Chronofold: a data structure for versioned text

Victor Grishchenko
N.N. Krasovskii Institute of Mathematics and Mechanics
Ekaterinburg, Russia
gritzko@imm.uran.ru

Mikhail Patrakeev
N.N. Krasovskii Institute of Mathematics and Mechanics
Ekaterinburg, Russia
patrakeev@mail.ru

Abstract

Collaborative text editing and versioning is known to be a tough topic. Diffs, OT and CRDT are three relevant classes of algorithms which all have their issues. CRDT is the only one that works correctly and deterministically in a distributed environment, at the unfortunate cost of data structure complexity and metadata overheads.

A *chronofold* is a data structure for editable linear collections based on the Causal Tree CRDT model. A chronofold maintains time-ordering and space-ordering of its elements. Simply put, a it is both a log and a text at the same time, which makes it very convenient for text versioning and synchronization. Being a simple array-based data structure with O(1) insertions, chronofold makes CRDT overheads acceptable for many practical applications.

ACM Reference format:
Victor Grishchenko and Mikhail Patrakeev. 2016. Chronofold: a data structure for versioned text. In *Proceedings of , , February 25, 2020 (DRAFT),* 7 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Even without the real-time collaboration, data structures for editable text is a vast field on its own. Plain text storage and transmission is not a challenge for modern computers; "War and Peace" weighs 3MB, on par with a smartphone photograph. Text editing is more demanding, as it needs fast writes and some basic versioning functionality (at least, for undo/redo functions). Naive implementations do not suffice; there is an entire class of editable-text data structures, such as gap buffers [15], piece tables [25], splay trees [32], ropes [13] and others. Integrated Development Environments (IDEs) have even more reasons to version the edited text; one of them is asynchronous communication between multiple worker threads or processes. Finally, there are Source Code Management systems (SCM), where texts have to be versioned and stored long-term. The underlying models of text versioning have plenty of overlap in these three classes of applications.

The classic plain text versioning model sees any document change (a *diff*) as a number of range insertions and deletions. Alternatively, that can be generalized to a number of range replacements (*splices*). The Myers algorithm [26] can calculate a diff from two versions of a text in $O(ND)$ time, where N is the combined size of the texts and D is the size of the changes; thus the worst case is $O(N^2)$. That is less of a problem for `diff`, `patch`, `svn`, `git`, etc, as their unit of change is a line of text. There are much less lines than characters and lines are more unique, so a number of optimizations and heuristics make Myers good enough in all the reasonable cases. If the unit of change is a *character*, Myers is much more of a challenge; e.g. Google's diff-match-patch [18] library uses timers to provide a good-enough result in acceptable time. Another issue with the diff approach is its non-determinism in case of concurrent changes. To integrate a change, `patch` relies on its position and *context* (the text around the changed spot). Concurrent changes may garble both causing mis-application of a patch. That is only partially solved by heuristics, so SCMs require manual merge of changes in any non-trivial cases.

Weave [29] is a classic data structure for text versioning. It was invented in the 1970s and used in SCCS, TeamWare and most recently BitKeeper as the main form of storage and by virtually every other SCM for merge of concurrent changes. A weave has a reputation of one of the most reinvented data structures in history. It is alternatively known as "interleaved deltas", "union string", and under other names. Its key idea is simple: annotate all pieces of a text (*deltas*) with their "birth" and "death" dates, keep the deleted pieces in their place. Then, one pass of such a collection can produce any version of the text, if all the "dead" and "yet-unborn" pieces are filtered out. The top issue with a weave is that it needs to be spliced on every edit (i.e. rewritten in full), very much like a plain string. The original SCCS weave was line-based, but we will use it as a broad term for this kind of a data structure, no matter line- or character-based.

Notably, the widely popular `git` SCM [5] has immutable binary blobs as its primary abstraction, no deltas. Still, it employs weave-like data structure to merge concurrent changes, while its internal storage format is organized around delta compression. It also supports line-based patches and blame maps. Ironically, declaring snapshots as its primary abstraction made `git` use deltas more, not less.

DOI: 10.1145/nnnnnnn.nnnnnnn

The *Operational Transformation* (OT) model [17] originated from the first experiments with real-time collaborative editors in the 80s. With OT, each single-character edit can be sent out immediately as an *operation* (op). OT needed deterministic merge of changes, despite any concurrent modification. Hence, it relied on positions, not contexts, to apply the changes. Positions are also affected by concurrent edits, so OT iteratively *transforms* the operations to keep them correct. That works reasonably well, except that concurrent modifications create combinatorially complex and highly counter-intuitive effects. For that reason, any practical OT implementation relies on a central server to transform the ops. Despite its somewhat torturous history, OT eventually led to such applications as Google Docs.

In 2006, the dissatisfaction with OT led to a new proposal named WithOut Operational Transforms (WOOT) [28]. WOOT represents a text as a directed acyclic graph of characters, each one referencing its left and right neighbors at the time of insertion. Unique per-character identifiers work as references. The order of identifiers resolves ties between concurrent insertions to the same location. Deleted characters get marked with *tombstones*. The approach was obviously correct, but highly impractical due to metadata overheads.

Causal Tree (CT) [20] aimed at improving WOOT, along with Logoot [34], TreeDoc [24], LSEQ [27] and other proposals. In particular, CT reduces per-character metadata to (a) *logical timestamp* (b) timestamp of the preceding character. Logical timestamps [22] are tuples $\langle t, a \rangle$ where t is the logical time value and p is the process id. The lexicographic order of timestamps forms an *arbitrary total order* [23] (ATO) consistent with the cause-effect ordering. CT employed fixed-width logical timestamps of various kinds (at least five varieties are known to the authors). Once the first draft of CT [19] appeared in 2008, it was immediately noted [14] that CT's inner workings are very reminiscent of a weave. In 2010, the Replicated Growable Array (RGA) [30] algorithm was proposed. In 2011, a broad term was introduced for this class of algorithms: Conflict-free Replicated Data Types (CRDT) [31]. In 2016, it was proven [11] that RGA and CT use the same algorithm (curiously, the paper uses another term for CT, a Timestamped Insertion Tree). Various OT-with-tombstones proposals effectively resulted in similar weave-building algorithms.

Despite the fact that both academics and the industry made circles around CRDTs for that many years, no standard solution emerged. The key issue remained the same: metadata overheads and cognitive costs of a distributed data model. So far, all the industry adoption of CRDTs had the air of a pilot project. By 2014, CT was deployed in the Yandex Live Letters collaborative editor [21] which was phased out several years later. Another CRDT based editor implementation was made [2] by 2019. CRDT was employed in the Atom editor [3] for its collaborative features. Apple Notes is known to sync with CRDTs [8]. Some deployed simplified ersatz-CRDT models [33]. As recently as in 2019, two high-profile CRDT-based editor projects fell short of the objectives (Google-associated *xi* [6] and GitHub's *xray* [4]). Authors cited data structure complexity as a major impediment.

CRDT's metadata and complexity issues become clear if we compare those to the SCCS weave [29]. The classic line-based weave was fundamentally a linear data structure, a stream of deltas interleaved with per-delta annotations. Today, it is considered inefficient, mostly because of metadata volume and $O(N)$ splicing writes. RGA, if applied to texts, has to build a weave with plenty of metadata: two timestamps and two pointers per character, according to the original article. A naive CT implementation may store a text as an actual tree of letters, with each letter being a rather complex object. Such implementations are known to exist and they don't work well. Another source of complexity is the fact that logical timestamps and change history are not linear. The use of version vectors to deal with partial orders is often too expensive to be practical. CRDT is not the only technology struggling with partial orders; `git` branch mechanics is notoriously labyrinthine.

All the efforts to optimize text CRDTs are too many to list them here. Some tried to work with ranges of characters [10] instead of single characters. Another approach was to compress ranges of timestamps [7] that are close in value. CT in particular survived multiple major revisions addressing the issue of overheads. Most CT implementations used flat data structures: strings, arrays and buffers [7, 12]. In particular, the 2012 JavaScript implementation [20, 21] used a peculiar optimization technique coding timestamps as character tuples, see Sec. 3. The 2017 RON CT implementation [7] borrowed the iterator heap technique [1] from LSMT databases. It merges i inputs by a single $O(N \log i)$ pass; the inputs might be versions, patches and/or single ops. The technique is perfect for batched server-side operations, not so much for real-time client-side use; that issue remained open.

This paper proceeds as follows. Sec. 2 explains the category of subjective linear orders and log timestamps, a logical timestamping scheme. That is the key to the article as it lets us use linear addressing in a distributed data structure. In Sec. 3, we introduce chronofold, a data structure for versioned text. Further, in Sec. 4 we put chronofold into the wider context of a complex editor or revision control system and explain how it works in lockstep with other data structures. Finally, Sec. 5 concludes with our findings.

2 CT, log time and subjective orders

In distributed systems, events happen "fast" while messages propagate "slowly". As a result, the perceived order of events is different for different observers. No wonder the seminal paper on distributed systems drew inspiration from relativistic physics. Another clear parallel is the concept of logical time being dependent on the frame of reference.

The CT model is defined in a way to be independent of any replica's local perceived event order (*subjective* order). CT works in terms of a *causal* partial order of events and a compatible timestamp-based ATO. That makes the model simple and its behavior self-evident. The unfortunate cost is that addressing, data structures, and versioning become non-linear and thus complex.

We found that the inner workings of the system might be greatly simplified if we rely on those subjective linear orders instead of ignoring them. As long as the system produces the same text, we have the best of both worlds: simplicity of linear addressing and resilience of a distributed model. In the Replicated Causal Tree model (RCT) we make subjective orders explicit and consider their properties.

We denote by $\mathbb{N}$ the set of natural numbers $\{1, 2, \ldots\}$; the variables i, j, k, l, m, n range over $\mathbb{N}$. Let PROC be an arbitrary set. The elements of PROC are called *processes* and the variables $\alpha, \beta, \gamma, \delta$ range over PROC. Processes create and exchange *events*, denoted by *timestamps*. Timestamps are ordered pairs $e = \langle n, \alpha \rangle$ abbreviated as n^α, where α is the *author* also denoted by $\mathsf{auth}(e)$, and n is the *author's index* also denoted by $\mathsf{andx}(e)$. The cartesian product $\mathbb{N} \times \mathsf{PROC}$ is the set of all (possible) events. An *op* is a record of an event that mentions its timestamp, a timestamp of its causal tree parent, and its value (a character). Each process keeps a subjectively ordered *log* of ops it creates or receives (also named a *replica*).

Definition 2.1. *Replicated Causal Tree, RCT,* is an ordered tuple $T = \langle E, \mathsf{val}, \mathsf{ref}, \mathsf{log} \rangle$ such that E is a set of events, val is a function with domain E, ref is a function from E to E, and log is a function from the set $\mathsf{proc}(T) := \{\mathsf{auth}(e) : e \in E\}$ to the set of injective sequences in E, which associates to every process $\alpha \in \mathsf{proc}(T)$ the sequence $\mathsf{log}(\alpha) = \langle \alpha_i \rangle_{i=1}^{\mathsf{lh}(\alpha)}$ of length $\mathsf{lh}(\alpha)$ of events in E with $\alpha_i \neq \alpha_j$ for $i \neq j$, and such that for all $\alpha, \beta \in \mathsf{proc}(T)$ and $n, m, i \in \mathbb{N}$ with $i \leqslant \mathsf{lh}(\alpha)$ the following three axioms are satisfied:

1. If $n^\alpha \in E$, then $n \leqslant \mathsf{lh}(\alpha)$ and $\alpha_n = n^\alpha$.
2. If $n^\alpha \in E$, then there is $j \leqslant n$ such that $\mathsf{ref}(n^\alpha) = \alpha_j$.
3. If $\alpha_i = m^\beta$, then $\mathsf{LOG}_m(\beta) \subseteq \mathsf{LOG}_i(\alpha)$, where $\mathsf{LOG}_k(\gamma)$ is the set $\{\gamma_1, \ldots, \gamma_k\}$ for $\gamma \in \mathsf{proc}(T)$ and $k \leqslant \mathsf{lh}(\gamma)$.

Three axioms define our timestamping scheme, the tree-forming relation and causal consistency, respectively. Then, a Causal Tree is a directed graph $\big\langle E, \{\langle e, \mathsf{ref}(e) \rangle : e \in E\} \big\rangle$. Note that formally the notation α_i is an abbreviation for $\big(\mathsf{log}(\alpha)\big)_i$, i.e. the i-th term in the sequence $\mathsf{log}(\alpha)$, and this notation can be used only for $\alpha \in \mathsf{proc}(T)$ and $i \leqslant \mathsf{lh}(\alpha)$. Again, the order is not total, hence $\alpha_i = i^\alpha$ does not hold.

Notation 2.2.

✎ $\mathsf{ndx}_\alpha(e) := i$ such that $e = \alpha_i$, if $e \in \mathsf{LOG}_{\mathsf{lh}(\alpha)}(\alpha)$;
✎ $\mathsf{ndx}_\alpha(e) := \infty$, if $e \notin \mathsf{LOG}_{\mathsf{lh}(\alpha)}(\alpha)$.

We call $\mathsf{ndx}_\alpha(e)$ the *index* of event e in the log of process α. Again, the order of events in a log is subjective.

Remark 2.3. *Suppose that* $T = \langle E, \mathsf{val}, \mathsf{ref}, \mathsf{log} \rangle$ *is an RCT and* $e \in E$. *Then:*

☞ $\mathsf{andx}(e) = \mathsf{ndx}_{\mathsf{auth}(e)}(e)$.
☞ $\mathsf{andx}(e) \leqslant \mathsf{ndx}_\alpha(e)$ *for all* $\alpha \in \mathsf{proc}(T)$.
☞ $\mathsf{andx}(e) \geqslant \mathsf{andx}(\mathsf{ref}(e))$ *if* $e \neq \mathsf{ref}(e)$.
☞ $\mathsf{andx}(i^\alpha) = i > \mathsf{andx}(\alpha_j)$, *for all* $j < i$. □

Note that an event's index in its author's log is the smallest index it has, in any log. Also, the index of an event in its author's log is greater than author indices of any preceding events in the same log, including its CT parent. Even with subjective ordering, these features hold because of causal consistency and the way we defined timestamps. We will rely on these features later.

Lemma 2.4. *Suppose that* T *is an RCT,* $\alpha \in \mathsf{proc}(T)$, $i \leqslant \mathsf{lh}(\alpha)$, *and* $\mathsf{ref}(\alpha_i) = k^\gamma$. *Then:*

☞ $\mathsf{LOG}_k(\gamma) \subseteq \mathsf{LOG}_i(\alpha)$.

That is, $\mathsf{LOG}_{\mathsf{andx}(\mathsf{ref}(\alpha_i))}\Big(\mathsf{auth}\big(\mathsf{ref}(\alpha_i)\big)\Big) \subseteq \mathsf{LOG}_i(\alpha)$. □

Notation 2.5.

✎ $\mathsf{pred}(e) := \alpha_{n-1}$, if $e = n^\alpha \in E$ and $n \geqslant 2$;
✎ $\mathsf{pred}(e) := e$, if $e = 1^\alpha \in E$.

We call $\mathsf{pred}(e)$ the *predecessor* of e in its author's log.

Proposition 2.6 (Causal Closedness of $\mathsf{LOG}_i(\alpha)$). *Suppose that* T *is an RCT. Let* $\alpha \in \mathsf{proc}(T)$ *and* $i \leqslant \mathsf{lh}(\alpha)$. *Then the set* $\mathsf{LOG}_i(\alpha)$ *is closed under functions* ref *and* pred. *That is, if* $e \in \mathsf{LOG}_i(\alpha)$, *then* $\mathsf{ref}(e) \in \mathsf{LOG}_i(\alpha)$ *and* $\mathsf{pred}(e) \in \mathsf{LOG}_i(\alpha)$. □

Historically, CT used at least five different logical timestamping schemes (Lamport [23], hybrid [16], abbreviated/char tuple [20], calendar based and others). Given their role in the system, even subtle details had a lot of impact. The scheme defined here is named *log timestamps.* Instead of incrementing the value of the greatest timestamp seen, like the Lamport scheme does, we set it to the event's index in its author's log. The lexicographic ordering of log timestamps is compatible with the causal order. In addition to that, it also provides a lower bound for the event's index in any log. Pragmatically speaking, it is the same number most of the time, as the level of contention between replicas of a text tends to be small. That makes it possible for us to switch from log timestamps to log indices and back, with very little friction.

The importance of this becomes clear when we consider our two uses for timestamps: referencing events and forming the ATO. Locally, referencing by index is much more convenient. The convenience of using an index for ordering depends on whether it matches the author's index. If $\alpha_n = n^\beta$ then the index is also the most lexicographically significant part of the timestamp. If not, it provides an upper bound on that part of the timestamp due to Rem. 2.3. That is enough to determine the ATO in the absolute majority of the cases. So most of the time, a CT implementation may use indices instead of timestamps.

Given that, a process may convert logical timestamps to its log indices once it receives an operation from another process. Then, it proceeds with the indices. When sending operations out, it performs the reverse conversion. This way, we *insulate* data structures from the complexity of a distributed environment.

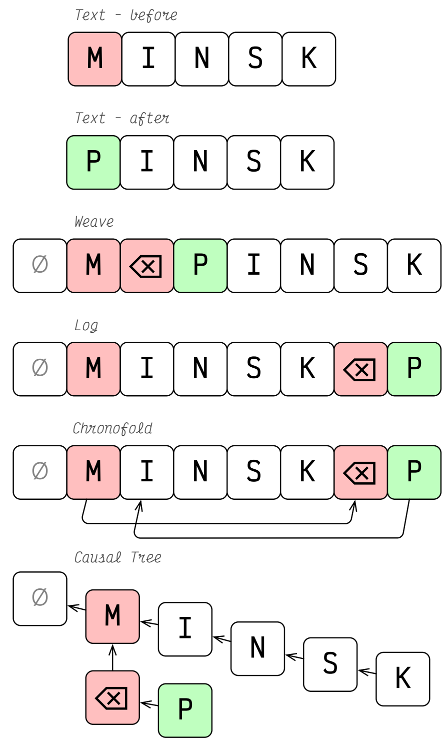
Another important feature of log indices is their stability. The main source of the famous OT complexity is its reliance on a linear addressing system that is not stable between edits. We avoid that here, to our great advantage, see Sec. 4. Given all of the above, we believe it is as natural to use log timestamps for versioned texts as using quaternions for 3D graphics and modeling. In the next section, we introduce a versioned text data structure that is comparable to plain text in terms of complexity and overheads.

3 Chronofold

Every data structure for versioned text has its advantages and shortcomings. A plain text string is the most simple and the most extensively used data structure in the world. Unfortunately, a string is edited by splicing; once we insert a character in the middle, we have to rewrite half the text. Hence, text edits are $O(N)$ while comparing (diffing) two versions of a string by the Myers algorithm is $O(ND)$. Weave is the most natural data structure for diffing and merging, but editing a weave also requires splicing. A log is an append-only data structure, hence has $O(1)$ edits. But, recovering a text from a log of edits is not trivial. Notably, log is an *immutable* data structure in the sense that every prefix of a log is also its complete version. Similarly, any postfix of a log is a list of recent changes, which is very convenient for replication and synchronization. Piece tables as used by many text editors have either $O(1)$ or $O(\log N)$ edits and may provide very limited versioning functionality.

Replicating a data structure introduces additional complexity. The original CT paper employs a hashtable and a log to maintain the weave. Similarly, the original RGA paper describes a mix of a hashtable and a linked list. Namely, letters of a text are put into a hashtable where the letter's timestamp is the key and the value is a tuple of a letter itself and a pointer the next letter. The approach is obviously correct, but very impractical, due to disproportionate overheads even in C-family languages. The 2012/2014 CT implementation [20] avoids the use of a hashtable by scanning a string of character-tuple encoded timestamps. Extremely counter-intuitively, in a garbage-collected language (JavaScript) such a scan introduces less overheads than a hashtable lookup. The reason is the number of tiny objects a hashtable has to create (one per key, one per value). That causes $O(N)$ of bookkeeping overheads and recurring maintenance. If rarely used, a full-scan turns less expensive than bookkeeping. On top of that, iterating a hashtable is very cache-unfriendly as there is no correlation between the character's position in the text and its position in the hashtable. That is yet another reason why our switch to indices makes a big difference.

Figure 1. Versioned text: "MINSK" corrected to "PINSK", stored in different data structures ($\varnothing$: root, $\lhd$: tombstone).



So ideally, we want a replicated versioned text data structure that is stored in an array, addressed by indices, needs no splicing, allows access to past versions of the text and merge of remote changes. We achieve that by combining a weave and a log: a *chronofold*[1] is a subjectively ordered log of tuples $\langle \mathsf{val}(\alpha_i), \mathsf{ndx}_\alpha(\mathsf{w}(\alpha_i)) \rangle$, $i \leqslant \mathsf{lh}(\alpha)$, where $\mathsf{w}(\alpha_i)$ is the event following α_i in the weave. So, the second element of the tuple forms a linked list that contains the weave and thus any version of the text. In the C notation, a text chronofold may look like:

```
struct {
    uint32_t codepoint; // UTF-32 character
    uint32_t next_ndx;  // weave linked list
} * cfold;
```

By reading a chronofold like a log, we see the history of changes. By reading it along the linked list, we may see any version of the text. A chronofold has the good properties of a log, a weave and a piece table: it is splicing-free, versioned and very convenient for synchronization. A chronofold entry

[1]The name of the data structure was decided by a popular vote [9].

takes less space than an op due to the absence of timestamps. We further optimize that in Sec. 4.

The algorithm for merging new ops into a chronofold repeats well-known CT/RGA algorithms. Once process α receives an op i^β, it appends an entry to its chronofold. Next, it has to find the op's position in the weave and relink the linked list to include the new op at that position. It locates the new op's CT parent $\mathsf{ref}(i^\beta) = k^\gamma = \alpha_j$ at the index j in the local log. Here, $k < i$ and $k \leq j$; most of the time we simply have $j = k$. It inserts the op after its parent, unless it finds *preemptive CT siblings* at that location (those are ops also having k^γ as their parent, but having greater timestamps). If found, the new op is inserted after preemptive siblings and their CT subtrees.

If explained in RGA terms [30], the CT parent becomes "left cobject" while preemptive siblings become "succeeding nodes" with greater timestamps/vectors. In the terms of the 2010 paper, those are "parent" and "unaware siblings". Either way, preemptive siblings is the only tricky case in the scheme.

Note that the chronofold building algorithm uses information that is not included into the chronofold itself. Namely, that is the tree-forming ref relation and the timestamp-to-index mapping $\mathsf{ndx}_\alpha : k^\gamma \to j$. It may also need $\mathsf{ndx}_\alpha^{-1} : j \to k^\gamma$ for the case of preemptive siblings. Exporting edits to other replicas needs ndx_α^{-1} to produce the timestamps.

Importantly, if ops are neither imported nor exported, the chronofold itself suffices. Namely, as per Remark 2.3, the timestamp of a new locally authored op is greater than other timestamps in the log. That excludes the case of preemptive siblings, so ndx_α^{-1} is not needed. The index of the preceding character is already known, so ndx_α is not needed either.

Then, the data for $\mathsf{ref}, \mathsf{ndx}_\alpha, \mathsf{ndx}_\alpha^{-1}$ can be kept in a separate data structure thus removing it from the hot code path. From the perspective of a text editor, that makes perfect sense: an op is merged once, then read many times. This is exactly the insulation layer we mentioned in Sec. 2.

The simplest way to store that metadata is to keep a *secondary* log of op timestamps and their ref indices. To implement $\mathsf{ndx}_\alpha^{-1} : j \to k^\gamma$ we simply read that log at the index j. To implement $\mathsf{ndx}_\alpha : k^\gamma \to j$, we may need to scan it from position k to the end, in the worst case. That costs $O(N)$ and from that perspective we might be tempted to store that mapping in a hashtable. That would solve the problem on paper but, as it was described earlier, that may not be a good idea in practice. One way to avoid those worst-case scans is to keep a separate sorted table of index *shifts*. Namely, once $\mathsf{ndx}_\alpha(i^\beta) - i > T$ for some threshold value T, make a shift table entry $\mathsf{shft}_\alpha : i^\beta \to \mathsf{ndx}_\alpha(i^\beta) - i$. Having that entry, we will know that for j^β if $j \geqslant i$ then $\mathsf{ndx}_\alpha(j^\beta) \geqslant \mathsf{shft}_\alpha(i^\beta) + i$. As long as this correction keeps us within T steps from the target, we do not need additional entries for β. This technique is improved in Sec. 4.

With a shift table, ndx has complexity $O(\log N)$, which means $O(\log N)$ insertions, except for one adversarial scenario. Namely, if one op has $O(N)$ CT childen which are fed into our replica in the reverse timestamp order. Then, the case of preemptive siblings turns into the bubble sort algorithm: $O(N)$ per op, $O(N^2)$ total. The scenario corresponds to lots of concurrent insertions into the same point in the text. Due to the properties of causal consistency, one author can not send ops out-of-order, see Def. 2.1, Lemma. 2.4. So this scenario should probably include a Sybil attack too. There is another chronofold-building algorithm that lacks this unfortunate corner case; we have to skip it as it depends on many techniques not explained here.

To illustrate what we achieved here, let us consider two typical versioning operations: recovering a past version and deriving a difference of two versions. Having a CT weave, we would need timestamps and version vectors to filter non-effective ops (removed, yet-unborn) and produce a version of a text or a difference thereof. Having a chronofold, we may iterate its linked list while ignoring all the ops past certain index. This way, we produce a version or a diff using indices only. (Albeit, this only applies to the versions we observed in our subjective linearization of the history; to work with other linearizations we have to build their respective chronofolds.) As we have mentioned earlier, local editing does not use timestamps either. That should make CRDT overheads acceptable for the use cases of undo/redo, real-time collaborative editing, or full-scale revision control.

But, whether we speak of editors, collaborative editors or revision control systems, there is more than plain text. There is also formatting, highlighting, annotations, versioning. In this regard, log timestamps make the data structure extremely flexible and adaptable, see Sec. 4.

4 Co-structures

All but the most basic editors overlay the text with various kinds of formatting. That might be syntax highlighting, spelling errors, compiler warnings, authorship and versioning info, revisions, etc. Differently from embedded markup (e.g. HTML), overlays are decoupled from the text stream, as they merely *reference* text ranges. Sometimes, the code responsible for such overlays may be computationally expensive, so it runs asynchronously in separate threads, processes or remotely on servers. Sometimes, these overlays are stored separately. Either way, as the text keeps changing, the referenced ranges become slightly off. Effectively, some editors have to run miniature OT engines to correct for that effect.

Fortunately, log indices create a stable addressing system for the edited text. As long as we stay with the same replica and same linearization, the indices are not affected by edits. That let us build *co-structures*, overlay data structures linked to the text through log indices. Co-structures reference text ranges, but instead of text positions they use log indices, so no correction needed. This is an improvement over past CT

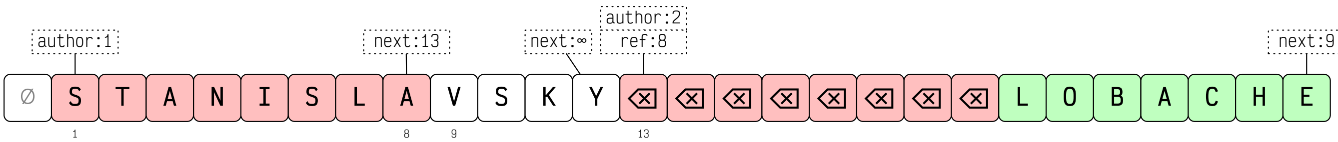


Figure 2. Chronofold and co-structures: "STANISLAVSKY" written by author 1 corrected to "LOBACHEVSKY" by author 2. Co-structures store timestamps (author, shift), the weave (next), and the tree-forming relation ref. Most values are implicit.

editor engines that used logical timestamps to denote such ranges. Again, we evaded the use of distributed primitives.

As a simple example, we may track a binary attribute by keeping a bitmap (e.g. whether a letter is bold). For richer attributes we may use a vector, etc. Although, keeping track of individual characters may not be the most convenient approach. In case we need to reference character ranges, one possible data structure is a *range map*. Namely, we divide a weave into a number of semi-intervals $[a_i, b_i)$ so $a_i = b_{i-1}, i \neq 0$. Each interval has uniform formatting f_i. That formatting we keep in a sorted map $a_i \to f_i$. When iterating the chronofold in the weave order, we check the range map for formatting changes.

As text editors tend to operate in (row,column) coordinates, we may dedicate another co-structure to that purpose. Namely, if we cache log indices of all the newlines in their weave order, we can start iteration from an arbitrary line's beginning. Again, co-structures make a chronofold extremely flexible.

Note that a slightly out-of-date co-structure can still be applied to the chronofold if that makes sense. As co-structures are decoupled from the chronofold, they can be (de)activated, (re)stored, re(built) and/or updated, all independently from the editing process. The only limitation is that the local linearization must stay the same.

Interestingly, this co-structure technique may serve to optimize the chronofold itself. In part, we already did that in the Sec. 3 by offloading metadata to a secondary log. As a next step, we may offload *next* pointers to a co-structure of their own. A typical verioned text consists of *spans* of sequentially typed characters: words, sentences, paragraphs. Simultaneous typing in a real-time collaborative editor may produce messier patterns. But, based on our experience with deployed systems, that is a rare exception, not the rule. In a typical chronofold, most of *next* pointers would be equal to $i + 1$. Instead of spending memory for every such value, we may offload them to a separate sparse array, where only non-trivial pointers are mentioned. In the resulting implementation, a *thinned* chronofold is simply a log of UTF-32 codepoints. As yet another optimization, we may notice that non-BMP codepoints are very rare. If so, we may reduce the core chronofold to a UTF-16 string where all non-BMP codepoints are marked with a special value and stored in a yet another sparse array.

Let's return to the secondary log carrying op timestamps and ref indices (Sec. 3). Author's indices tend to match our local log indices in practice. Even if not, spans will have the same index *shift* due to concurrent edits present in our log before the span. So, instead of individual timestamps we may store timestamp ranges in two range maps (authors and shifts respectively), thus avoiding per-character metadata. In this case, ndx_α^{-1} takes $O(\log N)$ as it only needs two range map lookups. ndx_α is formally $O(N)$ as it may potentially need to scan both range maps to find an op that was shifted from beginning to end. Optimizing is possible, but hardly worth it as we need ndx_α for head-of-span insertion only.

As the final optimization in this paper, we will use the fact that all the co-structures are addressed by the log index. That means, it is possible to put several of them into the same container, to amortize costs. Fig. 2 shows a chronofold with its secondary log and weave pointer co-structures. These can be stored in the same sorted map, assuming the integer key has two flag bits to differentiate between co-structures.

Assuming the cost of co-structures sifficiently amortized, a chronofold's footprint becomes closer to that of a plain non-versioned UTF-16 string, as used in Java, JavaScript, etc.

5 Conclusion

As a data structure, chronofold addresses the shortcomings of weave-based CRDTs. It is a simple array-based data structure with $O(1)$ inserts that might work faster than a plain string in many cases. It works like a piece table for editing, like a log for replication and like a weave for versioning. The authors are looking forward to see chronofold applications in the domains of revision control systems, collaborative software and development environments.

The greatest surprise to the authors though is that linear addressing is applicable to a partially ordered system. The concepts of log timestamps and subjective linear orders mitigate the cognitive and computational costs of a distributed data model. That may potentially find applications beyond the domain of text versioning.

References

[1] Cassandra - MergeIterator. https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/utils/MergeIterator.java.

[2] Codefest19, A CRDT text editor goes to production (Russian). https://2019.codefest.ru/lecture/1501.

[3] Collaborate in real time in atom, https://teletype.atom.io/.

[4] An experimental next-generation electron-based text editor. https://github.com/atom-archive/xray.

[5] Global information tracker, https://git-scm.com/.

[6] A modern editor with a backend written in rust. https://xi-editor.io/.

[7] Replicated object notation 2.0.1, https://github.com/gritzko/ron.

[8] Revisiting Apple Notes (3): Embedded tables, https://ciofecaforensics.com/2020/01/14/apple-notes-revisited-embedded-tables/.

[9] A twitter vote https://twitter.com/gritzko/status/1156782170041700352.

[10] Luc André, Stéphane Martin, Gérald Oster, and Claudia-Lavinia Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 50–59. IEEE, 2013.

[11] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 259–268, 2016.

[12] Alexei Baboulevich. Data laced with history, http://archagon.net/blog/2018/03/24/data-laced-with-history/#causal-trees-in-depth.

[13] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.

[14] Bram Cohen. LiveJournal. https://bramcohen.livejournal.com/49535.html?thread=765311#t765311.

[15] Charles Crowley. Data structures for text sequences https://www.cs.unm.edu/ crowley/papers/sds.pdf.

[16] Murat Demirbas, Marcelo Leone, Bharadwaj Avva, Deepak Madeppa, and Sandeep Kulkarni. Logical physical clocks and consistent snapshots in globally distributed databases. 2014.

[17] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 1989.

[18] Neil Fraser. The diff match patch library https://github.com/google/diff-match-patch.

[19] Victor Grishchenko. Causal trees: towards real-time read-write hypertext.

[20] Victor Grishchenko. Deep hypertext with embedded revision control implemented in regular expressions. *Proceedings of the 6th International Symposium on Wikis and Open Collaboration - WikiSym 10*, 2010.

[21] Victor Grishchenko. Citrea and swarm: Partially ordered op logs in the browser. In *1st Workshop on Principles and Practice of Eventual Consistency (PaPEC). https://doi. org/10.1145/2596631.2596641*, 2014.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[23] Leslie Lamport and Clocks Time. the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[24] M Mihai Letia, N Preguica, and M Shapiro. Crdts: Consistency without concurrency control. *RR-6956, INRIA*, 2009.

[25] J Strother Moore. "My" best ideas. https://www.cs.utexas.edu/users/moore/best-ideas/structure-sharing/text-editing.html.

[26] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[27] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. Lseq: An adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng '13, page 37–46, New York, NY, USA, 2013. Association for Computing Machinery.

[28] Gérald Oster, Pascal Urso, Pascal Molli, Hala Skaf-Molli, and Abdessamad Imine. Optimistic Replication for Massive Collaborative Editing. Research Report RR-5719, INRIA, 2005.

[29] Marc J Rochkind. The source code control system. *IEEE transactions on Software Engineering*, (4):364–370, 1975.

[30] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011.

[31] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[32] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[33] Evan Wallace. How figma's multiplayer technology works. https://www.figma.com/blog/how-figmas-multiplayer-technology-works/.

[34] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, 2010.