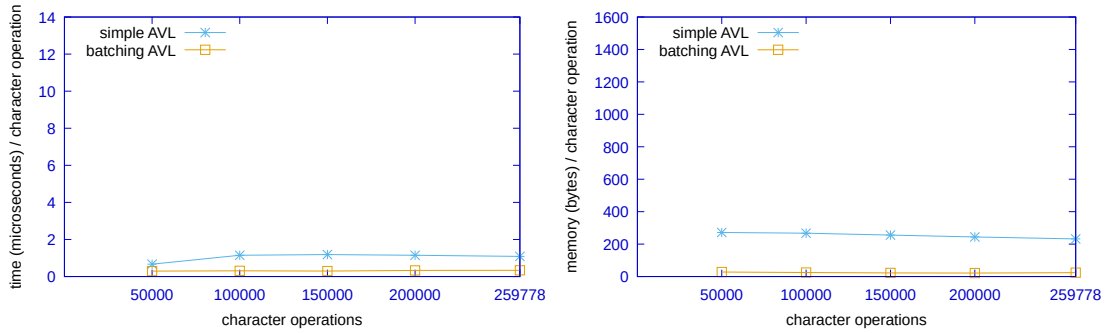
(a) time (b) memory

Figure 5.4.: Benchmark results for real world editing trace comparing the simple algorithm, the batching algorithm and the simple AVL algorithm



(a) time (b) memory

Figure 5.5.: Benchmark results for real world editing trace with the simple AVL algorithm



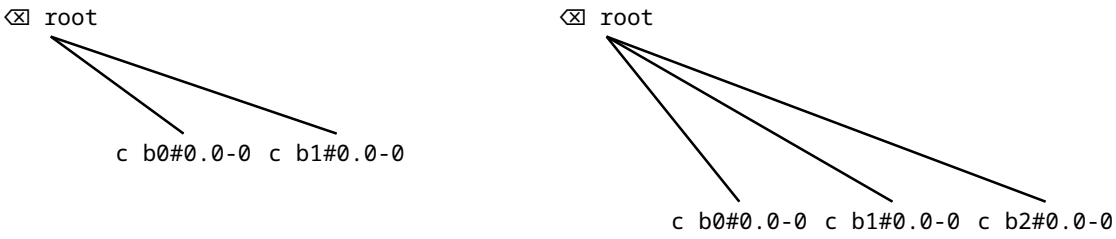
(a) time (b) memory

Figure 5.6.: Benchmark results for real world editing trace comparing the simple AVL algorithm and the batching AVL algorithm

5.2. Optimization Using a Look-Up Datastructure

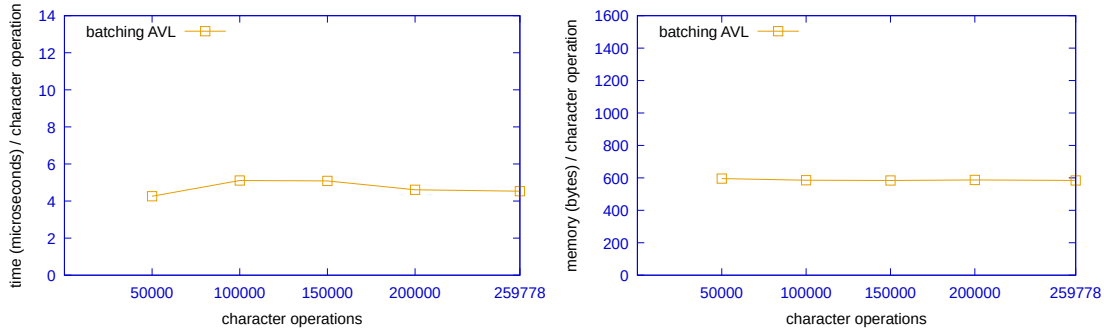
For this optimization a data structure is needed, that can quickly retrieve the node based on its index in the text and also allows quick insertions and deletions at arbitrary positions. This is similar to a binary search tree with the difference that the index of a node shifts when inserting a node to the left of it. Therefore, instead of storing the index of a node, it stores the size of all (visible) subnodes in the search tree. Then a binary search on that size finds the insert position. This also means that an insertion needs to update all sizes up to the root. An AVL tree was chosen as the binary search tree because it has logarithmic asymptotic complexity in all cases and more complex and potentially faster binary search trees such as B-trees do not have better asymptotic complexity. The batching optimization is excluded to be able to isolate the performance changes to the algorithmic changes.

This results in a very low time per character operation as shown in Figure 5.4 in comparison to the two other approaches with the real world benchmark. As it is not possible to read the values for the simple AVL algorithm there, Figure 5.5 shows only the simple AVL algorithm with the full text, so much more operations, and a different y-axis scale. The CPU profile in Appendix A.5 shows that there is not a single hot location, but execution is distributed over many methods. The memory overhead is still very high, because a new node in the AVL tree and the Fugue tree needs to be created for every character. Figure 5.5 shows a memory usage of about 250 bytes per character operation. Note that this also includes the full insertion and deletion history and not only the tree itself.



(a) before (b) after

Figure 5.7.: Example for edge case with many children

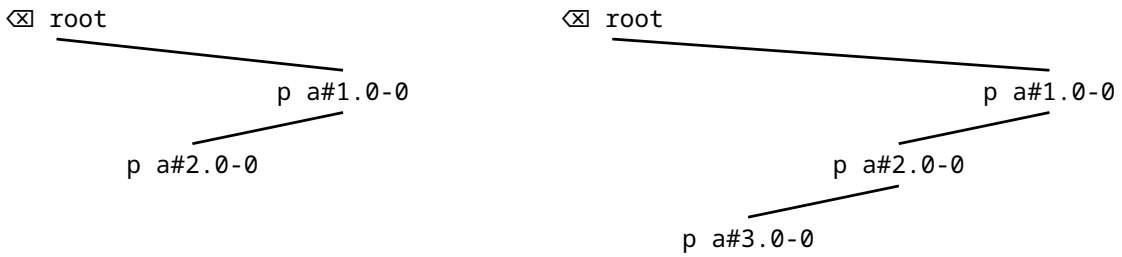


(a) time (b) memory

Figure 5.8.: Benchmark results of an edge case with many children

5.3. Combined Optimizations

Combining the AVL tree optimization and node batching improves the memory usage and runtime. The results are shown in Figure 5.6 for the real world benchmark. The runtime per operation is one microsecond, thus one million operations can be handled per second. The memory usage per operation is about 25 bytes per operation. This concludes our optimization of the common execution path.



(a) before (b) after

Figure 5.9.: Example for edge case for insertion to the left of the root

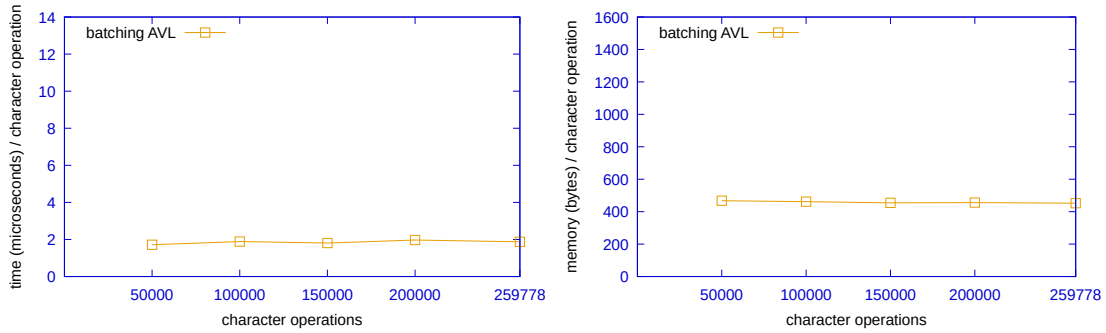
```
val firstRightChild = leftOrigin.firstRightChild()
var side: Side | Null = null
val origin = if (firstRightChild == null) {
  side = Side.Right
  leftOrigin
} else {
  side = Side.Left
  firstRightChild.leftmostDescendant()
}
```

Listing 5.3.: Code excerpt of an edge case for insertion to the left of the root

5.4. Performance Edge Cases

An optimal algorithm must perform efficiently in *all* cases. Therefore, efficiently handling edge cases is essential. This is important because remote users can send arbitrary operations. Therefore, a malicious user could use that to attack the algorithm and render the text editing unusable. The following are specific cases for our algorithm. Other algorithms need to be analyzed case by case.

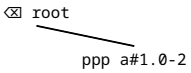
Edge case with many children Child insertions need to be efficient even after many children are inserted at the same side of the same node as shown in Figure 5.7 with the benchmark results in Figure 5.8. Therefore, the children are stored in a `mutable.SortedSet`, so a binary search tree. This results in logarithmic insertion.



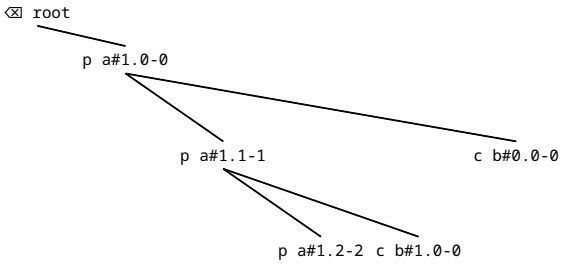
(a) time (b) memory

Figure 5.10.: Benchmark results of an edge case for insertion to the left of the root

Edge case for insertion to the left of the root Another case is repeatedly inserting at position 0 as shown in Figure 5.9 with the benchmark results in Figure 5.10. As the root node has a right child after the first insertion, further nodes need to be inserted to the left of that child. To find the node before the child our algorithm retrieves the leftmost descendant of it as shown in Listing 5.3. This requires a recursive traversal down the leftmost child, which is a linear operation. Therefore, our algorithm uses a cache for the leftmost descendant of every node in the tree. As all nodes in the path from the node to its leftmost descendant have the same leftmost descendant, one cache is used for this group of nodes. As shown later, it needs to be possible to split the cache up, if a child is inserted somewhere in that path to the left. The cache also uses an AVL tree with the specialty of storing a parent reference in each AVL tree node and the root node storing a reference to the leftmost descendant of all nodes of that AVL tree. Therefore, the leftmost descendant of this group of nodes can be efficiently retrieved and updated, the cache can be efficiently split up by splitting the AVL tree and new nodes can be efficiently inserted.

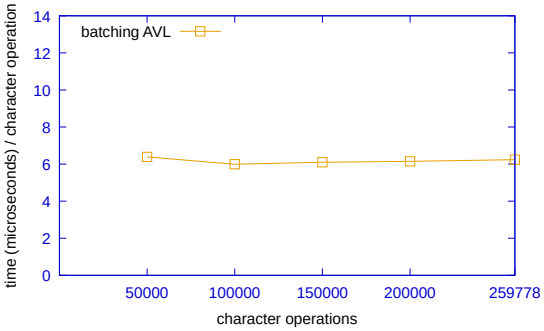


(a) before

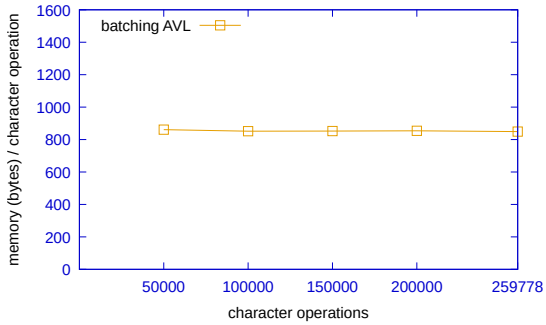


(b) after

Figure 5.11.: Example for edge case for concurrent insertion to the right



(a) time

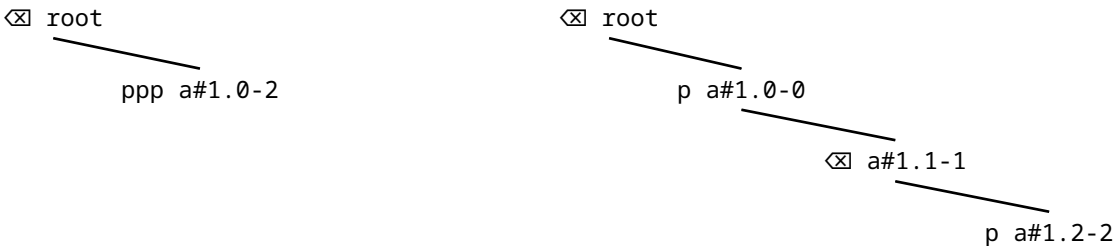


(b) memory

Figure 5.12.: Benchmark results of an edge case for concurrent insertion to the right

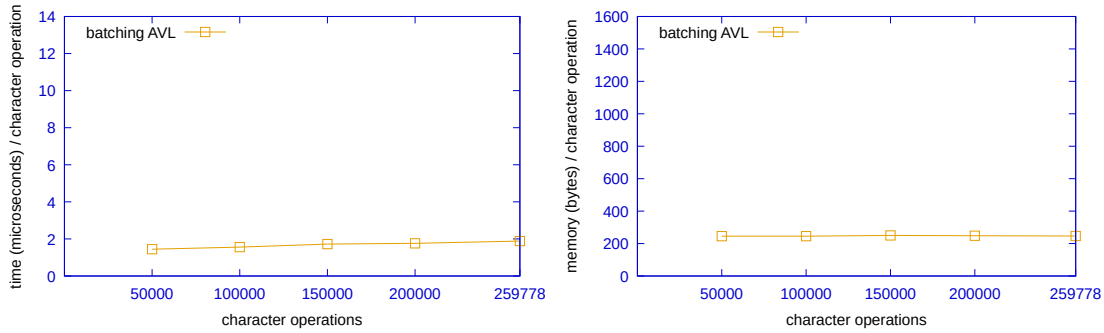
```
val base = if (rightChildrenBuffer.nn.isEmpty || before.isEmpty) {
  parent
} else {
  BatchingAVLTreeNodeSingle(before.get, before.get.value.to)
    .rightmostDescendant().complexTreeNode
}
```

Listing 5.4.: Code excerpt of an edge case for concurrent insertion to the right



(a) before (b) after

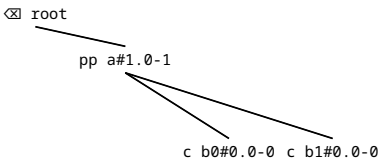
Figure 5.13.: Example for edge case for node splitting



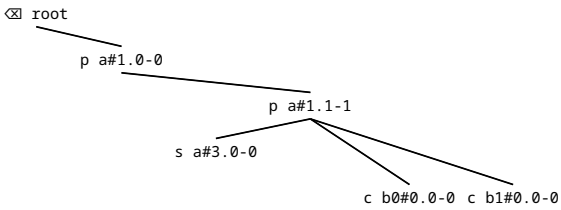
(a) time (b) memory

Figure 5.14.: Benchmark results of an edge case for node splitting

Edge case for concurrent insertion to the right In the edge case in Figure 5.11 with the benchmark results in Figure 5.12 the p nodes were first inserted and then c nodes were inserted concurrent to them. This means for every c node insertion, the node needs to be inserted at the correct position in the AVL tree to preserve the correct character ordering. For example as this is a concurrent insertion, the first c node needs to be inserted after the subtree of the child to the left of it. Therefore, the last node in the subtree of its left child needs to be retrieved, which requires to get the rightmost descendant of that child as shown in Listing 5.4. Therefore, this also needs the optimization as explained for the previous edge case.

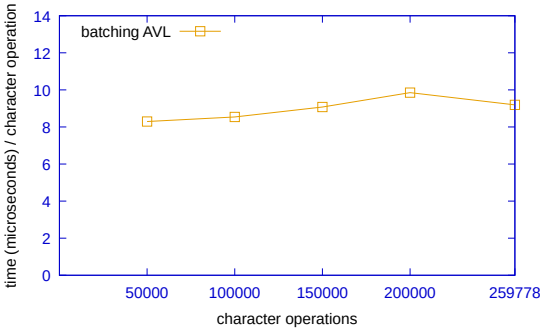


(a) before

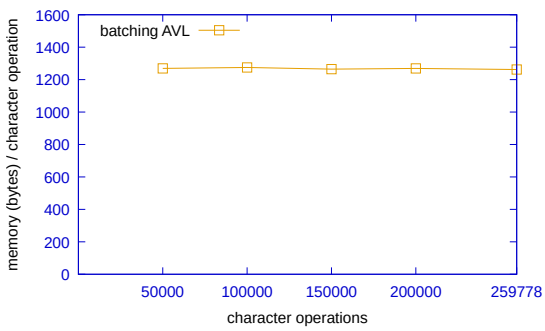


(b) after

Figure 5.15.: Example for edge case for node splitting with many right children



(a) time



(b) memory

Figure 5.16.: Benchmark results of an edge case for node splitting with many right children

Edge case for node splitting Splitting a batched node as shown in Figure 5.13 with the benchmark results in Figure 5.14 needs to be efficiently handled. The consecutive elements are stored in an `ArrayBuffer` and splitting it would be a linear operation. Therefore, instead of splitting it, nodes reference a subpart of the buffer. This means splitting a node only requires creating and inserting a new node and updating a few references to the buffer start and end, inserting it into the AVL tree and updating the descendant cache. The disadvantage is that the memory for deleted nodes is not reclaimed.

Edge case for node splitting with many right children A previous version of the algorithm stored a reference to the parent in each node. Therefore, splitting a node as shown in Figure 5.15 with the benchmark results in Figure 5.16 required updating the parent of all its former children. The parent reference is not required for the batching AVL algorithm, therefore it was simply removed.

```
final case class BatchingAVLTreeNode[V](
  replicaId: RID | Null,
  counter: Int,
  var _values: ArrayBuffer[V] | Null,
  var offset: Int,
  var to: Int,
  side: Side,
  var leftChildrenBuffer: SortedSet[AVLTreeNode[BatchingAVLTreeNode[V]]]
                        | AVLTreeNode[BatchingAVLTreeNode[V]] | Null,
  var rightChildrenBuffer: SortedSet[AVLTreeNode[BatchingAVLTreeNode[V]]]
                        | AVLTreeNode[BatchingAVLTreeNode[V]] | Null,
  var allowAppend: Boolean,
  var leftDescCache: AVL2TreeNode[AVLTreeNode[BatchingAVLTreeNode[V]]],
  var rightDescCache: AVL2TreeNode[AVLTreeNode[BatchingAVLTreeNode[V]]],
)
```

Listing 5.5.: Code excerpt of node data structure for batching AVL algorithm

Closing remarks It is important to note that there is no guarantee this covers all edge cases. Except for formal verification, the most feasible way is to thoroughly look at the source code and check that each possible operation is able to compute in the expected time. Appending to a batched node in our algorithm can be $O(n)$ in the case that the `ArrayBuffer` requires resizing, but our algorithm intentionally targets *amortized* $O(\log(n))$ as it is not relevant if a single operation takes a bit longer. Also, resizing the `ArrayBuffer` is fast as it only consists of a memory copy.

All these data structures also lead to a high per-node memory overhead, so it may be interesting if there are better ways to achieve the same performance goal. Note especially the last edge case where almost 1300 bytes are needed per character operation. Through optimization, probably in an ahead-of-time compiled language and not Scala or another JVM based language, this can probably be reduced at least a bit.

5.5. Node Data Structure Including All Optimizations

In Listing 5.5 we show our node data structure for the batching AVL algorithm that combines the batching with the look-up tree optimization. The fields that are from the batching node data structure shown in Listing 5.2 have the same meaning as explained in Section 5.1. For the look-up tree optimization, the `leftDescCache` and `rightDescCache` store an AVL tree for quickly retrieving the respective descendant. The `leftChildrenBuffer` and `rightChildrenBuffer` use a `SortedSet` to insert nodes in $\log(n)$ and have an optimization for single or no children to save memory. They also store the children in an `AVLTreeNode` for the fast node retrieval using an AVL tree.

6. Evaluation

We chose to evaluate our approach by benchmarking with JMH[1] as that is the de facto Java benchmarking tool. We ran the benchmarks on four Intel Xeon Gold vCPUs with 8 GB RAM rented from Hetzner Cloud[2] (type cx32).

We use the JMH support for async-profiler[3] because async-profiler is not affected by the Safepoint bias problem[4] which can lead to bias in the profiler results. Additionally, its allocation profiling does not influence Escape Analysis[5] or prevent JIT optimizations like allocation elimination and therefore measures only actual heap allocations[3].

The Scala.js output was not considered in the analysis given the inherent challenges arising from the additional layer of indirection created by the transpilation from Scala to JavaScript. This indirection likely affects performance and complicates optimization efforts because they potentially only affect the transpiled version rather than the original. Furthermore, the resulting code from the transpilation is highly unreadable, which makes it difficult to correlate it with the original code especially when it involves standard library functionality. Benchmarking the JavaScript transpiled output would have had the advantage of being able to directly compare with most other research results, as there is a popular framework by Kevin Jahns[6] that many publications use [25, Section 5.1].

For our final benchmarks the generic parameter which specified the type of the elements in the list data structure was removed and specialized for text. This reduces memory usage a bit as Scala otherwise needs to create an object per character. This leads to an overhead because of the required metadata per object and because a character object is two bytes large, but many characters only need a single byte.

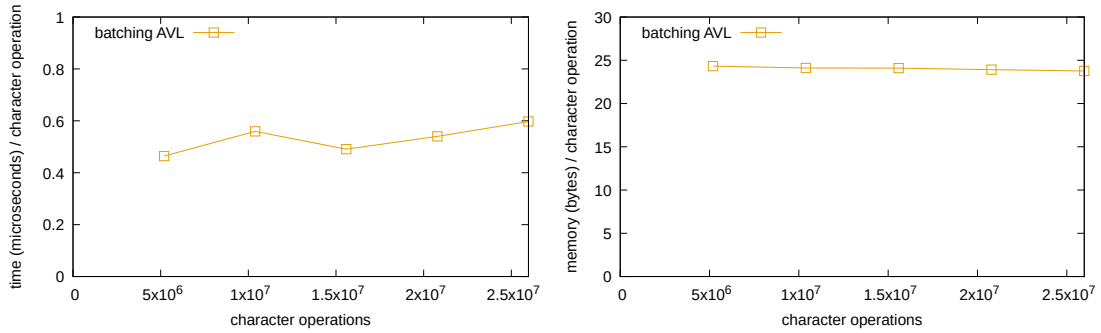
[1] https://github.com/openjdk/jmh
[2] https://www.hetzner.com/cloud/
[3] https://github.com/async-profiler/async-profiler
[4] https://psy-lob-saw.blogspot.com/2016/02/why-most-sampling-java-profilers-are.html
[5] https://blogs.oracle.com/javamagazine/post/escape-analysis-in-the-hotspot-jit-compiler
[6] https://github.com/dmonad/crdt-benchmarks/

```
ManagementFactory
    .getPlatformMBeanServer()
    .nn
    .invoke(
      new ObjectName("com.sun.management:type=DiagnosticCommand"),
      "gcClassHistogram",
      Array[Object | Null](null),
      Array("[Ljava.lang.String;")
    )
```

Listing 6.1.: Code excerpt of memory usage measurement

```
     #instances         #bytes  class name (module)
-------------------------------------------------------
        2957804       94649728  text_rdt.avl2.AVL2TreeNode
        1609332       88413312  [B (java.base@21.0.3)
        1478902       82818512  text_rdt.ComplexAVLTreeNode
        1478902       59156080  text_rdt.avl.AVLTreeNode
        1058700       50817600  text_rdt.ComplexAVLMessage$Insert
        1254001       50160040  scala.collection.mutable.RedBlackTree$Node
        1596802       38323248  java.lang.StringBuilder (java.base@21.0.3)
        1478903       35493672  text_rdt.avl2.AVL2Tree
        1596801       25548816  scala.collection.mutable.StringBuilder
         710800       22745600  text_rdt.ComplexAVLMessage$Delete
         538103       17219296  scala.collection.mutable.HashMap$Node
         538105       12914520  scala.Tuple2
         538101       12914424  text_rdt.SimpleID
           2270        9637384  [Ljava.lang.Object; (java.base@21.0.3)
         313201        7516824  scala.collection.mutable.RedBlackTree$Tree
         313201        7516824  scala.collection.mutable.TreeSet
         182315        4375560  text_rdt.FixtureOperation$Insert
              2        4194368  [Lscala.collection.mutable.HashMap$Node;
          77463        1239408  text_rdt.FixtureOperation$Delete
...
Total  17763864      627984600
```

Listing 6.2.: Memory usage for batching AVL algorithm



(a) time (b) memory

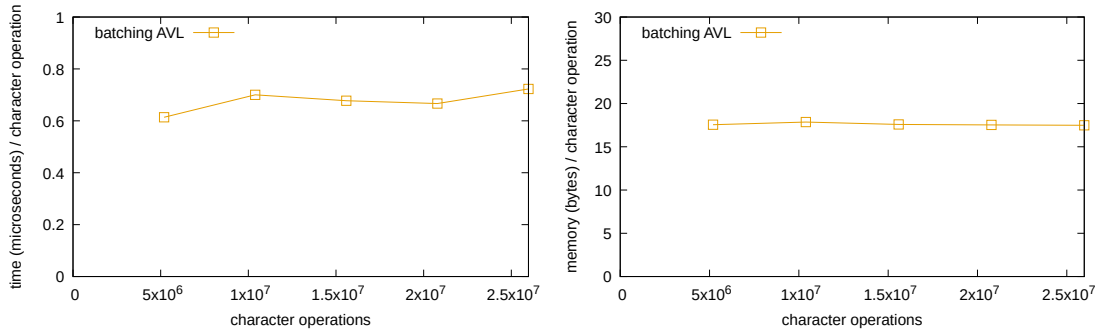
Figure 6.1.: Benchmark results for repeatedly concatenated real world text inserted locally with the batching AVL algorithm

6.1. Measuring Maximum Memory Usage

The memory usage is calculated using the code in Listing 6.1, which is equivalent to `jcmd PID GC.class_histogram`. It is measured before and after running the operations and the difference is then visualized in our graphs. The memory usage is returned using JMH `AuxCounters`[7] to ensure it is measured for exactly the same case as the CPU benchmarks.

For the 100 times consecutively written real-world benchmark the memory usage is as shown in Listing 6.2. The `avl2` types are used for the leftmost and rightmost descendant cache which indicates that optimizing these would improve memory usage considerably, see Chapter 7. The `ComplexAVLTreeNode` is created for every batched node and the `AVLTreeNode` is needed for the AVL lookup tree and also created for every batched node. The byte arrays (`[B`) in combination with `StringBuilder` are used to store the underlying text. The `ComplexAVLMessage` stores the history of all messages. The `HashMap$Node`, `[Lscala.collection.mutable.HashMap$Node`, `Tuple2` and `SimpleID` are used to associate IDs with the respective nodes. The `RedBlackTree` and `TreeSet` are used for multiple same-side children and for quickly retrieving the correct node when a batching node has been split. The `FixtureOperation` is the underlying test data and therefore does not count towards the memory usage when measuring the memory usage difference before and after running the test.

[7] `https://github.com/openjdk/jmh/blob/master/jmh-core/src/main/java/org/openjdk/jmh/annotations/AuxCounters.java`



(a) time (b) memory

Figure 6.2.: Benchmark results for repeatedly concatenated real world text inserted remotely with the batching AVL algorithm

6.2. Results

Section 5.4 already looked at performance edge cases and artificial cases which are important to cover in the context of decentralized algorithms, so there is no case that could severely reduce the performance of the algorithm which could lead to it becoming unusable. Otherwise, edit actions that hit such an edge case by chance, attackers, or even just a large amount of activity could lead to this slowdown.

Figure 6.1 shows the real world text repeated 100 times to match benchmark B4x100 from Jahns benchmark framework[8]. Figure 6.2 shows the same but simulating that the edit operations are received from a remote replica. Both benchmarks show that operations are performant independent of the text size with one microsecond per operation. Memory usage is acceptable with about 25 bytes per operation but could likely be improved further.

A real world editing trace with *concurrent edits* in an offline context would be useful to analyze performance in that case, but unfortunately we are not aware of such a dataset. As our algorithm has a time complexity of $O(n \log(n))$ for n character operations *in all cases* this would only allow more accurate measurements, for example for the expected memory usage per operation.

[8] `https://github.com/dmonad/crdt-benchmarks/`

6.3. Investigating Prior Benchmarks

The prior benchmarks based on Jahns benchmark framework[9] have several issues. First, they do not give any indication about asymptotic behavior as they are only executed with one relatively small choice for N which parameterizes the repetition of operations or client count. Optimizing asymptotic behavior is much harder in general than achieving acceptable performance for the common choice of $N = 6000$ on modern CPUs with multiple billion instruction cycles per second. Also, they do not use a trusted benchmark framework but use self-written warmup and benchmark code which is likely affecting the accuracy of the benchmark as they run in the context of a JIT compiler similar to the JVM. The JMH framework is designed to have as accurate results as possible.

[9] `https://github.com/dmonad/crdt-benchmarks/`

7. Future Work

In this chapter we look at what is missing and which aspects could be researched further.

Investigating OT Algorithms In their review of the Fugue paper, Sun shows that the claims in the Fugue paper [25] about OT being interleaving are not correct [19, 23, 22, 20]. First, they show that mistakes were made in the Fugue paper when applying the OT algorithms which render their results regarding OT invalid [23, 22]. They also show that interleaving has been examined and documented before and can be solved in OT, usually by having operations based on strings and not single characters, but this is also possible when operating on single characters [22]. Therefore, investigating OT algorithms, especially in a *non-realtime* setting could be interesting.

Necessary Non-interleaving Properties for Intent-Preserving Text Editing The review by Sun also suggests that not all the properties that are proposed in the Fugue paper (especially multi-user relay interleaving and backward interleaving) are necessary or useful for user intent preserving text editing [22]. While the examples we show in Section 2.1 and Section 2.2 are realistic, we do not know which properties are strictly necessary, as required properties seriously limit the freedom in the design of suitable algorithms. For example, the Fugue paper proposes a property of maximally non-interleaving that produces a unique order with the least possible interleaving. While it is interesting that this property produces a unique order it is unclear whether this is useful in practice. Our example in Section 2.2 also shows that this property is not sufficient for non-interleaving when deletions are involved. Future work could investigate how this property could be adapted to better model non-interleaving in such cases.

Privacy One big problem we see with all these algorithms is that it is hard or impossible to properly delete data in case a user wishes to do so while still being able to converge and preserve user intentions. Future work could investigate which possibilities exist to actually remove deleted text. One possibility could be to clear the deleted characters in the tree. This would also work for the causal broadcast messages but then undo would not be possible anymore. Therefore, maybe more control is needed for end users whether they want to do a normal deletion or a permanent deletion, which would break undo, and also to see which data is still visible in the internal data structure or in the message log.

As the messages are only required to be processed by the peers themselves, adding encryption should be comparatively easy. This could also route messages over a server and store them there without the server being able to read the contents. Some thought should still be put into what can be inferred from metadata like message timing and size. For example, it would likely be possible for the server to find out which user writes how many characters at what time.

Correctness Currently, there is little protection against messages that do not conform to the expected rules. For example if two peers send different characters with the same ID this will create inconsistencies or potentially also crashes. Also peers can easily send characters for other peers as the peer ID is not verified to be only used by the respective peer. This should be tested more, for example using fuzzing tests that can send arbitrary messages that do not conform to the rules. Also, inconsistencies by different characters with the same ID should be avoided, for example by making the character part of the ID.

While our property tests seemed to find all relevant issues, they were pretty limited for tests with multiple replicas and could not check the exact expected outcome in that case. Therefore, it may be interesting to find ways to more thoroughly test this while also testing non-interleaving.

Usability Rich text is probably the largest missing feature that may also lead to many design challenges. First, there is inline formatting like bold, underlined, italic, strike-through, subscript or superscript text. But there is also structural formatting like headings, subheadings, ordered and unordered lists, tables, etc. Both create new challenges with user intent. While for OT there is a lot of previous work which is also successfully used in production e.g. Google Docs[1], for CRDTs there is not much previous research [9]. The Peritext paper [9] investigates inline formatting and shows some problems in prior

[1] `https://www.google.com/docs/about/`

algorithms with correctly preserving user intentions [9]. For example the Yjs algorithm based on Yet Another Transformation Approach (YATA) [12] adds markers where inline formatting starts and where it ends into the text. This fails to handle a simple case where a bold text is unbolded and concurrently part of that bold text is unbolded which then leads to unrelated text getting bold [9, Section 2.3.2].

In a collaborative context it needs to be possible to undo arbitrary actions by any user and not only the last action like it is usually the case in traditional editors. Therefore, support for so-called selective undo is needed. For OT algorithms, transformations need to be applied to the correct document context [16]. This means the control algorithms need to properly handle this and transformation functions potentially need to uphold specific properties [16].

When part of a text is moved and concurrently part of that text is edited it would make sense that these edits are correctly preserved. As normal copy and paste does not track this state this needs a special operation or needs to store the necessary metadata in the clipboard. Also, this needs support at the CRDT level [1, 5].

Instead of operating on a character level it could make sense to operate on a string level. This would be more efficient and could have better semantics for range deletions, copy and paste or moving text. For OT this seems to often be done but is much more complicated, especially in combination with undo [21].

While we did not look at this in this thesis, it is not hard to serialize and deserialize our representation. It may be interesting to find out which parts of the data structures, that are only needed to improve lookup performance, should be persisted to storage and which parts can be quickly rebuilt on loading.

Performance While in some cases the full editing history needs to be kept to be able to attribute all changes, in other cases it can be reduced as much as possible without causing causality problems. The approach of the antimatter[2] algorithm is to combine operations that have been seen by the same group of peers by tracking acknowledgements. In case peers go offline but come online at some point later it can potentially still combine operations.

Our current performance measurements only test non-concurrent actions. It may be beneficial to either find or create some real-world editing trace with concurrent actions or generate some artificial trace like in YATA [12, Section 6.1].

[2] `https://web.archive.org/web/20240623153539/https://braid.org/antimatter`

The memory usage per character is pretty high, even for the real world benchmark. Except for using a low-level language it could also make sense to investigate how to only create the cache for the leftmost and rightmost descendant if they are deeply nested which based on Section 6.1 would likely save large amounts of memory.

When the data is larger than the available memory, our algorithm currently can only be used with swapping. Future work could look into alternatives, for example to store currently not edited parts to disk.

8. Conclusion

This thesis shows that efficient collaborative plain text editing in a decentralized and *non-realtime* setting while preserving user intentions is possible. The optimization to logarithmic runtime per operation in relation to the text length ensures that this is also efficient for extremely large text. This thesis also shows that prior benchmarks do not measure asymptotic complexity and do not cover all algorithmic performance edge cases and proposes to include both in future benchmarks. This is especially an issue in decentralized networks, as there is only limited control over all messages and peers can send you messages with malicious content that triggers these edge cases.

The WebRTC implementation shows a practical example of text editing in P2P networks and allows easy experimentation.

Section 2.1 shows that interleaving for the *maximally non-interleaving* property [25] is indeed possible when deletions are involved. Therefore, a more accurate property should be researched to ensure non-interleaving.

Significant parts that are common in text editing are still missing, the largest being rich text support. Rich text support likely leads to further implementation and optimization challenges, and it is not clear whether these are solvable while preserving the same asymptotic complexity in all cases. Additionally, the preservation of user intentions of formatting actions likely has similar challenges as ensuring non-interleaving has. The interaction of rich text and being able to undo arbitrary actions likely also poses further challenges.

While testing whether the algorithm converges is comparably simple, testing intent preservation and non-interleaving without reimplementing the algorithm in the test is challenging. As testing is a critical part to ensure correctness, more focus needs to be put on testing text editing algorithms.

Acknowledgments

I would like to thank everyone who reviewed drafts of this thesis. I would also like to thank my human and non-human rubber ducks for their help in debugging my code.

Acronyms

CRDT conflict-free replicated data type 8–10, 13, 16, 17, 52, 53

DTN delay tolerant network 7, 8

MANET mobile ad hoc network 7

OT operational transformation 8–10, 16, 17, 51–53

P2P peer-to-peer 7, 24, 55

RDT replicated data type 26

RGA Replicated Growable Array 8, 17

WOOT WithOut Operational Transforms 8

YATA Yet Another Transformation Approach 53

Bibliography

[1] Parwat Singh Anjana, Adithya Rajesh Chandrassery, and Sathya Peri. “An Efficient Approach to Move Elements in a Distributed Geo-Replicated Tree”. In: *IEEE 15th International Conference on Cloud Computing, CLOUD 2022, Barcelona, Spain, July 10-16, 2022*. Ed. by Claudio Agostino Ardagna et al. IEEE, 2022, pp. 479–488. DOI: 10.1109/CLOUD55607.2022.00071. URL: https://doi.org/10.1109/CLOUD55607.2022.00071.

[2] Jim Bauwens, Kevin De Porre, and Elisa Gonzalez Boix. “[Short paper] Towards improved collaborative text editing CRDTs by using Natural Language Processing”. In: *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2023, Rome, Italy, 8 May 2023*. Ed. by Elisa Gonzalez Boix and Pierre Sutra. ACM, 2023, pp. 51–55. DOI: 10.1145/3578358.3591330. URL: https://doi.org/10.1145/3578358.3591330.

[3] Kenneth Birman, André Schiper, and Pat Stephenson. “Lightweight causal and atomic group multicast”. In: *ACM Trans. Comput. Syst.* 9.3 (1991), pp. 272–314. ISSN: 0734-2071. DOI: 10.1145/128738.128742. URL: https://doi.org/10.1145/128738.128742.

[4] Loïck Briot, Pascal Urso, and Marc Shapiro. “High Responsiveness for Group Editing CRDTs”. In: *Proceedings of the 2016 ACM International Conference on Supporting Group Work*. GROUP ’16. Sanibel Island, Florida, USA: Association for Computing Machinery, 2016, pp. 51–60. ISBN: 9781450342766. DOI: 10.1145/2957276.2957300. URL: https://doi.org/10.1145/2957276.2957300.

[5] Liangrun Da and Martin Kleppmann. “Extending JSON CRDT with Move Operations”. In: *CoRR* abs/2311.14007 (2023). DOI: 10.48550/ARXIV.2311.14007. arXiv: 2311.14007. URL: https://doi.org/10.48550/arXiv.2311.14007.

[6] Colin J. Fidge. *Timestamps in Message-Passing Systems That Preserve the Partial Ordering*. 1988.

[7] Martin Kleppmann et al. “Interleaving anomalies in collaborative text editors”. In: *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2019, Dresden, Germany, March 25-28, 2019*. ACM, 2019, 6:1–6:7. DOI: 10.1145/3301419.3323972. URL: https://doi.org/10.1145/3301419.3323972.

[8] Martin Kleppmann et al. “Local-first software: you own your data, in spite of the cloud”. In: *Onward!* ACM, 2019, pp. 154–178.

[9] Geoffrey Litt et al. “Peritext: A CRDT for Collaborative Rich Text Editing”. In: *Proc. ACM Hum. Comput. Interact.* 6.CSCW2 (2022), pp. 1–36. DOI: 10.1145/3555644. URL: https://doi.org/10.1145/3555644.

[10] Friedemann Mattern. *Virtual Time and Global States of Distributed Systems*. 1988.

[11] David A. Nichols et al. “High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System”. In: *ACM Symposium on User Interface Software and Technology*. ACM, 1995, pp. 111–120.

[12] Petru Nicolaescu et al. “Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types”. In: *Proceedings of the 19th International Conference on Supporting Group Work, Sanibel Island, FL, USA, November 13 - 16, 2016*. 2016, pp. 39–49. DOI: 10.1145/2957276.2957310. URL: https://doi.org/10.1145/2957276.2957310.

[13] Gérald Oster et al. “Data consistency for P2P collaborative editing”. In: *Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006, Banff, Alberta, Canada, November 4-8, 2006*. Ed. by Pamela J. Hinds and David Martin. ACM, 2006, pp. 259–268. DOI: 10.1145/1180875.1180916. URL: https://doi.org/10.1145/1180875.1180916.

[14] Hyun-Gul Roh et al. “Replicated abstract data types: Building blocks for collaborative applications”. In: *J. Parallel Distributed Comput.* 71.3 (2011), pp. 354–368.

[15] Chengzheng Sun et al. “Real Differences between OT and CRDT under a General Transformation Framework for Consistency Maintenance in Co-Editors”. In: *CoRR* abs/1905.01518 (2019). arXiv: 1905.01518. URL: http://arxiv.org/abs/1905.01518.

[16] David Sun and Chengzheng Sun. “Context-Based Operational Transformation in Distributed Collaborative Editing Systems”. In: *IEEE Trans. Parallel Distributed Syst.* 20.10 (2009), pp. 1454–1470. DOI: 10.1109/TPDS.2008.240. URL: https://doi.org/10.1109/TPDS.2008.240.

[17] David Sun et al. “Real Differences between OT and CRDT in Building Co-Editing Systems and Real World Applications”. In: *CoRR* abs/1905.01517 (2019). arXiv: 1905.01517. URL: http://arxiv.org/abs/1905.01517.

[18] David Sun et al. “Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors”. In: *CoRR* abs/1905.01302 (2019). arXiv: 1905.01302. URL: http://arxiv.org/abs/1905.01302.

[19] Dr. Chengzheng Sun. *A Critical Examination of “the Fugue Paper” in Relation to OT*. 2023. URL: https://web.archive.org/web/20240603141053/https://medium.com/codox/a-critical-examination-of-the-fugue-paper-in-relation-to-ot-157f6ccaed95 (visited on 06/03/2024).

[20] Dr. Chengzheng Sun. *Dispelling Misconceptions in the Fugue Paper about GOT and OT*. 2023. URL: https://web.archive.org/web/20240603143419/https://medium.com/codox/dispelling-misconceptions-in-the-fugue-paper-about-got-and-ot-16e362609f6f (visited on 06/03/2024).

[21] Dr. Chengzheng Sun. *Operational Transformation Frequently Asked Questions and Answers*. 2024. URL: https://web.archive.org/web/20240603145105/https://www3.ntu.edu.sg/scse/staff/czsun/projects/otfaq/ (visited on 06/03/2024).

[22] Dr. Chengzheng Sun. *Unveiling Issues with the Fugue Paper Regarding Jupiter-OT*. 2023. URL: https://web.archive.org/web/20240603142535/https://medium.com/codox/unveiling-issues-with-the-fugue-paper-regarding-jupiter-ot-72565337b923 (visited on 06/03/2024).

[23] Dr. Chengzheng Sun. *What’s wrong with the Fugue Paper about adOPTed and OT?* 2023. URL: https://web.archive.org/web/20240603141850/https://medium.com/codox/whats-wrong-with-the-fugue-paper-about-adopted-and-ot-9e74ffa0f828 (visited on 06/03/2024).

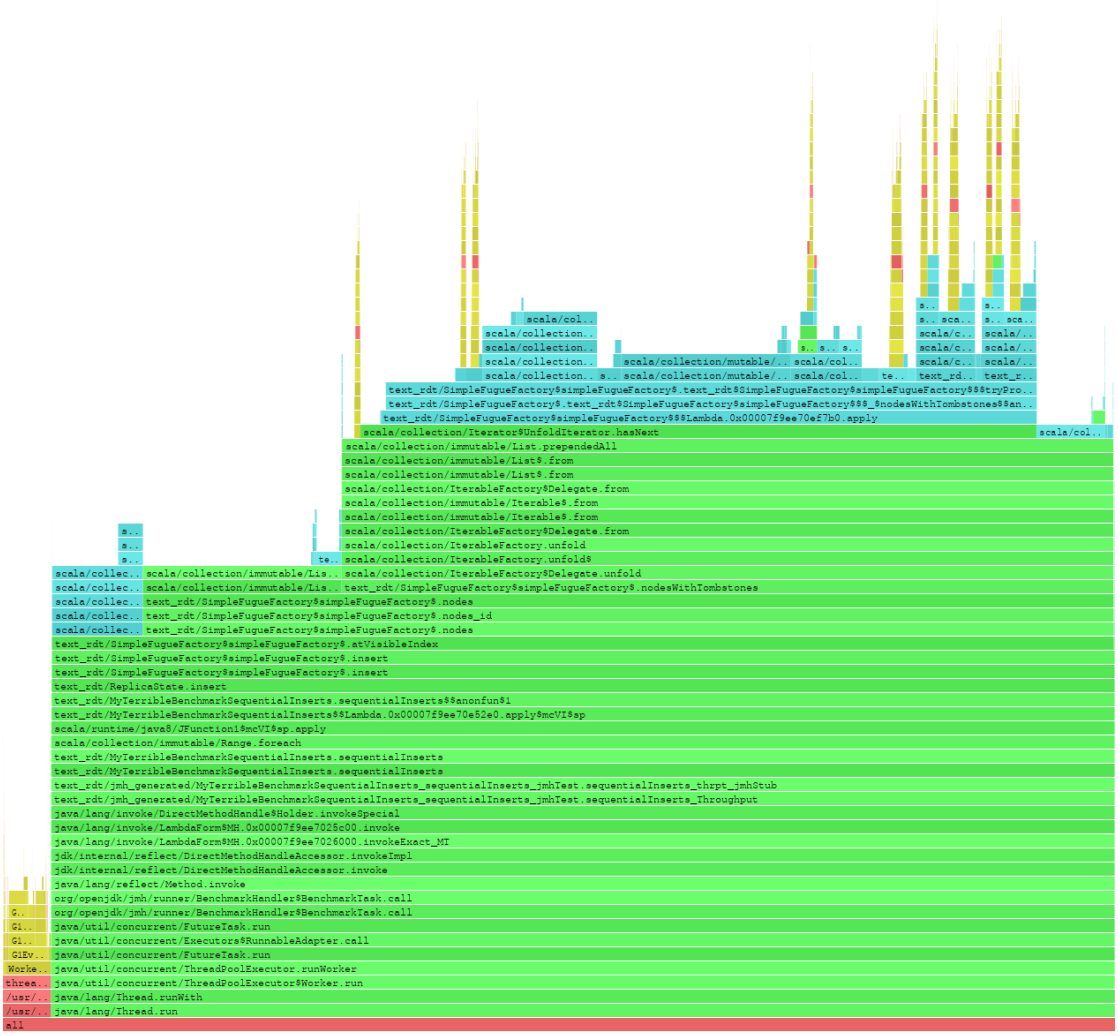
[24] Andrew S. Tanenbaum. *Distributed Systems: Pearson New International Edition : Principles and Paradigms*. 2013. URL: https://elibrary.pearson.de/book/99.150005/9781292038001.

[25] Matthew Weidner, Joseph Gentle, and Martin Kleppmann. “The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing”. In: *CoRR* abs/2305.00583 (2023). DOI: 10.48550/ARXIV.2305.00583. arXiv: 2305.00583. URL: https://doi.org/10.48550/arXiv.2305.00583.

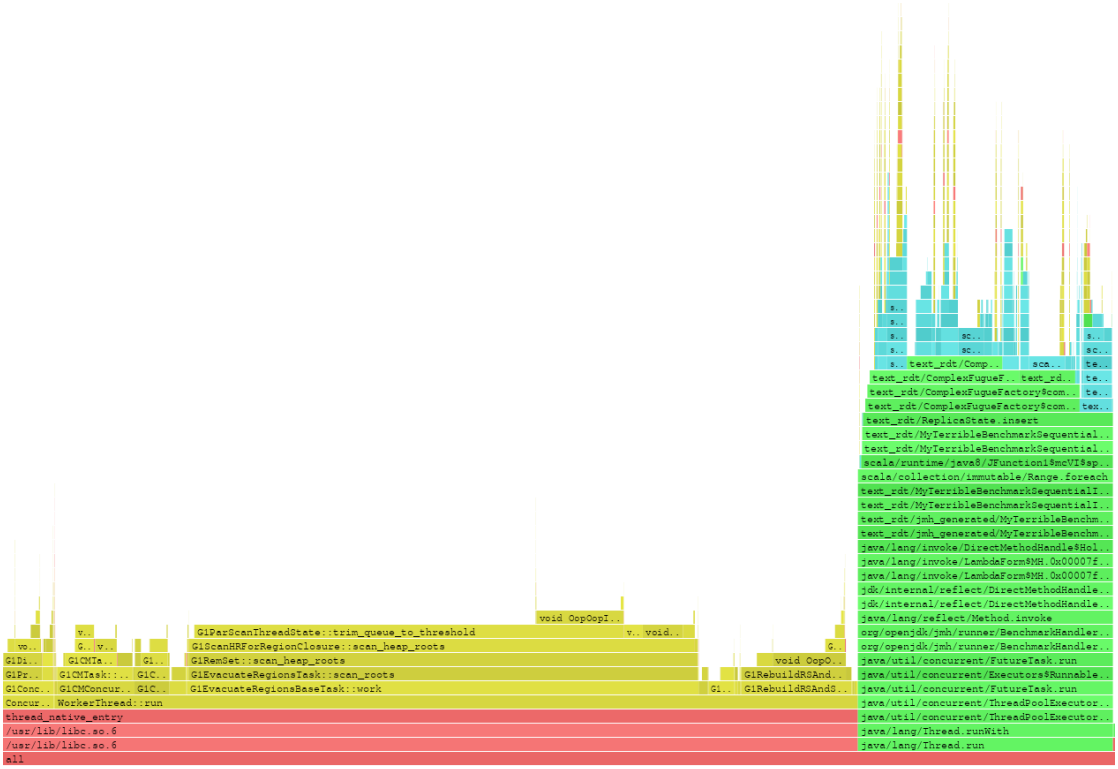
[26] Stéphane Weiss, Pascal Urso, and Pascal Molli. “Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks”. In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 2009, pp. 404–412. DOI: 10.1109/ICDCS.2009.75. URL: https://doi.org/10.1109/ICDCS.2009.75.

A. Appendix

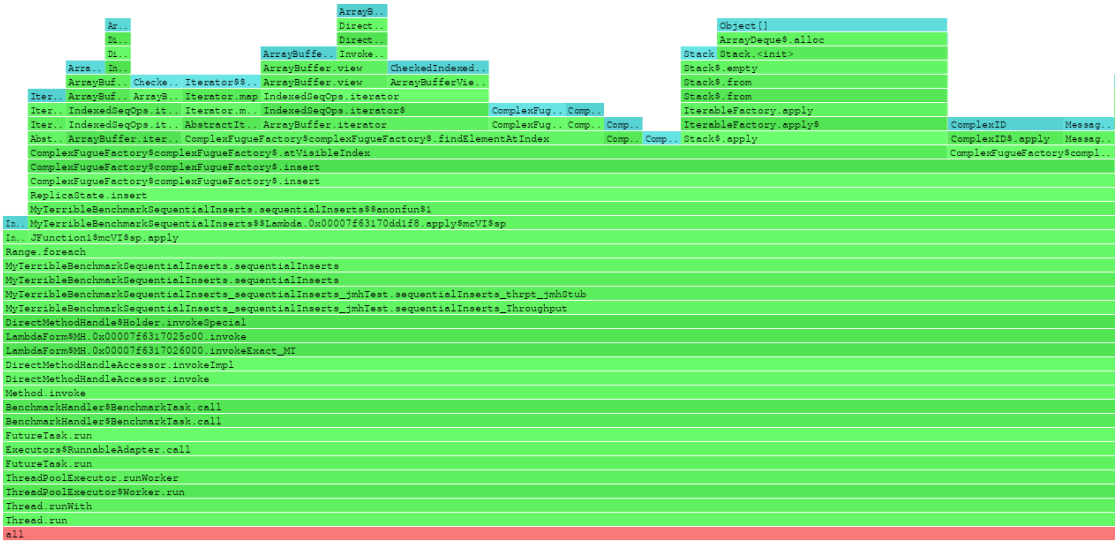
A.1. CPU Profile for Simple Algorithm with Sequential Insertions



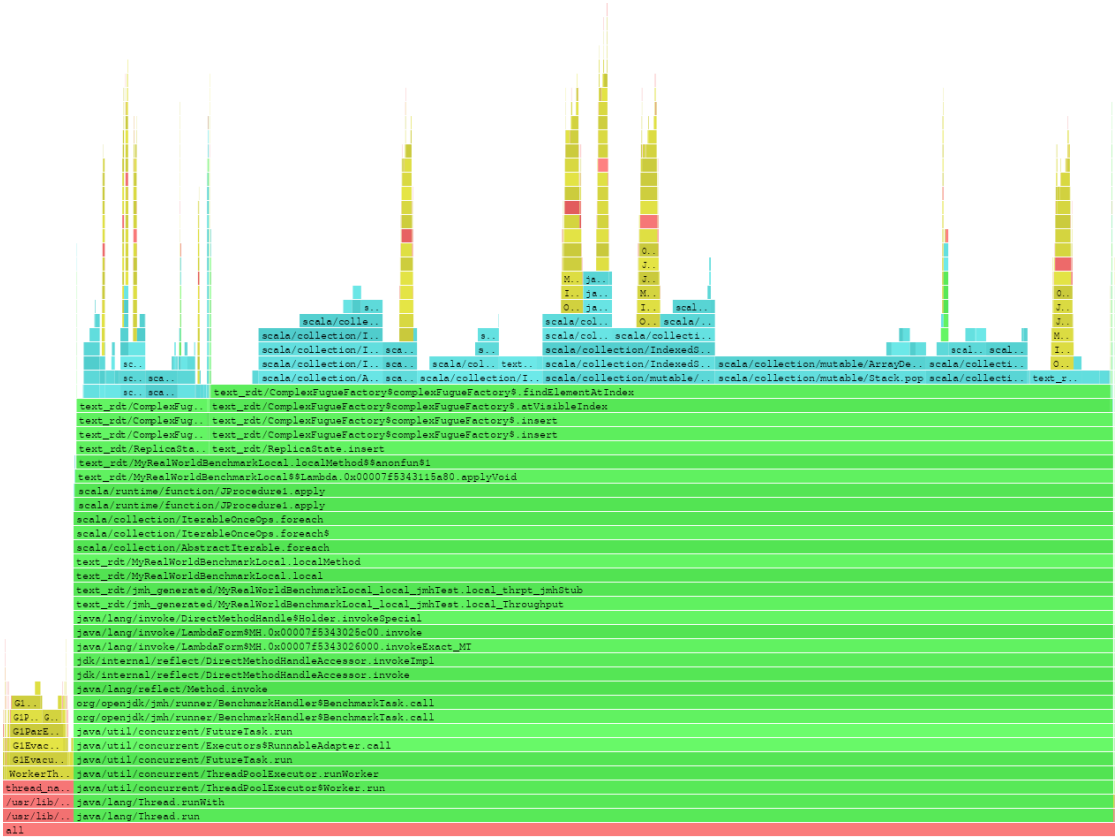
A.2. CPU Profile for Batching Algorithm with Sequential Insertions



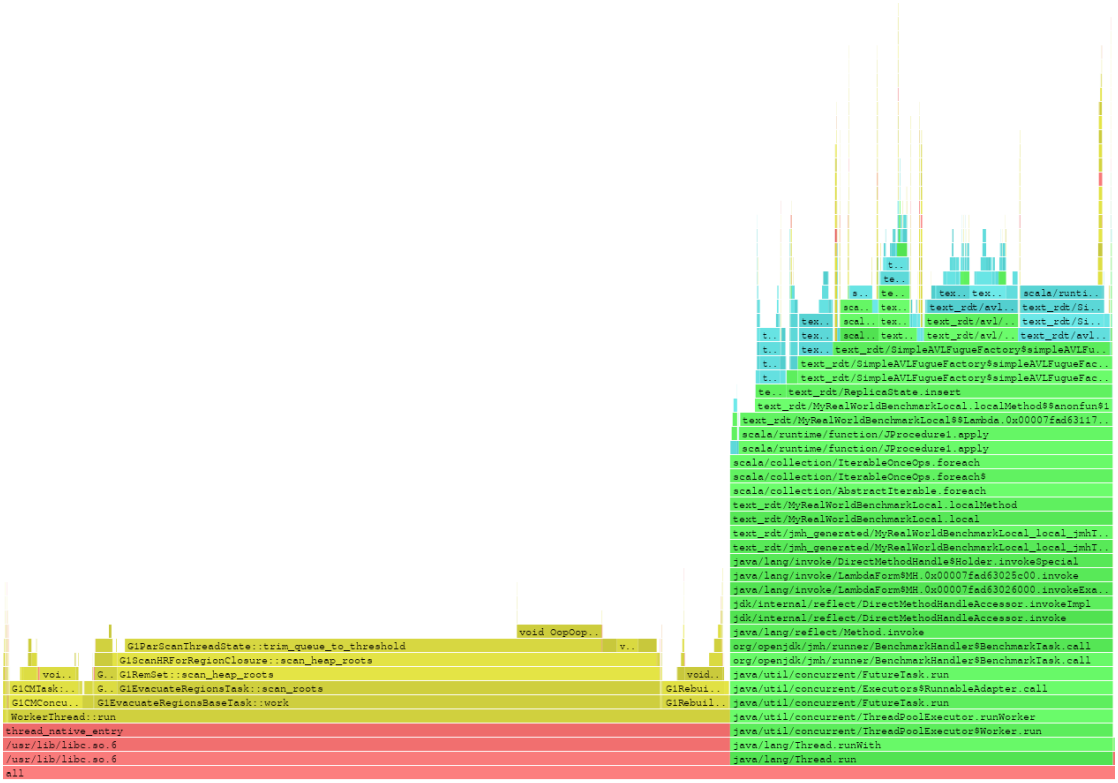
A.3. Allocation Profile for Batching Algorithm with Sequential Insertions



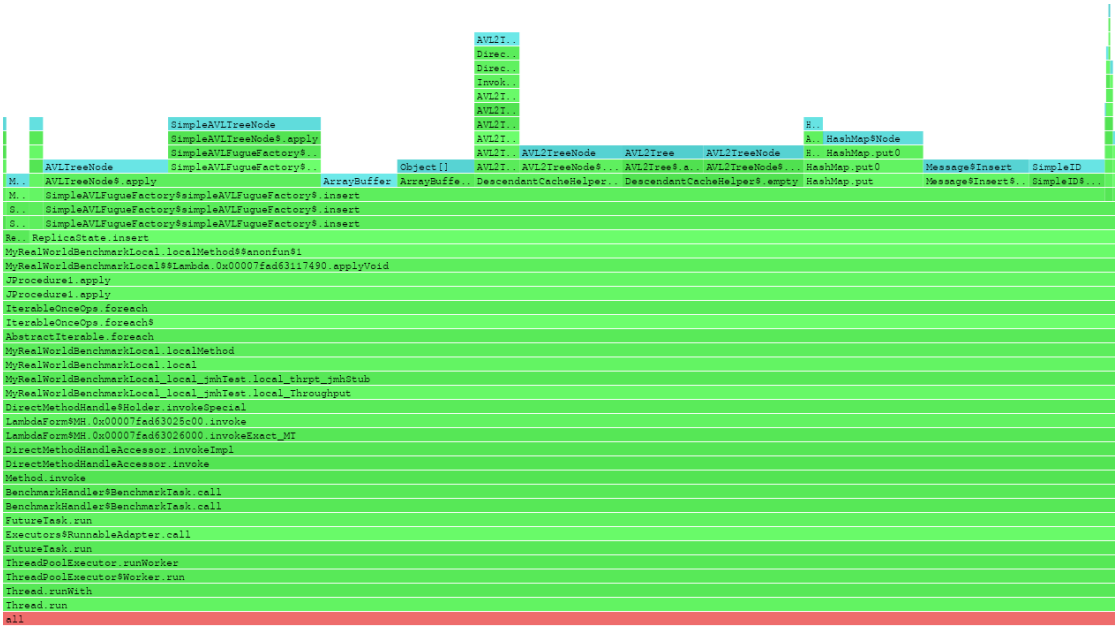
A.4. CPU Profile for Batching Algorithm with Real World Dataset



A.5. CPU Profile for Simple AVL Algorithm with Real World Dataset



A.6. Allocation Profile for Simple AVL Algorithm with Real World Dataset



A.7. Code Showing FugueMax Is Interleaving

```
let rng = seedrandom("42");
let docA = new CRuntime({
  debugReplicaID: ReplicaIDs.pseudoRandom(rng),
});
let ctextA = docA.registerCollab(
  "text",
  (init) => new FugueMaxSimple(init)
);
let docB = new CRuntime({
  debugReplicaID: ReplicaIDs.pseudoRandom(rng),
});
let ctextB = docB.registerCollab(
  "text",
  (init) => new FugueMaxSimple(init)
);
let messageA: Uint8Array = null!
docA.on("Send", (e) => {
  messageA = e.message
})
let messageB: Uint8Array = null!
docB.on("Send", (e) => {
  messageB = e.message
})
docA.transact(() => {
  ctextA.insert(0, 'S')
  ctextA.insert(1, 'h')
  ctextA.insert(2, 'o')
  ctextA.insert(3, 'p')
  ctextA.insert(4, 'p')
  ctextA.insert(5, 'i')
  ctextA.insert(6, 'n')
  ctextA.insert(7, 'g')
})
docB.receive(messageA)
```

```
docB.transact(() => {
  ctextB.insert(8, '*')
  ctextB.insert(9, 'b')
  ctextB.insert(10, 'r')
  ctextB.insert(11, 'e')
  ctextB.insert(12, 'a')
  ctextB.insert(13, 'd')
  ctextB.delete(7)
  ctextB.insert(7, 'g')
  ctextB.insert(8, 'B')
  ctextB.insert(9, 'a')
  ctextB.insert(10, 'k')
  ctextB.insert(11, 'e')
  ctextB.insert(12, 'r')
  ctextB.insert(13, 'y')
  ctextB.insert(14, ':')
})
docA.transact(() => {
  ctextA.insert(8, '*')
  ctextA.insert(9, 'a')
  ctextA.insert(10, 'p')
  ctextA.insert(11, 'p')
  ctextA.insert(12, 'l')
  ctextA.insert(13, 'e')
  ctextA.insert(14, 's')
  ctextA.insert(8, 'F')
  ctextA.insert(9, 'r')
  ctextA.insert(10, 'u')
  ctextA.insert(11, 'i')
  ctextA.insert(12, 't')
  ctextA.insert(13, ':')
})
docB.receive(messageA)
docA.receive(messageB)
console.log([...ctextA.values()].join(""))
console.log([...ctextB.values()].join(""))
```