Collaborative Text Editing: Better, Faster, Smaller

Joseph Gentle
me@josephg.com

Martin Kleppmann
University of Cambridge, UK
martin@kleppmann.com

Abstract

Collaborative text editing algorithms allow several users to concurrently modify a text file, and automatically merge concurrent edits into a consistent state. Existing algorithms fall in two categories: Operational Transformation (OT) algorithms are slow to merge files that have diverged substantially due to offline editing; CRDTs are slow to load and consume a lot of memory. We introduce Eg-walker, a collaboration algorithm for text that avoids these weaknesses. Compared to existing CRDTs, it consumes an order of magnitude less memory in the steady state, and loading a document from disk is orders of magnitude faster. Compared to OT, merging long-running branches is orders of magnitude faster. In the worst case, the merging performance of Eg-walker is comparable with existing CRDT algorithms. Eg-walker can be used everywhere CRDTs are used, including peer-to-peer systems without a central server. By offering performance that is competitive with centralised algorithms, our result paves the way towards the widespread adoption of peer-to-peer collaboration software.

1 Introduction

Real-time collaboration has become an essential feature for many types of software, including document editors such as Google Docs, Microsoft Word, or Overleaf, and graphics software such as Figma. In such software, each user's device locally maintains a copy of the shared file (e.g. in a tab of their web browser). A user's edits are immediately applied to their own local copy, without waiting for a network round-trip, so that the user interface is responsive regardless of network latency. Different users may therefore make edits concurrently, and the software must merge such concurrent edits in a way that preserves the users' intentions, and ensure that all devices converge to the same state.

For example, in Figure 1, two users initially have the same document "Helo". User 1 inserts a second letter "l" at index 3, while concurrently user 2 inserts an exclamation mark at index 4. When user 2 receives the operation $Insert(3, \text{"}l\text{"})$ it can apply it to obtain "Hello!", but when user 1 receives $Insert(4, \text{"!"})$ it cannot apply that operation as-is, since that would result in the state "Hell!o", which would be inconsistent with the other user's state and the intended insertion position. Due to the concurrent insertion at an earlier index, user 1 must insert the exclamation mark at index 5.

One way of solving this problem is to use *Operational Transformation* (OT): when user 1 receives $Insert(4, \text{"!"})$ that operation is transformed with regard to the concurrent

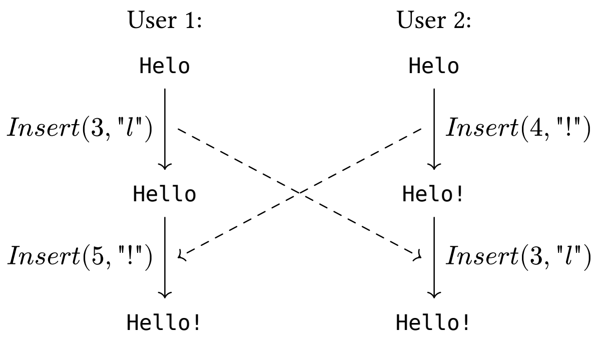


Figure 1. Two concurrent insertions into a text document.

insertion at index 3, which increments the index at which the exclamation mark is inserted. OT is an old and widely-used technique: it was introduced in 1989 [17], and the OT algorithm Jupiter [40] forms the basis of real-time collaboration in Google Docs [16].

OT is simple and fast in the case of Figure 1, where each user performed only one operation since the last version they had in common. In general, if the users each performed n operations since their last common version, merging their states using OT has a cost of at least $O(n^2)$, since each of one user's operations must be transformed with respect to all of the other user's operations. Some OT algorithms have a merge complexity that is cubic or even slower [37,47,52]. This is acceptable for online collaboration where n is typically small, but if users may edit a document offline or if the software supports explicit branching and merging workflows [38], an algorithm with complexity $O(n^2)$ can become impracticably slow. In Section 4 we show a real-life example document that takes one hour to merge using OT.

Conflict-free Replicated Data Types (CRDTs) have been proposed as an alternative to OT. The first CRDT for collaborative text editing appeared in 2006 [43], and over a dozen text CRDTs have been published since [31]. These algorithms work by giving each character a unique identifier, and using those IDs instead of integer indexes to identify the position of insertions and deletions. This avoids having to transform operations, since IDs are not affected by concurrent operations. Unfortunately, these IDs need to be held in memory while a document is being edited. Even with careful optimisation, this metadata uses more than 10 times as much memory as the document text, and makes documents much slower to load from disk. Some CRDT algorithms also need to retain IDs of deleted characters (*tombstones*).

In this paper we propose *Event Graph Walker* (Eg-walker), a collaborative editing algorithm that has the strengths of both OT and CRDTs but not their weaknesses. Like OT, Eg-walker uses integer indexes to identify insertion and deletion positions, and transforms those indexes to merge concurrent operations. When two users concurrently perform n operations each, Eg-walker can merge them at a cost of $O(n \log n)$, much faster than OT's cost of $O(n^2)$ or worse.

Eg-walker merges concurrent edits using a CRDT algorithm we designed. Unlike existing algorithms, we invoke the CRDT only to perform merges of concurrent operations, and we discard its state as soon as the merge is complete. We never write the CRDT state to disk and never send it over the network. While a document is being edited, we only hold the document text in memory, but no CRDT metadata. Most of the time, Eg-walker therefore uses 1–2 orders of magnitude less memory than a CRDT. During merging, when Eg-walker temporarily uses more memory, its peak memory use is comparable to the best known CRDT implementations.

Eg-walker assumes no central server, so it can be used over a peer-to-peer network. Although all existing CRDTs and a few OT algorithms can be used peer-to-peer, most of them have poor performance compared to the centralised OT used in production software such as Google Docs. In contrast, Eg-walker's performance matches or surpasses that of centralised algorithms. It therefore paves the way towards the widespread adoption of peer-to-peer collaboration software, and perhaps overcoming the dominance of centralised cloud software that exists in the market today.

Collaboration on plain text files is the first application for Eg-walker. We believe that our approach can be generalised to other file types such as rich text, spreadsheets, graphics, presentations, CAD drawings, and more. More generally, Eg-walker provides a framework for efficient coordination-free distributed systems, in which nodes can always make progress independently, but converge eventually [27].

This paper makes the following contributions:

- In Section 3 we introduce Eg-walker, a hybrid CRDT/OT algorithm for text that is faster and has a vastly smaller memory footprint than existing CRDTs.
- Since there is no established benchmark for collaborative text editing, we are also publishing a suite of editing traces of text files for benchmarking. They are derived from real documents and demonstrate various patterns of sequential and concurrent editing.
- In Section 4 we use those editing traces to evaluate the performance of our implementation of Eg-walker, comparing it to selected CRDTs and an OT implementation. We measure CPU time to load a document, CPU time to merge edits from a remote replica, memory usage, and file size. Eg-walker improves the state of the art by orders of magnitude in the best cases, and is only slightly slower in the worst cases.
- We prove the correctness of Eg-walker in Appendix B.

2 Background

We consider a collaborative plain text editor whose state is a linear sequence of characters, which may be edited by inserting or deleting characters at any position. Such an edit is captured as an *operation*; we use the notation $Insert(i, c)$ to denote an operation that inserts character c at index i, and $Delete(i)$ deletes the character at index i (indexes are zero-based). Our implementation compresses runs of consecutive insertions or deletions, but for simplicity we describe the algorithm in terms of single-character operations.

2.1 System model

Each device on which a user edits a document is a *replica*, and each replica stores its full editing history. When a user makes an insertion or deletion, that operation is immediately applied to the user's local replica, and then asynchronously sent over the network to any other replicas that have a copy of the same document. Users can also edit their local copy while offline; the corresponding operations are then enqueued and sent when the device is next online.

Our algorithm makes no assumptions about the underlying network via which operations are replicated: any reliable broadcast protocol (which detects and retransmits lost messages) is sufficient. For example, a relay server could store and forward messages from one replica to the others, or replicas could use a peer-to-peer gossip protocol. We make no timing assumptions and can tolerate arbitrary network delay, but we assume replicas are non-Byzantine.

A key property that the collaboration algorithm must satisfy is *convergence*: any two replicas that have seen the same set of operations must be in the same document state (i.e., a text consisting of the same sequence of characters), even if the operations arrived in a different order at each replica. If the underlying broadcast protocol ensures that every non-crashed replica eventually receives every operation, the algorithm achieves *strong eventual consistency* [48].

2.2 Event graphs

We represent the editing history of a document as an *event graph*: a directed acyclic graph (DAG) in which every node is an *event* consisting of an operation (insert/delete a character), a unique ID, and a set of IDs of its *parent nodes*. When a is a *parent* of b, we also say b is a *child* of a, and the graph contains an edge from a to b. We construct events such that the graph is transitively reduced (i.e., it contains no redundant edges). When there is a directed path from a to b we say that a *happened before* b, and write $a \rightarrow b$ as per Lamport [36]. The $\rightarrow$ relation is a strict partial order. We say that events a and b are *concurrent*, written $a \parallel b$, if both events are in the graph, $a \neq b$, and neither happened before the other: $a \nrightarrow b \wedge b \nrightarrow a$.

The *frontier* is the set of events with no children. Whenever a user performs an operation, a new event containing that operation is added to the graph, and the previous fron-

$$\begin{tikzcd}
 & e_1 : Insert(0, \text{"}H\text{"}) \arrow[d] & \\
 & e_2 : Insert(1, \text{"}e\text{"}) \arrow[d] & \\
 & e_3 : Insert(2, \text{"}l\text{"}) \arrow[d] & \\
 & e_4 : Insert(3, \text{"}o\text{"}) \arrow[dl] \arrow[dr] & \\
e_5 : Insert(3, \text{"}l\text{"}) & & e_6 : Insert(4, \text{"!"})
\end{tikzcd}$$

Figure 2. The event graph corresponding to Figure 1.

tier in the replica's local copy of the graph becomes the new event's parents. The new event and its parent edges are then replicated over the network, and each replica adds them to its copy of the graph. If any parent events are missing, the replica waits for them to arrive before adding them to the graph; the result is a simple causal broadcast protocol [11,13]. Two replicas can merge their event graphs by taking the union of their sets of events. Events in the graph are immutable; they always represents the operation as originally generated, and not as a result of any transformation.

For example, Figure 2 shows the event graph corresponding to Figure 1. The events e_5 and e_6 are concurrent, and the frontier of this graph is the set of events $\{e_5, e_6\}$.

The event graph for a substantial document, such as a research paper, may contain hundreds of thousands of events. It can nevertheless be stored in a very compact form by exploiting the typical editing patterns of humans writing text: characters tend to be inserted or deleted in consecutive runs. Many portions of a typical event graph are linear, with each event having one parent and one child. We describe the storage format in more detail in Section 3.8.

2.3 Document versions

Let G be an event graph, represented as a set of events. Due to convergence, any two replicas that have the same set of events must be in the same state. Therefore, the document state (sequence of characters) resulting from G must be $\mathsf{replay}(G)$, where replay is some pure (deterministic and non-mutating) function. In principle, any pure function of the set of events results in convergence, although a replay function that is useful for text editing must satisfy additional criteria (see Section 3.1).

Consider the event $Delete(i)$, which deletes the character at position i in the document. In order to correctly interpret this event, we need to determine which character was at index i at the time when the operation was generated.

More generally, let e_i be some event. The document state when e_i was generated must be $\mathsf{replay}(G_i)$, where G_i is the set of events that were known to the generating replica at the time when e_i was generated (not including e_i itself). By definition, the parents of e_i are the frontier of G_i, and thus G_i is the set of all events that happened before e_i, i.e., e_i's parents and all of their ancestors. Therefore, the parents of e_i unambiguously define the document state in which e_i must be interpreted.

To formalise this, given an event graph (set of events) G, we define the *version* of G to be its frontier set:

$$\mathsf{Version}(G) = \{e_1 \in G \mid \nexists e_2 \in G : e_1 \to e_2\}$$

Given some version V, the corresponding set of events can be reconstructed as follows:

$$\mathsf{Events}(V) = V \cup \{e_1 \mid \exists e_2 \in V : e_1 \to e_2\}$$

Since an event graph grows only by adding events that are concurrent to or children of existing events (we never change the parents of an existing event), there is a one-to-one correspondence between an event graph and its version. For all valid event graphs G, $\mathsf{Events}(\mathsf{Version}(G)) = G$.

The set of parents of an event in the graph is the version of the document in which that operation must be interpreted. The version can hence be seen as a *logical clock*, describing the point in time at which a replica knows about the exact set of events in G. Even if the event graph is large, in practice a version rarely consists of more than two events.

2.4 Replaying editing history

Collaborative editing algorithms are usually defined in terms of sending and receiving messages over a network. The abstraction of an event graph allows us to reframe these algorithms in a simpler way: a collaborative text editing algorithm is a pure function $\mathsf{replay}(G)$ of an event graph G. This function can use the parent-child relationships to partially order events, but concurrent events could be processed in any order. This allows us to separate the process of replicating the event graph from the algorithm that ensures convergence. In fact, this is how *pure operation-based CRDTs* [9] are formulated, as discussed in Section 5.

In addition to determining the document state from an entire event graph, we need an *incremental update* function. Say we have an existing event graph G and corresponding document state $doc = \mathsf{replay}(G)$. Then an event e from a remote replica is added to the graph. We could rerun the function to obtain $doc' = \mathsf{replay}(G \cup \{e\})$, but it would be inefficient to process the entire graph again. Instead, we need to efficiently compute the operation to apply to *doc* in order to obtain *doc′*. For text documents, this incremental update is also described as an insertion or deletion at a particular index; however, the index may differ from that in the original event due to the effects of concurrent operations, and a deletion may turn into a no-op if the same character has also been deleted by a concurrent operation.

Both OT and CRDT algorithms focus on this incremental update. If none of the events in G are concurrent with e, OT is straightforward: the incremental update is identical to the operation in e, as no transformation takes place. If there is concurrency, OT must transform each new event with regard to each existing event that is concurrent to it.

In CRDTs, each event is first translated into operations that use unique IDs instead of indexes, and then these operations are applied to a data structure that reflects all of the operations seen so far (both concurrent operations and those that happened before). In order to update the text editor, these updates to the CRDT's internal structure need to be translated back into index-based insertions and deletions. Many CRDT papers elide this translation from unique IDs back to indexes, but it is important for practical applications.

Regardless of whether the OT or the CRDT approach is used, a collaborative editing algorithm can be boiled down to an incremental update to an event graph: given an event to be added to an existing event graph, return the (index-based) operation that must be applied to the current document state so that the resulting document is identical to replaying the entire event graph including the new event.

3 The Event Graph Walker algorithm

Eg-walker is a collaborative text editing algorithm based on the idea of event graph replay. The algorithm builds on a replication layer that ensures that whenever a replica adds an event to the graph, all non-crashed replicas eventually receive it. The state of each replica consists of three parts:

1. **Event graph:** Each replica stores a copy of the event graph on disk, in a format described in Section 3.8.
2. **Document state:** The current sequence of characters in the document with no further metadata. On disk this is simply a plain text file; in memory it may be represented as a rope [12], piece table [44], or similar structure to support efficient insertions and deletions.
3. **Internal state:** A temporary CRDT structure that Eg-walker uses to merge concurrent edits. It is not persisted or replicated, and it is discarded when the algorithm finishes running.

Eg-walker can reconstruct the document state by replaying the entire event graph. It first performs a topological sort, as illustrated in Figure 3. Then each event is transformed so that the transformed insertions and deletions can be applied in topologically sorted order, starting with an empty document, to obtain the document state. In Git parlance, this process "rebases" a DAG of operations into a linear operation history with the same effect. The input of the algorithm is the event graph, and the output is this topologically sorted sequence of transformed operations. While OT transforms one operation with respect to one other, Eg-walker uses the internal state to transform operations efficiently.

In graphs with concurrent operations there are multiple possible sort orders. Eg-walker guarantees that the final document state is the same, regardless which of these orders is chosen. However, the choice of sort order may affect the performance of the algorithm, as discussed in Section 3.7.

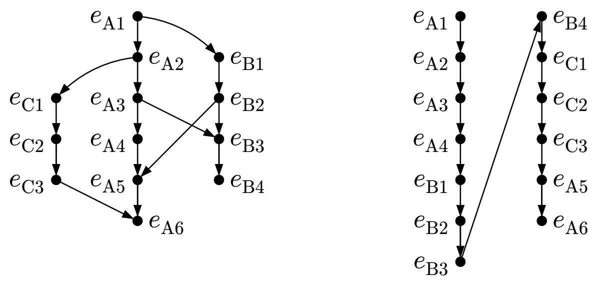


Figure 3. An event graph (left) and one possible topologically sorted order of that graph (right).

For example, the graph in Figure 2 has two possible sort orders; Eg-walker either first inserts "l" at index 3 and then "!" at index 5 (like User 1 in Figure 1), or it first inserts "!" at index 4 followed by "l" at index 3 (like User 2 in Figure 1). The final document state is "Hello!" either way.

Event graph replay easily extends to incremental updates for real-time collaboration: when a new event is added to the graph, it becomes the next element of the topologically sorted sequence. We can transform each new event in the same way as during replay, and apply the transformed operation to the current document state.

3.1 Characteristics of Eg-walker

Eg-walker ensures that the resulting document state is consistent with Attiya et al.'s *strong list specification* [8] (in essence, replicas converge to the same state and apply operations in the right place), and it is *maximally non-interleaving* [55] (i.e., concurrent sequences of insertions at the same position are placed one after another, and not interleaved).

One way of achieving this goal would be to track the state of each branch of the editing history in a separate CRDT object. The CRDT for a given branch could translate events from the event graph into the corresponding CRDT operations. When branches fork, the CRDT object would need to be cloned in memory. When branches merge, CRDT operations from one branch would be applied to the other branch's CRDT state. Essentially, this approach simulates a network of communicating CRDT replicas and their states. This approach produces the correct result, but it performs poorly, as we need to store and update a full copy of the CRDT state for every concurrent branch in the event graph.

Eg-walker improves on this approach in two ways:

1. Eg-walker avoids the need to clone and merge multiple CRDT objects. Instead, the algorithm maintains a single data structure that can transform and merge events from multiple branches.
2. In portions of the event graph that have no concurrency (which, in many editing histories, is the vast majority of events), events do not need to be transformed at all, and we can discard all of the internal state accumulated so far.

Moreover, Eg-walker does not need the event graph and the internal state when generating new events, or when adding an event to the graph that happened after all existing events. Most of the time, we only need the current document state. The event graph can remain on disk without using any space in memory or any CPU time. The event graph is only required when handling concurrency, and even then we only have to replay the portion of the graph since the last ancestor that the concurrent operations had in common.

Eg-walker's approach contrasts with existing CRDTs, which require every replica to persist the internal state (including the unique ID for each character) and send it over the network, and which require that state to be loaded into memory in order to both generate and receive operations, even when there is no concurrency. This uses significant amounts of memory and makes documents slow to load.

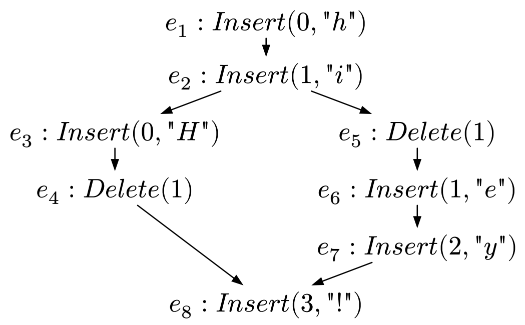
OT algorithms avoid this internal state; similarly to Eg-walker, they only need to persist the latest document state and the history of operations that are concurrent to operations that may arrive in the future. In both Eg-walker and OT, the event graph can be discarded if we know that no event we may receive in the future will be concurrent with any existing event. However, OT algorithms are very slow to merge long-running branches (see Section 4). Some OT algorithms are only able to handle restricted forms of event graphs, whereas Eg-walker handles arbitrary DAGs.

3.2 Walking the event graph

For the sake of clarity we first explain a simplified version of Eg-walker that replays the entire event graph without discarding its internal state along the way. This approach incurs some CRDT overhead even for non-concurrent operations. In Section 3.6 we show how the algorithm can be optimised to replay only a part of the event graph.

First, we topologically sort the event graph in a way that keeps events on the same branch consecutive as much as possible: for example, in Figure 3 we first visit $e_{\mathrm{A1}} \dots e_{\mathrm{A4}}$, then $e_{\mathrm{B1}} \dots e_{\mathrm{B4}}$. We avoid alternating between branches, such as $e_{\mathrm{A1}}, e_{\mathrm{B1}}, e_{\mathrm{A2}}, e_{\mathrm{B2}} \dots$, even though that would also be a valid topological sort. For this we use a standard textbook algorithm [15]: perform a depth-first traversal starting from the oldest event, and build up the topologically sorted list in the order that events are visited. When a node has multiple children in the graph, we choose their order based on a heuristic so that branches with fewer events tend to appear before branches with more events in the sorted order; this can improve performance (see Section 3.7) but is not essential. We estimate the size of a branch by counting the number of events that happened after each event.

The algorithm then processes the events one at a time in topologically sorted order, updating the internal state and outputting a transformed operation for each event. The internal state simultaneously captures the document at two versions: the version in which an event was generated (which we call the *prepare* version), and the version in which all events seen so far have been applied (which we call the *effect* version). If the prepare and effect versions are the same, the transformed operation is identical to the original one. In general, the prepare version represents a subset of the events of the effect version.



$$\begin{tikzcd}
 & e_1 : Insert(0, "h") \arrow[d] & \\
 & e_2 : Insert(1, "i") \arrow[dl] \arrow[dr] & \\
e_3 : Insert(0, "H") \arrow[d] & & e_5 : Delete(1) \arrow[d] \\
e_4 : Delete(1) \arrow[ddr] & & e_6 : Insert(1, "e") \arrow[d] \\
 & & e_7 : Insert(2, "y") \arrow[dl] \\
 & e_8 : Insert(3, "!") &
\end{tikzcd}$$

Figure 4. An event graph. Starting with document "hi", one user changes "hi" to "hey", while concurrently another user capitalises the "H". After merging to the state "Hey", one of them appends an exclamation mark to produce "Hey!".

The internal state can be updated with three methods, each of which takes an event as argument:

- apply(e) updates the prepare version and the effect version to include e, assuming that the current prepare version equals $e.parents$, and that e has not yet been applied. This method interprets e in the context of the prepare version, and outputs the operation representing how the effect version has been updated.
- retreat(e) updates the prepare version to remove e, assuming the prepare version previously included e.
- advance(e) updates the prepare version to add e, assuming that the prepare version previously did not include e, but the effect version did.

The effect version only moves forwards in time (through apply), whereas the prepare version can move both forwards and backwards. Consider the example in Figure 4, and assume that the events $e_1 \dots e_8$ are traversed in order of their subscript. These events can be processed as follows:

1. Start in the empty state, and then call apply(e_1), apply(e_2), apply(e_3), and apply(e_4). This is valid because each event's parent version is the set of all events processed so far.
2. Before we can apply e_5 we must rewind the prepare version to be $\{e_2\}$, which is the parent of e_5. We can do this by calling retreat(e_4) and retreat(e_3).
3. Now we can call apply(e_5), apply(e_6), and apply(e_7).
4. The parents of e_8 are $\{e_4, e_7\}$; before we can apply e_8 we must therefore add e_3 and e_4 to the prepare state again by calling advance(e_3) and advance(e_4).
5. Finally, we can call apply(e_8).

In complex event graphs such as the one in Figure 3 the same event may have to be retreated and advanced several times, but we can process arbitrary DAGs this way. In general, before applying the next event e in topologically sorted order, compute $G_{\text{old}} = \mathsf{Events}(V_p)$ where V_p is the current prepare version, and $G_{\text{new}} = \mathsf{Events}(e.parents)$. We then call retreat on each event in $G_{\text{old}} - G_{\text{new}}$ (in reverse topological sort order), and call advance on each event in $G_{\text{new}} - G_{\text{old}}$ (in topological sort order) before calling apply(e).

3.3 Representing prepare and effect versions

The internal state implements the apply, retreat, and advance methods by maintaining a CRDT data structure. This structure consists of a linear sequence of records, one per character in the document, including tombstones for deleted characters. Runs of characters with consecutive IDs and the same properties can be run-length encoded to save memory. A record is inserted into this sequence by apply(e_i) for an insertion event e_i. Subsequent deletion events and retreat/advance calls may modify properties of the record, but records in the sequence are not removed or reordered once they have been inserted.

When the event graph contains concurrent insertions, we use a CRDT to ensure that all replicas place the records in this sequence in the same order, regardless of the order in which the event graph is traversed. For example, RGA [47] or YATA [41] could be used for this purpose. Our implementation of Eg-walker uses a variant of the Yjs algorithm [28], itself based on YATA, that we conjecture to be maximally non-interleaving. We leave a detailed analysis of this algorithm to future work, since it is not core to this paper.

Each record in this sequence contains:

- the ID of the event that inserted the character;
- $s_p \in \{\texttt{NotInsertedYet}, \texttt{Ins}, \texttt{Del 1}, \texttt{Del 2}, ...\}$, the character's state in the prepare version;
- $s_e \in \{\texttt{Ins}, \texttt{Del}\}$, the state in the effect version;
- and any other fields required by the CRDT to determine the order of concurrent insertions.

The rules for updating s_p and s_e are:

- When a record is first inserted by apply(e_i) with an insertion event e_i, it is initialised with $s_p = s_e = \texttt{Ins}$.
- If apply(e_d) is called with a deletion event e_d, we set $s_e = \texttt{Del}$ in the record representing the deleted character. In the same record, if $s_p = \texttt{Ins}$ we update it to $\texttt{Del 1}$, and if $s_p = \texttt{Del}\ n$ it advances to $\texttt{Del}\ (n+1)$, as shown in Figure 5.
- If retreat(e_i) is called with insertion event e_i, we must have $s_p = \texttt{Ins}$ in the record affected by the event, and we update it to $s_p = \texttt{NotInsertedYet}$. Conversely, advance($e_i$) moves s_p from $\texttt{NotInsertedYet}$ to $\texttt{Ins}$.
- If retreat(e_d) is called with a deletion event e_d, we must have $s_p = \texttt{Del}\ n$ in the affected record, and we update it to $\texttt{Del}\ (n-1)$ if $n > 1$, or to $\texttt{Ins}$ if $n = 1$. Calling advance(e_d) performs the opposite.

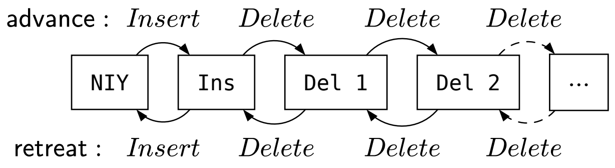


Figure 5. State machine for internal state variable s_p.

"H"	"h"	"i"
id : 3	id : 1	id : 2
s_p : Ins	s_p : Del 1	s_p : Ins
s_e : Ins	s_e : Del	s_e : Ins

retreat(e_4)
retreat(e_3)
$\Longrightarrow$

"H"	"h"	"i"
id : 3	id : 1	id : 2
s_p : NIY	s_p : Ins	s_p : Ins
s_e : Ins	s_e : Del	s_e : Ins

Figure 6. Left: the internal state after applying $e_1...e_4$ from Figure 4. Right: after retreat(e_4) and retreat(e_3), the prepare state is updated to mark "H" as NotInsertedYet, and the deletion of "h" is undone. The effect state is unchanged.

"H"	"h"	"e"	"y"	"!"	"i"
id : 3	id : 1	id : 6	id : 7	id : 8	id : 2
s_p : Ins	s_p : Del 1	s_p : Ins	s_p : Ins	s_p : Ins	s_p : Del 1
s_e : Ins	s_e : Del	s_e : Ins	s_e : Ins	s_e : Ins	s_e : Del

Figure 7. The internal Eg-walker state after replaying all of the events in Figure 4.

As a result, s_p and s_e are Ins if the character is visible (inserted but not deleted) in the prepare and effect version respectively; $s_p = \texttt{Del}\ n$ indicates that the character has been deleted by n concurrent delete events in the prepare version; and $s_p = \texttt{NotInsertedYet}$ indicates that the insertion of the character has been retreated in the prepare version. s_e does not count the number of deletions and does not have a NotInsertedYet state since we never remove the effect of an operation from the effect version.

For example, Figure 6 shows the state after applying $e_1...e_4$ from Figure 4, and how that state is updated by retreating e_4 and e_3 before e_5 is applied. In the effect state, the lowercase "h" is marked as deleted, while the uppercase "H" and the "i" are visible. In the prepare state, by retreating e_4 and e_3 the "H" is marked as NotInsertedYet, and the deletion of "h" is undone ($s_p = \texttt{Ins}$).

Figure 7 shows the state after replaying all of the events in Figure 4: "i" is also deleted, the characters "e" and "y" are inserted immediately after the "h", e_3 and e_4 are advanced again, and finally "!" is inserted after the "y". The figures include the character for the sake of readability, but Eg-walker actually does not store text content in its internal state.

3.4 Mapping indexes to character IDs

In the event graph, insertion and deletion operations specify the index at which they apply. In order to update Eg-walker's internal state, we need to map these indexes to the correct record in the sequence, based on the prepare state s_p. To produce the transformed operations, we need to map the positions of these internal records back to indexes again – this time based on the effect state s_e.

A simple but inefficient algorithm would be: to apply a $Delete(i)$ operation we iterate over the sequence of records and pick the ith record with a prepare state of $s_p = \texttt{Ins}$ (i.e., the ith among the characters that are visible in the prepare state, which is the document state in which the operation should be interpreted). Similarly, to apply $Insert(i, c)$ we skip over $i - 1$ records with $s_p = \texttt{Ins}$ and insert the new record after the last skipped record (if there have been concurrent insertions at the same position, we may also need to skip over some records with $s_p = \texttt{NotInsertedYet}$, as determined by the list CRDT's insertion ordering).

To reduce the cost of this algorithm from $O(n)$ to $O(\log n)$, where n is the number of characters in the document, we construct a B-tree whose leaves, from left to right, contain the sequence of records representing characters. We extend the tree into an *order statistic tree* [15] (also known as *ranked B-tree*) by adding two integers to each node: the number of records with $s_p = \texttt{Ins}$ contained within that subtree, and the number of records with $s_e = \texttt{Ins}$ in that subtree. Every time s_p or s_e are updated, we also update those numbers on the path from the updated record to the root. As the tree is balanced, this update takes $O(\log n)$.

Now we can find the ith record with $s_p = \texttt{Ins}$ in logarithmic time by starting at the root of the tree, and adding up the values in the subtrees that have been skipped. Moreover, once we have a record in the sequence we can efficiently determine its index in the effect state by going in the opposite direction: working upwards in the tree towards the root, and summing the numbers of records with $s_e = \texttt{Ins}$ that lie in subtrees to the left of the starting record. This allows us to efficiently transform the index of an operation from the prepare version into the effect version. If the character was already deleted in the effect version ($s_e = \texttt{Del}$), the transformed operation is a no-op.

The above process makes $\mathsf{apply}(e_i)$ efficient. We also need to efficiently perform $\mathsf{retreat}(e_i)$ and $\mathsf{advance}(e_i)$, which modify the prepare state s_p of the record inserted or deleted by e_i. While advancing/retreating we cannot look up a target record by its index. Instead, we maintain a second B-tree, mapping from each event's ID to the target record. The mapping stores a value depending on the type of the event:

- For delete events, we store the ID of the character deleted by the event.
- For insert events, we store a pointer to the leaf node in the first B-tree that contains the corresponding record. When nodes in the first B-tree are split, we update the pointers in the second B-tree accordingly.

On every $\mathsf{apply}(e)$, after updating the sequence as above, we update this mapping. When we subsequently call $\mathsf{retreat}(e)$ or $\mathsf{advance}(e)$, that event e must have already been applied, and hence $e.id$ must appear in this mapping. This map allows us to advance or retreat in logarithmic time.

3.5 Clearing the internal state

As described so far, the algorithm retains every insertion since document creation forever in its internal state, consuming a lot of memory, and requiring the entire event graph to be replayed in order to restore the internal state. We now introduce a further optimisation that allows Eg-walker to completely discard its internal state from time to time, and replay only a subset of the event graph.

We define a version $V \subseteq G$ to be a *critical version* in an event graph G iff it partitions the graph into two subsets of events $G_1 = \mathsf{Events}(V)$ and $G_2 = G - G_1$ such that all events in G_1 happened before all events in G_2:

$$\forall e_1 \in G_1 : \forall e_2 \in G_2 : e_1 \rightarrow e_2.$$

Equivalently, V is a critical version iff every event in the graph is either in V, or an ancestor of some event in V, or happened after *all* of the events in V:

$$\forall e_1 \in G : e_1 \in \mathsf{Events}(V) \lor (\forall e_2 \in V : e_2 \rightarrow e_1).$$

A critical version might not remain critical forever; it is possible for a critical version to become non-critical because a concurrent event is added to the graph.

A key insight in the design of Eg-walker is that critical versions partition the event graph into sections that can be processed independently. Events that happened at or before a critical version do not affect how any event after the critical version is transformed. This observation enables two important optimisations:

- Any time the version of the event graph processed so far is critical, we can discard the internal state (including both B-trees and all s_p and s_e values), and replace it with a placeholder as explained in Section 3.6.
- If both an event's version and its parent version are critical versions, there is no need to traverse the B-trees and update the CRDT state, since we would immediately discard that state anyway. In this case, the transformed event is identical to the original event, so the event can simply be emitted as-is.

These optimisations make it very fast to process documents that are mostly edited sequentially (e.g., because the authors took turns and did not write concurrently, or because there is only a single author), since most of the event graph of such a document is a linear chain of critical versions.

The internal state can be discarded once replay is complete, although it is also possible to retain the internal state for transforming future events. If a replica receives events that are concurrent with existing events in its graph, but the replica has already discarded its internal state resulting from those events, it needs to rebuild some of that state. It can do this by identifying the most recent critical version that happened before the new events, replaying the existing events that happened after that critical version, and finally applying the new events. Events from before that critical version are not replayed. Since most editing histories have critical versions from time to time, this means that usually only a small subset of the event graph is replayed. In the worst case, this algorithm replays the entire event graph.

3.6 Partial event graph replay

Assume that we want to add event e_{new} to the event graph G, that $V_{\text{curr}} = \mathsf{Version}(G)$ is the current document version reflecting all events except e_{new}, and that $V_{\text{crit}} \neq V_{\text{curr}}$ is the latest critical version in $G \cup \{e_{\text{new}}\}$ that happened before both e_{new} and V_{curr}. Further assume that we have discarded the internal state, so the only information we have is the latest document state at V_{curr} and the event graph; in particular, without replaying the entire event graph we do not know the document state at V_{crit}.

Luckily, the exact internal state at V_{crit} is not needed. All we need is enough state to transform e_{new} and rebase it onto the document at V_{curr}. This internal state can be obtained by replaying the events since V_{crit}, that is, $G - \mathsf{Events}(V_{\text{crit}})$, in topologically sorted order:

1. We initialise a new internal state corresponding to version V_{crit}. Since we do not know the the document state at this version, we start with a single placeholder record representing the unknown document content.
2. We update the internal state by replaying events from V_{crit} to V_{curr}, but we do not output transformed operations during this stage.
3. Finally, we apply the new event e_{new} and output the transformed operation. If we received a batch of new events, we apply them in topologically sorted order.

The placeholder record we start with in step 1 represents the range of indexes $[0, \infty]$ of the document state at V_{crit} (we do not know the length of the document at that version, but we can still have a placeholder for arbitrarily many indexes). Placeholders are counted as the number of characters they represent in the order statistic tree construction, and they have the same length in both the prepare and the effect versions. We then apply events as follows:

- Applying an insertion at index i creates a record with $s_p = s_e = \texttt{Ins}$ and the ID of the insertion event. We map the index to a record in the sequence using the prepare state as usual; if i falls within a placeholder for range $[j, k]$, we split it into a placeholder for $[j, i-1]$, followed by the new record, followed by a placeholder for $[i, k]$. Placeholders for empty ranges are omitted.
- Applying a deletion at index i: if the deleted character was inserted prior to V_{crit}, the index must fall within a placeholder with some range $[j, k]$. We split it into a placeholder for $[j, i-1]$, followed by a new record with $s_p = \texttt{Del 1}$ and $s_e = \texttt{Del}$, followed by a placeholder for $[i+1, k]$. The new record has a placeholder ID that only needs to be unique within the local replica, and need not be consistent across replicas.
- Applying a deletion of a character inserted since V_{crit} updates the record created by the insertion.

Before applying an event we retreat and advance as usual. The algorithm never needs to retreat or advance an event that happened before V_{crit}, therefore every retreated or advanced event ID must exist in second B-tree.

If there are concurrent insertions at the same position, we invoke the CRDT algorithm to place them in a consistent order as discussed in Section 3.3. Since all concurrent events must be after V_{crit}, they are included in the replay. When we are seeking for the insertion position, we never need to seek past a placeholder, since the placeholder represents characters that were inserted before V_{crit}.

3.7 Algorithm complexity

Say we have two users who have been working offline, generating k and m events respectively. When they come online and merge their event graphs, the latest critical version is immediately prior to the branching point. If the branch of k events comes first in the topological sort, the replay algorithm first applies k events, then retreats k events, applies m events, and finally advances k events again. Asymptotically, $O(k+m)$ calls to apply/retreat/advance are required regardless of the order of traversal, although in practice the algorithm is faster if $k < m$ since we don't need to retreat/advance on the branch that is visited last.

Each apply/retreat/advance requires one or two traversals of first B-tree, and at most one traversal of the second B-tree. The upper bound on the number of entries in each tree (including placeholders) is $2(k+m)+1$, since each event generates at most one new record and one placeholder split. Since the trees are balanced, the cost of each traversal is $O(\log(k+m))$. Overall, the cost of merging branches with k and m events is therefore $O((k+m)\log(k+m))$.

We can also give an upper bound on the complexity of replaying an event graph with n events. Each event is applied exactly once, and before each event we retreat or advance each prior event at most once, at $O(\log n)$ cost. The worst-case complexity of the algorithm is therefore $O(n^2 \log n)$, but this case is unlikely to occur in practice.

3.8 Storing the event graph

To store the event graph compactly on disk, we developed a compression technique that takes advantage of how people typically write text documents: namely, they tend to insert or delete consecutive sequences of characters, and less frequently hit backspace or move the cursor to a new location. Eg-walker's event graph storage format is inspired by the Automerge CRDT library [25,33], which in turn uses ideas from column-oriented databases [6,50]. We also borrow some bit-packing tricks from the Yjs CRDT library [28].

We first topologically sort the events in the graph. Different replicas may sort the graph differently, but locally to one replica we can identify an event by its index in this sorted order. Then we store different properties of events in separate byte sequences called *columns*, which are then combined into one file with a simple header. Each column stores some different fields of the event data. The columns are:

- *Event type, start position, and run length.* For example, "the first 23 events are insertions at consecutive indexes starting from index 0, the next 10 events are deletions at consecutive indexes starting from index 7," and so on. We encode this using a variable-length binary encoding of integers, which represents small numbers in one byte, larger numbers in two bytes, etc.
- *Inserted content.* An insertion event contains exactly one character (a Unicode scalar value), and a deletion does not. We concatenate the UTF-8 encoding of the characters for insertion events in the same order as they appear in the first column, and LZ4-compress.
- *Parents.* By default we assume that every event has exactly one parent, namely its predecessor in the topological sort. Any events for which this is not true are listed explicitly, for example: "the first event has zero parents; the 153rd event has two parents, namely events numbers 31 and 152;" and so on.
- *Event IDs.* Each event is uniquely identified by a pair of a replica ID and a per-replica sequence number. This column stores runs of event IDs, for example: "the first 1085 events are from replica A, starting with sequence number 0; the next 595 events are from replica B, starting with sequence number 0;" and so on.

Replicas can optionally also store a copy of the final document state reflecting all events. This allows documents to be loaded from disk without replaying the event graph.

We send the same data format over the network when replicating the entire event graph. When sending a subset of events over the network (e.g., a single event during real-time collaboration), references to parent events outside of that subset need to be encoded using event IDs of the form $(replicaID, seqNo)$, but otherwise the encoding is similar.

4 Evaluation

We created a TypeScript implementation of Eg-walker optimised for simplicity and readability [19], and a production-ready Rust implementation optimised for performance [20]. The TypeScript version omits the run-length encoding of internal state, B-trees, and topological sorting heuristics.

To evaluate the correctness of Eg-walker we proved that the algorithm complies with Attiya et al.'s *strong list specification* [8] (see Appendix B). We also performed randomised property testing on the implementations, including checking that our implementations converge to the same result.

To evaluate its performance, we compare the Rust implementation of Eg-walker with two popular CRDT libraries: Automerge v0.5.9 [1] (Rust) and Yjs v13.6.10 [28] (JavaScript).[1] We only test their collaborative text datatypes, and not the other features they support. However, the performance of these libraries varies widely. In an effort to distinguish between implementation differences and algorithmic differences, we have also implemented our own performance-optimised reference CRDT library. This library shares most of its code with our Rust Eg-walker implementation, enabling a more like-to-like comparison between the traditional CRDT approach and Eg-walker. Our reference CRDT outperforms both Yjs and Automerge.

We have also implemented a simple OT library using the TTF algorithm [42]. (We do not use the server-based Jupiter algorithm [40] or the popular OT library ShareDB [21] because they do not support the branching and merging patterns that occur in some of our dataset.)

We compare these implementations along 3 dimensions:[2]

Speed The CPU time to load a document into memory, and to merge a set of updates from a remote replica.

Memory usage The RAM used while a document is loaded and while merging remote updates.

Storage size The number of bytes needed to persistently store a document or replicate it over the network.

4.1 Editing traces

As there is no established benchmark for collaborative text editing, we collected a set of editing traces from real documents. We have made these traces freely available on GitHub [22]. For this evaluation we use seven traces, which fall into three categories:

Sequential Traces (S1, S2, S3): One author, or multiple authors taking turns (no concurrency).

[1]We also tested Yrs [53], the Rust rewrite of Yjs by the original authors. It performed worse than Yjs, so we omitted it from our results.

[2]Experimental setup: We ran the benchmarks on a Ryzen 7950x CPU running Linux 6.5.0-28 and 64GB of RAM. We compiled Rust code with rustc v1.78.0 in release mode with `'-C target-cpu=native'`. Rust code was pinned to a single CPU core to reduce variance across runs. For JavaScript (Yjs) we used Node.js v22.2.0. All reported time measurements are the mean of at least 100 test iterations (except for the case where OT takes an hour to merge trace A2, which we ran 10 times). The standard deviation for all benchmark results was less than 1.2% of the mean, except for the Yjs measurements, which had a stddev of less than 6%. Error bars on our graphs are too small to be visible.

Concurrent Traces (C1, C2): Multiple users concurrently editing the same document with ≈1 second latency. Many short-lived branches with frequent merges.

Asynchronous Traces (A1, A2): Event graphs derived from branching/merging Git commit histories. Multiple long-running branches and infrequent merges.

We recorded the sequential and concurrent traces with keystroke granularity using an instrumented text editor. To make the traces easier to compare, we normalised them so that each trace contains ≈500k inserted characters (about 100 printed pages). We extended shorter traces to this length by repeating them several times. See Appendix A for details.

4.2 Time taken to load and merge changes

The slowest operations in many collaborative editors are:

- merging a large set of edits from a remote replica into the local state (e.g. reconnecting after working offline);
- loading a document from disk into memory so that it can be displayed and edited.

To simulate a worst-case merge, we start with an empty document and then merge an entire editing trace into it. In the case of Eg-walker this means replaying the full trace. Figure 8 shows the merge time for each implementation.

After completing this merge, we saved the resulting local replica state to disk and measured the CPU time to load it back into memory. In the CRDT implementations we tested, loading a document from disk is equivalent to merging the remote events, so we do not show CRDT loading times separately in Figure 8. In these algorithms, the CRDT metadata needs to be in memory for the user to be able to edit the document, or to apply any updates received from other replicas (even when there is no concurrency). In contrast, OT and Eg-walker can load documents orders of magnitude faster than CRDTs by caching the final document state on disk, and loading just this data (essentially a plain text file). Eg-walker and OT only need to load the event graph when merging concurrent changes or to reconstruct old document versions. Document edits by the local user or applying non-concurrent remote events do not need the event graph.

We can see in Figure 8 that Eg-walker and OT are very fast to merge the sequential traces (S1, S2, S3), since they simply apply the operations with no transformation. However, OT performance degrades dramatically on the asynchronous traces (6 seconds for A1, and 1 hour for A2) due to the quadratic complexity of the algorithm, whereas Eg-walker remains fast (160,000× faster in the case of A2).

On the concurrent traces (C1, C2) and asynchronous trace A2, the merge time of Eg-walker is similar to that of our reference CRDT, since they perform similar work. Both are significantly faster than the state-of-the-art Yjs and Automerge CRDT libraries; this is due to implementation differences and not fundamental algorithmic reasons.

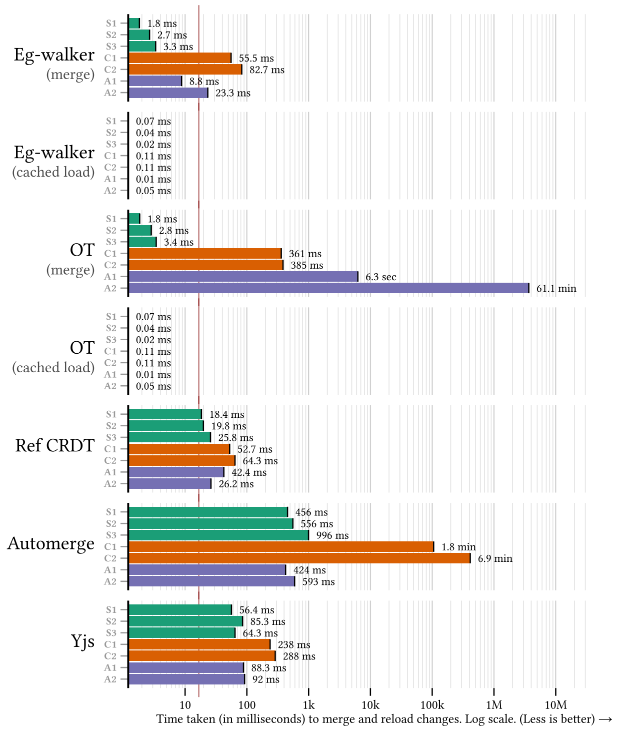


Figure 8. The CPU time taken by each algorithm to merge all events in each trace (as received from a remote replica), or to reload the resulting document from disk. The CRDT implementations (Ref CRDT, Automerge and Yjs) take the same amount of time to merge changes as they do to subsequently load the document. The red line at 16 ms indicates the time budget available to an application that wants to show the results of an operation by the next frame, assuming a display with a 60 Hz refresh rate.

On the sequential traces Eg-walker outperforms our reference CRDT by a factor of 7–10×, and on trace A1 (which contains large sequential sections) Eg-walker is 5× faster. Comparing to Yjs or Automerge, this speedup is greater still. This is due to Eg-walker's ability to clear its internal state and skip all of the internal state manipulation on critical versions (Section 3.5). To quantify this effect, we compare Eg-walker's performance with a version of the algorithm that has these optimisations disabled. Figure 9 shows the time taken to replay all our traces with this optimisation enabled and disabled. We see that the optimisation is effective for S1, S2, S3, and A1, whereas for C1, C2, and A2 it makes little difference (A2 contains no critical versions).

Automerge's merge times on traces C1 and C2 are outliers. This appears to be a bug, which we have reported.

When merging an event graph with very high concurrency (like A2), the performance of Eg-walker is highly dependent on the order in which events are traversed. A poorly chosen traversal order can make this trace as much

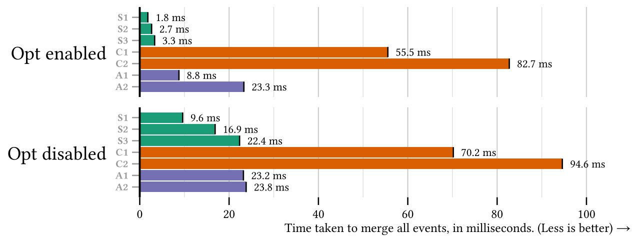


Figure 9. Time taken for Eg-walker to merge all events in a trace, with and without the optimisations from Section 3.5.

as 8× slower to merge. Our topological sort algorithm (Section 3.2) tries to avoid such pathological cases.

4.3 RAM usage

Figure 10 shows the memory footprint (retained heap size) of each algorithm. The memory used by Eg-walker and OT is split into peak usage (during the merge process) and the "steady state" memory usage, after temporary data such as Eg-walker's internal state is discarded and the event graph is written out to disk. For the CRDTs the figure shows steady state memory usage; peak usage is up to 25% higher.

Eg-walker's peak memory use is similar to our reference CRDT's steady state: slightly lower on the sequential traces, and approximately double for the concurrent traces. However, the steady-state memory use of Eg-walker is 1–2 orders of magnitude lower than the best CRDT. This is a significant result, since the steady state is what matters during normal operation while a document is being edited. Note that peak memory usage would be lower when replaying a subset of an event graph, which is the common case.

Yjs has 2–3× greater memory use than our reference CRDT on most traces, and Automerge an order of magnitude greater. Automerge's very high memory use on C1 and C2 is probably a bug. The computer we used for benchmarking had enough RAM to prevent swapping in all cases.

OT has the same memory use as Eg-walker in the steady state, but significantly higher peak memory use on the C1, C2, and A2 traces (6.8 GiB for A2). The reason is that our OT implementation memoizes intermediate transformed operations to improve performance. This memory use could be reduced at the cost of increased merge times.

4.4 Storage size

Our binary encoding of event graphs (Section 3.8) results in smaller files than the equivalent internal CRDT state persisted by Automerge or Yjs. To ensure a like-for-like comparison we have disabled Eg-walker's built-in LZ4 and Automerge's built-in gzip compression. Enabling this compression further reduces the file sizes.

Automerge stores the full editing history of a document, and Figure 11 shows the resulting file sizes relative to the raw concatenated text content of all insertions, with and without a cached copy of the final document state (to en-

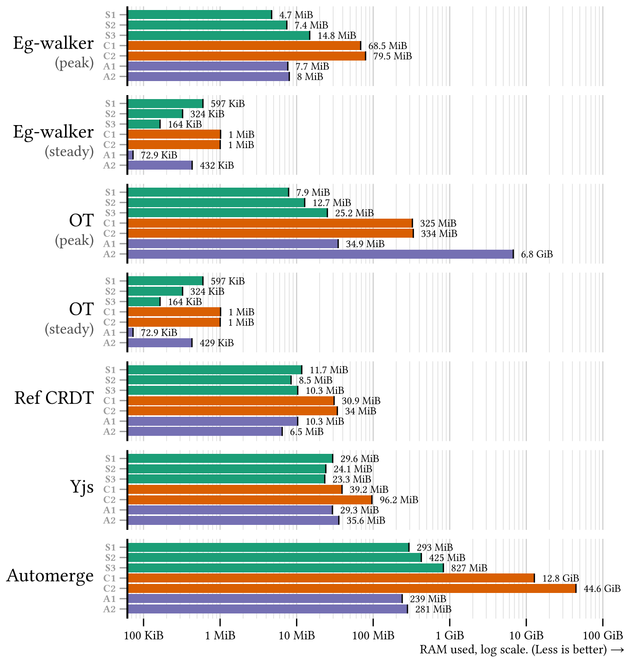


Figure 10. RAM used while merging an editing trace received from another replica. Eg-walker and OT only retain the current document text in the steady state, but need additional RAM at peak while merging concurrent changes.

able fast loads). Even with this additional document text, Eg-walker's files are smaller on all traces except S1.

In contrast, Yjs only stores the text of the final, merged document. This results in a smaller file size, at the cost of making it impossible to reconstruct earlier document states. Figure 12 compares Yjs to the equivalent event graph encoding in which we only store the final document text and operation metadata. Our encoding is smaller than Yjs on all traces. The overhead of storing the event graph is between 20% and 3× the final plain text file size.

5 Related Work

Eg-walker is an example of a *pure operation-based CRDT* [9], which is a family of algorithms that capture a DAG (or partially ordered log) of operations in the form they were generated, and define the current state as a query over that log. However, existing publications on pure operation-based CRDTs [7,10] present only datatypes such as maps, sets, and registers; Eg-walker adds a list/text datatype to this family.

MRDTs [49] are similarly based on a DAG, and use a three-way merge function to combine two branches since their lowest common ancestor; if the LCA is not unique, a recursive merge is used. MRDTs for various datatypes have been defined, but so far none offers text with arbitrary insertion and deletion.

Toomim's *time machines* approach [54] shares a conceptual foundation with Eg-walker: both are based on travers-

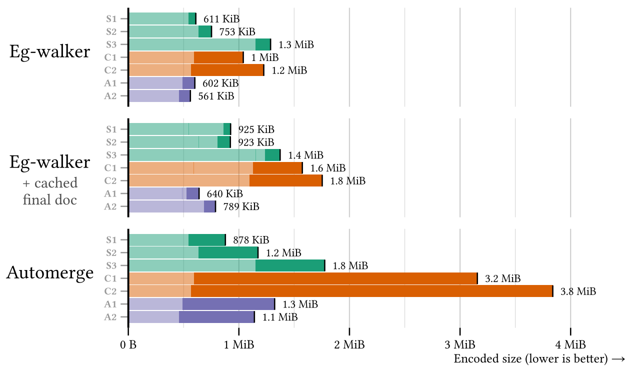


Figure 12. File size storing edit traces using Eg-walker's event graph encoding (with and without final document caching) compared to Automerge. The lightly shaded region in each bar shows the concatenated length of all stored text. This acts as lower bound on the file size.

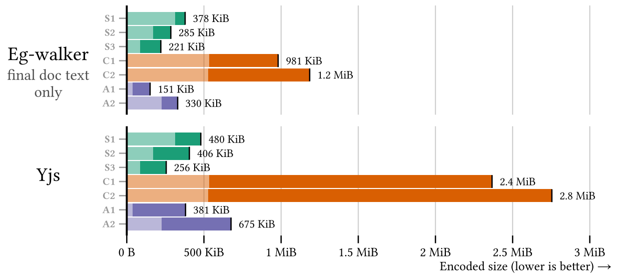


Figure 12. File size storing edit traces in which deleted text content has been omitted, as is the case with Yjs. The lightly shaded region in each bar is the size of the final document, which is a lower bound on the file size.

ing an event graph, with operations being transformed from their original form into a form that can be applied in topologically sorted order. Toomim also points out that CRDTs can implement this transformation. Eg-walker is a concrete, optimised implementation of the time machine approach; novel contributions of Eg-walker include updating the prepare version by retreating and advancing, as well as the details of internal state clearing and partial event graph replay.

Eg-walker is also an *operational transformation* (OT) algorithm [17]. OT has a long lineage of research going back to the 1990s [40,46,51]. To our knowledge, all existing OT algorithms consist of a set of *transformation functions* that transform one operation with regard to one other operation, and a *control algorithm* that traverses an editing history and invokes the necessary transformations. A problem with this architecture is that when two replicas have diverged and each performed n operations, merging their states unavoidably has a cost of at least $O(n^2)$; in some OT algorithms the cost is cubic or even worse [37,47,52]. Eg-walker departs from the transformation function/control algorithm architecture and instead performs transformations using an internal CRDT state, which reduces the merging cost to $O(n \log n)$ in most cases; the upper bound of $O(n^2 \log n)$ is unlikely to occur in practical editing histories.

Other collaborative text editing algorithms [45,47,55,56] belong to the family of *conflict-free replicated data types* (CRDTs) [48]. To our knowledge, all existing CRDTs for text work by assigning each character a unique ID, and translating index-based insertions and deletions into ID-based ones. These unique IDs need to be held in memory when a document is being edited, persisted for the lifetime of the document, and sent to all replicas. In contrast, Eg-walker uses unique IDs only transiently during replay but does not persist or replicate them, and it can free all of its internal state whenever a critical version is reached. Eg-walker needs to store the event graph as long as concurrent operations may arrive, but this takes less space than CRDT state, and it only needs to be in-memory while merging concurrent operations. Most of the time the event graph can remain on disk.

Gu et al.'s *mark & retrace* method [26] builds a CRDT-like structure containing the entire editing history, not only the parts being merged. Differential synchronization [18] relies on heuristics such as similarity-matching of text to perform merges, which is not guaranteed to converge.

Version control systems such as Git [14], Pijul [39], and Darcs [2] also track the editing history of text files. However, they do not support real-time collaboration, and they are line-based (good for code), whereas Eg-walker is character-based (which is better for prose). Git uses a three-way merge, which is not reliable on files containing substantial repeated text [29]. Merges in Darcs have worst-case exponential complexity [35], and Pijul merges using a CRDT that assigns a unique ID to every line [3].

6 Conclusion

Eg-walker is a new approach to collaborative text editing that has characteristics of both CRDTs and OT. It is orders of magnitude faster than existing algorithms in the best cases, and competitive with the fastest existing implementations in the worst cases. Compared to existing CRDTs, it uses orders of magnitude less memory in the steady state, files are vastly faster to load for editing, and in documents with largely sequential editing edits from other users are merged much faster. Compared to OT, large merges (e.g., when two users each did a significant amount of work while offline) are much faster, and Eg-walker supports arbitrary branching/merging patterns (e.g., in peer-to-peer collaboration).

Since Eg-walker stores a fine-grained editing history of a document, it allows applications to show that history to the user, and to restore arbitrary past versions of a document by replaying subsets of the graph. The underlying event graph is not specific to the Eg-walker algorithm, so we expect that the same data format will be able to support future collaborative editing algorithms as well. The core idea of Eg-walker is not specific to plain text; we believe it can be extended to other file types such as rich text, graphics, or spreadsheets.

Until now, many applications have been implemented using centralised server-based OT to avoid the overheads of CRDTs. Eg-walker is the first CRDT to surpass OT's performance, and it requires no server. This breakthrough makes it possible for decentralised, local-first software [32] to become competitive with traditional cloud software.

Acknowledgements

This work was made possible by the generous support from Michael Toomim, the Braid community and the Invisible College. None of this would have been possible without financial support and the endless conversations we have shared about collaborative editing. Thank you to Matthew Weidner and Joe Hellerstein for feedback on a draft of this paper.

References

[1] Automerge CRDT. Retrieved from https://automerge.org/

[2] Darcs. Retrieved from https://darcs.net/

[3] The Pijul manual: Theory. Retrieved from https://pijul.org/manual/theory.html

[4] Node.js source code: src/node.cc. Retrieved from https://github.com/nodejs/node/blob/main/src/node.cc

[5] Makefile for Git. Retrieved from https://github.com/git/git/blob/master/Makefile

[6] Daniel J Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024

[7] Paulo Sérgio Almeida. 2023. Approaches to Conflict-free Replicated Data Types. (October 2023). Retrieved from https://arxiv.org/abs/2310.18220

[8] 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. 259–268. https://doi.org/10.1145/2933057.2933090

[9] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. 2017. Pure Operation-Based Replicated Data Types. Retrieved from https://arxiv.org/abs/1710.04469

[10] Jim Bauwens and Elisa Gonzalez Boix. 2023. Nested Pure Operation-Based CRDTs. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, July 2023. Schloss Dagstuhl, 1–26. https://doi.org/10.4230/LIPIcs.ECOOP.2023.2

[11] Kenneth Birman, André Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems* 9, 3 (August 1991), 272–314. https://doi.org/10.1145/128738.128742

[12] 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

[13] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (second ed.). Springer.

[14] James Coglan. 2019. *Building Git*. Retrieved from https://shop.jcoglan.com/building-git/

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). MIT Press.

[16] John Day-Richter. 2010. What's different about the new Google Docs: Making collaboration fast. Retrieved from https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html

[17] C A Ellis and S J Gibbs. 1989. Concurrency control in groupware systems. In *ACM International Conference on Management of Data (SIGMOD)*, 1989. 399–407. https://doi.org/10.1145/67544.66963

[18] Neil Fraser. 2009. Differential synchronization. In *9th ACM Symposium on Document Engineering (DocEng)*, 2009. ACM, 13–20. https://doi.org/10.1145/1600193.1600198

[19] Joseph Gentle. Reference Eg-walker implementation in Typescript. Retrieved from https://github.com/josephg/reference-reg

[20] Joseph Gentle. Diamond Types: A fully featured realtime editing library. Retrieved from https://github.com/josephg/diamond-types

[21] Joseph Gentle. ShareDB. Retrieved from https://github.com/share/sharedb

[22] Joseph Gentle. Editing Traces (github repository). Retrieved from https://github.com/josephg/editing-traces

[23] Joseph Gentle. 2021. 5000x faster CRDTs: An Adventure in Optimization. Retrieved from https://josephg.com/blog/crdts-go-brrr/

[24] Victor B F Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, OOPSLA (October 2017). https://doi.org/10.1145/3133933

[25] Alex Good and Andrew Jeffery. Automerge Binary Document Format. Retrieved from https://automerge.org/automerge-binary-format-spec/

[26] Ning Gu, Jiangming Yang, and Qiwei Zhang. 2005. Consistency maintenance based on the mark & retrace technique in groupware systems. In *ACM International Conference on Supporting Group Work (GROUP)*, November 2005. ACM, 264–273. https://doi.org/10.1145/1099203.1099250

[27] Joseph M Hellerstein. 2010. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *ACM SIGMOD Record* 39, 1 (September 2010), 5–19. https://doi.org/10.1145/1860702.1860704

[28] Kevin Jahns. Yjs Shared Editing. Retrieved from https://yjs.dev/

[29] Sanjeev Khanna, Keshav Kunal, and Benjamin C Pierce. 2007. A Formal Investigation of Diff3. In *27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, December 2007. Springer, 485–496. https://doi.org/10.1007/978-3-540-77050-3_40

[30] Martin Kleppmann and Alastair R Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (April 2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

[31] Martin Kleppmann, Annette Bieniusa, and Marc Shapiro. CRDT Papers. Retrieved from https://crdt.tech/papers.html

[32] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You own your data, in spite of the cloud. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*, October 2019. ACM, 154–178. https://doi.org/10.1145/3359591.3359737

[33] Martin Kleppmann. 2019. Experiment: columnar data encoding for Automerge. Retrieved from https://github.com/automerge/automerge-perf/blob/master/columnar/README.md

[34] Martin Kleppmann. 2020. Benchmarking resources for Automerge. Retrieved from https://github.com/automerge/automerge-perf

[35] Eric Kow. Understanding Darcs. Retrieved from https://en.wikibooks.org/wiki/Understanding_Darcs/Print_Version

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

[37] Du Li and Rui Li. 2006. A performance study of group editing algorithms. In *12th International Conference on Parallel and Distributed Systems (ICPADS)*, July 2006. https://doi.org/10.1109/icpads.2006.18

[38] Karissa Rae McKelvey, Scott Jenson, Eileen Wagner, Blaine Cook, and Martin Kleppmann. 2023. *Upwelling: Combining real-time collaboration with version control for writers.* Retrieved from https://www.inkandswitch.com/upwelling/

[39] Pierre-Étienne Meunier and Florent Becker. Pijul. Retrieved from https://pijul.org/

[40] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *8th Annual ACM Symposium on User Interface and Software Technology (UIST)*, 1995. 111–120. https://doi.org/10.1145/215585.215706

[41] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *19th International Conference on Supporting Group Work (GROUP 2016)*, November 2016. ACM, 39–49. https://doi.org/10.1145/2957276.2957310

[42] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. 2006. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *9th IEEE International Conference on Collaborative Computing (CollaborateCom)*, November 2006. https://doi.org/10.1109/colcom.2006.361867

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

[44] Peng Lyu. 2018. Text Buffer Reimplementation. Retrieved from https://code.visualstudio.com/blogs/2018/03/23/text-buffer-reimplementation

[45] 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)*, 2009. 395–403. https://doi.org/10.1109/icdcs.2009.20

[46] 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)*, 1996. 288–297. https://doi.org/10.1145/240080.240305

[47] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *Journal of Parallel and Distributed Computing* 71, 3 (March 2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

[48] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*, October 2011. 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[49] Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. 2022. Certified mergeable replicated data types. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022. ACM, 332–347. https://doi.org/10.1145/3519939.3523735

[50] Michael Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth O'Neil, Patrick O'Neil, Alexander Rasin, Nga Tran, and Stanley Zdonik. 2005. C-Store: A Column-oriented DBMS. In *31st International Conference on Very Large Data Bases (VLDB)*, 2005. 553–564.

[51] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. 1998. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction* 5, 1 (March 1998), 63–108. https://doi.org/10.1145/274444.274447

[52] David Sun, Chengzheng Sun, Agustina Ng, and Weiwei Cai. 2020. Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW1 (May 2020), 1–30. https://doi.org/10.1145/3392825

[53] Bartosz Sypytkowski, Kevin Jahns, and John Waidhofer. Y CRDT: Rust port of Yjs. Retrieved from https://github.com/y-crdt/y-crdt

[54] Michael Toomim. 2024. CRDT and OT generalize as Time Machines. Retrieved from https://braid.org/time-machines

[55] Matthew Weidner and Martin Kleppmann. 2023. The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing. Retrieved from https://arxiv.org/abs/2305.00583

[56] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2010. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems* 21, 8 (2010), 1162–1174. https://doi.org/10.1109/tpds.2009.173

Table 1. The text editing traces used in our evaluation. *Repeats*: Number of times the original trace was repeated to normalise its length relative to the other traces. *Events*: total number of editing events, in thousands, including repeats. Each inserted or deleted character counts as one event. *Average concurrency*: mean number of concurrent branches per event in the trace. *Graph runs*: number of sequential runs of events (linear event sequences without branching/merging). *Authors*: number of users who added at least one event. *Chars remaining*: percentage of inserted characters that remain in the document (i.e., are never deleted) after all events have been merged. *Final size*: Resulting document size in kilobytes after all events have been merged.

Name	Type	Repeats	Events (k)	Avg Concurrency	Graph runs	Authors	Chars remaining (%)	Final size (kB)
S1	sequential	3	779	0	1	2	57.5	307.2
S2	sequential	3	1105	0	1	1	26.7	166.3
S3	sequential	1	2218	0	1	2	7.5	84.4
C1	concurrent	25	652	0.43	92101	2	90.1	521.5
C2	concurrent	25	608	0.44	133626	2	93	516.3
A1	asynchronous	1	947	0.1	101	194	7.8	37.2
A2	asynchronous	2	698	6.11	2430	299	49.6	222

A Editing Traces

Table 1 gives an overview of the editing traces used in our evaluation (Section 4). All traces are freely available for benchmarking collaborative text editing algorithms on GitHub [22]. The traces represent the editing history of the following documents:

Sequential Traces These traces have no concurrency. They were recorded using an instrumented text editor that recorded keystroke-granularity editing events. Trace S1 is the LaTeX source of a journal paper [30,34] written by two authors who took turns. S2 is the text of an 8,800-word, single-author blog post [23]. S3 is the Typst source of this paper that you are currently reading.

Concurrent Traces Trace C1 is two users in the same document, writing a reflection on an episode of a TV series they have just watched. C2 is two users collaboratively reflecting on going to clown school together. We recorded these real-time collaborations with keystroke granularity, and we added 1 sec (C1) or 0.5 sec (C2) artificial latency between the collaborating users to increase the incidence of concurrent operations.

Asynchronous Traces We reconstructed the editing trace of some files in Git repositories. The event graph mirrors the branching/merging of Git commits. Since Git does not record individual keystrokes, we generated the minimal edit operations necessary to perform each commit's diff. Trace A1 is `src/node.cc` from the Git repository for Node.js [4], and A2 is `Makefile` from the Git repository for Git itself [5].

We recorded the sequential and concurrent traces ourselves, collaborating with friends or colleagues. All contributors to the traces have given their consent for their recorded keystroke data to be made publicly available and to be used for benchmarking purposes. The asynchronous traces are derived from public data on GitHub.

The recorded editing traces originally varied a great deal in length. To allow easier comparison of measurements between traces, we have attempted to roughly standardise the sizes of all editing traces to contain approximately 500k inserted characters (with the exception of S3, which is approximately twice this size). We did this by duplicating the shorter event graphs multiple times in our data files, without introducing any concurrency (that is, all events from one run of the trace happen either before or after all events from another run). We repeat the original S1 and S2 traces 3 times, the original C1 and C2 traces 25 times, and the original A2 trace twice. The statistics given in Table 1 are after duplication.

B Proof of Correctness

We now demonstrate that Eg-walker is a correct collaborative text algorithm by showing that it satisfies the *strong list specification* proposed by Attiya et al. [8], a formal specification of collaborative text editing. Informally speaking, this specification requires that replicas converge to the same document state, that this state contains exactly those characters that were inserted but not deleted, and that inserted characters appear in the correct place relative to the characters that surrounded it at the time it was inserted. Assuming network partitions are eventually repaired, this is a stronger specification than *strong eventual consistency* [48], which is a standard correctness criterion for CRDTs [24].

With a suitable algorithm for ordering concurrent insertions at the same position, Eg-walker is also able to achieve maximal non-interleaving [55], which is a further strengthening of the strong list specification. However, since that algorithm is out of scope of this paper, we also leave the proof of non-interleaving out of scope.

B.1 Definitions

Let Char be the set of characters that can be inserted in a document. Let $\mathsf{Op} = \{Insert(i, c) \mid i \in \mathbb{N} \land c \in \mathsf{Char}\} \cup \{Delete(i) \mid i \in \mathbb{N}\}$ be the set of possible operations. Let ID be the set of unique event identifiers, and let $\mathsf{Evt} = \mathsf{ID} \times \mathcal{P}(\mathsf{ID}) \times \mathsf{Op}$ be the set of possible events consisting of a unique ID, a set of parent event IDs, and an operation. When $e \in G$ and $e = (i, p, o)$ we also use the notation $e.id = i$, $e.parents = p$, and $e.op = o$.

Definition 1: An event graph $G \subseteq \mathsf{Evt}$ is *valid* if:

1. every event $e \in G$ has an ID $e.id$ that is unique in G;
2. for every event $e \in G$, every parent ID $p \in e.parents$ is the ID of some other event in G;
3. the graph is acyclic, i.e. there is no subset of events $\{e_1, e_2, ..., e_n\} \subseteq G$ such that e_1 is a parent of e_2, e_2 is a parent of e_3, ..., and e_n is a parent of e_1; and
4. for every event $e \in G$, the index at which $e.op$ inserts or deletes is an index that exists (is not beyond the end of the document) in the document version defined by the parents $e.parents$.

Since event graphs grow monotonically and we never remove events, it is easy to ensure that the graph remains valid whenever a new event is added to it.

Attiya et al. make a simplifying assumption that every insertion operation has a unique character. We use a slightly stronger version of the specification that avoids this assumption. We also simplify the specification by using our event graph definition instead of the original abstract execution definition (containing message broadcast/receive events and a visibility relation). These changes do not affect the substance of the proof: each node of our event graph corresponds to a *do* event in the original strong list specification, and the transitive closure of our event graph is equivalent to the visibility relation.

Given an event graph G we define a replay function $\mathsf{replay}(G)$ as introduced in Section 2.4, based on the Eg-walker algorithm. It iterates over the events in G in some topologically sorted order, transforming the operation in each event as described in Section 3, and then applying the transformed operation to the document state resulting from the operations applied so far (starting with the empty document). In a real implementation, replay returns the final document state as a concatenated sequence of characters. For the sake of this proof, we define replay to instead return a sequence of (id, c) pairs, where id is the unique ID of the event that inserted the character c. This allows us to distinguish between different occurrences of the same character. The text of the document can be recovered by simply ignoring the id of each pair and concatenating the characters.

We can now state our modified definition of the strong list specification:

Definition 2: A collaborative text editing algorithm with a replay function $\mathsf{replay}(G)$ satisfies the *strong list specification* if for every valid event graph $G \subset \mathsf{Evt}$ there exists a relation $lo \subset \mathsf{ID} \times \mathsf{ID}$ called the *list order*, such that:

1. For event $e \in G$, let $G_e = \{e\} \cup \mathsf{Events}(e.parents)$ be the subset of G consisting of e and all events that happened before e. Let $doc_e = \mathsf{replay}(G_e) = \langle(id_0, c_0), ..., (id_{n-1}, c_{n-1})\rangle$ be the document state immediately after locally generating e, where $c_i \in$ Char and $id_i \in$ ID. Then:
   (a) doc_e contains exactly the elements that have been inserted but not deleted in G_e:
   $$\begin{gathered}(\exists i \in [0, n-1] : doc_e[i] = (id, c)) \iff \\ (\exists a \in G_e, j \in \mathbb{N} : a.id = id \land a.op = Insert(j, c)) \land \\ (\nexists b \in G_e, k \in \mathbb{N} : b.op = Delete(k) \land \\ \mathsf{replay}(\mathsf{Events}(b.parents))[k] = (id, c)).\end{gathered}$$
   (b) The order of the elements in doc_e is consistent with the list order:
   $$\forall i, j \in [0, n-1] : i < j \implies (id_i, id_j) \in lo.$$
   (c) Elements are inserted at the specified position:
   $$\forall i, c : e.op = Insert(i, c) \implies doc_e[i] = (e.id, c)$$
2. The list order lo is transitive, irreflexive, and total, and thus determines the order of all insert operations in the event graph.

B.2 Proving Convergence

Lemma 3: Let e be an event in a valid event graph such that $e.op = Delete(i)$. In the internal state immediately before applying e (in which all events that happened before e have been advanced and all others have been retreated), either the record that e will update has $s_p =$ `Ins`, or it is part of a placeholder (which behaves like a sequence of $s_p =$ `Ins` records).

Proof: If we had $s_p =$ `NotInsertedYet`, that would imply that we retreated the insertion of the character deleted by e, which contradicts the fact that the insertion of a character must happen before any deletion of the same character. Furthermore, if we had $s_p =$ `Del` k for some k, that would imply that an event that happened before e already deleted the same character, in which case it would not be possible to generate e. This leaves $s_p =$ `Ins` or placeholder as the only options that do not result in a contradiction. □

Lemma 4: Let S_0 be some internal Eg-walker state, and let a and b be two concurrent events. Let S_1 be the internal state resulting from updating S_0 with retreat and advance calls so that the prepare version of S_1 equals the parents of b. Let S_2 be the internal state resulting from first replaying a on top of S_0, and then retreating and advancing so that the prepare version of S_2 equals the parents of b. Then the only difference between S_1 and S_2 is in the record inserted or updated by a (and possibly the split of a placeholder that this record falls within); the rest of S_1 and S_2 is the same.

Proof: Since S_0 is produced by Eg-walker, it contains records for all characters that have been inserted or deleted by events since the last critical version prior to a and b, it contains placeholders for any characters inserted but not deleted prior to that critical version, and it does not contain anything for characters that were deleted prior to that critical version. By the definition of critical version, any event e that is concurrent with a or b must be after the critical version, and therefore the record that is updated by e must exist in S_0.

S_1 has the same record sequence and the same s_e in each record as S_0, since retreating and advancing do not change those things. The s_p values in S_1 are set so that every record inserted by an event concurrently with b has $s_p = \texttt{NotInsertedYet}$, every record whose insertion happened before b but which was not deleted before b has $s_p = \texttt{Ins}$, and every record that was deleted by $k > 0$ separate events before b has $s_p = \texttt{Del}\ k$. To achieve this it is sufficient to consider events that happened after the last critical version. Thus, the s_p values in S_1 do not depend on the s_p values in S_0, and they do not depend on any events that are concurrent with b.

Replaying a on top of S_0 involves first updating the s_p values to set the prepare version to the parents of a (which may differ from the parents of b), and then applying a, which either inserts or updates a record in the internal state, and possibly splits a placeholder to accommodate this record. S_2 is then produced by updating all of the s_p values in the same way as for S_1. As these s_p values depend only on $b.parents$ and not on a, S_2 is identical to S_1 except for the record inserted or updated by a. ☐

Lemma 5: Let a and b be two concurrent events such that $a.op = Insert(i, c_i)$ and $b.op = Insert(j, c_j)$. If we start with some internal state and document state and then replay a followed by b, the resulting internal state and document state are the same as if we had replayed b followed by a.

Proof: To replay a followed by b, we first retreat/advance so that the prepare state corresponds to $a.parents$, then apply a, then retreat a, then retreat/advance so that the prepare state corresponds to $b.parents$, then apply b. Applying a inserts a record into the internal state, and after retreating a this record has $s_p = \texttt{NotInsertedYet}$ and $s_e = \texttt{Ins}$. Since b is concurrent to a, a cannot be a critical version, and therefore the internal state is not cleared after applying a. When b is applied, the presence of the record inserted by a is the only difference between the internal state when applying b after a compared to applying b without applying a first (by Lemma 4). When determining the insertion position in the internal state for b's record based on b's index j, the record inserted by a does not count since it has $s_p = \texttt{NotInsertedYet}$. Therefore, b's record is inserted into the internal state at the same position relative to its neighbours, regardless of whether a has been applied previously. By similar argument the same holds for a's record.

As explained in Section 3.3, the internal state uses a CRDT algorithm to place the records in the internal state in a consistent order, regardless of the order in which the events are applied. The details of that algorithm go beyond the scope of this paper. The key property of that algorithm is that the final sequence of internal state records is the same, regardless of whether we apply first a and then b, or vice versa. For example, if we first apply a then b, and if the final position of b's record in the internal state is after a's record, then the CRDT algorithm has to skip over a's record (and potentially other, concurrently inserted records) when determining the insertion position for b's record. This process never needs to skip over a placeholder, since placeholders represent characters that were inserted before the last critical version. It only ever needs to skip over records for insertions that are concurrent with a or b; by the definition of critical versions, all such insertion events appear after the last critical version (and hence after the last internal state clearing) in the topological sort, and therefore they are represented by explicit internal state records, not placeholders.

Now we consider the document state. WLOG assume that the record inserted by a appears at an earlier position in the internal state than the record inserted by b (regardless of the order of applying a and b). Let i' be the transformed index of $a.op$ when a is applied first, and let j' be the transformed index of $b.op$ when b is applied first.

Say we replay a before b. When computing the transformed index for b, the internal state record for a has $s_p = \texttt{NotInsertedYet}$, and hence it is not counted when mapping $b.op$'s index j to b's internal state record. However, a's record *is* counted when mapping b's internal state record back to an index, since a's record has $s_e = \texttt{Ins}$ and it appears before b's record. Therefore the transformed index for $b.op$ is $j' + 1$ when applied after a. On the other hand, if we replay b before a, the record for b appears after the record for a in the internal state, so the transformed index for a is i', unaffected by b. Thus, we have the situation as shown in Figure 1, and the effect of the two insertions a and b on the document state is the same regardless of their order. ☐

Lemma 6: Let a and b be two concurrent events such that $a.op = Insert(i, c)$ and $b.op = Delete(j)$. If we start with some internal state and document state and then replay a followed by b, the resulting internal state and document state are the same as if we had replayed b followed by a.

Proof: Since a and b are concurrent, the character being deleted by b cannot be the character inserted by a. We therefore only need to consider two cases: (1) the record inserted by a has an earlier position in the internal state than the record updated by b; or (2) vice versa.

Case (1): If we replay a before b, we first apply a, then retreat a, then apply b (and also retreat/advance other events before applying, like in Lemma 5). Applying a inserts a record into the internal state, and after retreating a this record has $s_p = \texttt{NotInsertedYet}$ and $s_e = \texttt{Ins}$. When subsequently applying b we update an internal state record at a later position. The record inserted by a is not counted when mapping b's index to an internal record, but it is counted when mapping the internal record back to a transformed index, resulting in b's transformed index being one greater than it would have been without earlier applying a (by Lemma 4). On the other hand, if we replay b before a, the record updated by b appears after a's record in the internal state, so the transformation of a is not affected by b. The transformed operations therefore converge.

Case (2): If we replay b before a, we first apply b, then retreat b, then apply a (plus other retreats/advances). Applying b updates an existing record in the internal state (possibly splitting a placeholder in the process). Before applying b this record must have $s_p = \texttt{Ins}$ (by Lemma 3), and it can have either $s_e = \texttt{Ins}$ (in which case, the transformed operation for b is $\mathit{Delete}(j')$ for some transformed index j') or $s_e = \texttt{Del}$ (in which case, b is transformed into a no-op). After applying and retreating b this record has $s_p = \texttt{Ins}$ and $s_e = \texttt{Del}$ in any case. We next apply a, which by assumption inserts a record into the internal state at a later position than b's record. If we had $s_e = \texttt{Del}$ before applying b, the process of applying and retreating b did not change the internal state, so the transformed operation for a is the same as if b had not been applied, which is consistent with the fact that b was transformed into a no-op. If we had $s_e = \texttt{Ins}$ before applying b, b's record is counted when mapping a's index to an internal record position, but not counted when mapping the internal record back to a transformed index, resulting in a's transformed index being one less than it would have been without earlier applying b (by Lemma 4), as required given that b has deleted an earlier character. On the other hand, if we replay a before b, the record inserted by a appears after b's record in the internal state, so the transformation of b is not affected by a, and the transformed operations converge. □

Lemma 7: Let a and b be two concurrent events such that $a.\mathit{op} = \mathit{Delete}(i)$ and $b.\mathit{op} = \mathit{Delete}(j)$. If we start with some internal state and document state and then replay a followed by b, the resulting internal state and document state are the same as if we had replayed b followed by a.

Proof: WLOG we need to consider two cases: (1) the record updated by a has an earlier position in the internal state than the record updated by b; or (2) a and b update the same internal state record. The case where a's record has a later position than b's record is symmetric to (1).

Case (1): We further consider two sub-cases: (1a) the record that a will update has $s_e = \texttt{Ins}$ prior to applying a; or (1b) the record has $s_e = \texttt{Del}$.

Case (1a): Say we replay a before b. Before applying a, the record that a will update must have $s_p = \texttt{Ins}$ (by Lemma 3). After applying and retreating a, the record updated by a has $s_p = \texttt{Ins}$ and $s_e = \texttt{Del}$, and the transformed operation for a is $\mathit{Delete}(i')$ for some transformed index i'. We subsequently apply b, which by assumption updates an internal state record that is later than a's. a's record is therefore counted when mapping the index of $b.\mathit{op}$ to an internal record position, but not counted when mapping the internal record back to a transformed index. If a had not been replayed previously, it would have been counted during both mappings (by Lemma 4). Thus, if the record updated by b has $s_e = \texttt{Ins}$, the transformed operation for b is $\mathit{Delete}(j' - 1)$, where j' is the transformed index of b's operation if a had not been replayed previously, and $j' - 1 \geq i'$, as required. If b's record previously has $s_e = \texttt{Del}$, it is transformed into a no-op. On the other hand, if we replay b before a, the record updated by b appears later than a's record in the internal state, so the transformation of a is not affected by b.

Case (1b): Say we replay a before b. Before applying a, the record that a will update must have $s_p = \texttt{Ins}$ (by Lemma 3), and $s_e = \texttt{Del}$ by assumption. After applying and retreating a, the record updated by a remains in the same state ($s_p = \texttt{Ins}$, $s_e = \texttt{Del}$), and the transformed operation for a is a no-op. When we subsequently apply b, the transformed operation is therefore the same as if a had not been applied, as required. On the other hand, if we replay b before a, the record updated by b appears later than a's record in the internal state, so the transformation of a is not affected by b.

Case (2): Before replaying both of the events, the record that both events update may have $s_e = \texttt{Ins}$ or $s_e = \texttt{Del}$, but after applying the first event it definitely has $s_e = \texttt{Del}$. The second event will therefore be transformed into a no-op. The same happens regardless of whether a or b is replayed first, so the result does not depend on the order of replay of the two events. □

Lemma 8: Given a valid event graph G, $\mathsf{replay}(G)$ is a deterministic function. In other words, any two replicas that have the same event graph converge to the same document state and the same internal state.

Proof: The algorithms to transform an operation and to apply a transformed operation to the document state are by definition deterministic. This leaves as the only source of nondeterminism the choice of topologically sorted order (G is valid and hence acyclic, thus at least one such order exists, but there may be several topologically sorted orders if

G contains concurrent events). We show that all sort orders result in the same final document state.

Let $E = \langle e_1, e_2, ..., e_n \rangle$ and $E' = \langle e'_1, e'_2, ..., e'_n \rangle$ be two topological sort orders of $G = \{e_1, e_2, ..., e_n\}$. Then E' must be a permutation of E. Both sequences are in some causal order, that is: if $e_i \to e_j$ (e_i happens before e_j, as defined in Section 2.2), then e_i must appear before e_j in both E and E'. If $e_i \parallel e_j$ (they are concurrent), the events could appear in either order. Therefore, it is possible to transform E into E' by repeatedly swapping two concurrent events that are adjacent in the sequence. We show that at each such swap we maintain the invariant that the document state and the internal state resulting from replaying the events in the order before the swap are equal to the states resulting from replaying the events in the order after the swap. Therefore, the document state and the internal state resulting from replaying E are equal to those resulting from E'.

Let $\langle e_1, e_2, ..., e_i, e_{i+1}, ..., e_n \rangle$ be the sequence of events prior to one of these swaps, and e_i, e_{i+1} are the events to be swapped. Replaying the events in the prefix $\langle e_1, e_2, ..., e_{i-1} \rangle$ is a deterministic algorithm resulting in some document state and some internal state. Next, we replay either e_i followed by e_{i+1}, or e_{i+1} followed by e_i. Since e_i and e_{i+1} are concurrent, it is not possible for only one of the two to be contained in a critical version, and therefore no state clearing will take place between applying these two events. If e_i and e_{i+1} are both insertions, the resulting states in either order are the same by Lemma 5. If one of e_i and e_{i+1} is an insertion and the other is a deletion, we use Lemma 6. If both e_i and e_{i+1} are deletions, we use Lemma 7. Finally, replaying the suffix $\langle e_{i+2}, ..., e_n \rangle$ is a deterministic algorithm. This shows that concurrent operations commute. □

B.3 Satisfying the Strong List Specification

Lemma 9: Let G be a valid event graph, let $doc = \mathsf{replay}(G)$ be the document state resulting from replaying G, and let S be the internal state after replaying G. Then the ith element in doc corresponds to the ith record with $s_e = \texttt{Ins}$ in the internal state (counting placeholders as having $s_e = \texttt{Ins}$, and not counting records with $s_e = \texttt{Del}$). Moreover, the set of elements in doc is exactly the elements that have been inserted but not deleted in G:

$$\begin{gathered}(\exists i \in [0, n-1] : doc[i] = (id, c)) \iff \\ (\exists a \in G, i \in \mathbb{N} : a.id = id \land a.op = Insert(i, c)) \land \\ (\nexists b \in G, i \in \mathbb{N} : b.op = Delete(i) \land \\ \mathsf{replay}(\mathsf{Events}(b.parents))[i] = (id, c)).\end{gathered}$$

Proof: Let $E = \langle e_1, e_2, ..., e_n \rangle$ be some topological sort of G, and assume that we replay G in this order. By Lemma 8 it does not matter which of the possible orders we choose. We then prove the thesis by induction over n, the number of events in G. The base case is trivial: $G = \{\}$, $doc = \langle\rangle$, so there are no events, no records in the internal state, and no elements in the document state.

Inductive step: Let $E_k = \langle e_1, e_2, ..., e_k \rangle$ with $k < n$ be a prefix of E. Since the set of events in E_k also forms a valid event graph, we can assume the inductive hypothesis, namely that replaying E_k results in a document corresponding to the records with $s_e = \texttt{Ins}$ in the resulting internal state, and the document contains exactly those elements that have been inserted but not deleted by an operation in E_k. We now add e_{k+1}, the next event in the sequence E, to the replay. We do this by transforming e_{k+1} using the internal state obtained by replaying E_k, and applying the transformed operation to the document state from E_k. We need to show that the invariant is still preserved in the following two cases: either (1) $e_{k+1}.op = Insert(j, c)$ for some j, c, or (2) $e_{k+1}.op = Delete(j)$ for some j. We also have to consider the case where the internal state is cleared, but we begin with the case where no state clearing occurs.

Case (1): The set of elements that have been inserted but not deleted grows by $(e_{k+1}.id, c)$ and otherwise stays unchanged. The transformation of an insertion operation is always another insertion operation. The document state is therefore updated by inserting the same element $(e_{k+1}.id, c)$, and otherwise remains unchanged. Moreover, the transformed index of that insertion is computed by counting the number of internal state records with $s_e = \texttt{Ins}$ that appear before the new record in the internal state, and the new record also has $s_e = \texttt{Ins}$, and the s_e property of no other record is updated, so the correspondence between internal state records and document state is preserved.

Case (2): The element being deleted is at index j in the document at the time e_{k+1} was generated, which is $\mathsf{replay}(\mathsf{Events}(e_{k+1}.parents))$. We compute this element by retreating and advancing events until the prepare version equals $e_{k+1}.parents$, and then finding the jth (zero-indexed) record in the internal state that has $s_p = \texttt{Ins}$. The records with $s_p = \texttt{Ins}$ are those that have been inserted but not deleted in events that happened before e_{k+1}, and therefore the jth such record is the record corresponding to $\mathsf{replay}(\mathsf{Events}(e_{k+1}.parents))[j]$. Before applying e_{k+1}, this record may have either $s_e = \texttt{Ins}$ or $s_e = \texttt{Del}$. If $s_e = \texttt{Ins}$, we update it to $s_e = \texttt{Del}$, and transform e_{k+1} into a deletion whose index is the number of $s_e = \texttt{Ins}$ to the left of the target record in the internal state; by the inductive hypothesis, this is the correct document element to be deleted. If $s_e = \texttt{Del}$ before applying e_{k+1}, that event is transformed into a no-op, since another operation in E_k has already deleted the element in question from the document state. In either case, we preserve the invariants of the induction.

If e_{k+1} is a critical version, we clear the internal state and replace it with a placeholder. By the definition of critical version, every event in E_k and e_{k+1} happened before every event in the rest of E. Therefore, after retreating and advancing any event after e_{k+1}, any internal state record with $s_e = \texttt{Del}$ will also have $s_p = \texttt{Del}\ k$ for some $k > 0$, and any internal state record with $s_e = \texttt{Ins}$ will also have $s_p = \texttt{Ins}$ unless it is deleted by an event after e_{k+1}. Since an internal state with $s_e = \texttt{Del}$ can never move to state $s_e = \texttt{Ins}$, this means that any records with $s_e = \texttt{Del}$ as of the critical version can be discarded, since they will never again be needed for transforming the index of an operation after e_{k+1}. Moreover, since all of the remaining records have $s_e = s_p = \texttt{Ins}$ as of the critical version, and since the replay of the remaining events in E will never need to advance or retreat an event prior to the critical version, all of the records in the internal state can all be replaced by a single placeholder while still preserving the invariants of the induction. □

Theorem 10: The Eg-walker algorithm satisfies the strong list specification (Definition 2).

Proof: Given a valid event graph G, let $\mathsf{replay}(G)$ be the replay function based on Eg-walker, as introduced earlier. We must show that there exists a list order $lo \subset \mathsf{ID} \times \mathsf{ID}$ that satisfies the conditions given in Definition 2. We claim that this list order corresponds exactly to the sequence of records and placeholders in the internal state after replaying the entire event graph G. By Lemma 8, this internal state exists and is unique. This correspondence is more apparent if we assume a variant of Eg-walker that does not clear the internal state on critical versions, but we also claim that performing the optimisations in Section 3.5 preserves this property.

To begin, note that the internal state is a totally ordered sequence of records, and that (aside from clearing the internal state) we only ever modify this sequence by inserting records or by updating the s_p and s_e properties of existing records. Thus, if a record with ID id_i appears before a record with ID id_j at some point in the replay, the order of those IDs remains unchanged for the rest of the replay. We define the list order lo to be the ordering relation among IDs in the internal state after replaying G using a Eg-walker variant that does not clear the internal state. This order exists, is unique (Lemma 8), and is by definition transitive, irreflexive, and total, so it meets requirement (2) of Definition 2.

Let $e \in G$ be any event in the graph, and let $G_e = \{e\} \cup \mathsf{Events}(e.parents)$ be the subset of G consisting of e and all events that happened before e. Note that G_e satisfies the conditions in Definition 1, so it is also valid. Let $doc_e = \mathsf{replay}(G_e) = \langle (id_0, c_0), ..., (id_{n-1}, c_{n-1}) \rangle$ be the document state immediately after locally generating e. Since replay is deterministic (Lemma 8), doc_e exists and is unique.

By Lemma 9, doc_e contains exactly the elements that have been inserted but not deleted in G_e, which is requirement (1a) of Definition 2. Also by Lemma 9, the ith element in doc_e corresponds to the ith record with $s_e = \texttt{Ins}$ in the internal state obtained by replaying G_e. Since any pair of IDs that are ordered by the internal state derived from G_e retain the same ordering in the internal state derived from G, we know that the ordering of elements in doc_e is consistent with the list order lo, satisfying requirement (1b) of Definition 2.

Finally, to demonstrate requirement (1c) of Definition 2 we assume that $e.op = Insert(i, c)$, and we need to show that $doc_e[i] = (e.id, c)$. Since G_e contains only e and events that happened before e, but no events concurrent with e, we know that immediately before applying e, every record in the internal state will have $s_p = \texttt{Ins}$ if and only if it has $s_e = \texttt{Ins}$ (because there are no events that are reflected in the effect version but not in the prepare version $e.parents$). Therefore, the set of records that are counted while mapping the original insertion index i to an internal state record equals the set of records that are counted while mapping the internal record back to a transformed index. Thus, the transformed index of the insertion is also i, and therefore the new element is inserted at index i of the document as required. This completes the proof that Eg-walker satisfies the strong list specification. □