Merge What You Can, Fork What You Can't: Managing Data Integrity in Local-First Software

Nicholas Schiefer
MIT CSAIL
Cambridge, MA, USA
schiefer@mit.edu

Geoffrey Litt
MIT CSAIL
Cambridge, MA, USA
glitt@mit.edu

Daniel Jackson
MIT CSAIL
Cambridge, MA, USA
dnj@mit.edu

Abstract

In a local-first architecture that prioritizes availability in the presence of network partitions, there is a tension between two goals: merging concurrent changes without user intervention and maintaining data integrity constraints. We propose a synchronization model called *forking histories* which satisfies both goals in an unconventional way. In the case of conflicting writes, the model exposes multiple event histories that users can see and edit rather than converging to a single state. This allows integrity constraints to be maintained within each history while giving users flexibility in deciding when to manually reconcile conflicts. We describe a class of applications for which these integrity constraints are particularly important and propose a design for a system that implements this model.

CCS Concepts: • **Information systems** → *Distributed database transactions.*

Keywords: CRDTs, Optimistic Concurrency Control

ACM Reference Format:
Nicholas Schiefer, Geoffrey Litt, and Daniel Jackson. 2022. Merge What You Can, Fork What You Can't: Managing Data Integrity in Local-First Software. In *Principles and Practice of Consistency for Distributed Data (PaPoC '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3517209.3524041

1 Introduction

Local-first software [8] prioritizes availability: users can freely access and modify data locally on a client device, then optionally synchronize that data with other devices when connected to a network. This architecture enables offline access, low latency UIs, and other benefits for users, but also introduces challenges around data consistency, since users may concurrently make incompatible edits that must be reconciled.

There are a variety of existing approaches that help application developers grapple with this challenge. For example, Conflict-Free Replicated Data Types (CRDTs) [12, 13] avoid conflicts entirely by modeling data in terms of commutative operations. Meanwhile, systems like Bayou [14] and Ice Cube [7] allow developers to specify application-specific logic for detecting and resolving conflicts. Other systems like Diamond [15] abandon complete local control and provide only limited offline functionality. While these approaches differ in the details of the model they expose to developers, they share a fundamental goal: to ensure that all replicas converge to the same state as quickly as possible given network availability and to maximally preserve user intent.

Digital gardens pose unique sync challenges. While existing techniques have proven useful in many contexts, we argue that they do not offer good solutions for a large class of applications we call *digital gardens*, which include media library managers for photos, music, books, and academic references; personal knowledge management tools for taking and organizing notes; and even filesystem browsers. Digital gardens grow over time, often containing large volumes of data that make it hard for a user to manually audit the state of the system. Digital gardens also manage immensely valuable personal data, where data integrity is paramount.

This combination of features makes it difficult to sync data in a digital garden application. If the system merges changes in an undesirable way, the problem can be hard to detect and can have catastrophic long-lasting implications. Meanwhile, in order to avoid a tedious user experience, it is essential that unrelated changes are still merged automatically without manual action by the user. In Section 2 we demonstrate this tension with an example scenario of two users editing a shared photo library.

This problem seems fundamental, but it relates to a key assumption made by most data sync systems about consistency: that all users should see a single shared state as soon as possible. We argue that this property is not essential in digital gardens—although users typically want to *eventually* converge to a single consistent view, they can accept extended periods where they see different views instead. This

PaPoC '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9256-3/22/04.
https://doi.org/10.1145/3517209.3524041

is familiar to any programmer who has used a version control system like Git: the process of synchronizing changes is not tightly coupled to merging those changes into a single history. By relaxing the assumption that end-users would like to converge immediately upon syncing, we can imagine new strategies for synchronization.

Achieving data integrity with forking histories. We propose *forking histories*: a new model where a synchronization layer could expose *multiple* co-existing states to the user through the application layer. This would enable entirely new techniques for managing concurrent writes—changes could still be automatically merged when possible, but conflicts could result in forking histories which users could inspect, edit, and manually merge back together. It would allow for maintaining strong consistency within each history, avoiding problematic merges. At the same time, users could continue to make progress on any history, not just the primary one, and have control over when to do the work of resolving conflicts.

In effect, we decouple the physical and logical synchronization processes: one replica can receive write events from another and access them if needed, without reconciling them with the local changes. Instead, we defer the logical reconciliation until the user is ready to perform it, without needing to take the device offline. We believe this unconventional model could present an attractive set of tradeoffs in the context of digital garden applications; we elaborate further on this claim in Section 3.

TreeDB: an instance of forking histories. To concretely illustrate how this idea might work in practice, in Section 4 we describe the design of a proposed synchronization system called TreeDB that implements the forking history model. Instead of computing a single consistent state across replicas, we compute a shared tree of reified write events that represents *multiple* possible histories, each one internally consistent and satisfying application-specific integrity constraints. The system can play any history into a materialized state and expose it to the application, so users can view any of the forking histories and not just the "primary" one. Furthermore, users can choose to manually resolve conflicts by patching up the tree and moving events across histories, but they can also defer conflict resolution and continue working on a separate state of the application.

TreeDB generalizes the event log structure of distributed version control systems like Git. It expresses its conflicts using *conflict sets*, a conflict model found in consistent distributed systems with optimistic and multi-version concurrency control [2, 9, 16]. The system maintains a key property of eventual consistency: regardless of the order in which the events are received at different nodes, all nodes will eventually agree on the placement of events in the tree. We discuss this property in more detail in Section 4.3.

TreeDB's events are sufficiently general that they can be used to implement other approaches to conflict detection and resolution—for example, commutative operations can be used to lower the frequency of conflicts, even to the point that the system *always* converges to a single history. In Section 5, we show how simple CRDTs can be simulated in TreeDB. TreeDB can also be seen as an adaptation of Git's commit graph to non-textual structured data; we also discuss this relationship in Section 5.

It remains to be seen if forking histories can be made ergonomic enough for end-users to understand in the context of a real application. We suspect that careful user interface design can enable users to reason about forking histories, and that TreeDB—or perhaps a similar system—provides critical properties for local-first digital gardens.

2 Sync challenges in digital gardens

Consider an application like Apple Photos or Adobe Lightroom that helps a dedicated photographer manage and organize their photo library. This application has several characteristics that distinguish it from other kinds of software.

Accretion over time: A photo library manages data that *grows over time*. Several of the authors manage photos libraries that span over multiple decades, and the value of decade-old photos is no less than that of photos taken last week. In contrast, data from old projects is usually not of critical importance in a project tracker: the current state of the task list is really what matters.

Unbounded workflows: Photo managers also handle workflows that are *unbounded in scope*. Once a user finishes writing a paper, the feedback they received from editors during the writing process becomes irrelevant, because the writing process ends at some discrete moment. In contrast, the scope of what a user might want to do with their photos expands indefinitely with time: perhaps a user might edit their photos into a yearly calendar and a once-a-decade photo book to share with friends and family. A photo library is a living digital artifact that the user works with intermittently and in diverse ways.

Medium Data: The data in a photo manager is too voluminous to manage carelessly but still small enough to fit on a modern mobile device. We might call this *Medium Data*, in contrast to Big Data. With Medium Data, a user cannot exhaustively check that the data is not corrupted or damaged, since going through tens of thousands of records could take weeks. In contrast, manually checking that a project tracker is in a valid state could be done in a few minutes in almost all cases.

Valuable data: Lastly, we note that photo libraries manage *immensely valuable* data. Much of the data is irreplaceable and a user must trust that the application won't accidentally lose it. Furthermore, even the metadata of a photo library might represent hundreds or thousands of hours of

creative effort. A photo library manager carries a heavy burden of responsibility to the user on matters of data integrity.

We believe that these four properties—accretion over time, unbounded scope, medium size, and immense value—characterize an important class of applications that we call *digital gardens*.[1] A user interacts with their digital garden repeatedly and intermittently over a long period of time, as one would plant, weed, and trim a garden. We think that many serious uses of library managers (for photos, music, or other artifacts) are central examples of digital gardens. Some people also express serious creative effort as they organize their files in a traditional files-and-folders filesystem browser, so we think that some uses of a filesystem would similarly qualify.

2.1 Example scenario

In this section, we illustrate some of the challenges of syncing data for a digital garden. Alice and Bob are using an application that allows users to organize and edit a shared photo collection. It uses a local-first architecture that allows offline editing and syncs data when a network is available. A photo has two visual attributes: a saturation and contrast value. Additionally, users can create albums that contain references to photos; a photo can exist in multiple albums at once.

```
Photo { cont: int, sat: int }
Album { name: string, photos: Set<Photo> }
```

Alice and Bob are traveling home from a vacation. Before heading home, they synchronized all of their photos between their laptops. Now they can each spend their train ride home editing photos offline. Alice creates an album containing several dozen photos. She makes a bulk edit to all the photos in her album, reducing the contrast on all the photos to 70% to create a faded look. Meanwhile, Bob is performing his own edits. He creates his own album of several dozen photos, and then applies a bulk edit to all photos in the album, raising the saturation to 130% to make the colors more vibrant.

Later, when Alice and Bob regain internet access, they synchronize their changes. It turns out that Alice and Bob chose some of the same photos to be included in both of their albums, which means that some of their actions affected an overlapping set of photos. How should these edits be merged? We present three options, shown in Figure 1.

Independent style properties. One solution is to minimize conflicts. We could model saturation and contrast for a given photo as two independent Last-Writer-Wins Registers [13]. In this case, because Alice and Bob edited different properties, their writes do not even touch the same register and are trivially mergeable. Using this data model, the application can merge the two users' changes without conflicts, and the users can view and edit their photos after merging.

However, a week later, Alice is showing a slideshow of her album to some friends and notices that some of the photos look strange. The problem is that Alice and Bob's edits to saturation and contrast were *both* applied to those photos, resulting in a visual result that neither user wanted. Alice hadn't noticed this problem earlier upon merging, since there were too many photos to manually review.

This sync system satisfied a useful property: after synchronizing, both users were able to continue editing, without needing to do any manual conflict resolution. However, the system failed to keep the data in a state that the users find acceptable. Neither user wanted to modify saturation or contrast in isolation; their judgement about the quality of the result depended on the value of the other parameters.

Style as a single value. Another option could be to treat the visual style of a photo as an atomic value. For example, we could use an LWW-Register CRDT to hold a value containing both the saturation and the contrast. This model does not allow concurrent edits to contrast and saturation to be merged together; instead, the system converges to contain either Alice or Bob's preferred overall style for a photo, arbitrarily choosing one of their edits using a totally ordered property like a physical timestamp.

We now avoid merging visual edits to the same photo, but there is still a problem. Imagine that the photo app has synchronized using this approach. A week after syncing with Alice, Bob is looking through his album and notices that some photos look very different from the others, ruining the visual uniformity of his album. The reason is that some of his edits have been overridden by Alice's. This system did a better job preserving intent at the level of the individual photo, but it failed to preserve Bob's higher-level intent: to *edit in bulk* all of the photos in an album to look the same way. Bulk edits are common in digital gardens, and so it is often important to reason about intent over groups of records rather than individual records in isolation.

Bulk edits. To solve the above problem, we can consider further coarsening the granularity of our conflict detection. We can model editing all photos in an album as a "bulk action" that must atomically succeed or fail, to ensure a consistent visual look throughout the album. For example, this idea can be naturally implemented in a system like Bayou [14] or IceCube [7], in which developers can describe domain events that execute transactionally, with application-specific logic for conflict detection and resolution. In this new model, Alice and Bob's visual edits are incompatible; the system arbitrarily picks Alice's write as the winner and discards the entirety of Bob's bulk edit action as a conflict.

This approach avoids the data integrity problems with the earlier solutions, but it creates a subpar user experience for

[1] The term "digital gardens" has been used informally on the Web to refer to a variety of applications. One history of the term [1] notes that some people have used the term "digital garden" to refer to photo albums and private folder collections that people organize over time, and we adopt that meaning here. There have also been other usages that refer primarily to public collections of notes resembling a blog.

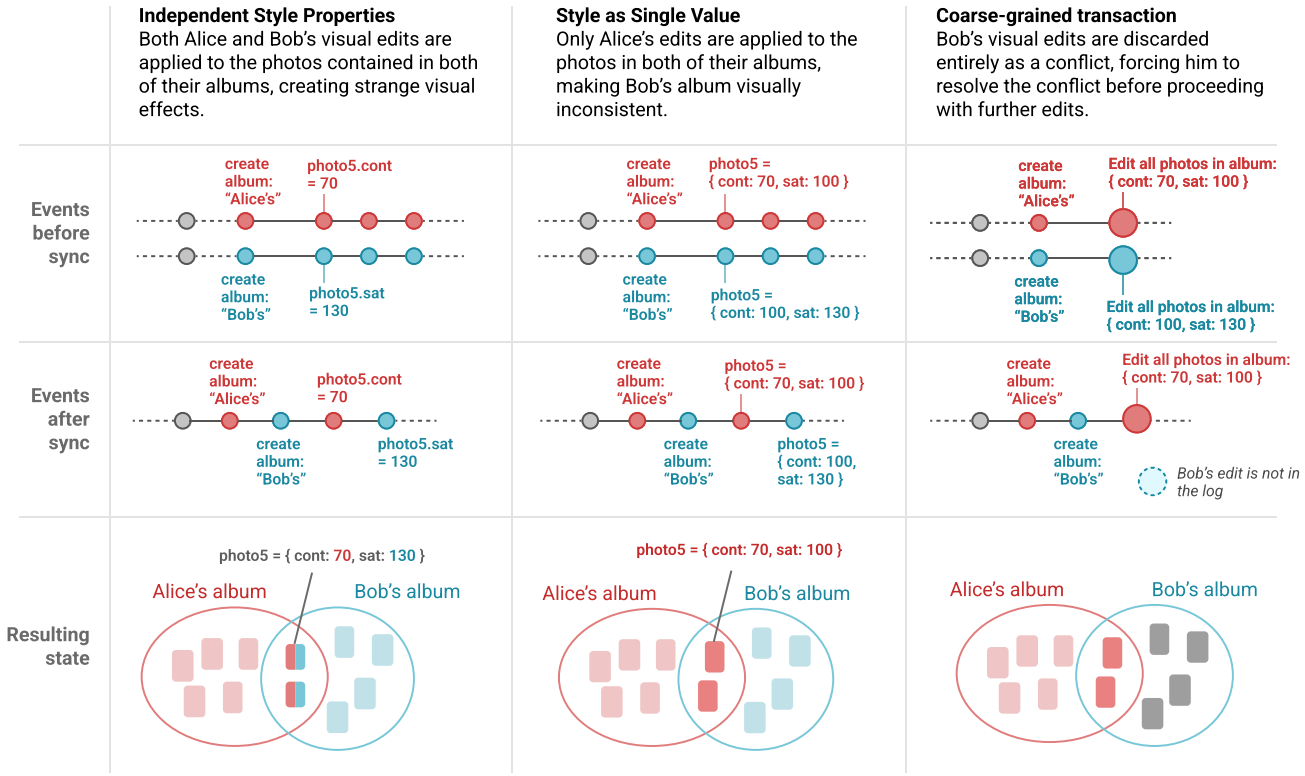

Figure 1. An overview of different conflict resolution options in the example scenario.

Bob. Bob had planned to continue his editing work after he gets home. After synchronizing, he sees his careful editing work disappear from the UI because his entire bulk edit was discarded as a conflict. Of course, his edit actions may not be permanently lost—in this case the photo app allows him to recover them in a history view—but in order to do any work that builds on his changes, Bob must *first* do the manual conflict resolution work needed to bring his changes back into the application state. This coarse-grained transaction approach has achieved a stronger level of data integrity, but lost a valuable property from the earlier options: the ability for users to smoothly continue their work after synchronizing changes with another user.

In sum, all three solutions are flawed in different ways. On the one hand, being overly permissive in merging changes results in data integrity problems that are difficult to detect and clean up. On the other hand, being overly conservative blocks users with conflict resolution work and prevents them from proceeding with their work.

3 The forking histories model

Based on the scenario above, we claim that an ideal synchronization model for digital gardens would satisfy these three properties:

Availability. Data can be edited freely offline, e.g. on mobile devices, while disconnected from other replicas.

Data integrity. The application carefully models user intent and avoids merging together changes in ways that might cause undetected problems.

Deferred conflict resolution. After exchanging edits, the application allows users to continue working freely, without first resolving all conflicts.

Achieving these properties with forking histories. These properties are difficult to reconcile, but we propose a model of *forking histories* that achieves all three properties. The key insight is that all of the options presented in Section 2 share the assumption that the application should converge to a single shared state as soon as the users synchronize their changes. This assumption makes sense in some contexts like withdrawing from a bank balance or booking a meeting room, where it is useful to converge on a single state as quickly as possible. However, this assumption does not necessarily apply in a photo editing app. Alice and Bob would like to *eventually* converge, but they would rather choose when to do so, rather than be forced to converge immediately upon syncing. The idea is to "merge what you can, fork what you can't."

Merge what you can: When user changes do not conflict, they should be automatically merged into a single shared state. The system should allow app developers to associate metadata with write events in a way that maximally captures user intent and relevant integrity constraints, so that automatic merges do not cause surprising results.

Fork what you can't: When user changes do conflict with one another in a way that the system cannot automatically resolve, the application state *forks* into multiple co-existing histories. The application UI displays the multiple histories across all users and devices, so everyone is aware that the fork has occurred. The system can still arbitrarily designate one history as the default across all replicas, but this is merely a lightweight tag.

Users can switch to other histories and see those other states reflected in the application; they can even do further work in the application, extending that non-default history. Users can see diff views between histories and do manual work to reconcile events between them, and can manually designate any history as the new default view shown to all users. The physical action of sharing events between replicas has been decoupled from the logical action of merging those events into a common history. Instead, we have independent notions of event sharing: another user's events can be *available* without having been *applied*.

The forking histories model moves constraint checking from the write path to the read path. In a traditional, centralized transaction processor, constraints are maintained on write. This is impossible in a system with availability during network partitions. CRDTs avoid the issue by weakening integrity constraints, and Bayou/IceCube include conflict resolution logic to maintain constraints during late-arriving writes. In contrast, in a forking histories system, any write is allowed, but might only be visible from histories where it did not cause a conflict.

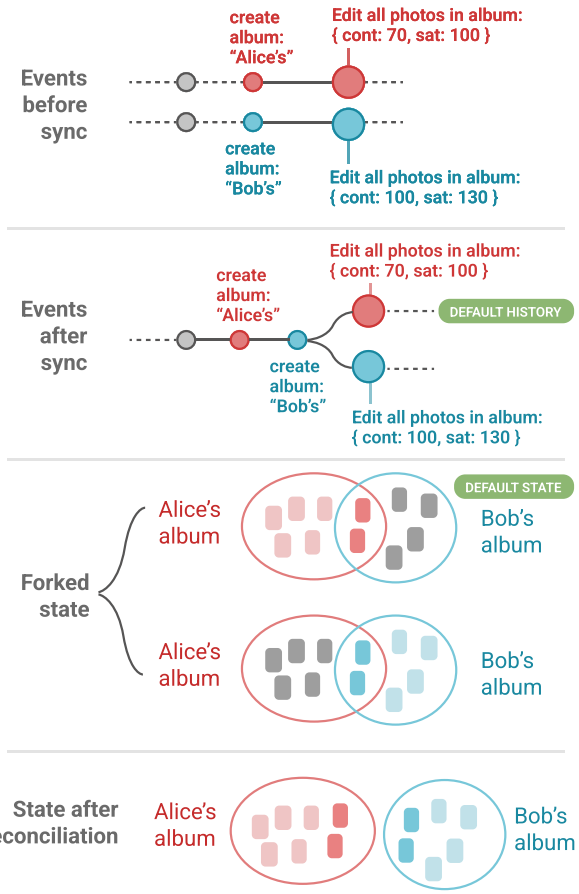

Figure 2. After syncing, the application state is forked into two histories, one where each user's change was applied. The history with Alice's change is tagged as the default. After the users manually reconcile, the albums no longer overlap.

3.1 Forking histories in our example

How would a system based on the forking history model handle the example scenario in Section 2.1? The system creates both Alice and Bob's albums in a single non-forked history, since album creation events do not conflict. However, the bulk edits of the photos contend on shared state, because the application developer has defined all edits of photo appearance metadata as conflicting. Since it can't merge them, it forks starting from the last shared state and create two histories: one with Alice's edits, and the other with Bob's, illustrated in Figure 2. It arbitrarily chooses the history with Alice's edits applied as the default.

After syncing, Bob sees these histories in his UI and decides that he does not want to deal with merging them quite yet. He clicks a toggle in the application to switch back to the one which contains only his edits, and not Alice's. He continues to make further edits, building on his prior work.

Later on, Alice and Bob decide that they are ready to incorporate Bob's changes back into the main history, and they coordinate on a solution which Bob can execute on his history. First, he performs an undo on his bulk edit action. Next, he selects the photos in his album which overlap with Alice's photos and replaces them with duplicated copies. Then, he re-executes his bulk edit action on the album with its new contents. After these adjustments, Bob's history no longer contains any events that conflict with the main history, so he can safely merge it into the main history.

As we can see, Alice and Bob still eventually converged to a single shared state. However, they were able to decide when to merge histories, which enabled them to maintain data consistency without being blocked from continuing their work.

3.2 Design considerations

The forking histories paradigm presents a tradeoff for users. It imposes a cost by requiring users to reason about multiple states of the application simultaneously. In exchange, users gain significant benefits: they can be confident in the integrity of their data, while always being able to access and edit their data offline. In a situation where data integrity isn't particularly important or where incorrect merges can be easily corrected, the cost may outweigh the benefit. However, in the context of digital gardens we believe that the tradeoff is clearly worthwhile; it is better to give users an accurate view that acknowledges the challenges of maintaining data consistency than to pretend that data can always be automatically merged without issues.

Programmers may be familiar with the benefits of forking history from using distributed version control systems (DVCSs) like Git and Mercurial, which are indeed instances of the forking histories model for data that can be represented as trees of text files. In fact, our proposed TreeDB can be seen as a generalization of Git's commit graph using more general mechanisms for detecting and automatically reconciling conflicts, especially outside of the domain of plain text; we elaborate on this relationship in Section 5.

However, DVCSs are also notoriously difficult to learn [4, 5, 11], which raises another question: is it really possible for typical end-users to use a system with forking histories? While outside the scope of this paper, we think this presents an opportunity for HCI and design work—there are many possible interfaces for reasoning about diverging versions of the same data. Some of these interfaces are already deployed in industry: for example, Track Changes in Microsoft Word and Suggested Edits in Google Docs are UIs for reasoning about different versions of a text document, and the Figma collaborative drawing tool has a user interface for reconciling merge conflicts which shows live previews of conflicting UI elements.

4 Our proposal: TreeDB

To show why we think it is possible to create a sync system using forking histories on the kinds of structured data that show up in digital gardens, we propose *TreeDB* as a concrete example of such a system. We believe that TreeDB can be implemented efficiently and that it has several interesting properties, such as supporting many common CRDTs as a complement to the forking history model.

4.1 Events, versions, and conflict sets

All writes to TreeDB are done within the context of an *event*. Each event consists of a set of write operations to be performed. All writes within an event are performed *transactionally*: that is, they are performed at the same time and all together or not at all. Writes within a single event are coalesced to their final value.

A *version* is a single, consistent view of the data in TreeDB. For now, think of versions as abstract labels that identify the data on some history at some point in time. There is a partial order on versions: a version v_1 may precede a version v_2, which we write as $v_1 < v_2$. When $v_1 < v_2$, all of the writes that were visible at v_1 are also visible at v_2.

Each event has an associated *sequence version*. The sequence version of an event identifies the latest event that was visible when it was created. As we will see in Section 4.3, the sequence version acts as a logical clock for ensuring causal ordering of events.

In addition to the set of writes, an event keeps track of its *conflict set*. Our conflict set is based on a design that is commonly used in consistent distributed systems, and particularly the key-value store FoundationDB [3, 6, 16]. A conflict set consists of a set of keys, each of which uniquely identifies some data in TreeDB: for example, the keys might be keys in the sense of a relational database (corresponding to rows in a table), or something coarser- or finer-grained. A conflict set consists of two parts:

Read conflict set. A set of pairs of keys and versions. Intuitively, a key in the read conflict set was "looked at" or otherwise depended on during this event; the version identifies the most recent change to the key. Note that every version must precede the sequence version of the event.

Write conflict set. A set of keys. A key in the write conflict set is one that will create conflicts if it was read during a simultaneous event. Generally, any key that should be changed in the process of applying the event should appear in the write conflict set.

Critically, the sequence version and conflict set completely define the causal semantics of the event; that is, the allowable positions for this event relative to others (past, present, and future) in a serialized event log. For example, an event with empty conflict sets can, by definition, never conflict with any other concurrent event. An application developer can adjust the semantics of an event by manipulating the read version and conflict sets; in Section 5 we give some examples of crafting conflict sets to simulate simple CRDTs.

A set of events is called an *event set*. The primary mechanism for distributed synchronization in TreeDB is sharing events between event sets on different nodes.

4.2 Conflicts and serialization

Creating an event is equivalent to a transaction commit; there is more to be done before it can be considered a "write".

An *augmented event* is a pair of an event and a *write version*, which is also a version. As of the write version, all writes from the event are applied: that is, if our event has a sequence version s and a write version $w > s$ then any read at a version $v > w$ will see the write from our event. Conversely, no version $v < w$ will see the writes from our event; a version for which neither $v < w$ nor $v > w$ may or may not see the event. The write versions are assigned by a component

called the *sequencer* discussed in Section 4.3, which runs independently (but deterministically) at every node.

With the notion of a write version, we can define a *conflict*. Consider a set of augmented events $E = \{e_1, e_2, \ldots, e_n\}$ on a key k with respective read versions $\{v_1, v_2, \ldots, v_n\}$ and respective write versions $\{w_1, w_2, \ldots, w_n\}$. We have a conflict in this set at k if there exist any pairs of events $e_i \neq e_j$ such that $v_i < w_j < w_i$ and k is in both the read set of e_i and the write set of e_j.

Put less precisely: a conflict exists when another event wrote to a key that an event read before it wrote to it. If there exists a way of assigning write versions to events so that no conflicts exist, then the write versions define a partial order on the set of events. Furthermore, any total order on those events that respects this partial order of write versions is a serializable history of those events.

4.3 Sequencing

Every node in TreeDB runs a program called a sequencer: it is the sequencer's job to arrange the *event set* (which has no further structure) into a rooted *event tree*. Each path from the root of the tree to an event defines a *history*; the tree must be constructed so that it has two key properties.

Causal order. For every history in the tree, and any two events $e_1 \neq e_2$ with respective sequence versions s_1 and s_2 where e_1 appears before e_2 in the history, we must not have $s_2 < s_1$; that is, either $s_1 < s_2$ or s_1 and s_2 are incomparable.

History serializability. For every history, there are no conflicts among the events in the history.

Put simply, the event tree arranges the events so that they respect causal order and have no conflicts.

Notice that the sequencer has considerable latitude in how it decides to organize the events in the event tree. We do not even require that each event appear only once in the event tree, although obviously each event must appear only once in each history. While the precise details of the sequencing process are an implementation detail, these two properties ensure that the causal semantics of the events are respected. In practice, it will be useful to use this latitude to produce semantics that are minimally surprising to the user (or at least well-communicated) and efficient to compute.

We impose one key requirement on the sequencer: the sequencer's output must be a deterministic function of the event set. If we think of a sequencer as an incremental computation running at every node, this means that the sequencer must produce the same event tree regardless of the order in which the events arrived at that node. This property is critical to the eventual consistency of TreeDB. As long as all events eventually arrive at all nodes, all nodes will eventually converge on the same event tree. In some sense, TreeDB can be viewed as a sort of "meta-CRDT", since it is a CRDT on the *tree of histories* rather than the domain data. This meta-CRDT property also ensures that meta-actions taken by users on the tree (e.g., merging changes across histories) will also result in a converging tree across all nodes.

4.4 Materializing

Lastly, we close the loop by connecting the event tree to the versioned read system. Each history is a path through the tree from the root to an event e. Replaying those events defines a view of the data as of e's write version w.

Suppose that we try to read a key k from that view of the data at a version $v > w$. The read conflict imposed by that key is the pair (k, v_k), where v_k is the write version of the most recent ancestor event e_k that wrote to k. That is, we conflict with any writes that might have changed the value of k after this event reads it.

From the perspective of a developer writing an application with TreeDB, there is a single consistent view of the data "at" the sequence version of the event. Any writes in an event that read at that version is played on top of data that is identical to the data at that version.

The component that is responsible for presenting this consistent view is called the *materializer*: its job is to consume a history in the event tree and materialize a consistent view of the keys. In practice, we expect that a materializer will work incrementally, applying diffs based on events to quickly materialize an appropriate version.

5 Simulating existing systems

Common CRDTs. Some apps do not need strong integrity semantics, or might need them only very selectively. We think TreeDB is a general enough framework that it can subsume these other use cases. For example, common CRDTs can be implemented in TreeDB's event framework by carefully defining the conflict sets. If all write events to a key k are "blind"—that is, have an empty read conflict set and a write conflict set of $\{k\}$, then those writes will never conflict. The sequencer has the latitude to determine a "last writer" deterministically from the event set, and this last writer necessarily respects the sequence versions and therefore respects causal order within the history. Thus, the value of k is a last-writer-wins (LWW) register [12].

We can similarly implement an OR-Set CRDT where adds win over removes [12], albeit only by assuming particular behavior in the sequencer. The event add(x) has an empty read conflict set and a write conflict set of $\{x\}$, as in the LWW register. The event remove(x) with sequence version s has a read conflict set of $\{(x, s)\}$ and a write conflict set of $\{x\}$. Suppose that we have a concurrent addition and removal of the same value x. Because the key x is in both the read conflict set of remove(x) and the write conflict set of add(x), we have a conflict in any history where the addition precedes the removal. Therefore, the only event trees that respect both causal order and non-conflicting histories are (1) a fork into two histories, one for the add event and one for the remove

event, and (2) a single history with the removal preceding the addition.

If we assume that the sequencer avoids unnecessary forks, then it chooses the second option, and TreeDB therefore simulates the behaviour of an OR-Set. We believe that TreeDB can simulate other CRDTs as well, although we have not yet determined the precise power of TreeDB's event model.

We believe that building a CRDT on top of TreeDB's event system provides the application developer and possibly the end-user a remarkable amount of flexibility in defining conflict semantics. For example, a particular data structure might generally support conflict-free real-time collaboration using a CRDT but also allow the user to enter a "strict mode" in which normally non-conflicting operations conflict, in cases where they are performing particularly delicate changes.

Distributed version control systems. As mentioned in Section 3.2, distributed version control systems (DVCSs) are prominent examples of the forking history model. In fact, TreeDB can be seen as an implementation of a DVCS commit graph with different logic for detecting and (when possible) automatically resolving conflicts. Here, we use Git to lay out this relationship concretely.

Git organizes changes in a *commit graph*, which encodes the data and defines the forking and merging semantics. Each change is represented as a *commit*, which consists of a diff (i.e., a set of lines to add to and remove from each file) and a reference to one or more previous commits, called the parents. Users copy the entire commit graph to their machine and can make new commits on top of any of those other commits, usually by "checking out" a particular commit into their working filesystem. When a user shares these commits with other users, the commits are reconciled if they did not appear to conflict.[2] When the changes do conflict, there is a logical fork in the history: Git retains both commits as peers but identifies a "merge conflict" and prompts the user to reconcile the conflicting edit.

Crucially, a merge conflict is an illusion from the perspective of the underlying commit graph. By the time a merge conflict has been identified, the commits have already been created without conflicting. Both commits are equally valid, and the disagreement is over which one receives the label of the branch.

TreeDB is nearly[3] a direct translation of this commit graph to structured, rather than textual, data. TreeDB events are analogous to Git commits, with structured rather than text-based semantics. Instead of identifying conflicts using diffs and resolving non-conflicting conflicts using patches, TreeDB identifies conflicts using structured conflict sets and resolves them using application-defined business logic. TreeDB and Git are therefore both concrete instantiations of a meta-algorithm for managing event graphs that is parameterized by the conflict detection ("diff") and event application semantics. TreeDB is much better suited to highly structured data where the text-based diff and patch semantics are unlikely to maintain appropriate integrity constraints: for example, it would be quite difficult to encode a relational database into text so that foreign key constraints are preserved by Git's merge and patch. However, one should be able to model text as a type of structured data; this could allow a reimplementation of Git within TreeDB, albeit with diff and merge algorithms that have stronger semantics [10].

Confusingly, the user interface for Git (primarily the command line utility) uses the word "branch" to refer to the floating label, not an actual graph-theoretic branch in the commit graph. This language is likely inspired by earlier version control systems with a different conceptual model, like CVS, where forks happen when a user explicitly creates a branch to avoid conflicts and registers it in a centralized system. We posit that the conceptual mismatch between CVS-style explicit branches and the structure of the commit graph explains some of the difficulties that users have in using Git. In Git, a "branch" (such as `main`) is a floating label that applies to a particular commit: in fact, any commit can serve as a parent for any other commit.

One interesting question is how a user can simulate a Git-style branch, for example to perform some experimental work that they might decide to throw away. In TreeDB, a Git-style branch can be created by a "branch creation event" that performs no writes but has all keys in both its read and write conflict sets: this effectively creates a "serialization point" that bars any simultaneous events on that history.

[2]The precise definition of a conflict in Git is extremely complex and depends on the semantics of Git's diff and patch implementations. We do not discuss them here.

[3]One incidental difference between Git and TreeDB is that the Git commit graph is a directed acyclic graph, while TreeDB's event tree is a tree. In Git, a commit can actually have multiple parents. However, TreeDB uses its sequence versions only for causal order delivery, and so a "merge event" would just have a sequence version that is one higher than the highest sequence version of its parents. If we wanted finer-grained lineage tracking, we could track this metadata out-of-band.

Acknowledgments

We thank Martin Kleppmann and the anonymous reviewers for helpful feedback on this paper. Nicholas Schiefer was supported by a Simons Investigator Award. Geoffrey Litt was supported by an NSF GRFP Fellowship.

References

[1] Maggie Appleton. 2020. A Brief History & Ethos of the Digital Garden. (2020).

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Pub. Co, Reading, Mass.

[3] Christos Chrysafis, Ben Collins, Scott Dugas, Jay Dunkelberger, Moussa Ehsan, Scott Gray, Alec Grieser, Ori Herrnstadt, Kfir Lev-Ari, Tao Lin, Mike McMahon, Nicholas Schiefer, and Alexander Shraer.

2019. FoundationDB Record Layer: A Multi-Tenant Structured Datastore. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam Netherlands, 1787–1802. https://doi.org/10.1145/3299869.3314039

[4] Santiago Perez De Rosso and Daniel Jackson. 2016. Purposes, Concepts, Misfits, and a Redesign of Git. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, Amsterdam Netherlands, 292–310. https://doi.org/10.1145/2983990.2984018

[5] Sukru Eraslan, Julio César Cortés Ríos, Kamilla Kopec-Harding, Suzanne M. Embury, Caroline Jay, Christopher Page, and Robert Haines. 2020. Errors and Poor Practices of Software Engineering Students in Using Git. In *Proceedings of the 4th Conference on Computing Education Practice 2020*. ACM, Durham United Kingdom, 1–4. https://doi.org/10.1145/3372356.3372364

[6] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems* 30, 2 (June 2005), 492–528. https://doi.org/10.1145/1071610.1071615

[7] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. 2001. The IceCube Approach to the Reconciliation of Divergent Replicas. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing - PODC '01*. ACM Press, Newport, Rhode Island, United States, 210–218. https://doi.org/10.1145/383962.384020

[8] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2019*. ACM Press, Athens, Greece, 154–178. https://doi.org/10.1145/3359591.3359737

[9] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[10] Russell O'Connor. 2011. Git Is Inconsistent. (April 2011).

[11] Santiago Perez De Rosso and Daniel Jackson. 2013. What's Wrong with Git?: A Conceptual Design Analysis. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! '13*. ACM Press, Indianapolis, Indiana, USA, 37–52. https://doi.org/10.1145/2509578.2509584

[12] Nuno Preguiça. 2018. Conflict-Free Replicated Data Types: An Overview. *arXiv:1806.10254 [cs]* (June 2018). arXiv:cs/1806.10254

[13] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. Report. Inria – Centre Paris-Rocquencourt ; INRIA.

[14] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *ACM SIGOPS Operating Systems Review* 29, 5 (Dec. 1995), 172–182. https://doi.org/10.1145/224057.224070

[15] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M Levy. Diamond: Automating Data Management and Storage for Wide-area, Reactive Applications. (????), 16.

[16] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event China, 2653–2666. https://doi.org/10.1145/3448016.3457559