

Local-first Collaboration on Relational Data

by

Veronika Sirotkina

Bachelor Thesis in Computer Science

Submission: April 25, 2024

Supervisor: Anton Podkopaev

Statutory Declaration

Family Name, Given/First Name	Sirotkina, Veronika
Matriculation number	30006541
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

.....
Date, Signature

1 Abstract

Nowadays many apps need an internet connection to work properly, especially those that sync data between devices and let people collaborate in real-time. Without internet these apps often become unusable. Local-first software aims to change that by letting users edit shared documents offline and syncing changes when they are back online.

This paper explores the ways of enabling local-first real-time collaboration over abstract relational data in the context of GanttProject, an open-source tool for creating Gantt charts.

Contents

1	Abstract	iii
2	Introduction	1
3	Background	1
3.1	Operational Transformation	1
3.1.1	OT for plain text	1
3.1.2	Google Wave	2
3.2	Local-first Software	3
3.3	CRDT	4
3.3.1	State-based CRDT	4
3.3.2	Operation-based CRDT	4
3.3.3	Automerger	7
3.3.4	CRDT for Relational Data	7
4	Approaches	9
4.1	CRDT approach	9
4.2	Time Warp approach	10
4.3	Transaction replay approach	11
4.3.1	Postgres transaction isolation levels	11
4.4	Data Branching	12
5	GanttProject	12
5.1	Colloboque	12
5.2	Data model	14
6	Implementation	16
6.1	jOOQ	16
6.2	Restoring the latest version	16
6.3	Transaction replay	16
6.4	Protocol	17

2 Introduction

With internet becoming more accessible and cloud-based software gaining momentum, more apps started to heavily depend on the presence of internet connection to work properly. This is especially true for apps that need to synchronize clients' data across multiple devices or apps that allow for real-time collaboration between clients.

The most obvious and maybe the easiest way to enable synchronization across devices is naturally to store an authoritative copy of the data in the cloud and provide a centralized server to manage changes coming from the clients. Real-time collaboration can be achieved by implementing operational transformation algorithms [14].

This approach is, of course, viable and is widely used in practice, but it has a big downside. Without internet connection access to the data is either lost completely or it is only available in a read-only mode. The access can also be lost if the provider company's servers are down, or if the company stops supporting the product, or if the company finds the contents of the document inappropriate [3].

Local-first software [10] is a new approach to designing collaborative software. In the context of this paper the most notable feature of local-first software is that it allows clients to edit shared documents even when they are offline. The changes can be integrated with the upstream version once the connectivity is reestablished.

GanttProject [6, 5] is an open-source tool for building Gantt diagrams. It's originally local-only, but at the current time it also provides a cloud storage [4]. A work-in-progress module called Colloboque will enable real-time collaboration in GanttProject.

The goal of this project is to enable local-first real-time collaboration in GanttProject by further developing Colloboque.

3 Background

3.1 Operational Transformation

Operational Transformation [14] or OT is a technology for managing concurrent updates, particularly in collaboration software. OT algorithms assume that there are multiple clients. All clients have a replica of the same document and they can modify the document with a predetermined set of operations independently from each other.

When a client receives an update from another client, he doesn't apply the received operation immediately. He compares the incoming change against other observed concurrent changes, transforms the operation and then applies it. The transformation algorithm must ensure that the states of all clients converge.

3.1.1 OT for plain text

In this section I will present an example of how OT can be used for managing concurrent insertions in plain text documents.

There are two clients, Alice and Bob, and each of them has a copy of a document. The document contains a single sentence: "Hello World". The only defined operation is an insertion of a single character at a specified position: `insert(char, pos)`.

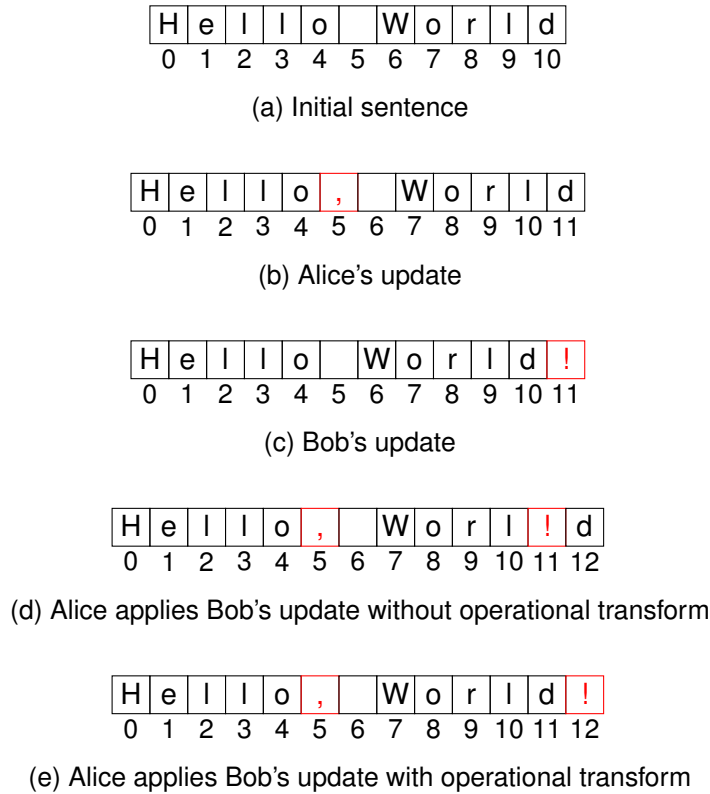


Figure 1: Demonstrating Operational Transformation on plain text

Alice inserts a comma after "Hello" with `insert(',', 5)` and gets "Hello, World", while Bob inserts an exclamation mark after "World" with `insert('!', 11)` and gets "Hello World!". After that they exchange the performed operations.

If Alice and Bob both apply each other's changes as is, Alice will get "Hello, Wor!ld", and Bob will get "Hello, World!" - the two documents now diverge. Instead, when Alice gets `insert('!', 11)` she compares it with the operation she performed concurrently `insert(',', 5)`, sees that the indexing has shifted and transforms `insert('!', 11)` into `insert('!', 12)`. After applying this transformed operation she will get "Hello, World!".

Bob performs similar actions to learn that he doesn't need to transform Alice's operation, applies it as is and also gets "Hello, World!".

3.1.2 Google Wave

Google Wave [7, 8, 15] was a communication platform developed by Google using Operational Transformation approach. It was launched in September 2009. Google Wave combined elements of email, instant messaging, wikis, and social networks into a single platform.

Google Wave had features such as real-time collaboration, allowing users to work together on documents known as "waves" by typing messages, inserting images, and adding other content. It supported the embedding of media objects like videos, maps, and polls directly into waves.

Despite its innovative features, Google Wave failed to gain popularity. Google discontinued Google Wave in August 2010 due to low user adoption, but its technology and concepts influenced subsequent Google products like Google Docs and Google Drive.

3.2 Local-first Software

Local-first software was first formalized in 2019. The original paper [10] suggests 7 principles of local-first software. Here I will list 4 which I find the most relevant.

- **Synchronization across devices**

Imagine a client who uses a document editor on multiple devices. Synchronization involves tracking changes made by the client on one device and ensuring these changes are reflected across all other devices used by the client. In the end data on all devices must reach the same state. This is a very convenient feature that lets clients access their work from any device.

- **The network is optional**

Nowadays it is normal for many apps to lose most of their functionality if internet connection is unstable. For example, take an online LaTeX editor called Overleaf. In Overleaf if the connection is lost, the page waits for a few seconds before it tries to connect again. In those few seconds the user is still allowed to modify the text fields, but when connection is re-established, the user is met with an immediate error, notifying them that the project went out of sync. Overleaf makes no effort to integrate the changes made by the user while they were offline. This leads to a very frustrating experience where local changes are either discarded or the user has to manually copy them from a special pop-up.

For local-first software it is important that it should retain its core functionality even when the device is offline. The client is still able to view and edit documents as he pleases, and the changes made while being offline are integrated with other replicas when the device connects to the network again.

- **Seamless collaboration**

Real-time collaboration is a very attractive feature that lets multiple people work on a single document simultaneously. Some notable examples of web-apps that allow for real-time collaboration are Google Docs, Google Sheets, Figma, etc. Usually such apps don't allow to edit documents offline with some Google products like Google Docs being an exception. For local-first software the aim is to provide collaboration functionality on par with cloud-based apps like Figma while retaining optionality of the network.

- **Ultimate ownership and control**

As a popular saying goes, there is no cloud - it's just someone else's computer. When user data is stored in the cloud, the ultimate ownership of the data belongs to the corporation that provides the software. The user might lose access to their data because of technical problems, or, what's more concerning, the company might bar the user from accessing their data on a whim. For example, by introducing a paid subscription or banning the user because their data presumably violates the service's policies [3]. With local-first software, the data physically belongs to the user and the service provider can't take the user's access.

3.3 CRDT

CRDT stands for Conflict-Free Replicated Data Type [13]. It's a type of data structure designed for distributed systems. Objects of CRDTs can be replicated across multiple nodes, and each node can independently update the object. When information about the updates or, depending on the flavour of a particular CRDT specification, the state of each node is shared, CRDTs guarantee that all replicas will eventually reach the same state.

CRDT object has an identifier, physical content and an initial state. Any two objects with the same identifiers are called replicas of each other. CRDT specification defines a set of update operations, as well as read-only queries.

A CRDT can implement any logical type like Counter, Set, List, etc. The state of a CRDT's logical type is represented by its abstract state. For example, a Counter CRDT can be implemented as a tuple of values, where each replica can only change its designated value. The abstract state of such CRDT is the sum of all values in a tuple. You can see an example implementation of such counter on Figure 2

CRDT guarantees that all well-formed update operations are commutative and idempotent with respect to the abstract state of an object. Because of these properties, replicas can make updates independently, without communicating with other replicas. As long as all updates are eventually communicated, all replicas are guaranteed to end up with the same abstract state.

CRDT types can be further categorized into two groups: state-based CRDTs and operation-based CRDTs. It is worth noting that state-based CRDTs can always be emulated by operation-based CRDTs and operation-based CRDTs can always be emulated by state-based CRDTs. But for some structures it's simpler to think in terms of state-based CRDT, while some feel more natural as an operation-based CRDT.

3.3.1 State-based CRDT

For state-based CRDT one has to specify a merge function, which will merge together states of two different replicas. Requirements for the merge function can be formulated in terms of causal history.

Definition 3.1 (Causal History — state-based). For any replica x_i of x :

- Initially, $C(x_i) = \emptyset$.
- After executing update operation f , $C(f(x_i)) = C(x_i) \cup \{f\}$.
- After executing merge against states x_i, x_j , $C(merge(x_i, x_j)) = C(x_i) \cup C(x_j)$.

The example implementation of a Counter CRDT on Figure 2 is state-based.

3.3.2 Operation-based CRDT

Operation-based CRDTs are not required to have a merge function, but their updates must consist of two phases. The first phase called `atSource` is executed completely at a local replica and produces no side-effects. The second phase called `downstream` is executed locally immediately after the `atSource` phase. After that the `downstream` update is propagated to other replicas and is executed asynchronously by them, changing the downstream state.


```

1  import java.lang.Integer.max
2
3  const val MAX_REPLICAS = 3
4
5  class StateCounter(
6      // One of {0, 1, ..., MAX_REPLICAS - 1}
7      // The problem of assigning a unique id
8      // is not considered in the context of this example
9      private val id: Int = 0
10 ) {
11     // Initial state
12     private val counter = MutableList(MAX_REPLICAS) { 0 }
13
14     // Update operation
15     fun increment() {
16         counter[id] += 1
17     }
18
19     // Query
20     fun value() = counter.sum()
21
22     // Merge function
23     fun merge(other: StateCounter) {
24         for (i in 0 until MAX_REPLICAS) {
25             counter[i] = max(counter[i], other.counter[i])
26         }
27     }
28 }

```

Figure 2: Example implementation of a state-based Counter CRDT in Kotlin

```

1  import kotlin.random.Random
2
3  typealias OperationId = Long
4
5  class OperationCounter {
6
7      // Initial state
8      private val increments = mutableSetOf<OperationId>()
9
10     // Update operation
11     fun increment() {
12         val operationId = incrementAtSource()
13         incrementDownstream(operationId)
14     }
15
16     // atSource phase of the increment operation
17     private fun incrementAtSource(): OperationId {
18         return Random.nextLong()
19     }
20
21     // downstream phase of the increment operation
22     fun incrementDownstream(operationId: OperationId) {
23         increments.add(operationId)
24     }
25
26     // Query
27     fun value() = increments.size
28 }

```

Figure 3: Example implementation of a state-based Counter CRDT in Kotlin

This specification can be interpreted in the following way: the `atSource` phase generates an update that follows the internal protocol of the respective CRDT, while the `downstream` phase tries to apply the update to the object.

Figure 3 demonstrates an example implementation of an operation-based Counter CRDT.

Causal history can also be defined for operation-based CRDTs.

Definition 3.2 (Causal History — operation-based). For any replica x_i of x :

- Initially, $C(x_i) = \emptyset$.
- After executing the downstream phase of operation f at replica x_i , $C(f(x_i)) = C(x_i) \cup \{f\}$.

TODO: ELABORATE ON THE USAGE OF DEFINITIONS AND DEFINE CONVERGENCE OR DROP THEM COMPLETELY

3.3.3 Automerge

Automerge [1] is a library that provides a JSON-like CRDT for JavaScript. It was designed specifically for implementing local-first software. Automerge documents support all JSON primitive data types, as well as Map, List, Text, and Counter.

Automerge documents support both merging full documents and applying individual changes. To define rules for merging documents, it's enough to define rules for merging Maps, Lists, Text and Counters. Full explanation of the merging rules in Automerge can be found in Automerge documentation [11].

Some conflicts can't be automatically resolved, for example if two replicas concurrently set the same key. In events like this Automerge produces a Conflict object, which contains updates from both replicas. They can be used to allow the user to pick the value manually, but this is not mandatory. Conflict objects can be completely ignored, in which case a single value is picked automatically from the list of available options presented by the Conflict object. The library guarantees that if two replicas encounter the same Conflict, it will be resolved in a consistent way, ensuring that the two replicas still converge.

The core of Automerge is written in Rust. Consider an example using Automerge's Rust API demonstrating Conflict objects on Figure 4

3.3.4 CRDT for Relational Data

Some work on designing CRDT for relational data have been done before [16]. In the paper the authors present a two layered architecture. The first layer is the Application Relation layer, or AR layer. The application can interact with the application layer as if it was a normal database. The second layer, called Conflict-free Replicated Relation layer or CRR layer, handles the logic of applying and propagating updates. The CRR layer contains an AR layer schema, augmented with additional data to implement CRDT logic.

Read-only queries can be executed directly on the application layer. Update requests are applied atomically to both AR and CRR schemas. Later they are propagated to other replicas.

The CRR layer is responsible for handling conflicts between local updates and concurrent updates propagated by other replicas. To achieve that the CRR layer stores rows in a special Set CRDT called CLSet, and all row attributes are stored in a Last-Writer-Wins register CRDT. Some numeric attributes are represented by Counter CRDT.

CLSet maintains a counter for each object in the set. This number is called the *causal length* of an object. Successful inserts and removals increment causal length, while unsuccessful operations have no effect. For example, removing an object that doesn't belong to the set has no effect, while removing an object that does belong to the set increments the object's causal length. Following this specification, we can conclude that an object belongs to the set if and only if its causal length is odd. When handling conflicting updates, an update with the greatest causal length wins.

Another problem that the CRR layer needs to solve is maintaining integrity constraints. If some constraints are violated when applying a local update the update can be simply rolled back. The main challenge comes from integrity constraint violations happening when merging updates from other replicas. In this case the CRR layer generates a new update that undoes the conflicting update and propagates it to other replicas.

```

1 use automerge::ReadDoc;
2 use automerge::transaction::Transactable;
3
4 fn main() {
5     // Create an automerge document and its replica
6     let mut first_doc = automerge::Automerge::new();
7     let mut second_doc = first_doc.fork();
8
9     // Set property "key" to "first value" in the original document
10    first_doc.transact::<_, _, automerge::AutomergeError>(|doc| {
11        Ok(doc.put(automerge::ROOT, "key", "first value").unwrap())
12    }).unwrap();
13    // Set property "key" to "second value" in the replica
14    second_doc.transact::<_, _, automerge::AutomergeError>(|doc| {
15        Ok(doc.put(automerge::ROOT, "key", "second value").unwrap())
16    }).unwrap();
17
18    // Replicas merge each other's changes
19    first_doc.merge(&mut second_doc).unwrap();
20    second_doc.merge(&mut first_doc).unwrap();
21
22    // Get the conflict object
23    let conflict1: Vec<_> = first_doc
24        .get_all(automerge::ROOT, "key").unwrap();
25    // Get the resolved conflict value
26    let value1 = first_doc
27        .get(automerge::ROOT, "key").unwrap().unwrap().0;
28
29    // Get the replica's conflict object
30    let conflict2: Vec<_> = second_doc
31        .get_all(automerge::ROOT, "key").unwrap();
32    // Get the resolved conflict value
33    let value2 = second_doc
34        .get(automerge::ROOT, "key").unwrap().unwrap().0;
35
36    println!("{:?}", conflict1);
37    // [(Scalar(Str("first value")), Id(1, ActorID("aaa"), 0)),
    ↪ (Scalar(Str("second value")), Id(1, ActorID("bbb"), 1))]
38    println!("{:?}", conflict2);
39    // [(Scalar(Str("first value")), Id(1, ActorID("aaa"), 1)),
    ↪ (Scalar(Str("second value")), Id(1, ActorID("bbb"), 0))]
40    println!("{:?}", value1); // Scalar(Str("second value"))
41    println!("{:?}", value2); // Scalar(Str("second value"))
42 }

```

Figure 4: Conflict objects in Automerge/Rust

The paper discusses three kinds of integrity constraints violations.

The first one is uniqueness constraint violation. When two updates conflict because of a uniqueness constraint violation, update with a higher timestamp is reverted.

The second one is referential integrity constraint violation. It can happen when one replica updates or inserts a row with a foreign key f , while another replica deletes the row containing f from the referenced table. In that case the update/insert operation is reverted, because the delete operation might have been the result of undoing an insert after uniqueness constraint violation. In this case undoing the delete operation will lead to encountering a uniqueness constraint violation again.

The third one is numeric constraint violation like lower and upper bounds. Since some numeric attributes behave like counters with additive updates, it's possible to get overflow or underflow after merging concurrent updates. In this case the more recent updates are reverted.

The approach with issuing undo updates is not fit for all cases. For example some operations might have further side-effects, like sending an email or charging a bank account. An undo operation can't revert the impact made on the external world. A paper on CRDT [13] suggests that this problem could be solved by introducing quotas for certain operation. For example, in case of a counter supports both increments and decrements, but mustn't go below zero, let's impose the following restriction: a client can't issue more decrements than the number of increments he personally issued. If a device wants to decrement more, it can contact other clients to redistribute the allowed decrements. Still, this restriction is overly strong. The other solution to maintain the global invariant is to synchronize, but that defeats the purpose of CRDT.

4 Approaches

In this section I will describe approaches considered during the course of my work.

4.1 CRDT approach

The first approach to enabling local-first collaboration in GanttProject is to implement a CRDT for relational data. With a robust relational data CRDT, the task would be very straightforward. Each client can store a replica of a CRDT object and propagate its updates to other clients either through a centralized server or a peer-to-peer connection.

The main challenge comes from implementing a relational data CRDT. The paper referenced in Section 2.3.4 gives a general direction on how such a CRDT can be implemented, but this is still no easy task. It's also unclear whether additional problems could arise due to the limited support of SQL features that the described CRDT provides. For example, it seems impossible to implement an auto-increment attribute. When there is a need to generate unique ids for newly inserted rows the authors of the paper generate random ids instead. There are probably many more such examples, but listing them requires further analysis.

The described relational data CRDT is implemented in a library for Elixir called Crecto. It can't be used easily since GanttProject is implemented in Java and Kotlin.

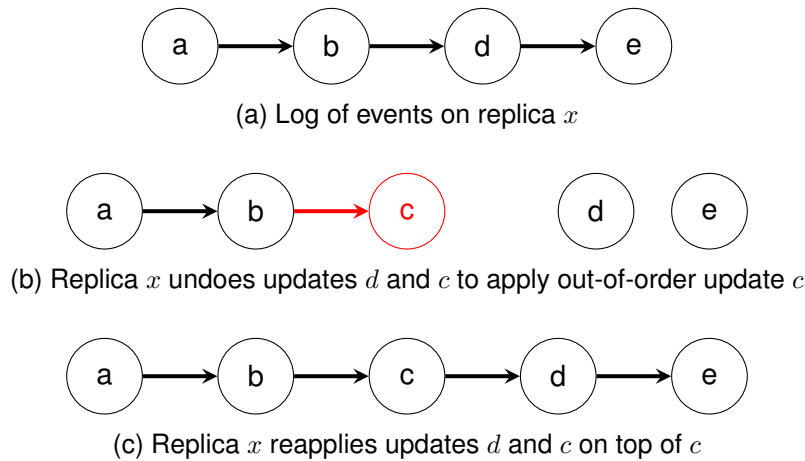


Figure 5: Demonstration of time warp progression

Taking into account the amount of work required to employ this approach and the associated concerns it was decided to think of other approaches which would be easier to implement and integrate with the existing codebase.

4.2 Time Warp approach

In distributed systems, "time warp" refers to a technique used to handle inconsistencies in the ordering of events across multiple nodes. In such systems, events may occur concurrently at different nodes. Due to network partitions and communication delays, the order in which events are observed can be different at different nodes.

Time warp allows a node to rewind its state to apply an out-of-order event. After that the node can also reapply the events that were undone during the rewind stage. See Figure 5 for a demonstration of time warping.

This technique can be used to allow local-first collaboration with the help of a centralized server. Assume that the clients are able to perform time warp. The centralized server receives update messages from clients and writes them to an append-only log. Let clients adhere to the following algorithm when producing updates:

1. Add an update to the local log
2. Notify the server about the update
3. Wait for the server to process the update
4. Recieve response from the server that the update was successfully committed
5. Mark the update as committed

Alternatively if the client went out of sync:

1. Add an update to the local log
2. Notify the server about the update
3. Wait for the server to process the update
4. Recieve response from the server that the update is out of sync. The server also sends the updates that must be applied by this client to sync with the server.

5. Client uses time warp to undo local updates and apply sync updates.
6. If local updates produce no conflicts, they are applied again, otherwise the user can be prompted to resolve the conflicts. The original updates are discarded, and if the user resolves the conflict, new updates are generated.

Notice that even if the update that the server received is out of sync, it can still be committed if it produces no conflicts. When the server sends the client a list of sync updates, it can also inform the client that the out of sync update was committed successfully after the sync updates. Then the client can also apply this local update after applying all the sync updates - there is no need to discard it. TODO: MAYBE UNCLEAR, ADD PICTURE?

To employ this approach, it is crucial that the client is able to revert recent updates. Unfortunately, this functionality doesn't come built-in with most relational databases, including Postgres. To allow time warp on a database, one would have to be able to generate undo operations for all possible transactions. One notable challenge is recovering rows that were deleted as a side-effect of removing a foreign key they were referencing. Another challenge comes from the fact that transactions usually can't be organized into a strictly ordered sequence. In Postgres, multiple transaction isolation levels are supported, and only the strongest level, called `SERIALIZABLE`, has this property. See [Serializable Isolation Level documentation \[12\]](#) for more details.

4.3 Transaction replay approach

The Colloboque server stores a history of committed Postgres database transactions and the state of the project after each transaction. When a client loses connection to the server, he keeps committing transactions locally. When connection is re-established, the set of changes applied by the client is sent to the server. The server then takes the state of the project that the out-of-sync client based its changes on and uses it to start two transactions. The first transaction contains all the changes that the client applied locally, while the second transaction contains all the changes processed by the server while the client was disconnected.

4.3.1 Postgres transaction isolation levels

In the described approach transactions have to make use of isolation level `REPEATABLE READ`, which is stronger than the default transaction isolation level.

The default mode for transactions in Postgres is `READ COMMITTED` [12]. It means that a transaction is allowed to read changes that were committed by other transactions after the current transaction has started.

```
1 BEGIN;
2 SELECT * FROM data; -- 0 rows
3 -- another transaction commits inserting a row
4 SELECT * FROM data; -- 1 row
5 COMMIT;
```

When using `REPEATABLE READ` transaction mode, the changes from newly committed transactions are not visible, but if the current transaction makes a conflicting update, it will lead to a serialization error.

```

1 BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
2 SELECT * FROM data; -- 1 row
3 -- another transaction commits deleting a row
4 SELECT * FROM data; -- 1 row
5 UPDATE data SET col = 1;
6 -- ERROR: could not serialize access due to concurrent delete

```

With this approach the Postgres database can reliably identify whether the changes that the client made offline produce conflicts. If there are conflicts, the server discards the changes. The client will then have to sync with the server. An out-of-sync version of the document can be saved as a copy so that the user can compare the two versions manually.

4.4 Data Branching

Data branching in databases refers to the practice of creating divergent versions of data within a database, often for the purpose of testing, experimentation, or parallel development. This concept is similar to branching in version control systems like Git, but applied to data within a database rather than code in a repository.

Data branching could solve some problems posed by the Time Warp and Transaction replay approaches. In case of Time Warp, there is no need to undo operations if there exists a branch that contains the required version of the project. In case of Transaction replay, there would be no need to store the state of the project after each transaction.

One example of data branching for Postgres is Neon [2]. Unfortunately, the free version of Neon only allows to create 10 branches, which is not enough for the purposes of GanttProject.

TODO: WRITE MORE?

5 GanttProject

GanttProject is an open-source desktop application for creating Gantt charts. A Gantt chart is a type of bar chart that illustrates a project schedule. It's a visual representation of the tasks and activities that need to be completed, along with their corresponding start and end dates. Gantt charts often include dependencies between tasks, showing which tasks must be completed before others can start. These dependencies are typically represented by arrows or lines connecting the bars. Some Gantt charts also include information about resource allocation, showing which resources (such as personnel or equipment) are assigned to each task.

GanttProject is written in Java and Kotlin and uses Postgres for the database. It supports saving projects in the cloud, but they can't be edited simultaneously by multiple people. GanttProject interface is shown on Figure 6

5.1 Colloboque

To enable real-time collaboration in GanttProject, a module called Colloboque is being developed. It is written in Kotlin. Before I started working on Colloboque, it worked as

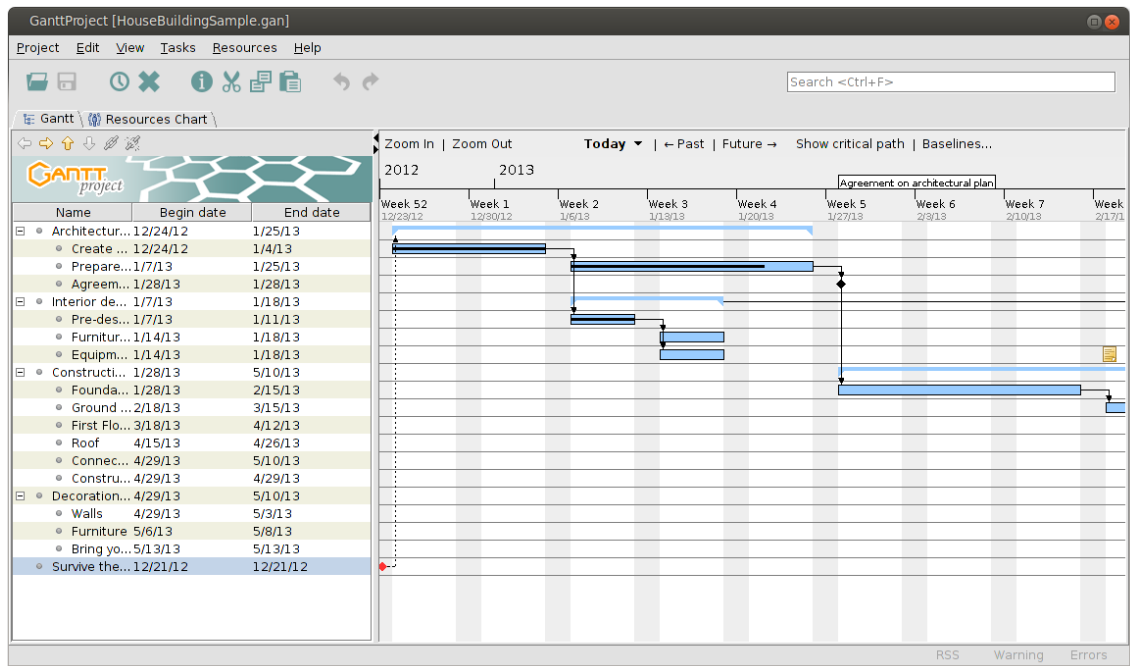


Figure 6: GanttProject interface

follows.

Clients connect to a centralized Colloboque server and the server responds with an XML file describing the current version of the project. It was supposed to be the last version of the project, but the stub implementation responded with the same default project every time. The client recieved and loaded the project XML sent by the server. After that every change made by the user is wrapped into an update message and sent to the server. The client waits for the server to respond with a commit message. Until a response from the server is recieved, new changes are not allowed.

The server recieves messages from clients. Each message contains information about a transaction that the client wants to perform. The server assigns a transaction id to each transaction, and the client's message includes the transaction id that the client last observed. If the server's last transaction id and the base transaction id in the update message do not match, the server informs the client about the error.

If the transaction ids match, the server executes the transaction, generates a commit message and propagates it to all connected clients.

This implementation had a few problems:

1. Imagine a simple situation. A new project is created, let's denote the initial state of the project as p . Then one client connects and makes a change, producing a second version of the project - let's denote the second version p' . After that a second client connects to the same project. While it is expected that the server would share the latest version of the project p' with the new client, this wasn't the case. In reality, the server would always respond to all newly connected clients with the same default project template p .
2. The transactions performed by the server were not logged. This means that if a

```

1 CREATE TABLE Task (
2     uid          VARCHAR          NOT NULL,
3     num          INTEGER          NOT NULL,
4     name         VARCHAR          NOT NULL,
5     color        VARCHAR          NULL,
6     shape        VARCHAR          NULL,
7     is_milestone BOOLEAN          NOT NULL DEFAULT false,
8     is_project_task BOOLEAN       NOT NULL DEFAULT false,
9     start_date   DATE             NOT NULL,
10    duration     INTEGER          NOT NULL,
11    completion   INTEGER          NULL,
12    earliest_start_date DATE       NULL,
13    priority     VARCHAR          NOT NULL DEFAULT '1',
14    web_link     VARCHAR          NULL,
15    cost_manual_value NUMERIC(1000, 2) NULL,
16    is_cost_calculated BOOLEAN     NULL,
17    notes        VARCHAR          NULL,
18
19    PRIMARY KEY (uid)
20 );

```

Figure 7: Task table definition on a GanttProject client

client was disconnected during some update, then this update is completely lost for this client. The server doesn't have the ability to generate an XML file describing the project from the database state, so recovery was not possible in this situation.

3. When a client performs an update, he has to wait for confirmation from the server. During that time he can't introduce more updates. This makes collaboration in GanttProject not local-first. While it was never intended to be local-first in the first place, it is a nice feature that can be introduced.

5.2 Data model

Due to the way relational databases work, conflicts may arise in places where, intuitively, they shouldn't. For example, if one client tries to change task duration, while another tries to change task description, they will run into a conflict. This is because in the database all information about a task is stored in a single row. When clients try to concurrently change task duration and task description, they are essentially trying to change the same row. This will lead to an error if the two transactions happen to run in parallel on the server.

To avoid that, the data model on the Colloboque server differs from the one on the client. The client's Task table can be found on Figure 7. On the server this table is separated into multiple tables, each table containing uid of the task and some of the attributes of the original Task table, grouped either by logic or attribute type. A view object joins all the tables and provides an interface that allows to query all the different tables as if they were a single Task table. Additionally, triggers are specified to handle updates and insertions called on the view, so that they are performed on the underlying tables. The server's Task view definition can be found on Figure 8

```

1  -- Task start date and duration shall be changed as a whole
2  CREATE TABLE TaskDates (
3      uid          VARCHAR(128) PRIMARY KEY REFERENCES TaskName,
4      start_date    DATE          NOT NULL,
5      duration_days INT          NOT NULL DEFAULT 1,
6      earliest_start_date DATE
7  );
8
9  -- Integer valued properties
10 CREATE TABLE TaskIntProperties (
11     uid          VARCHAR(128) REFERENCES TaskName,
12     prop_name     TaskIntPropertyName,
13     prop_value    INT,
14     PRIMARY KEY (uid, prop_name)
15 );
16
17 -----
18 -- Other component tables are omitted --
19 -----
20
21 -- Updatable view which collects all task properties in a single row.
22 -- Inserts and updates are processed with INSTEAD OF triggers.
23 CREATE VIEW Task AS
24 SELECT
25     TaskName.uid,
26     num,
27     name,
28     start_date,
29     duration_days AS duration,
30     earliest_start_date,
31     is_cost_calculated,
32     cost_manual_value,
33     is_milestone,
34     is_project_task,
35     MAX(TIP.prop_value) FILTER (WHERE TIP.prop_name = 'completion') AS completion,
36     MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'priority') AS priority,
37     MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'color') AS color,
38     MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'shape') AS shape,
39     MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'web_link') AS web_link,
40     MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'notes') AS notes
41 FROM TaskName
42     JOIN TaskDates USING (uid)
43     LEFT JOIN TaskIntProperties TIP USING (uid)
44     LEFT JOIN TaskTextProperties TTP USING (uid)
45     LEFT JOIN TaskCostProperties TCP USING (uid)
46     LEFT JOIN TaskClassProperties TCLP USING (uid)
47 GROUP BY TaskName.uid, TaskDates.uid, TCP.uid, TCLP.uid;

```

Figure 8: Task view definition on a GanttProject Colloboque server

6 Implementation

TODO: Implementation is a huge work in progress and the results are not very tangible as of now.

6.1 jOOQ

The first step in the implementation of local-first collaboration in GanttProject was to integrate jOOQ library [9] with the existing server schema.

jOOQ can generate Java classes based on the database schema, describing tables, columns, and so on. These generated classes provide type-safe access to the database objects, eliminating the need for manual SQL string concatenation and reducing the risk of SQL injection attacks. jOOQ also abstracts away the differences between SQL dialects. This allows to write database-agnostic code that can be easily ported.

TODO: DESCRIBE THIS PARAGRAPH MORE FORMALLY OR EDIT OUT. There were some problems due to jOOQ being bugged. Enum classes were not visible outside of a schema and enum class names had to be written in a specific way for them to be recognized at the class generation step. For example `TaskIntPropertyName` had to be written as `"taskintpropertyname"` in the actual script (Figure 8, line 12).

6.2 Restoring the latest version

The next step was to make the server return the latest version of the project to newly connected clients.

To do that the server first had to persist transaction log. This would allow the server to calculate the relevant version of the project by taking the initial XML project file and repeatedly applying transactions to it.

The source code of the client contains logic for loading project from XML into an in-memory H2 database. This logic was exposed in Colloboque. When a new client connected, the server would load the XML project template into an in-memory database, apply transactions from the transaction log table to it, save the resulting project as an XML and send it to the client.

This way new clients would receive a relevant version of the project, not a project template, as it was before.

6.3 Transaction replay

The last step is to implement transaction replay approach and enable local-first collaboration.

I've implemented a function, which takes a database and two sets of update logs. It creates two connections and starts a transaction on each of the connections. Both transactions run with isolation level REPEATABLE READ. After starting the transactions, the function applies one set of updates in one transaction, and the second set of updates in another transaction. Finally it tries to commit both transactions, and either returns true if both commits were successful, or false if there was an error.

Unfortunately, the execution never manages to reach the commit stage. When the second set of updates is being applied, the code hangs indefinitely. It seems that there is some kind of a deadlock in the database. Changing transaction isolation level to READ COMMITTED doesn't affect this behavior. This problem needs further investigation. See function code on Figure 9

6.4 Protocol

Assume we can tell from the transaction ids whether two transactions are concurrent or in a happens-before relationship and if yes then which happens before.

All updates on the server are processed synchronously. They are either in-order updates which can be directly applied, or out of order updates. Out-of-order updates are checked for compatibility with tryMerge and applied if they are compatible. After each update is committed, connected clients are notified. If an update is incompatible, the author of the update is notified.

The client processes all updates, both local and external, synchronously from a queue. When a local update is performed, it is queued with all the other updates. If a client reaches a local update in the queue and successfully applies it, information about the update is sent to the server. There is a track list for local updates that were sent to the server but the client has not yet received an acknowledgement for them.

Sometimes a client pulls a server's update from the queue, but can't apply it because his local state is different from the state required by the update. In this case there is a track list for server updates which were received but couldn't be applied. We say that the client diverges from the server iff the track list for unapplied server updates is not empty.

If the client pulls a server update from the queue, a few situations are possible.

First, the base transaction of the server's update matches the last transaction on the client and the client doesn't diverge from the server. Then the client can directly apply the update.

Second, the base transaction of the server's update matches the last transaction on the client and the client currently diverges from the server. If the client diverges then the last committed transaction on the client always corresponds to the last local update that the client performed. If the server successfully committed the client's last update, it means that his updates don't conflict with the previous server updates. So, the client can apply all updates stored in the unapplied server updates track list. Now the client is not in a divergent state.

Third, the transaction is concurrent with the client's last transaction. In this case the client adds this update to a track list of unapplied server updates.

Fourth, the message informs the client that one of his local updates couldn't be committed. In this case the client discards the local copy of the project (maybe saving a backup) and requests the latest version of the project from the server.

Fifth, the base transaction is newer than the client's state in terms of the happens-before relation. This means that the client is missing some updates. Then the client requests the most recent state from the server. He can also communicate his last transaction id so that the server can send missing updates only.

```

1  override fun tryMergeConcurrentUpdates(
2      database: SqlProjectDatabaseImpl,
3      serverTransaction: List<XlogRecord>,
4      clientTransaction: List<XlogRecord>
5  ): Boolean {
6      val serverConnection = database.createConnection()
7      val clientConnection = database.createConnection()
8      serverConnection.transactionIsolation =
9      ↪ Connection.TRANSACTION_REPEATABLE_READ
10     clientConnection.transactionIsolation =
11     ↪ Connection.TRANSACTION_REPEATABLE_READ
12     val serverDsl = dsl(serverConnection)
13     val clientDsl = dsl(clientConnection)
14     serverDsl.startTransaction().execute()
15     clientDsl.startTransaction().execute()
16     try {
17         serverTransaction.forEach {
18             it.colloboqueOperations.forEach {
19                 LOG.debug("... applying operation={}", it)
20                 serverDsl.execute(generateSqlStatement(serverDsl, it))
21             }
22         }
23         clientTransaction.forEach {
24             it.colloboqueOperations.forEach {
25                 LOG.debug("... applying operation={}", it)
26                 // Hangs here.
27                 // When `serverDsl.commit().execute()` is moved
28                 // before the `clientTransaction.forEach { ... }`
29                 // block, the whole function executes successfully
30                 // and returns true even if there
31                 // should have been a conflict.
32                 // Transaction isolation level setting
33                 // doesn't affect this behaviour.
34                 clientDsl.execute(generateSqlStatement(clientDsl, it))
35             }
36         }
37         LOG.debug("... committing server's transaction")
38         if (serverDsl.commit().execute() != 0) return false
39         LOG.debug("... committing client's transaction")
40         return clientDsl.commit().execute() == 0
41     } catch (e: Exception) {
42         LOG.info("Failed to execute transactions in parallel! Reason:
43         ↪ ${e.localizedMessage}")
44     }
45     return false
46 }

```

Figure 9: Function for trying transaction replay

References

- [1] *Automerge*. <https://github.com/automerge/automerge>.
- [2] *Data branching for Postgres by Neon*. <https://neon.tech/branching>.
- [3] Brian Fung. “A mysterious message is locking Google Docs users out of their files”. In: *The Washington Post* (2017). URL: <https://www.washingtonpost.com/news/the-switch/wp/2017/10/31/a-mysterious-message-is-locking-google-docs-users-out-of-their-files/>.
- [4] *GanttProject Cloud official web-page*. <https://ganttproject.cloud/>.
- [5] *GanttProject GitHub repository*. <https://github.com/bardsoftware/ganttproject>.
- [6] *GanttProject official web-page*. <https://www.ganttproject.biz/>.
- [7] *Google Wave on Wikipedia*. https://en.wikipedia.org/wiki/Google_Wave.
- [8] *Google Wave Overview*. <https://youtu.be/p6pgxLaDdQw?si=QcI7beHUIwMFTto2o>. 2009.
- [9] *jOOQ library*. <https://www.jooq.org/>.
- [10] Martin Kleppmann et al. “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. Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178. ISBN: 9781450369954. DOI: [10.1145/3359591.3359737](https://doi.org/10.1145/3359591.3359737). URL: <https://doi.org/10.1145/3359591.3359737>.
- [11] *Merging rules for objects in Automerge documents*. https://automerge.org/docs/under-the-hood/merge_rules/.
- [12] *PostgreSQL Transaction Isolation Levels documentation*. <https://www.postgresql.org/docs/current/transaction-iso.html>.
- [13] Marc Shapiro et al. “A comprehensive study of Convergent and Commutative Replicated Data Types”. In: (Jan. 2011).
- [14] Chengzheng Sun and Clarence Ellis. “Operational transformation in real-time group editors: issues, algorithms, and achievements”. In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. CSCW ’98. Seattle, Washington, USA: Association for Computing Machinery, 1998, pp. 59–68. ISBN: 1581130090. DOI: [10.1145/289444.289469](https://doi.org/10.1145/289444.289469). URL: <https://doi.org/10.1145/289444.289469>.
- [15] David Wang, Alex Mah, and Soren Lassen. *Google Wave Operational Transformation*. <https://svn.apache.org/repos/asf/incubator/wave/whitepapers/operational-transform/operational-transform.html>. July 2010.
- [16] Weihai Yu and Claudia-Lavinia Ignat. “Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge”. In: *IEEE International Conference on Smart Data Services, 2020 IEEE World Congress on Services*. Beijing, China, Oct. 2020. URL: <https://inria.hal.science/hal-02983557>.