Text Editing with Character IDs: Collaboration Made Simple

MATTHEW WEIDNER*, Carnegie Mellon University, United States

Text editors typically model their text as an array of characters addressed by index. Even specialized text-editor data structures, such as ropes and piece tables, have an index-based API. Programming with this data model is challenging because each character's address—its index—constantly changes.

We argue for a different data model: a text editor should assign each character an immutable ID and use that as its address. Character IDs let one treat each character as a stable entity that moves around over time, matching users' mental model. We describe how character IDs make it easier to implement annotations, diffs, and especially collaborative editing. We also provide an open-source TypeScript library, Articulated, that implements an optimized data structure for storing the list of IDs, and we show that it has practical performance on a real text-editing trace.

Our collaborative text-editing algorithm (which also works for general lists) is simpler than prior work on Operational Transformation (OT) and Conflict-free Replicated Data Types (CRDTs): it merely combines character IDs with a general-purpose collaboration technique called server reconciliation. We nonetheless show that one of its decentralized variants is equivalent to the popular RGA CRDT.

CCS Concepts: • **Theory of computation** → **Data structures design and analysis**; **Distributed algorithms**; • **Software and its engineering** → *Software libraries and repositories.*

Additional Key Words and Phrases: text editor data structures, server reconciliation, Operational Transformation, Conflict-free Replicated Data Types

1 MOTIVATION

Computers naturally represent text as an array of characters.[1] To address a character, you use its array index.

However, the typical operations in a text editor are not array operations like `text[i] = char`. Instead, they are insert and delete operations, which shift all subsequent characters, updating their array indexes (Figure 1).

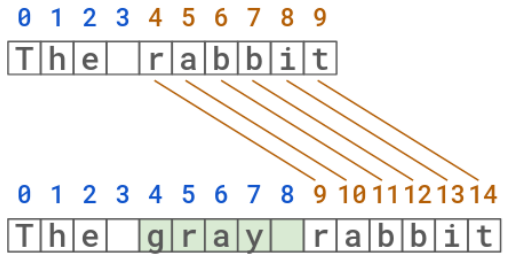

Fig. 1. Inserting characters in a text document changes the indexes of all subsequent characters.

Specialized text-editor data structures, such as the rope [Boehm et al. 1995] and piece table [Crowley 1998], use more advanced data layouts than a simple array to improve the efficiency of insert and delete operations. For example, a rope organizes its text in a binary tree, allowing logarithmic-time edits. However, these specialized data structures still have an index-based API:

*Work completed while the author was at Common Curriculum, Inc.

[1]Technically, Unicode text strings are arrays of code units, while a character is a code point that may consist of multiple code units. We ignore this distinction throughout the paper.

Author's address: Matthew Weidner, Carnegie Mellon University, Pittsburgh, PA, United States, maweidne@andrew.cmu.edu.

their core methods are `charAt(index)`, `insertAt(index, char)`, and `deleteAt(index)`. Data structures that model rich text as a tree (e.g., XML) likewise use indexes to address children within each node and characters within leaf nodes.

Index-based APIs make it challenging to implement many advanced text-editing features. For example:

Annotations Annotations, such as inline comments or highlight notes, associate some additional content with a range of text. It is often convenient to store an annotation separately from the main text (e.g., to restrict view permissions), referencing its text range via `from` and `to` indexes. But then those indexes must be updated every time the main text changes.

Diffs Given two versions of a text document—either before and after states, or divergent branches—we want to see a diff between them. One can construct an exact character-by-character diff from the history of insert and delete operations, but doing so is nontrivial.

Collaborative Editing When two users edit a collaborative text document such as a Google Doc, their edits may be concurrent: due to network latency, each user performs an operation while unaware of the other's. Figure 2 shows an example in which Alice and Bob concurrently insert a word, with Alice's operation reaching the server first. If we use index-based operations, and the server applies those operations to its own state literally in the order that it receives them, then it will compute the nonsensical result "The gray rabbit j highumped."

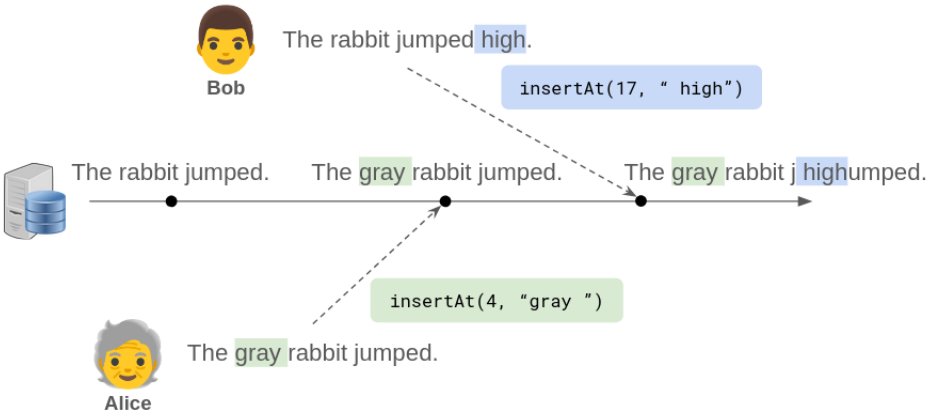

Fig. 2. Bob submits the operation `insertAt(17, " high")` to the central server. But before his operation arrives, the server applies Alice's concurrent operation `insertAt(4, "gray ")`. If the server applies Bob's operation literally—still inserting at index 17—then the result is nonsensical. The correct (user-intended) insertion position has shifted to index 22.
Database server clipart by lyte [Public Domain], via Openclipart (openclipart.org/163711).

2 CHARACTER IDS, PLUS TOMBSTONES

This paper advocates for an obvious but underused strategy: assign each character a unique immutable ID and address the character by that ID, instead of by its ever-changing index. Put another way, a text editor's core data structure should be a list of unique entities, each rendered as a character, instead of a list of literal characters. This matches users' mental model, in which each character has a unique identity that is stable over time ("this specific letter q"). Aligning mental models in this way makes programming more intuitive.

Due to our interest in collaborative editing, we prefer *globally* unique IDs (in the style of UUIDs), though for a strictly local document, one could assign IDs using a counter.

Unique IDs are already common for list elements in web applications, such as the slides in a slideshow or tasks in a Kanban column. Like text characters, those elements change array indexes over time, so IDs are necessary for data normalization, client-server communication, and UI rendering.

Actually, our strategy has one extra component: in addition to the IDs of present characters, store the IDs of deleted characters—still in order relative to the present IDs (Figure 3). We call these deleted-but-remembered IDs **tombstones**, borrowing a name from collaborative text-editing algorithms [Shapiro et al. 2011a]. A tombstone lets one refer to a specific position in the text, defined by its relation to other IDs, even when that position does not correspond to a present character.



Fig. 3. Illustration of tombstones in a text document that uses character IDs. After deleting "gray " from "The gray rabbit", the deleted characters' IDs are still stored in order relative to the remaining IDs, but they are not rendered and they do not count towards indexes.

Let us see how character IDs (with tombstones) simplify the advanced features above:

Annotations. Store an annotation's range as a pair `(fromId, toId)`, indicating the character IDs where it starts and ends. This describes the same range of characters even when those characters' indexes change due to edits earlier in the document. It also behaves reasonably after characters are deleted or inserted within the range: the range contracts or expands while preserving its endpoints.

Tombstones ensure that the range is still comprehensible (contracting as usual) after its starting or ending character is deleted.

Diffs. We can diff two versions of a text document by diffing their lists of character IDs. An ID present in the left version and tombstoned in the right must have been deleted on the right; an ID present in the left version and unknown in the right must have been inserted on the left.

Collaborative Editing. Instead of index-based operations, send ID-based editing operations to a collaboration server. In particular, send operations:

`insertAfter(beforeId, id, char)` Inserts the character `char` with a unique new ID `id` immediately after `beforeId`.

`delete(id)` Deletes the character with ID `id`, making its ID a tombstone.

Figure 4 shows the same scenario as Figure 2, but using these ID-based operations instead of index-based operations. This time, the resulting text is reasonable. Thanks to tombstones, we likewise get a reasonable result if an insertion operation references a concurrently-deleted `beforeId`, as shown in Figure 5.

We argue that these ID-based operations are more natural than index-based operations. Indeed, when a user inserts a character, they usually intend to insert it after a specific character on their screen, while being completely unaware of the corresponding index. Likewise, when a user deletes a character, they intend to delete that specific character regardless of its index. ID-based operations capture this intent, hence their results make sense to users even when they are applied to a slightly further-ahead state on the server.

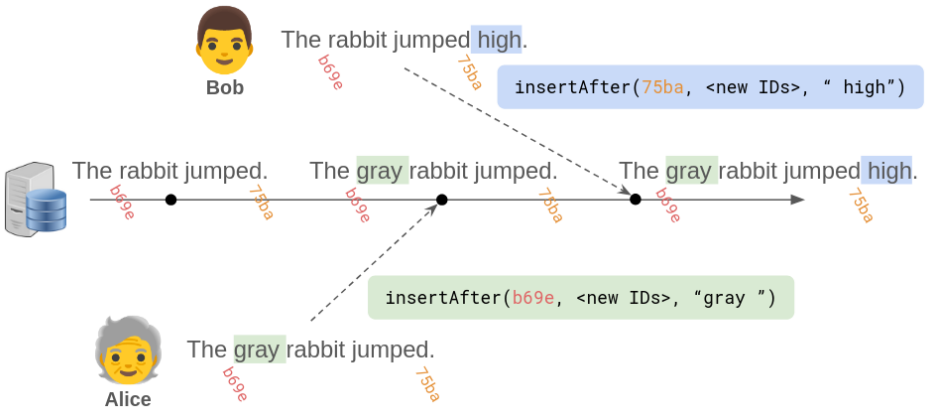

Fig. 4. Bob submits the operation `insertAfter(75ba, <new IDs>, " high")` to the central server, where 75ba is the ID of the 'd' in "jumped". (For brevity, we show a bulk operation that inserts multiple characters sequentially.) But before his operation arrives, the server applies Alice's concurrent operation `insertAfter(b69e, <new IDs>, "gray ")`, where b69e is the ID of the first space. This time, applying Bob's operation literally gives the expected result, because the ID 75ba shifts along with its character.

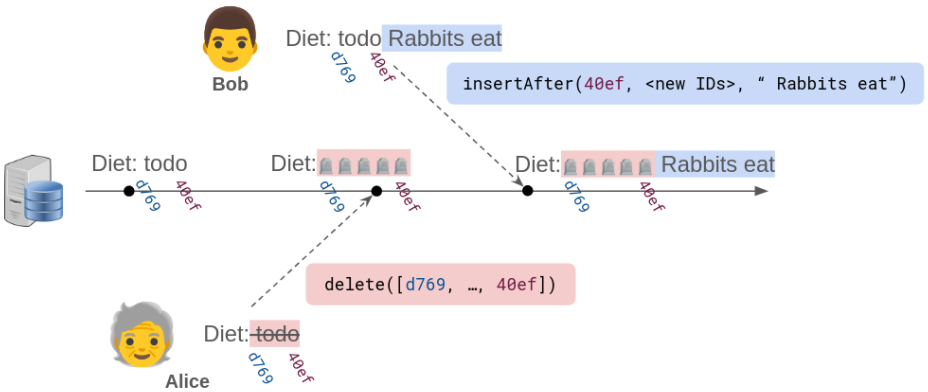

Fig. 5. Bob submits the operation `insertAfter(40ef, <new IDs>, "Rabbits eat")` to the central server, where `40ef` is the ID of the last 'o' in "todo". But before his operation arrives, the server applies Alice's concurrent operation `delete([d769, ..., 40ef])`, which deletes all characters in " todo". (For brevity, we show a bulk operation that deletes multiple IDs.) The server is still able to apply Bob's operation because it remembers the ID 40ef as a tombstone, yielding the reasonable combined result "Diet: Rabbits eat".

We will use these operations in Section 4 as part of a complete collaborative text-editing algorithm.

3 GENERATING AND STORING IDS

To use character IDs in practice, we need a way to generate globally unique IDs and a list-of-IDs data structure that stores them in list order (including tombstones). Table 1 shows the core API required. We choose to store the IDs separately from the actual text so that the same list-of-IDs data structure can be used for both text and general lists.

Table 1. The Core API for a List-of-IDs Data Structure

Method	Description
`insertAfter(beforeId, id)`	Inserts `id` immediately after `beforeId`, or at the beginning of the list if `beforeId = null`.
`insertBefore(afterId, id)`	Inserts `id` immediately before `afterId`, or at the end of the list if `afterId = null`.
`delete(id)`	Marks the given ID as a tombstone, without changing its order in the list.
`idAt(index)`	Returns the ID currently at the given index.
`indexOf(id, bias?)`	Returns the current index of the given ID. If the ID is a tombstone, returns the index on the side given by `bias` if provided (`"left"` or `"right"`), else −1.
`has(id)`	Returns whether the given ID is currently present (not a tombstone).
`length()`	Returns the number of present IDs.
`values()`	Iterates over all present IDs in list order.
`valuesWithIsDeleted()`	Iterates over all IDs including tombstones in list order, emitting pairs `(id, isDeleted)`.

A simple implementation could use random UUIDv4s [Leach et al. 2005] as the IDs and store them in an array of pairs `(id, isDeleted)`. However, this implementation is inefficient: all operations take linear time, and we are storing an extra 16-byte UUIDv4 for every character. We suspect that this storage overhead is the main reason text editors typically avoid character IDs in practice, despite the advantages described earlier.

We provide a more efficient implementation, in the form of an open-source TypeScript library named Articulated.[2] The library was developed with assistance from the Claude chatbot and Claude Code, primarily to author tests and to summarize inline documentation for the readme. In addition, Claude Code and GitHub Copilot were used as code reviewers. All generated content was proofread manually.

Our library essentially implements a rope [Boehm et al. 1995], but for IDs instead of characters:

- IDs inserted from left to right by a single user, as in normal typing, are generated as pairs `(bunchId, 0)`, `(bunchId, 1)`, `(bunchId, 2)`, …, where `bunchId` is a globally-unique value shared by this entire **bunch** of IDs.
- Neighboring IDs that have the same `bunchId`, sequential counters, and the same `isDeleted` status are stored in compressed form, as a single struct with fields `(bunchId, startCounter, count, isDeleted)`.
- Sequential structs with the same bunch ID but alternating `isDeleted` status are collected into arrays, and those arrays are arranged into a balanced tree. The tree allows most operations to run in time $O(\text{poly}(\log(n)) + s)$, where n is the number of IDs including tombstones and s is the maximum array size.

Similar optimizations are already present in collaborative text-editing algorithms [Briot et al. 2016; Jahns 2020; Sobo 2022]. Our contribution is to provide them in a generic list-of-IDs data structure, instead of as the internals of a specific collaboration framework. Benchmark results appear in Section 6.

[2] https://github.com/mweidner037/articulated

Generating bunch IDs is left to library consumers, but we recommend that each client generate them in the form `clientId_c`, where `clientId` is a per-client UUID and `c` is a client-local counter. This reduces the entropy in the set of all IDs, aiding compression over the network and on disk.[3]

As an additional feature, our implementation is a persistent data structure, which is useful for collaborative editing (Section 4.1). To attain persistence, the binary tree is a B+ Tree [Comer 1979], modified to omit the keys (since IDs are sorted *by fiat* instead of by an intrinsic key order) and to track the number of present and total IDs within each subtree. A persistent tree cannot use parent pointers because they would need to be rewritten on every change; instead, we assign each node a fixed internal sequence number (a simple local counter), and we store two extra maps as persistent red-black trees:

`parentSeqs` Maps each inner node's sequence number to its parent's sequence number.

`leafMap` Maps each leaf node's first ID to the leaf node's current object and its parent's sequence number. Keys are ordered such that we can get the leaf given any of its IDs.

The function `has(id)` looks up the ID's current leaf node using `leafMap`, returning true if the leaf actually contains the ID and is not deleted. The function `indexOf(id, bias?)` first looks up the ID's leaf node and parent sequence number using `leafMap`. It then looks up all sequence numbers on the parent's path to the root using `parentSeqs`, recording them. Finally, it traverses the actual tree downwards from the root, each time selecting the child whose sequence number matches the path recorded previously. The nodes on this path store statistics sufficient to compute the ID's current index.

4 COLLABORATIVE EDITING

We showed in Section 2 how character IDs let a collaboration server handle concurrent operations in a reasonable way. We now describe the client side of collaboration, yielding a complete collaborative text-editing algorithm. The algorithm is inspired by the Thymer IDE, which uses an extension of Section 4.1 to implement collaboration on document outline trees [Cools 2024].

Later subsections extend the algorithm to support rich-text formatting, conflict-aware text-editing operations, and general lists with move operations. In Section 5, we adapt it to the case of decentralized (server-free) collaboration.

4.1 The Client Side: Server Reconciliation

When a user edits a collaborative text document, such as a Google Doc, they expect to see the result of their own operations immediately, without waiting for a network round-trip to the collaboration server. Specifically, their client should *optimistically* display the state that it expects the server will be in once the server processes the user's pending local operations. The actual result may differ due to concurrent operations or server-specific business logic; the client must eventually update to match that result.

Server reconciliation is a general-purpose technique for optimistic client states. In server reconciliation, the document state displayed to the user is always the result of applying the user's pending local operations on top of the latest state received from the server. Clients apply their pending local operations using a client-side copy of the server code, which is allowed to be approximate. This technique has been used in multiplayer computer games for decades [Gambetta [n. d.]]; Replicache provides it in a library for general collaborative web applications [Rocicorp LLC 2025].

To implement server reconciliation, each client stores two pieces of data:

[3]Each thread must use its own client ID to prevent local counter reuse. In a web application, each new browser tab typically requires a fresh client ID.

(1) The latest document state it received from the server; and
(2) Its pending local operations—operations performed by the local user that were not yet acknowledged by the server.

When a client receives an update from the server, it updates its copy of the server state to match and removes any acknowledged local operations from the pending list. It then recomputes the state displayed to the user by re-applying the remaining pending local operations on top of the new server state.[4] See Figure 6.

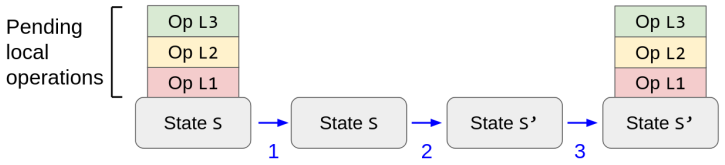

Fig. 6. The server reconciliation way to process a server state update `S -> S'` in the presence of pending local operations `L1`, `L2`, `L3`. Step 1: The client rewinds its state to the previous state received from the server, `S`. Step 2: The client applies the update, yielding the new server state `S'`. Step 3: The client re-applies any pending local operations that were not acknowledged by the update—here all of them.

Observe that server reconciliation essentially implements the *definition* of optimistic client states: each client displays the state that it expects the server will be in, once the server processes its user's pending local operations. Server reconciliation does so without imposing any mathematical requirements on the operations themselves—all clients will eventually converge to the server's state regardless—so it is suited to a wide variety of collaborative applications. While an application built on server reconciliation does need to support the rollback-and-reapply steps from Figure 6, that is an implementation concern separate from the semantics of the operations themselves. For example, clients can store state in persistent data structures, including the one provided by our library, or build on an application-agnostic server reconciliation library like Replicache [Rocicorp LLC 2025].

A collaborative application developer should still choose operations that give "reasonable" results on the server, even when applied to a slightly further-ahead state due to concurrency. This is an informal, application-driven requirement, not a mathematical correctness condition, so developers without specialized expertise can design operations and improve them in response to user feedback. The same operations will then produce reasonable optimistic client states in a server reconciliation architecture, since clients are just anticipating the server's future state.

We already saw that ID-based `insertAfter` and `delete` operations give reasonable results on the server, even in the face of concurrency (Figure 4). Hence they work well in a server reconciliation architecture. That combination is our complete collaborative text-editing algorithm.

The simplicity of this collaborative text-editing algorithm should not be taken for granted. Existing academic work on collaborative text editing focuses on Operational Transformation (OT, used by e.g. Google Docs) [Day-Richter 2010; Ellis and Gibbs 1989; Sun and Ellis 1998] and Conflict-free Replicated Data Types (CRDTs, used by e.g. the popular Yjs open-source library) [Jahns 2026; Oster et al. 2006b; Shapiro et al. 2011a]. These are both families of distributed algorithms with nontrivial mathematical correctness requirements; they are infamous for their complexity [Gentle

[4]Alternatively, the client may wait until no pending local operations remain (all are acknowledged), then directly replace its displayed state with the server state. This strategy, used by PowerSync's sync priorities 1–3 [PowerSync [n. d.]], is simpler to implement but may lead to starvation if the interval between user edits is less than the network round-trip time.

2011; Levien 2019]. Character IDs and server reconciliation let us instead use a naïve approach: the server applies operations literally in the order that it receives them, and clients optimistically apply their own operations on top.

4.2 Rich-Text Formatting Operations

Suppose our local text editor supports rich-text formatting, via a local method `markAt(fromIndex, toIndex, mark)` that applies a formatting mark (bold, font size, etc.) to the text in the given range.

We can make this method collaborative merely by replacing the indexes with character IDs. That is, we add a new operation alongside `insertAfter` and `delete`, still in the context of server reconciliation:

mark(fromId, toId, mark) Applies a formatting mark to the characters between the given character IDs.

Using character IDs ensures that a `mark` operation affects its intended range of characters even if a concurrent edit changes their indexes. Tombstones ensure that the range is still comprehensible (contracting to the remaining characters) after its starting or ending character is deleted.

Our `mark` operation is inspired by the Peritext rich-text CRDT [Litt et al. 2022], which uses a similar operation with character IDs derived from an underlying plain-text CRDT. We refer to their work for detailed advice about inclusive vs exclusive endpoints and other user expectations around collaborative rich-text formatting. Unlike Peritext, we do not explicitly store the complete set of marks and use it to construct the current formatting state. Instead, we trust each user's local text editor to maintain that state and apply it to new characters as needed (e.g., bolding new characters inserted in the middle of an active bold mark).

Remark 4.1. Besides formatting, rich text typically forms a structured document, which requires further operations. For example, the Quill rich-text editor models its state as a flat list of blocks (paragraph, heading, list item, etc.), each separated by a newline character labeled with the block's type [Quill contributors [n. d.]]. We can support that structure by assigning each newline an ID as usual; block split and merge operations then insert and delete newlines. However, it is not obvious how to move entire blocks without invalidating concurrent edits. Likewise, our current approach does not support operations on general trees such as XML. We leave it to future work to adapt character IDs to these more complex document structures.

4.3 Conflict-Aware Text-Editing Operations

We noted above that server reconciliation does not impose any mathematical requirements on the operations themselves, unlike OT and CRDT architectures. It follows that we can safely extend our collaboration algorithm with text-editing operations beyond `insertAfter` and `delete`.

While those operations give reasonable results in the face of concurrent edits to different parts of a document—avoiding false conflicts caused by changing array indexes—they are not always reasonable in the face of directly conflicting edits. For example, suppose one user deletes the word "color", while concurrently another user changes it to "colo**u**r". Our algorithm, like many others, results in a state with just the inserted "u", which is now a dangling letter. This result is in fact required by the strong list specification [Attiya et al. 2016], a popular correctness condition that we prove for our algorithm in Section 5.2. Users may nonetheless interpret it as data corruption [Clemmer 2025].

We can avoid the dangling letter by designing conflict-aware text-editing operations for use alongside `insertAfter` and `delete`:

`deleteRange(fromId, toId)` Deletes all IDs between `fromId` and `toId`, inclusive—including IDs inserted concurrently. A client issues this operation when its user deletes an entire word or sentence.

`editWord(before, after)` Given two subsequences of the text (with IDs), replaces the `before` sequence with `after`, but only if it is present unchanged in the input state with no new characters or tombstones. A client issues this operation when its user edits characters within an existing word, submitting the word's before and after states.

In the "colour" scenario above, the first user deletes the word using `deleteRange`, while the second inserts the "u" using `editWord`. Regardless of which order the server receives these operations, it computes a reasonable combined result: the word is completely deleted.

One can imagine further conflict-aware text-editing operations: find-and-replace, AST manipulation, LLM-driven edit, etc. We caution that complex operations may confuse users or violate each other's preconditions in the face of concurrency. Additionally, no operations can resolve true semantic conflicts, like that between an editor who polishes a document and a writer who changes its topic.

4.4 General Lists with Move Operations

Our ID-based collaborative text-editing algorithm works just as well for non-text lists, such as the list of slides in a slideshow or tasks in a Kanban column, as long as each list element has an immutable ID. However, such lists often allow an additional operation: move, which changes an existing element's position within the list.

While we could implement move as a delete followed by an insert, that would duplicate an element that two users move concurrently. A second option is to move the ID within the list-of-IDs data structure; however, that may break annotation ranges (by moving `toId` before `fromId`) or cause surprising behavior for concurrent inserts referencing that ID.

Instead, we advise the following strategy, adapted from an analogous CRDT [Kleppmann 2020]:

- Add a `position` field to each list element, storing a globally-unique value such as a UUID. This is in addition to the element's immutable ID.
- Store positions instead of IDs in the list-of-IDs data structure (Section 3). The order of list elements is the order of their current positions in this list.
- Move an element using an operation `moveAfter(beforePos, pos, id)`, where `id` is the ID of the list element being moved, `pos` is a unique new position, and `beforePos` indicates the move target. This operation inserts `pos` into the list-of-positions just after `beforePos` and sets `element.position = pos`.

Operations internal to a list element should continue to address it by ID, not position. However, annotation ranges should reference positions instead of IDs. Also, when a position is no longer used—either due to a move or because its element was deleted—it should become a tombstone.

With this strategy, and the usual server reconciliation, one checks that:

- Operations internal to a list element affect that element even if it is moved concurrently.
- Concurrent moves of the same element follow a last-writer-wins rule: the last move operation to reach the server sets the final position.
- In a list with elements $[A, B, C, D]$, suppose E is inserted at the end of the list (after D), while concurrently D is moved to the beginning of the list. Then the final list order is $[D, A, B, C, E]$: E's insertion is relative to D's original position, not its ID, and so E does not "follow" D's movement.

5 DECENTRALIZED COLLABORATIVE EDITING

5.1 Algorithm

In a decentralized setting, such as a peer-to-peer, federated, or local-first network, we cannot rely on a central server to order operations. Thus we cannot use the collaborative text-editing algorithm from Section 4.1. However, a related strategy works: clients establish a shared total order of operations in a decentralized way; each client's state is always the result of applying the operations that it knows about in that order.

Specifically, consider a distributed system whose events are operations on a shared data structure, such as a text document. We assume that each operation describes a deterministic change to the data structure and is labeled by a unique operation ID. Define a **decentralized total order** to be a total order $\prec$ on these operations with the following properties:

(1) The total order is a linear extension of the causal order [Lamport 1978]: if o happened before p, then $o \prec p$.
(2) If a client is aware of (sent or received) operations o, p, and all operations that happened before either operation, then they can compute whether $o \prec p$ or $p \prec o$, without additional network communication.

For example, one may order operations using Lamport timestamps (with client IDs as a tiebreaker) [Lamport 1978] or the depth-first topological sort of the causal order described in Appendix A (an incremental variant of Cormen et al. [2009]).

Given a decentralized total order $\prec$, we can implement decentralized collaboration using the following distributed algorithm:

- Users issue operations on their local client at any time.
- Clients broadcast operations to each other asynchronously while ensuring eventual delivery.
- A client computes the current state of their copy of the shared data structure as follows: (a) take the set of operations that the client is aware of; (b) restrict to those whose causal dependencies are satisfied, i.e., the client is also aware of all operations that happened before;[5] (c) apply those operations in $\prec$ order, starting from a fixed initial state.

This algorithm trivially satisfies Strong Eventual Consistency [Shapiro et al. 2011b]: clients that are aware of the same operations are in the same state. Note that because the algorithm does not involve locking or consensus, the order is not append-only—a client may receive an operation that belongs at an arbitrary point in its copy of the total order. Processing such an operation requires a rollback-and-reapply procedure similar to server reconciliation; see Figure 7.

Decentralized total orders are already used for collaboration on general data. For example, OpSets [Kleppmann et al. 2018] and the Automerge library [Automerge contributors 2026] use the Lamport timestamp order; the Matrix chat protocol [Jacob et al. 2020], Classic Jazz [Garden Computing 2026], and Wonlaw [2023] use various topological sorts of the causal order.

We can use our usual ID-based text-editing operations, `insertAfter` and `delete`, within a decentralized total order. Indeed, the Classic Jazz library [Eickhoff 2025] and a blog post by Wonlaw [2023] do so already. (Automerge instead uses operations from the RGA CRDT [Martin Kleppmann, personal communication].) We thus obtain a decentralized collaborative text-editing algorithm analogous to the centralized algorithm in Section 4.1.

[5] This step is not strictly necessary, but it simplifies programming and prevents some confusing user-facing behaviors [Lloyd et al. 2013].

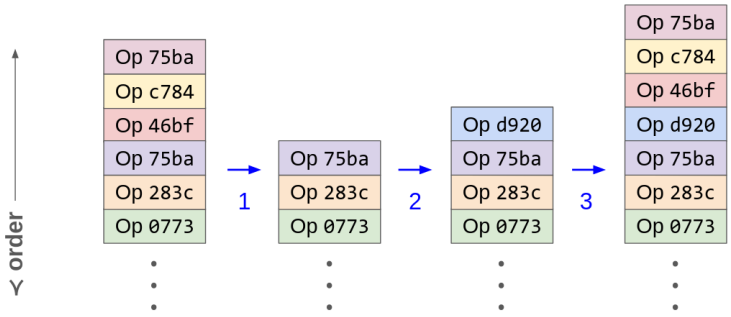

Fig. 7. The decentralized total order way to process a remote operation `d920` that belongs at an arbitrary point in the $\prec$ order. Step 1: Rollback to the state after the immediately prior operation. Step 2: Apply the new operation. Step 3: Re-apply all subsequent (greater) operations.

5.2 Correctness: The Strong List Specification

We claim that our decentralized collaborative text-editing algorithm gives results that are reasonable in the face of concurrency. Specifically, it satisfies the same correctness requirement expected of CRDTs and (decentralized) OT algorithms:

Theorem 5.1. For any decentralized total order $\prec$, the algorithm in Section 5.1 satisfies the strong list specification by Attiya et al. [2016].

For the proof, fix a distributed history of `insertAt(index, char)` and `deleteAt(index)` operations on a shared text document, specifying the user-initiated operations and when messages were delivered between clients. Formally, this is an **abstract execution satisfying causal broadcast** as defined by Attiya et al. [2016], except with unspecified return values. Suppose a group of clients execute this history using our decentralized algorithm, with decentralized total order $\prec$. That is, each client issues character ID-based operations corresponding to the given index-based operations, following the distributed algorithm from Section 5.1.

Consider a particular client's state at a particular moment. On the one hand, that state stores character IDs in some order, which we denote $<_\prec$. On the other hand, define a tree T as follows:

- There is one node for each character ID inserted so far, plus a special root node for `null`.
- The parent of `id` is `beforeId`, where `insertAfter(beforeId, id, char)` is the unique operation that inserted `id`.
- Sibling nodes are ordered in *reverse* of the $\prec$ order on their `insertAfter` operations. That is, IDs inserted later appear farther to the left.

Let $<_T$ be the order on character IDs corresponding to the depth-first pre-order traversal of T. That is, $<_T$ is the total order characterized by:

Depth-first If a and b are siblings with a less than b in the sibling order, and c is a descendant of a in T, then $a <_T c <_T b$.

Pre-order If a is the parent of d, then $a <_T d$.

Lemma 5.2. $<_\prec = <_T$.

Proof. It suffices to prove that $<_\prec$ satisfies the Depth-first and Pre-order properties above.

The pre-order property is easy: if a is the parent of d, then d was inserted by an operation `insertAfter(a, d)`, placing it after a in the client's list. Hence $a <_\prec d$.

For the depth-first property, let o_a, o_b, o_c denote the `insertAfter` operations for a, b, c. We have $o_b \prec o_a$ by the definition of the sibling order. Meanwhile, $o_a \prec o_c$ because o_a happened before o_c.

Because $o_b \prec o_a$, b is already present when processing a's `insertAfter` operation in $\prec$ order. Thus $a <_\prec b$: a is inserted directly after their common parent, which (by the pre-order property) is somewhere before b.

Next, consider the case where c is a child of a. Because $o_b \prec o_a \prec o_c$, b is already present when processing o_c = `insertAfter(a, c, char)` in $\prec$ order. Thus inserting after a (which is less than b) places c somewhere before b: $c <_\prec b$. Similarly, any child of c is later inserted directly after c, hence somewhere before b. By induction on the relative depth of an arbitrary descendant c of a, we see that $c <_\prec b$ for all such descendants. □

Proof of Theorem 5.1. The strong list specification mandates that there is a single total order $<$ for the entire execution such that for any client at any moment, that client's state is given by: the characters whose `insertAt(index, char)` operations were visible to that client, minus any with a visible `deleteAt(index)` operation, in $<$ order. Furthermore, `insertAt(index, char)` must initially insert `char` at the requested `index`.

We claim that the single total order $<$ is given by the depth-first pre-order traversal order $<_{T'}$, where T' is defined as above but containing all IDs appearing in the entire execution. Indeed, it is easy to see that T' extends the tree T corresponding to each specific state, hence the total order $<_{T'}$ extends each $<_T$. The claim then follows from Lemma 5.2.

Deletions are handled in the mandated way by our tombstones. Finally, characters are initially inserted at the correct index because we generate `insertAfter(beforeId, id, char)` operations using the obvious choice of `beforeId`. □

Remark 5.3. Attiya et al. [2016] show that for any algorithm satisfying their strong list specification, there are scenarios in which some client must store $\Omega(D)$ data even though only one character is visible, where D is the number of previously-deleted characters. Thus our tombstones cannot be optimized away, at least in the general decentralized case.

5.3 Equivalence with RGA CRDT

The strong list specification allows some freedom in the precise character order when there are concurrent operations. Thus it makes sense to ask whether two different algorithms always yield the same text, given the same history of user-initiated operations. Specifically, we say that two collaborative text-editing algorithms are **semantically equivalent** if they always yield the same text when given the same distributed history of `insertAt(index, char)` and `deleteAt(index)` operations.

Despite its different presentation, our decentralized approach yields an algorithm that is semantically equivalent to a popular text-editing CRDT:

Theorem 5.4. Consider the algorithm in Section 5.1 with $\prec$ set to the Lamport timestamp order. This algorithm is semantically equivalent to RGA [Roh et al. 2011] / Causal Trees [Grishchenko 2010].

Proof. As in the proof of Theorem 5.1, fix a distributed history of `insertAt(index, char)` and `deleteAt(index)` operations on a shared text document, and suppose a group of clients execute this history using the algorithm.

We consider RGA in the form described by Attiya et al. [2016]. In this description, the list order is the depth-first pre-order traversal of the following tree:

- There is one node for each inserted character, plus a special root node for the beginning of the list.

- The parent of a character is the character directly to its left when it was inserted.
- Sibling nodes are ordered in reverse of the Lamport timestamp order on their insertion operations.

This is precisely the tree T appearing in Lemma 5.2 when $\prec$ is the Lamport timestamp order. Additionally, both our algorithm and RGA handle deletions using tombstones. The theorem then follows from Lemma 5.2. □

5.4 Non-Interleaving Guarantees

Suppose two users collaborate on a text document using our algorithm (either centralized or decentralized). While collaborating, they work concurrently for a period of time, e.g., because one user is offline. During this period, each user inserts a complete sentence at the same starting position, typing from left to right. What text will result once they reconnect and share operations?

Several existing collaborative text-editing algorithms will interleave the characters from their two sentences, creating a jumbled mess [Kleppmann et al. 2018; Weidner and Kleppmann 2025]. Fortunately, this is not the case with our `insertAfter` operations: regardless of what order the server (or decentralized total order) chooses for such operations, one sentence will appear completely before the other. Indeed, our algorithm easily satisfies **forward non-interleaving** as defined by Weidner and Kleppmann [2025], so the claim follows from their Lemma 3.

For a direct proof, let p be the character ID at the starting position, and let $a_1 \dots a_m$ and $b_1 \dots b_n$ be the character IDs for the two users' concurrent sentences. The corresponding operations are

$$[\texttt{insertAfter}(p,\ a_1), \texttt{insertAfter}(a_1,\ a_2), \dots, \texttt{insertAfter}(a_{m-1},\ a_m)],$$
$$[\texttt{insertAfter}(p,\ b_1), \texttt{insertAfter}(b_1,\ b_2), \dots, \texttt{insertAfter}(b_{n-1},\ b_n)].$$

The first operation received by the server (or in the $\prec$ order) must be either $\texttt{insertAfter}(p,\ a_1)$ or $\texttt{insertAfter}(p,\ b_1)$; without loss of generality, it is $\texttt{insertAfter}(p,\ a_1)$. When $\texttt{insertAfter}(p,\ b_1)$ is received sometime later, it will update $pa_1 \dots a_k \rightarrow pb_1a_1 \dots a_k$ for some $1 \leq k \leq m$. This order of sentences (b before a) is then preserved by each remaining operation, so by induction, the final list of IDs is $pb_1 \dots b_n a_1 \dots a_m$.

Remark 5.5. Kleppmann et al. [2018] prove the same property for RGA—the CRDT from Theorem 5.4.

What about right-to-left insertions, which happen for example when two users concurrently prepend items to a shopping list [Weidner and Kleppmann 2025]? Each prepended item corresponds to an `insertAfter` operation with `beforeId = null` (the beginning of the list); as these operations are processed, their items end up in *reverse* operation order. Thus items are interleaved if and only if the users' operations are interleaved in the operation order.

Figure 8 illustrates that in the decentralized setting, when two users work concurrently for a period of time, Lamport timestamps interleave their operations, while a depth-first topological sort of the causal order does not. Thus, if two users concurrently prepend several items to a shopping list, our decentralized algorithm will interleave their items when using the Lamport timestamp order, but it avoids doing so when using a depth-first topological sort. This fact could justify the depth-first topological sort's more difficult implementation.

However, the depth-first topological sort still does not achieve the **maximal non-interleaving** property defined by Weidner and Kleppmann [2025], i.e., it will needlessly interleave backwards insertions in some cases. To see this, consider the collaborative text-editing scenario in their Figure 7. In that scenario, once the algorithm arbitrarily orders $A < B < C$, maximal non-interleaving mandates the final list order $AXYBC$. On the other hand, one allowed depth-first topological sort

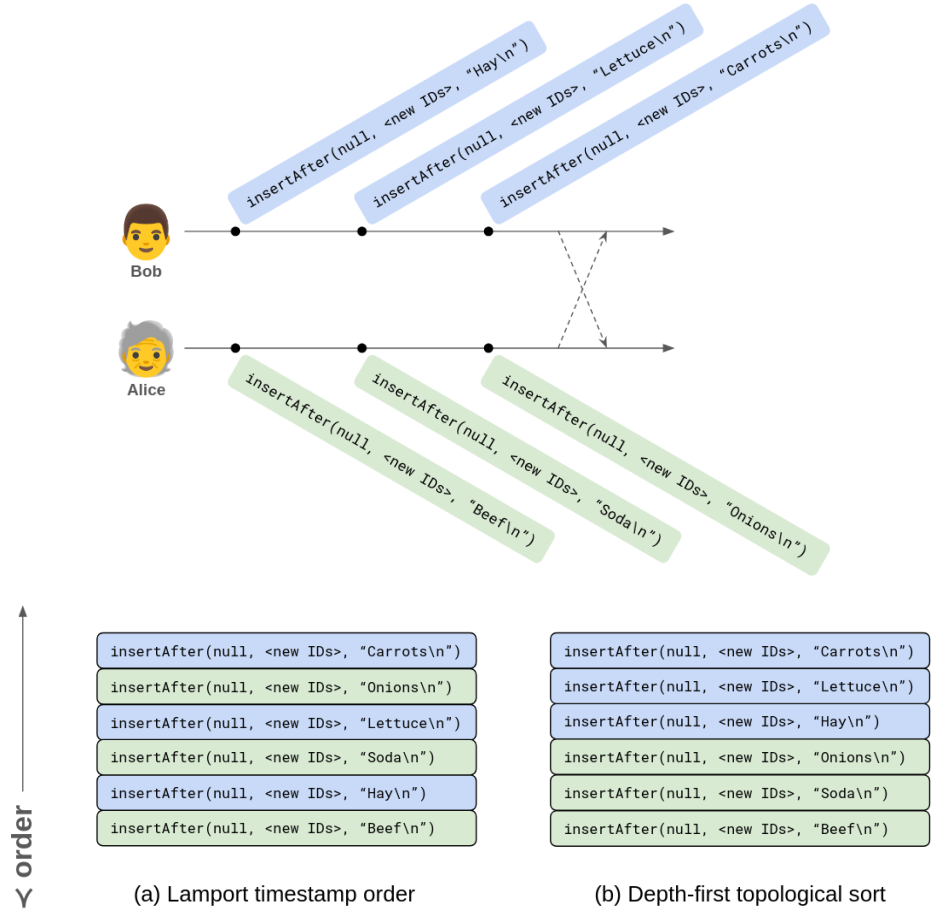

Fig. 8. Alice and Bob prepend items to a shopping list in concurrent branches before exchanging operations. The resulting operation order (bottom to top) and shopping list (top to bottom) depend on $\prec$. (a) Lamport timestamps are assigned sequentially in each branch, breaking ties by client ID. This interleaves operations and hence the prepended list items. (b) A depth-first topological sort orders one branch before the other. This avoids interleaving operations and hence the prepended list items.

of the causal order inserts in order $C \prec B \prec A \prec X \prec Y$. With the scenario's implied `insertAfter` operations, this order results in the final list order $AYXBC$ instead of $AXYBC$.

When using a central server, the degree of interleaving for concurrent right-to-left insertions is more ambiguous. On the one hand, the server can receive operations from two users in an arbitrary interleaved order, thus interleaving right-to-left insertions at the same starting position. On the other hand, simultaneously-online users typically observe each other's edits and manually avoid interleaving, while a well-programmed client will submit offline operations in a single batch upon reconnecting—again avoiding interleaving.

Remark 5.6. One can define a collaborative operation `insertBefore(afterId, id, char)` that is the reverse of `insertAfter` and has reversed non-interleaving guarantees. An application can

choose to use this operation when the user explicitly inserts from right to left, e.g., because they clicked a button with "prepend" semantics.

6 EVALUATION

Character IDs impose a performance cost on top of storing the characters alone. We suspect that this cost is the main limit on their adoption: with a naïve implementation, such as storing a UUIDv4 per character, the extra overhead is not worth the programming advantages described above. Thus we developed benchmarks to measure the cost of character IDs, both in absolute terms and relative to existing text-editor data structures.

The benchmarks use a text-editing trace by Martin Kleppmann.[6] It records every keystroke used to write a 17-page LaTeX paper, comprising 182,315 single-character insert operations and 77,463 single-character delete operations; the final document size is 104,852 characters. The trace is already used to benchmark text-editing CRDTs,[7] though note that our benchmarks are single-user—they measure local edits (or their repetition on a server or a collaborator's device), without any concurrency. When executing the trace, we reloaded each data structure every 1,000 edits, simulating a user who edits across multiple text editor sessions.

We benchmarked Articulated, our list-of-IDs data structure from Section 3, alongside traditional text-editor data structures that store only characters:

String A simple text string, edited with slice and concatenation operations.

Character Array An array of characters, edited with JavaScript's `Array.splice` method.

Rope The persistent rope implementation by Marijn Haverbeke,[8] used to store a list of characters.

ProseMirror The `EditorState` data structure used by the ProseMirror rich-text editor.[9] For our plain-text editing trace, it essentially amounts to an array of lines.

We also benchmarked existing optimized text-editing CRDTs. Like Articulated, these are used alongside a text editor's own data structure to enable collaboration, and they internally assign an ID to each character (see Section 7). However, they also store extra metadata specific to CRDTs—for example, Automerge stores the full operation history. We include the CRDT libraries to show an upper bound on what is possible in practice, not as a head-to-head comparison.

Automerge The Automerge CRDT's `string` type v3.5.0 [Automerge contributors 2026].

Yjs The Yjs CRDT's `Y.Text` type v13.6.33 [Jahns 2026].

All implementations are written in TypeScript or JavaScript, except for Automerge, whose core is written in Rust and compiled to WebAssembly. The benchmarking code was written in TypeScript with assistance from Claude Code, using GitHub Copilot for code review. All generated code was proofread manually, except for a script to extract retained size data from v8 heap snapshots, whose output was spot-checked against the same data reported in Google Chrome's heap snapshot viewer.

Results were gathered on an M4 MacBook Pro with macOS Sequoia v15.7.7 and 48 GB of RAM, running Node.js v24.16.0 via TSX[10] v4.23.13. Each metric reported below is an average over 10 trials, and error bars show a 95% confidence interval for the average. Unless otherwise specified, the average is arithmetic mean and the confidence interval is based on the Student t-distribution [Kounev et al. 2025]. Each trial was run in its own Node.js process, with the process running five

[6] https://github.com/automerge/automerge-perf
[7] https://github.com/dmonad/crdt-benchmarks/
[8] https://code.haverbeke.berlin/marijn/rope-sequence/releases/tag/1.3.4
[9] https://code.haverbeke.berlin/prosemirror/prosemirror-state/src/tag/1.4.4
[10] TSX (https://tsx.hirok.io/) is a command-line tool that runs TypeScript code in Node.js without requiring a separate command to transpile TypeScript to JavaScript.

Table 2. Memory Usage for the Final Document State

Category	Implementation	Memory Usage (kB)
Traditional text-editor data structure	String	105 ± 0
	Character Array	839 ± 0
	Rope	839 ± 0
	ProseMirror	382 ± 7
List of IDs	Articulated	2613 ± 2
CRDT	Yjs	3043 ± 0

warmup runs followed by the measured run. Trials were executed sequentially in pseudorandom order.

6.1 Memory and Storage

Our primary concern with character IDs is that they may take up too much space: either in memory while the text editor is running, or on disk when stored alongside a text document.

Memory. We measured the memory used by each data structure at the end of the text-editing trace, when the actual document size is 105 kB.[11] Specifically, we reloaded the data structure one last time at the end of the trace, recorded a v8 heap snapshot, and then extracted the data structure's retained size from the snapshot. Automerge was omitted because its WASM memory does not live on the heap and is more difficult to measure.

Table 2 shows the results. Articulated uses several times as much memory as any of the traditional text-editor data structures, including the ProseMirror text editor's state. However, its memory usage is still small in absolute terms: less than 3 MB for a 17-page document. We consider this acceptable in practice, and it is likely dominated by the memory usage of other parts of an application, such as UI elements. Articulated's memory usage is also within the upper bound demonstrated by the optimized Yjs CRDT.

Storage. Table 3 shows the saved state size for the document at the end of the trace, plus how long it takes to save and load that saved state. For the traditional text-editor data structures, the saved state is just the 105 kB text string; for Articulated, it represents the character IDs without any text; and for the CRDT libraries, it represents the text plus CRDT metadata. Articulated's default saved state is a simple but somewhat verbose JSON object, so we also show the effect of GZIP compression using the fflate JavaScript library.[12] (Automerge already uses Deflate compression internally [Good and Jeffery 2025], while Yjs uses a custom binary format that is not intended for further compression.)

Articulated's compressed saved state size is a reasonable 2.54× multiple of the compressed text size, and it is actually smaller than the uncompressed text. It is also within the upper bounds demonstrated by Automerge and Yjs. Note that even the uncompressed size is comparable to a single image file. However, the compressed state's save and load times are much longer than for any of the traditional text-editor data structures. The save time is especially concerning because saving is a frequent background operation and 21.7 ± 1.2 ms is long enough to drop a frame if done

[11] All trace characters are ASCII, so the document's 104,852 characters round to 105 kB both as a UTF-8 file and in v8's internal string storage—despite JavaScript's nominal use of UTF-16 [Thier 2025].
[12] https://github.com/101arrowz/fflate

Table 3. Storage Metrics for the Final Document State

Category	Implementation	Saved State Size (kB)	Save Time (ms)	Load Time (ms)
Actual Text	String	105 ± 0		
	String w/ GZIP	28.3 ± 0.0	2.82 ± 0.06	2.83 ± 0.16
List of IDs	Articulated	876 ± 0	2.35 ± 0.24	15.9 ± 0.8
	Articulated w/ GZIP	71.9 ± 0.0	21.7 ± 1.2	19.0 ± 0.7
CRDT	Automerge	156 ± 0	17.4 ± 0.1	441 ± 6
	Yjs	289 ± 0	4.77 ± 0.08	29.9 ± 1.2

Table 4. Throughput and Distribution of Operation Times

Category	Implementation	Throughput (ops/sec)	P90 (μs)	P99 (μs)	Max (μs)
Traditional text-editor data structure	String	$[629k, 633k]$	2.20 ± 0.02	2.54 ± 0.01	422 ± 82
	Character Array	$[282k, 285k]$	8.85 ± 0.04	10.9 ± 0.1	1964 ± 398
	Rope	$[211k, 222k]$	1.87 ± 0.04	51.4 ± 1.5	3125 ± 471
	ProseMirror	$[165k, 166k]$	11.5 ± 0.1	16.8 ± 0.1	16,510 ± 1407
List of IDs	Articulated	$[683k, 693k]$	1.58 ± 0.00	3.84 ± 0.04	543 ± 24
CRDT	Automerge	$[30.8k, 31.4k]$	35.5 ± 0.3	41.4 ± 0.4	4105 ± 34
	Yjs	$[91.8k, 93.4k]$	17.7 ± 0.1	29.0 ± 0.5	700 ± 20

on the JavaScript main thread. We plan to improve the size vs. save time trade-off in the future by replacing GZIP with a Yjs-inspired custom binary format.

6.2 Operation Times

Our next benchmarks measure the time per editing operation. Specifically, they measure the single-threaded CPU time needed to synchronously process each index-based insert and delete operation. For Articulated, inserting generates a new character ID and inserts it at the correct index, while deleting looks up an ID by index and marks it as a tombstone. For the CRDT libraries, each operation edits the text, updates CRDT metadata, and prepares a message for collaborators (which our benchmarks then ignore).

Table 4 shows per-trial aggregates across the entire text-editing trace. The 95% confidence intervals apply to the arithmetic mean of each aggregate across trials, except for throughput, where we computed a confidence interval for the average operation time and then inverted its endpoints. We focus on throughput and tail operation times as the most interesting aggregates. High throughput (equivalently, low CPU usage) allows a collaboration server to handle more documents and reduces power usage on mobile devices, while low tail operation times avoid user-perceived jitter from dropped frames.

We see that Articulated's maximum operation time is less than one millisecond, well within a single frame. Its throughput and tail latencies are comparable to the traditional text-editor data structures that it would be used alongside, and throughput is more than adequate for clients: at typical typing speeds of about 60 words per minute, even a collaborative document with dozens of active editors will experience less than 1,000 operations per second.

One drawback relative to CRDTs is that when using Articulated in a server reconciliation architecture, a client's rollback-and-reapply procedure (Figure 6) might apply the same pending

local operation multiple times before it is acknowledged by the server. However, Articulated's higher throughput than the CRDTs gives us leeway, and one can reduce re-applications by throttling the rollback-and-reapply procedure: deliver remote operations in occasional batches, e.g., at most once every 100 ms.

7 RELATED WORK

Operational Transformation. The most closely related work is found in collaborative text-editing algorithms, since collaborative operations must address characters consistently across network delays.

The study of collaborative text-editing algorithms began with Ellis and Gibbs [1989], who introduced Operational Transformation (OT). In this family of algorithms, clients send traditional index-based operations over the network. Concurrent scenarios like Figure 2 are solved via **transformation**: when the server receives Bob's operation `insertAt(17, " high")`, it first transforms that operation against Alice's concurrent operation `insertAt(4, "gray ")`, yielding the operation `insertAt(22, " high")`, which it applies literally. In general, the server transforms each received operation against all concurrent operations that it has already applied, using a type-specific transformation function. Likewise, a client transforms operations from the server against its own pending local operations before applying them to its local state. Clients in a decentralized setting use a more complicated series of transformations [Ressel et al. 1996].

There are two challenges when designing an OT algorithm. First, one must define a transformation rule for every pair of operations, hence the programming effort grows quadratically in the number of operation types. Second, to ensure that clients and the server eventually converge to the same state (despite transforming and applying operations in different orders), one must ensure a mathematical **transformation property** for every pair of operations. In the decentralized setting, there is a second transformation property required for every triple of operations; several published algorithms incorrectly claim to satisfy this second transformation property, and a valid solution was first found only via formal verification and the use of tombstones [Oster et al. 2006a].

Our ID-based editing operations sidestep the first challenge by allowing concurrent operations to make sense as-is, without requiring transformation. Server reconciliation (Section 4.1) and decentralized total orders (Section 5.1) sidestep the second challenge by using the same order of operations on all devices. Note that outside of the academic literature, some OT implementations already exploit a central server's total order, allowing them to skip the usual transformation properties [Haverbeke 2015; Zapp 2023].

Interestingly, Ellis and Gibbs were already aware of server reconciliation in their original paper on OT, under the name "reversible execution" [Sarin and Greif 1985]. They reject it by giving the example of two concurrent `deleteAt(index)` operations targeting the same index, which yield an unreasonable result—two characters are deleted instead of one—when applied literally in any serial order. As a consequence, they claim that collaborative text-editing requires "non-serializable correctness criteria", i.e., a fundamentally distributed approach like OT. Our perspective is that the issue is not with the serial order, but with indexes. When instead using character IDs, two identical `delete(id)` operations give the expected result when applied in either serial order.

Conflict-free Replicated Data Types. The first Conflict-free Replicated Data Type (CRDT) was introduced by Oster et al. [2006b] as an alternative to OT for collaborative text editing. While the definition of a text-editing CRDT is quite broad (indeed, the decentralized approach in Section 5.1 is technically a CRDT [Kleppmann et al. 2018]), published text-editing CRDTs typically have the following form [Preguiça et al. 2009; Roh et al. 2011; Weiss et al. 2009]:

(1) Each character is labeled by a unique and immutable element from an abstract total order. Different CRDTs use different total orders; often, it is a depth-first traversal of a special kind of tree.
(2) When a user types a character, their client labels it with a new total order element that sorts correctly relative to the neighboring characters' elements. The client instructs all others to store the character along with this element.
(3) When a user deletes a character, their client instructs all others to mark the character's total order element as a tombstone.

CRDTs' total order elements are conceptually similar to our character IDs. The main difference is the total order itself: our IDs are simply IDs, without an intrinsic total order; they are instead ordered *by fiat* on the server as it processes operations. This means that we can skip designing and implementing a total order of the required form, which is the complicated part of a text-editing CRDT.[13]

RGASplit [Briot et al. 2016] and Yjs [Jahns 2026] both compress total order elements that are inserted sequentially by the same user. We apply the same optimization to IDs in our list-of-IDs data structure (Section 3).

Eg-walker is a CRDT implementation technique that converts total order operations to index-based operations internally, so that they can be applied very quickly [Gentle and Kleppmann 2025]. It would be interesting to explore whether this technique also works with our collaborative editing algorithms.

ID-Based Collaborative Editing Algorithms. Recent implementations already use character or list element IDs for collaboration, in the manner described in this paper.

The Thymer IDE uses server reconciliation to implement collaboration on document outlines [Cools 2024]. Specifically, the outline is a tree in which each node has a UUID. Edits use the operation `insmov(id, parentId, siblingId)`, which inserts or moves the node with ID `id` so that it is a child of `parentId` ordered directly after `siblingId`. When inserts target the same `parentId`, they behave exactly like our `insertAfter` operations; thus sibling lists follow the algorithm in Section 4.1. This approach directly inspired our work.

The Classic Jazz CRDT library [Garden Computing 2026] uses a decentralized total order internally to implement various CRDTs. Its CoList and CoText CRDTs apply this total order to ID-based operations, matching Section 5.1 [Eickhoff 2025]. Wonlaw [2023] also describes a decentralized total order and gives ID-based list operations (with tombstones) as an example.

On the theoretical side, Attiya et al. [2016] use globally unique identifiers instead of characters when defining the strong list specification, so that they can unambiguously refer to list elements over time. Their specification still uses index-based insert operations, but it does use ID-based delete operations.

Text Editor Data Structures. Outside of collaborative editing, our list-of-IDs implementation (Section 3) is closely related to the rope text-editor data structure [Boehm et al. 1995]. A rope is a balanced binary tree whose leaves store substrings of a text string; the full text is the concatenation of those substrings in depth-first traversal order. The balanced tree allows one to perform common text-editor operations—insertion, deletion, and getting a character by index—in logarithmic time. Our list-of-IDs implementation is similar, except that the tree's leaves store compressed ID subsequences instead of substrings.

[13] While our list-of-IDs data structure uses a tree, it is a local tree that is solely an optimization—unlike the replicated total-ordering trees that appear in several text-editing CRDTs, which are a core part of their algorithms. Some CRDT implementations include a B+ Tree like ours as an optimization on top of their core total order [Sobo 2022].

The piece table is another text-editor data structure [Crowley 1998]. It stores characters in two buffers, a read-only "original buffer" and an append-only "add buffer"; a separate table describes the current text as a series of substrings from those buffers, referenced by their indexes within the buffers. Given a piece table, one can construct immutable character IDs as pairs `(which buffer, index into that buffer)`. Crowley [1998] remarks that these pairs can be used to implement undo histories, formatting spans, and document hyperlinks. However, it is not clear whether they can be used for collaborative editing, and we do not know of any text editors that do so.

Finseth [1991] describes a generic text-editor implementation and includes an API for "marks", defined as "object[s] that can remember a position". A mark lies between two characters and moves around with those characters, behaving like a cursor or the start/end of an annotation. CKEditor 5 has a similar concept called "markers", which are shared between users as a feature of their OT algorithm [Cofalik and Tomanek 2018]. Unlike our character IDs, one must create each mark/marker explicitly.

8 CONCLUSION

We have described character IDs (plus tombstones) and their applications to text editing. In particular, we described how ID-based editing operations are more natural than index-based operations, and they easily enable collaboration when used with server reconciliation or a decentralized total order of operations. The resulting collaborative text-editing algorithms provide a simple alternative to existing work based on Operational Transformation or Conflict-free Replicated Data Types. We also presented Articulated, an open-source TypeScript library to generate and store character IDs, and we demonstrated that it has practical performance on a real text-editing trace.

In future work, it would be interesting to apply IDs to more complex ordered data structures, such as abstract syntax trees or block-based text with movable blocks. The challenge is to design ID-based operations that behave reasonably when applied to a slightly further-ahead state on the server. Doing so would enable collaboration while also clarifying our own mental models of those data structures.

ACKNOWLEDGMENTS

I am grateful to Scott Ames-Messinger for numerous discussions about collaborative text-editing, server reconciliation, and ProseMirror, and for his encouragement of this work. I also thank Wim Cools for his description of the Thymer IDE's collaborative editing algorithm that inspired this paper, and Anselm Eickhoff for bringing my attention to Classic Jazz's related implementation. Michael Wang contributed to the Articulated repository. A demo of collaborative editing using Articulated, ProseMirror, and PowerSync was developed with funding from Journey Mobile, Inc.

REFERENCES

Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and Complexity of Collaborative Text Editing. In *ACM Symposium on Principles of Distributed Computing (PODC 2016)*. ACM, 259–268. https://doi.org/10.1145/2933057.2933090

Automerge contributors. 2026. Automerge. GitHub repository. https://github.com/automerge/automerge

Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An alternative to strings. *Software: Practice and Experience* 25, 12 (1995), 1315–1330. https://doi.org/10.1002/spe.4380251203 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380251203

Loïck Briot, Pascal Urso, and Marc Shapiro. 2016. High Responsiveness for Group Editing CRDTs. In *19th International Conference on Supporting Group Work (GROUP 2016)*. ACM, 51–60. https://doi.org/10.1145/2957276.2957300

Alex Clemmer. 2025. Lies I was Told About Collaborative Editing, Part 1: Algorithms for offline editing. https://www.moment.dev/blog/lies-i-was-told-pt-1, archived at https://perma.cc/PE9M-9DSH. Accessed: 2026-05-02.

Szymon Cofalik and Anna Tomanek. 2018. How collaborative editing drove CKEditor 5's architecture. https://ckeditor.com/blog/lessons-learned-from-creating-a-rich-text-editor-with-real-time-collaboration/, archived at https://perma.cc/2589-

7P2Z. Accessed: 2026-05-17.

Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. https://doi.org/10.1145/356770.356776

Wim Cools. 2024. Hacker News comment. https://news.ycombinator.com/item?id=41100477. Accessed: 2026-05-02.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Charles Crowley. 1998. *Data Structures for Text Sequences.* Technical Report. University of New Mexico. https://www.cs.unm.edu/~crowley/papers/sds.pdf

John Day-Richter. 2010. What's different about the new Google Docs: Making collaboration fast. https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html, archived at https://perma.cc/KT7F-2EAT.

Anselm Eickhoff. 2025. Tweet. https://x.com/anselm_io/status/1925227663720935532, archived at https://perma.cc/KQV9-RLQY. Accessed: 2026-05-02.

C. A. Ellis and S. J. Gibbs. 1989. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 399–407. https://doi.org/10.1145/67544.66963

Craig A. Finseth. 1991. *The craft of text editing: Emacs for the modern world.* Springer-Verlag, Berlin, Heidelberg. https://finseth.com/craft/

Gabriel Gambetta. [n. d.]. Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation. https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html, archived at https://perma.cc/4TVX-ATUE. Accessed: 2026-05-02.

Garden Computing. 2026. Classic Jazz. GitHub repository. https://github.com/garden-co/classic-jazz Accessed: 2026-05-03.

Joseph Gentle. 2011. ShareJS – Live concurrent editing in your app. https://sharejs.org/, archived at https://perma.cc/VRU2-4FP8. Accessed: 2026-05-02.

Joseph Gentle and Martin Kleppmann. 2025. Collaborative Text Editing with Eg-walker: Better, Faster, Smaller. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 311–328. https://doi.org/10.1145/3689031.3696076

Alex Good and Andrew Jeffery. 2025. Binary Document Format. https://automerge.org/automerge-binary-format-spec/, archived at https://perma.cc/9MV6-7F24. Accessed: 2026-10-12.

Victor Grishchenko. 2010. Deep Hypertext with Embedded Revision Control Implemented in Regular Expressions. In *6th International Symposium on Wikis and Open Collaboration (WikiSym 2010)*. ACM, Article 3. https://doi.org/10.1145/1832772.1832777

Marijn Haverbeke. 2015. Collaborative Editing in ProseMirror. https://marijnhaverbeke.nl/blog/collaborative-editing.html, archived at https://perma.cc/T8LE-SFFL. Accessed: 2026-05-02.

Florian Jacob, Luca Becker, Jan Grashöfer, and Hannes Hartenstein. 2020. Matrix Decomposition: Analysis of an Access Control Approach on Transaction-based DAGs without Finality. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies* (Barcelona, Spain) *(SACMAT '20)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3381991.3395399

Kevin Jahns. 2020. Are CRDTs suitable for shared editing? https://blog.kevinjahns.de/are-crdts-suitable-for-shared-editing/, archived at https://perma.cc/5GAY-MZWH. Accessed: 2026-05-02.

Kevin Jahns. 2026. Yjs. GitHub repository. https://github.com/yjs/yjs

Martin Kleppmann. 2020. Moving elements in list CRDTs. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data* (Heraklion, Greece) *(PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3380787.3393677

Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. 2018. OpSets: Sequential Specifications for Replicated Datatypes (Extended Version). *CoRR* abs/1805.04263 (2018). arXiv:1805.04263 http://arxiv.org/abs/1805.04263

Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. 2025. *Systems Benchmarking* (2 ed.). Springer International Publishing. https://doi.org/10.1007/978-3-031-85634-1

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

Paul J. Leach, Rich Salz, and Michael H. Mealling. 2005. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122. https://doi.org/10.17487/RFC4122

Raph Levien. 2019. GitHub Issue comment. https://github.com/xi-editor/xi-editor/issues/1187#issuecomment-491473599, archived at https://perma.cc/ZG5C-SGNM. Accessed: 2026-05-02.

Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2022. Peritext: A CRDT for Collaborative Rich Text Editing. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW2, Article 531 (Nov. 2022), 36 pages. https://doi.org/10.1145/3555644

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. A Short Primer on Causal Consistency. *;login:* 38, 4 (Aug. 2013), 41–43. https://www.usenix.org/publications/login/august-2013-volume-38-number-4/short-primer-causal-consistency

Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. 2006a. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 1–10. https://doi.org/10.1109/COLCOM.2006.361867

Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006b. Data consistency for P2P collaborative editing. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 259–268. https://doi.org/10.1145/1180875.1180916

PowerSync. [n. d.]. PowerSync: Consistency. https://docs.powersync.com/architecture/consistency, archived at https://perma.cc/RSC2-Y7M9. Accessed: 2026-05-02.

Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 395–403. https://doi.org/10.1109/ICDCS.2009.20

Quill contributors. [n. d.]. Designing the Delta Format. https://quilljs.com/docs/guides/designing-the-delta-format, archived at https://perma.cc/HQ8G-VB8B. Accessed: 2026-10-13.

Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. 1996. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 288–297. https://doi.org/10.1145/240080.240305

Rocicorp LLC. 2025. Replicache Documentation: How Replicache Works. https://doc.replicache.dev/concepts/how-it-works, archived at https://perma.cc/8292-58DU. Accessed: 2026-05-02.

Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

Sunil K. Sarin and Irene Greif. 1985. *Computer-based Real-time Conferences*. Technical Report. MIT LCS. https://dspace.mit.edu/handle/1721.1/149091

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report 7506. INRIA. http://hal.inria.fr/inria-00555588/

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Vol. 6976. Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29 Series Title: Lecture Notes in Computer Science.

Nathan Sobo. 2022. How CRDTs make multiplayer text editing part of Zed's DNA. https://zed.dev/blog/crdts, archived at https://perma.cc/S895-3XSY. Accessed: 2026-05-02.

Chengzheng Sun and Clarence Ellis. 1998. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 59–68. https://doi.org/10.1145/289444.289469

Patrick Thier. 2025. How we made JSON.stringify more than twice as fast. https://v8.dev/blog/json-stringify, archived at https://perma.cc/M3DS-3HDL. Accessed: 2026-10-12.

Matthew Weidner and Martin Kleppmann. 2025. The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing. *IEEE Transactions on Parallel and Distributed Systems* 36, 11 (2025), 2425–2437. https://doi.org/10.1109/TPDS.2025.3611880

Stephane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems*. 404–412. https://doi.org/10.1109/ICDCS.2009.75

Matt Wonlaw. 2023. A Framework for Convergence. https://vlcn.io/blog/crdt-substrate, archived at https://perma.cc/JVN8-97YL. Accessed: 2026-05-02.

Greg Zapp. 2023. ProseMirror Collab Performance. https://stepwisehq.com/blog/2023-07-25-prosemirror-collab-performance/, archived at https://perma.cc/E2NQ-AEKF. Accessed: 2026-05-02.

A AN INCREMENTAL DEPTH-FIRST TOPOLOGICAL SORT OF THE CAUSAL ORDER

When describing the decentralized collaborative text-editing algorithm in Section 5.1, we mentioned a depth-first topological sort of the causal order as an example of a decentralized total order. Cormen et al. [2009] describe an algorithm for a depth-first topological sort of an arbitrary DAG, which can be applied to the causal order in any particular state. However, what we actually require is an *incremental* algorithm that updates the topological sort to include a new operation while preserving the order of existing operations. This section describes such an algorithm.

Let us say that an operation is **active** on a client if that client is aware of (sent or received) that operation and all of its causal dependencies. (The active operations are precisely those that contribute to the current state in the decentralized collaborative algorithm.) The **immediate causal dependencies** of an operation are the maximal elements among its set of causal dependencies. Note that an operation's causal dependencies are satisfied if and only if its immediate causal dependencies are all active. Dually, the **immediate causal dependents** of an operation are the minimal elements among the set of operations that it happened before.

To avoid edge cases, we invent a special "root" operation that happened before all other operations, i.e., it is the minimum element of the causal order.

The state of our topological sorting algorithm on each client is a tree containing a node for each active operation, including the special root operation. The decentralized total order of operations $\prec$ is the depth-first pre-order traversal of this tree in which siblings are sorted in an arbitrary but deterministic way, e.g., by their senders' client IDs (we ensure below that siblings have distinct client IDs).

The tree's edges are constructed incrementally: when a client adds a newly active operation to the tree, that operation's parent is the immediate causal dependency that is *last* in the existing traversal order. If the operation has no immediate causal dependencies, its parent is the root.

Lemma A.1. *The above total order $\prec$ is consistent across time and across clients: if $o \prec p$ in one state where both are active, then $o \prec p$ in all states where both are active.*

Proof. We claim that for every operation q, q's parent node is the same in all client states where q is active. To see this, inductively assume that the claim is true for all operations that happened before q. Then the operations that happened before q form the same subtree in all client states. It follows that their tree traversal order is the same in all client states. In particular, in any client state just before q becomes active, its immediate causal dependencies have the same traversal order. Thus q is always assigned the same parent node when it becomes active.

Now consider two states in which both o and p are active. Restricting to o, p, and the union of their causal dependencies, the corresponding subtrees are identical in both states, by the above claim. Thus o and p are ordered identically in both states. □

Let us also verify the decentralized total order properties. For (1), it suffices to prove that $o \prec p$ whenever o is an immediate causal predecessor of p. This holds for the o that is last in the traversal order because o is the parent of p and we use a pre-order traversal. The other immediate causal predecessors are less than the last one by definition, hence also less than p. (2) is obvious from our algorithm description.

We claimed above that sibling operations have distinct client IDs, hence client IDs can be used as the arbitrary sibling order. Indeed, sibling operations p and r share an immediate causal predecessor o by the tree's construction. If p and r are not concurrent, so that without loss of generality p happened before r, then o is not maximal among the operations that happened before r, a contradiction.

Finally, we should justify that in each state, the above total order is actually a depth-first topological sort in the sense of Cormen et al. [2009], i.e., it is an allowed output of their algorithm applied to the causal order DAG. Indeed, their algorithm yields our total order when it makes the following arbitrary choices:

- The for-each loop that visits all nodes proceeds in order: the root node first, followed by all others in arbitrary order (immediately skipped as they were already visited recursively).
- The for-each loop over a node's outgoing edges proceeds in order: all of that node's immediate causal dependents in *reverse* order of their senders' client IDs, then that node's other

causal dependents in arbitrary order (immediately skipped as they were already visited recursively).