Relic: A Full-Stack Web Framework for Offline-First Applications

Vigo Vlugt
vigovlugt@gmail.com

June 26, 2024, 88 pages

Academic supervisor:	Zhiming Zhao, Z.Zhao@uva.nl
Daily supervisor:	Zhiming Zhao, Z.Zhao@uva.nl
Host organisation/Research group:	Multiscale Networked Systems, https://mns-research.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

In this thesis the Relic Offline-First web framework is presented. Offline-First applications bring important benefits to users by being offline capable, extremely responsive and supporting Real-Time collaboration. However, creating Offline-First applications is complex; developers need to handle issues such as device storage and data replication. To address these issues, we have designed and implemented a framework that handles this complexity so developers can focus on creating applications.

Existing Offline-First frameworks often do not support application-specific conflict resolution, which limits the types of applications that can be created using that framework. Furthermore, many Offline-First frameworks do not support a relational data model, posing challenges for applications with relationships in their data. Existing Offline-First frameworks often only work with one specific database, imposing limitations on the choice of database technology that developers can use. Finally, existing frameworks often provide an existing backend service that can only be configured, greatly reducing flexibility and customization, or frameworks require developers to write a backend from scratch, which is time-consuming and complex.

To provide device storage, the Relic framework uses an embedded reactive SQLite database, allowing developers to build their applications using a relational data model. Relic provides built-in replication by using client-side prediction and server reconciliation, allowing developers to take advantage of application-specific conflict resolution. The framework is full-stack, running on both the server and the client. By allowing developers to write a custom backend using the backend Relic library, Relic provides developers with flexibility and power, without requiring them to write a backend from scratch. Additionally, developers can bring their own backend database by using modular adapters. The framework provides end-to-end type safety using TypeScript to increase developer experience.

Contents

Chapter 1

Introduction

Web applications are everywhere. The World Wide Web offers many advantages that are not present in native applications, such as accessibility, cross-platform compatibility, and ease of distribution. Despite these advantages, the current generation of web applications also has several shortcomings. Features such as offline support, real-time collaboration, and responsiveness are often not fully addressed in this generation of web applications.

To address these issues, developers can write software "Offline-First". The Offline-First paradigm describes applications that function both online and offline. With Offline-First applications, users can still access data without a network connection. The Offline-First approach to creating software is already used in some of the most popular mobile applications, such as WhatsApp, Google/Apple Calendar, and many others. Users can resume using WhatsApp, read their messages and queue new messages while the network connection is inconsistent or disconnected. This allows users to resume work in environments such as in a subway tunnel or on an airplane where the internet is unavailable.

The network connection of mobile devices is often more unpredictable than desktop and laptop devices, but that does not mean that Offline-First does not have benefits for these users. Offline-First is not only about offline functionality; Offline-First applications store data on the local device. This greatly affects the application's responsiveness. Instead of applications being dependent on the network latency to the server for every interaction, Offline-First applications can respond instantaneously, reading from and writing to the local storage. Not having to wait for loading spinners has large benefits, increasing user experience and productivity.

Finally, the data model of Offline-First applications makes supporting real-time updates trivial. Adding real-time updates in other architectures can be difficult and time-consuming. Real-time updates enable users to instantly see changes made by other users without refreshing the application. This capability enables users to work collaboratively on shared data, increasing user engagement and experience.

1.1 Problem Statement

Offline-first applications provide important functionality, offline support, real-time collaboration, and significantly increased responsiveness. However, Offline-First also introduces new sources of complexity: state management and state replication.

Typical web applications do not store much data on the client. The client requests data from the server, the single source of truth, and displays the data. To enable offline usage of the application and remove this network latency, Local-First applications store the state of the application on the client. Many state management solutions for front-end applications are not created to handle the amount of data that is normally stored in a database on the server. Modern UI frameworks such as React [1] use immutable data and simple objects to track state changes in order to rerender efficiently. Storing state in simple immutable objects works for small amounts of state but is too inefficient to store the entire state of the application. This task is typically designated to a database that can trivially handle this amount of data. Therefore, Local-First applications require such a database on the client.

The second problem is the replication of data to enable collaboration between users. Clients must be able to update data while offline and still be able to synchronize that data to the other clients after

[1] `https://react.dev`

regaining connection. Thus, at any given moment, multiple clients, each having a local copy of the database, may or may not be connected to the internet and may have modified as much data as they wanted locally. The application must make sure that the states of these clients eventually converge to a single consistent state, as we do not want clients to see multiple different views of the database even after synchronization. Locally applied writes of multiple users may also conflict with each other. How to resolve this conflict is an important issue that has a significant impact on the technical implementation of the application, while it also has a large impact on the user experience.

These problems increase the adoption difficulty of Offline-First compared to traditional web architectures. This makes it costly to invest in Offline-First, even though it may be the right architecture for the application, and users would benefit from the features. To increase adoption, we propose creating a web framework designed especially for building Offline-First applications. This framework should handle the complexity of state management and replication. With this framework, application developers can create Offline-First applications as easily as traditional web applications, and end-users can benefit from the features.

1.1.1 Research Questions

This thesis aims to provide an answer to the following research question:
RQ: How to design a web framework for Offline-First applications?
To answer the main research question, we define the following research sub-questions:

- **RQ1**: How to structure, store and query large amounts of data in the frontend that supports many different types of applications?
- **RQ2**: How to synchronize updates between clients while ensuring eventual consistency of state and minimizing architectural and coding constraints on the application?
- **RQ3**: How to design an Offline-First framework API to facilitate developer productivity, ease-of-use, modularity, and safety.

1.1.2 Research Method

Firstly, a literature study covering the state of the art and related work is conducted. The topics covered in the literature study will include:

- Existing Offline-First web frameworks.
- Case studies describing existing Offline-First systems or systems with real-time collaboration.
- Papers describing storage mechanisms on edge devices such as smartphones and web browsers.
- Papers presenting solutions to replicate data between devices.

With the results of the literature study, an overview of both storage mechanisms and replication strategies will be created, focusing on the tradeoffs for Offline-First applications. These overviews will inform the answers to the research questions.

With the answers to these research questions, a web framework incorporating the selected storage and replication mechanism will be designed and implemented, creating a web framework for Offline-First applications.

This implementation will then be used to validate two characteristics of the framework. The first characteristic that should be validated is the properties of RQ3, ease-of-use, productivity, modularity and safety. We will conduct a case study to validate this, implementing a Local-First application in this framework. The aforementioned properties will be shown, and the advantages and disadvantages of these properties in the case study of the application implementation will be highlighted.

After validating the developer experience, the framework's performance will be validated. The application implemented in the case study will be benchmarked on both the client- and server-side.

1.2 Contributions

Thus, our research makes the following contributions:

- A literature study of existing literature and frameworks for Offline-First web applications.
- An overview of storage mechanisms for Offline-First web applications.
- An overview of replication strategies for Offline-First web applications.
- An implementation of an Offline-First web framework.

- An implementation of an example application implemented using the framework.
- A case study demonstrating developer experience using the framework.
- A demonstration of the performance of the web framework backend and frontend.

1.3 Outline

In chapter 2, we describe the theoretical background forming the foundation for the research in this thesis. In chapter 2, the results of the related work and state-of-the-art literature study are presented. Chapter 4 discusses the design of the storage mechanism, replication technique and the overall framework. The implementation choices and technical details of the framework are discussed in chapter 5. Then, we present the methodology and results of the experiments in chapter 6. The results of the experiments are discussed in chapter 7. Finally, the thesis is concluded in chapter 8

Chapter 2

Background

This chapter will outline a necessary theoretical background that will serve as a foundation for the thesis.

2.1 Local-First

The term Local-First was introduced in the paper [1]. In this paper, the motivation and ideals for Local-First software are outlined, together with several existing approaches and possible future approaches to achieve these ideals are described.

The motivation for Local-First software emerges from the recent rise of "cloud software". Cloud software brings important benefits, such as collaboration and making data accessible from anywhere, however the usage of cloud apps also brings new problems. In cloud apps, your data is not stored on your device, but on "someone else's computer", which is in control of that data. If the service is unavailable or shuts down it is not possible to use that software anymore.

The paper introduces the concept of "old-fashioned apps", where the data does not reside on the server but on the local disk, which means that you are the primary owner of the data. However, in ordinary old-fashioned apps, collaboration is not present, and data is not automatically accessible on other devices. This forms the motivation for Local-First software, which combines cloud apps and old-fashioned apps so the user has personal ownership of the data, while cross-device access and collaboration are built in.

Local-First software is based on seven ideals. These ideals are:

- **Fast interactions**: interaction speed is not limited by network latency
- **Multi-device access**: users can access their work on multiple devices
- **Offline operation**: applications continue to work with inconsistent or no internet connectivity
- **Collaboration**: multiple users are able to edit a document simultaneously in real-time, and not have to worry about conflicts
- **Data longevity**: your data is accessible even when the company that produced the software is gone
- **Privacy**: data is end-to-end encrypted, to foster privacy and increased security
- **User control**: users, not the service provider have ultimate ownership of their data.

2.2 Offline-First

Closely related to the Local-First movement is the Offline-First paradigm [2–4]. Offline-First predates Local-First, and is much more adopted in the industry. These approaches to building software have a lot in common. An essential property for both paradigms is to enable users to use the application without or with inconsistent internet access. The Local-First property of "fast interactions" is interestingly a result of this former property. If the interactions of an application work offline, all interactions can also be performed online, without waiting for network latency. The properties "multi-device access" and "collaboration" are also shared with the Offline-First paradigm, as synchronization between the server and the client is an important aspect of Offline-First.

Here, the similarities between the two paradigms stop. The key difference between the paradigms are the last three Local-First ideals: data longevity, privacy and user control. In Local-First, applications should work without using a centralized server. The emphasis is on decentralization, moving away from the current trend where the cloud is owner of the data of the user. For Offline-First, this is not a core focus. The concept of ownership is not explicitly addressed, and the primary goal is to ensure offline functionality.

This difference is an important one. The Local-First goal of moving ownership from cloud servers to end-users is admirable and could prove significant in reshaping software and data ownership in the future. On the other hand, it presents significant trade-offs that not every application will want to make. Moving from a centralized system to a decentralized system has implications regarding software architecture. Furthermore, business-wise, it also presents implications, much software is distributed using a Software as a Service (SaaS) model. This model is a better fit for a centralized client-server architecture than a peer-to-peer architecture. Finally, the Local-First paper presents several applications which Local-First is intended for: "documents or files (such as text, graphics, spreadsheets, CAD drawings, or music), or personal data repositories (such as notes, calendars, to-do lists, or password managers". The paper then states that Local-First is not intended for applications such as: "banking services, e-commerce, social networking, ride-sharing, or similar services" as these are better served by centralized systems. Offline-First also does not make sense for every application; however, due to its centralized nature, it can serve more application types. For these reasons, this thesis will explicitly not tackle the challenges of Local-First applications but will instead focus on solving the challenges presented by Offline-First.

Offline-First is commonly seen in mobile applications, where internet connectivity is often more uncertain than in desktop applications. Mobile applications such as WhatsApp, Apple Calendar, Google Calendar and many others are already Offline-First applications. Desktop applications often have a more stable internet connectivity than mobile devices. However, this does not mean that Offline-First is not valuable for desktop applications. There are several benefits of Offline-First applications besides offline operation. Offline-First applications do not have to wait for network latency on reads and writes [3]. While other network properties such as bandwidth have been and are still improving, latency is harder to reduce as it is defined by server location and speed of light. Offline-First applications use a lot of bandwidth the first time the site is opened, after which network latency is removed. Another important benefit of Offline-First is the relative simplicity of adding realtime updates to the site, thus enabling user collaboration.

2.3 Consistency & the CAP Theorem

The CAP theorem [5] describes the fundamental characteristics of distributed systems:

- **Consistency**: every read reads the most recent write or fails.
- **Availability**: every read succeeds but is not guaranteed to see the most recent write
- **Partition tolerance**: when the network fails, the system continues to operate

The CAP theorem states that distributed systems can only provide two of the three guarantees. As no distributed system is free from network failures, generally, partition tolerance is selected as one guarantee. Therefore, distributed systems can be described as AP (available and partition tolerant) or CP (consistent and partition tolerant).

A CP system can be described as a system that supports strong consistency. A strongly consistent system is the highest level of consistency; every read will return all writes up to that point. This makes strong consistency easy to reason about, allows for simplified application logic and allows for a high level of data integrity. However, to achieve this, strong consistency requires considerable coordination and overhead. Furthermore, when a partition between two nodes occurs, one node has to shut down until the partition is resolved in order to stay consistent.

Most AP systems are eventually consistent: a weaker level of guarantee than strong consistency. Consequently, a read can return an older version of data, and different clients can see different states of the system simultaneously. However, easing these guarantees allows nodes in an eventually consistent system to remain available under network partitions. There is also less coordination overhead, resulting in reduced latency and higher scalability. Eventually consistent systems must eventually converge to the same state. A challenge in eventually consistent systems is conflict resolution in case of conflicts. If two clients have independently written different values in the same register, this conflict must be resolved to allow the nodes to eventually converge to the same value.

In the context of Offline-First, the "Offline operation" ideal states that applications should continue to work with inconsistent or no internet connectivity. In other words, under a network partition the application should continue to remain available. A Offline-First system is, therefore, required to support availability and partition tolerance under the CAP theorem, thus choosing AP and eventual consistency.

2.4 Conflict Resolution

In eventually consistent applications, there are bound to be conflicts when users use the application concurrently. An example of a conflict is presented in the following situation. Two users share the same eventually to-do list. The to-do list contains a to-do item called ”Hello, world”. User A and B are both offline, user A changes the to-do item name to ”Hello”, while user B changes the name to ”World”. When the users reconnect and synchronize their to-do list, there is a conflict. Should the to-do item be called ”Hello” or ”World”? There are many possibilities to solve this conflict, and they have big impacts on both software architecture and end-user satisfaction. The techniques to solve these conflicts are called conflict resolution mechanisms.

2.4.1 CRDTs

As mentioned, eventually consistent distributed systems can introduce conflicts under concurrent updates. Conflict-free Replicated Data Types (CRDTs) are a class of general-purpose data structures with built-in conflict resolution [6]. Therefore, users can continue to make changes concurrently and have a guarantee that the conflicts are resolved and the two states eventually converge to a single common state.

There are two main types of CRDTs: Commutative Replicated Data Types (CmRDTs) and Convergent Replicated Data Types (CvRDTs) [7]. CmRDTs are operation-based. The state is propagated by transmitting every update operation a client creates. These update operations are commutative and replicated to other clients, which can then apply the operation to converge to the same state.

CvRDTs are state-based. Instead of sending operations, the full state is shared with other clients. The local and remote state are then merged with a merge function that is commutative, associative and idempotent, in order to ensure convergence.

2.4.2 Operational Transformation

Operational Transformation (OT) is a conflict resolution technique which was first introduced in 1989 to enable collaboration in software systems [8]. One system using Operational Transformation is Google Docs[1], to enable collaboration in their document editor.

In Operational Transformation based systems, clients have a copy of the shared document locally, enabling clients to apply *operations* to their documents locally. These operations, are replicated to other clients. When receiving an operation from another client, this operation is first transformed according to transformation functions defined by the application. These transformation functions must transform the operation to ensure that causality and admissibility are preserved. In this context, causality means that an operation that happened before one operation is executed in the same order, and admissibility states whether the operation may be executed in the current state. When these properties are preserved, convergence to the same application state is guaranteed.

[1]https://docs.google.com

Chapter 3

Related work

This chapter discusses the related work and state of the art regarding Offline-First software. We start by reviewing existing Offline-First frameworks, highlighting the main differences and shortcomings. Then we discuss literature in this field, from academic papers and industry case studies. This chapter contains four sections, in the first section, we discuss the methodology for finding literature and frameworks. Then, we discuss existing frameworks. Next, we discuss literature describing relevant storage mechanisms. Finally, we discuss replication solutions and outline the gaps in existing research and frameworks.

3.1 Methodology

We first define the objectives of the literature review:

- By finding existing Offline-First frameworks, we aim to identify approaches to storage and replication related to Offline-First, and to identify gaps in existing frameworks.
- By finding literature describing storage solutions, we aim to find an answer to RQ1.
- To answer RQ2, we review literature that describes replication in Offline-First systems and other systems.
- To find requirements and approaches to the framework API and tradeoffs, answering RQ3.

Existing frameworks have been identified using the official Local-First website[9]. The Local-First website presents a comprehensive overview of popular JavaScript data storage and data synchronization libraries and frameworks, both Local-First and Offline-First. We eliminated libraries to primarily focus on complete frameworks. By examining the websites of each framework, we used snowballing to find other frameworks that were cited by frameworks.

Literature regarding storage, replication and library design was found using several methods. Academic sources were primarily found by using Google Search and Semantic Scholar, using keywords: "Local-First", "Offline-First", "state management", "sqlite", "document store", "triple store", "optimistic replication", "replication", "eventual consistency", "conflict resolution", "CRDT", "operational transformation", "last write wins", "semantic conflict resolution", "developer experience", "software library design", "type safety", "static typing". With the results of these queries, we evaluated the paper's relevance by reviewing the title and abstract. We used snowballing to find more papers that were cited or referenced by earlier found papers. To ensure a comprehensive view, we accompanied sources from the industry. These sources include case studies describing Offline-First software, real-time collaboration systems, and essays related to Offline-First. Many of these sources were found on the sites of existing Offline-First frameworks. Again, snowballing was used to find more sources, by examining linked sources and discussions of the sources in HackerNews, where other relevant sources were often mentioned.

3.2 Existing Frameworks

In this section, we discuss the existing frameworks and their characteristics. We identified several significant properties related to storage, replication and implementation. For replication, we review how the different frameworks handle conflict resolution and the network topology. Regarding storage, every framework with a client-server network topology has two storage mechanisms: one for frontend (FE) storage and one for backend (BE) storage. An important differentiating property regarding implementation is how the backend for applications using the framework is implemented. Finally, we review whether or not the framework is open-source.

Framework	Conflict Resolution	Network Architecture	FE Storage	BE Infrastructure	BE Storage	Source
Replicache[a]	Conflict Avoidance	Client-Server	Document store	Custom	Custom[b]	Closed
RxDB[c]	Application-Specific	Any	Document store	Custom	Custom	Open
ElectricSQL[d]	CRDT	Client-Server	SQLite	ElectricSync	Postgres	Open
PowerSync[e]	Application-Specific	Client-Server	SQLite	PowerSync	Postgres	Closed
VLCN[f]	CRDT	Client-Server	SQLite	Custom	SQLite	Open
Instant[g]	Last Write Wins	Client-Server	Graph store	Instant	Instant	Closed
PouchDB[h]	Application-Specific	Client-Server	Document store	CouchDB	CouchDB	Open
Realm[i]	Last Write Wins	Client-Server	Document store	MongoDB	MongoDB	Open
SQLSync[j]	Conflict Avoidance	Client-Server	SQLite	Custom[k]	SQLite	Open
WatermelonDB[l]	Application-Specific	Client-Server	Document store	Custom	Custom	Open
Verdant[m]	Last Write Wins	Client-Server	Document store	Verdant Server	Sqlite	Open
Triplit[n]	Last Write Wins	Client-Server	Document store	Triplit Server	Sqlite	Open
Firestore[o]	Last Write Wins	Client-Server	Document store	Firestore Server	Firestore	Closed

Table 3.1: Overview of existing Offline-First frameworks (BE = Backend, FE = Frontend)

[a]`https://replicache.dev/`
[b]Any database that supports snapshot isolation transactions
[c]`https://rxdb.info/`
[d]`https://electric-sql.com/`
[e]`https://www.powersync.com/`
[f]`https://vlcn.io/`
[g]`https://www.instantdb.com/`
[h]`https://pouchdb.com/`
[i]`https://realm.io/`
[j]`https://sqlsync.dev/`
[k]Only WASM
[l]`https://watermelondb.dev/`
[m]`https://verdant.dev/docs/sync/server`
[n]`https://www.triplit.dev/`
[o]`https://firebase.google.com/docs/firestore`

The result of this review is presented in table 3.1, presenting an overview of 13 existing Offline-First frameworks and their key differentiating properties.

3.2.1 Conflict Resolution

Five of the frameworks reviewed use Last Write Wins as the conflict resolution mechanism. Four frameworks allow developers to write application-specific conflict resolution to resolve conflicts according to the context of the application. Two frameworks use CRDTs to handle conflict resolution. Finally, two frameworks use conflict avoidance to ensure there is never any conflict.

Last Write Wins

A Last-Writer-Wins Register [7] is a type of CRDT that resolves conflicts by setting the resulting value to the value that was last updated. In the example of a conflict where two offline clients have independently changed the title of the same to-do item, on synchronization, LWW will set the title to the title that was last updated. Each update is assigned a timestamp such that it can be compared with other updates. Last-Writer-Wins Registers are available as a CvRDT and CmRDT.

CRDTs

ElectricSQL and VLCN compose CRDTs in order to allow the entire database to be represented as a CRDT, to enable automatic conflict resolution. To represent a database as a CRDT, VLCN represents tables as a Grow Only Set. A Grow Only Set is a CRDT which holds a set of values that can only grow, in this case, storing rows as values [7]. Grow Only Sets can be trivially merged by taking the union of two Grow Only Sets. Rows can be represented as map CRDTs, which store keys as static strings and other CRDTs as values. Finally, columns are represented as Last-Write-Wins CRDTs.

Thus, by composing CRDTs together, the entire database can be represented as a CRDT.

Application-Specific Conflict Resolution

Application-specific conflict resolution is not a general conflict resolution mechanism such as Last Write Wins but is specific to an application. It is more work to implement for the developer but also gives more flexibility and control.

Conflict Avoidance

Some systems are implemented such that a mechanism for conflict resolution is unnecessary, as the system is designed to never create a conflict. This makes it also very simple to work with and reason about as a developer, but it does create some restrictions on the system.

3.2.2 Frontend Storage

The most observed solution for frontend storage is a document store. Most frameworks (Replicache, PouchDB, Realm, Verdant, Triplit and Firestore) implement their own custom document store, especially for the framework. RxDB and WatermelonDB allow the developer to select their own document store, and have several official adapters available.

One framework, Instant, uses a graph store based on triple stores, with a custom query language, InstaQL.

Finally, four frameworks use SQLite in the frontend, which WASM makes possible. This allows developers to query their data using SQL.

3.2.3 Network Architecture

All of the frameworks support a client-server network architecture. In this model, clients submit updates in the form of state or operations to the server. The server then broadcasts this update to all other clients. With client-server, the server is the final authoritative source of truth. The only framework to also support another topology, such as peer-to-peer, is RxDB. In a peer-to-peer network topology, there is no centralized server, and clients are connected directly to each other using the WebRTC protocol.

3.2.4 Backend Infrastructure

Another large difference is how the frameworks manage the infrastructure in the backend. Many frameworks provide an executable that is responsible for replication, data storage, and real-time updates. This reduces work for the developer but also limits flexibility and customization. On the other hand, some frameworks do not provide an executable and require the developer to implement their own backend. This approach offers greater control and flexibility but requires a considerable time investment.

3.2.5 Backend Storage

Custom backend storage

Many frameworks that allow for a custom backend implementation also allow for the usage of a custom database on the backend. Just like the backend, this provides developers with the greatest amount of flexibility and control.

SQLite

Two frameworks, SQLSync and VLCN, use SQLite as frontend storage and require SQLite as backend storage. This provides the benefit of a single database type and provides two benefits. The first benefit is that it allows for code reuse between the frontend and the backend. This allows SQLSync to run the same code on the front and backend without requiring developers to implement it twice. The other benefit that VLCN makes use of is that the underlying storage is the same. Therefore, data can easily be replicated between SQLite in the front and backend, without requiring the data to be mutated to fit a different database.

Postgres

Both ElectricSQL and PowerSync run SQLite in the frontend but store the data in Postgres[1] on the backend. These frameworks are tightly integrated with Postgres, making use of the Write Ahead Log (WAL) [10]. Using the WAL, ElectricSQL and PowerSync can listen to changes in the database, which can then be immediately replicated to clients.

Miscellaneous

Many frameworks require their own custom backend storage instead of allowing for custom storage using SQLite or Postgres. For example, PouchDB can only replicate to CouchDB, which it is based on. Realm, which is created by MongoDB, can only replicate to MongoDB. Other frameworks like Firestore and Instant are proprietary, therefore developers do not have any control over the backend infrastructure, or over the backend database.

3.2.6 Source Availability

Finally, an important property to assess these frameworks is whether they are Open-Source or not. 9 of the 13 frameworks are Open-Source. The proprietary, Closed-Source frameworks are Replicache, PowerSync, Instant and Firestore. Closed-Source solutions have several implications for developers. Licensing fees or subscription costs can significantly impact the budget of a project. Customers also have to be aware of vendor lock-in. The threat of vendor lock-in is especially present in the case of a framework, as migrating away from a framework takes much effort. Furthermore, Closed-Source frameworks limit the possibility for users to extend or customize functionality, as source code is not available. Finally, Closed-Source frameworks also raise concerns regarding the lack of transparency about privacy and security, as developers have less visibility about how user data is processed and managed.

3.3 Storage

We have concluded the review of the existing frameworks. In this section we will review both academic literature, and case studies from the industry regarding storage mechanisms for edge devices.

3.3.1 Databases in the Browser

The problem of state management in web applications is discussed in many sources [11–15]. In many web applications, state is distributed across multiple layers between the frontend and backend; state is present in a database, an ORM, client-side caches, and in-memory state. Every layer adds complexity, as data needs to be transformed between layers. When data changes, a state in every layer that is affected must be invalidated. Current state management is complex, and adding features such as reactivity, cache invalidation, and optimistic mutations makes it even more difficult. To simplify state management, these sources propose to use a database as state management system in the frontend.

Litt *et al.* [11] describes a state management system for Offline-First web applications called Riffle. Riffle manages the state of the web application in a client-side persisted relational database. State invalidation and reactive queries are built into this database. The solution is evaluated using a case study, validating the increased simplicity for the developer, and fast and consistent interactions for end-users.

Sverre [13] presents the reasoning for the creation of the SQLSync framework. SQLSync is built on top of SQLite, adding reactive queries and synchronization. The author highlights several benefits of SQLite in the browser, including indexes, constraints, triggers, foreign keys, constraints, full-text search, and query optimization.

Parunashvili [12, 14] proposes a solution to the discussed state management problems. This solution was adapted to become the Instant framework. The importance of relations in this solution and the possibility of using the database without specifying a schema upfront are emphasised.

Limits of SQL

There are conflicting views regarding the usage of SQL as a query language. In the validation of Litt *et al.*, some limitations of SQL were found: unlike graph query languages, SQL cannot produce tree-shaped

[1] `https://www.postgresql.org/`

results, querying relations requires verbose syntax, fragments cannot easily be created to be reused across queries, and there is no automatic type inference of SQL queries. Sverre makes a statement that conflicts with Riffle's viewpoint, stating that SQL can easily express complex queries over the data. Parunashvili brings forward several issues with SQLite, the first being the size of SQLite WASM binaries and the requirement of a data model schema. Besides SQLite, issues with SQL have also been identified, specifically the lack of recursive queries in order to fetch nested relations.

SQL alternatives

Litt *et al.*, Prokopov and Parunashvili present different solutions to the problems identified with SQL. Litt *et al.* has implemented a GraphQL layer on top of SQL, emphasizing its ability to create tree-shaped results, better association traversion, and TypeScript type safety. On the other hand, the downsides of GraphQL are also apparent as the addition of another query language adds significant complexity.

Parunashvili and Prokopov propose using a graph-based query language to replace SQL, and replacing SQLite with a triple store [16]. Triple stores do not need a schema and are much more lightweight than an SQLite WASM binary, and the implementation is much simpler than SQL. Datalog is a logic-based language built for use with triple stores. Parunashvili suggests that the need to learn the logic-based nature of the language and the data transfer method of triples is not ideal in the frontend where you rather work with objects. Therefore, they propose a new graph query language for triple stores, InstaQL. InstaQL is written in plain JavaScript objects and returns JavaScript Objects. Furthermore, using plain JavaScript allows for type safety.

3.3.2 Traditional State Managers

The Linear Offline-First application takes a different approach to state management. Instead of using a database such as SQLite, a triple store or a document database, Linear uses a traditional state manager: MobX[2] [17]. MobX is a state management library that does not rely on immutable data, which is the reason why many traditional state management solutions do not scale very well with a large amount of data. MobX provides mutable state management through typical JavaScript classes and adds reactivity through observables. This has the consequence that all data must be loaded into memory, but this may be desired due to the increased performance of memory lookups. Linear added a layer on top of MobX, which provides developers with relations using TypeScript decorators. These relations allow developers to write code to find all people related to a team through, for example, the "team.members" property, and allow the reverse lookup aswell, such as a "person.team" property. When team membership is changed, the reverse property is also reactively updated.

3.3.3 Synchronous & Asynchronous Storage

Whether to implement storage asynchronously or synchronously is debated in Litt *et al.* [11]. In an early prototype, the database was implemented in a Web Worker thread, which communicates asynchronously with the main UI thread. This communication incurs overhead. In the case of Riffle, this overhead was sometimes 2ms per query, which counts up if multiple SQL queries have to re-run, causing lag in interactions. Furthermore, the asynchronous mental model is more complex than synchronous state. It has effects on the UI of the application as components need to wait on the state, having to show a loading spinner or an empty state. To address these problems, Riffle switched to synchronous state management, storing the data in memory. Consequently, thread overhead was eliminated and the UI code was simpler. On the other hand, running the database on the same thread as the UI can halt the UI if queries are long-running. Furthermore, the size of memory limits the maximum amount of data that can be stored.

3.4 Replication

In this section, we discuss several properties of replication systems. We discuss conflict resolution, and the different methods of handling it. Then we discuss an important property of distributed systems, the number of leaders. Furthermore, we discuss an option to deal with convergence: operation scheduling. Finally, we discuss the main network architectures to consider for replicated Offline-First applications.

[2]`https://mobx.js.org/README.html`

3.4.1 Conflict Resolution

Operational Transformation

Operational Transformation (OT) is an algorithm that was primarily designed to enable collaboration for concurrent text editing. OT is used in production in several text-editing applications such as Google Docs [18], Etherpad[3] and the discontinued Google Wave [19]. Besides text-editing, Operational Transformation has also been adapted to handle collaboration for arbitrary JSON documents in the ShareDB library[4].

The previously mentioned Operational Transformation implementations all use a client-server architecture, as the underlying OT algorithm requires a central server. OT was initially designed for decentralized peer-to-peer communication. However, all decentralized algorithms have been proven incorrect, as they are unable to always achieve convergence [20, 21].

The root of this problem seems to be the complexity of Operational Transformation. To prove the correctness of an OT algorithm, a large combination of states and operations must be considered. This notion is captured by Li and Li, which states: "Due to the need to consider complicated case coverage, formal proofs are very complicated and error-prone, even for OT algorithms that only treat two characterwise primitives (insert and delete)" [22].

In the renowned essay *How Figma's multiplayer technology works* [23], Operational Transformation was considered for the implementation of the multiplayer technology for the Figma collaborative interface design application. Figma discarded using OT for the same reasons as listed previously: the complicated nature of OT and the difficulty of correct implementation. In the article, Figma states: "Since Figma isn't a text editor, we didn't need the power of OTs and could get away with something less complicated.".

In summary, OT is a proven technology that is used successfully in production by large applications. However, OT is complicated and correctly implementing OT is even harder. Furthermore, most research on OT is based on text-editing; synchronizing all application states through OT requires more operations, further increasing the complexity. Finally, OT requires a central server, as decentralized solutions have proven incorrect.

CRDTs

In response to the problems of Operational Transformation, a new direction in research was proposed: Conflict-Free Replicated Data Types (CRDTs) [20]. Opposed to OT, CRDTs are more easily verifiable [24]. Furthermore, CRDTs can be used peer-to-peer, fully decentralized, without requiring a central server. For these reasons, the Local-First paper [1] proposes utilizing CRDTs for Local-First software. The paper presents three case studies using CRDTs to create Local-First applications. All of these case studies were implemented with peer-to-peer CRDTs, using WebRTC [5] as a transport mechanism. The examples include a collaborative Trello-like Kanban board, a collaborative drawing application, and a collaborative media canvas. The exploratory results of the case studies show that CRDTs satisfy the ideals of Local-First, offering a great user experience through collaboration and offline interaction.

Like OT, CRDTs have been extensively used in the industry. CRDTs enable collaborative text editing in Juypter notebooks [25]. CRDTs are used to achieve conflict resolution under eventual consistency in databases such as Redis [26], Riak [27], and CosmosDB [28]. Other articles explore building applications on top of CRDT-based databases such as Riak [29]. A comprehensive overview CRDT-based applications and implementations is shown in *General-purpose CRDT libraries* [30].

Where Operational Transformation in academia and the industry seems to be primarily concentrated on collaborative text editing, CRDTs seem to be used in a wider array of applications in both academia and the industry. The use-case of providing eventual consistency for distributed databases is of special interest for building Offline-First software.

CRDTs are not without problems; operation-based CRDTs require an ever-growing state, presenting performance, memory and disk-space issues [1, 23, 31]. While optimizations such as history compression and truncation are being researched, CRDTs will always have an overhead due to their decentralized nature.

Another CRDT problem in literature is the preservation of user intent [32–35]. CRDTs can be mathematically proven to always converge, but that does not guarantee that the resulting state is correct according to the user's intent. General CRDTs do not know about the context of the changes made to it and, therefore, do not know how to optimally preserve user intent. For example, imagine a Last

[3]`https://etherpad.org`
[4]`https://share.github.io/sharedb/`
[5]`https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API`

Writer Wins CRDT, which keeps track of a number. If two users increment this number concurrently, one increment operation will be overwritten by the other. Thus, after convergence, the number only increases by one. This problem is solvable by using another CRDT: a counter CRDT [7], which allows for concurrent increments. CRDTs do not always preserve user intent due to their context-unaware, generic nature.

CRDTs pose problems if invariants need to be kept [23, 36–38]. For example, a general counter CRDT can not enforce an invariant, such as the counter cannot exceed the number 10. To enforce this invariant, a new CRDT has to be designed, for example, a bounded counter [38]. Some invariants are much more difficult to solve within CRDTs than others. If working with a tree data structure CRDT, two concurrent local operations in separate clients that re-parent a node can create a cycle [23, 33].

CRDTs are difficult to customize [31, 36, 39], due to the complexity of implementing custom CRDTs. Implementing one of the simplest CRDTs, such as a Last Write Wins register CRDT, has a surprising amount of commonly made mistakes [40]. Therefore, most users of CRDTs use implementations of CRDT libraries which handle the complexity of CRDT implementations. However, that also means that users are locked into this library and the CRDTs provided by this library. For example, the most popular JavaScript CRDT library Yjs[6], does not provide a counter CRDT. Therefore, users must implement it themselves or use a suboptimal workaround such as using a sequence CRDT of numbers to represent a counter [36]. "Most CRDTs are good for one particular problem, but if that's not the problem you have, they're hard to extend." [36]

Literature is conflicting regarding CRDT's capability to compose. Some sources successfully combine various CRDTs to create new data structures [23, 41, 42]. Other sources comment on the difficulty of composing CRDTs due to their coupled data and merge operation [39]. Butler comments on the increased difficulty of maintaining invariants as composed CRDTs can break invariants that are satisfied without composition.

In the Local-First paper, the problem of network communication is mentioned [1]. This is not specific to CRDTs, but this is a problem that occurs when using peer-to-peer networks. Peer-to-peer networks suffer from unreliable NAT traversal [43]. Furthermore, setting up a peer-to-peer WebRTC connection in the browser still requires a centralized server for signalling [44]. CRDTs are network-agnostic and do not have to be transported using peer-to-peer communication. They would also work in a client-server model, preventing the issues related to peer-to-peer. For Local-First software, not Offline-First, peer-to-peer is needed to fully realize the ideal of Longevity, allowing Local-First software to work without a centralized cloud server that users do not control and have ownership over. Therefore, the Local-First paper sees peer-to-peer as the logical end goal for Local-First. Butler remarks that browsers are inherently not peer-to-peer. Every website a browser connects to is located on a centralized server. Furthermore, with a centralized server, CRDTs are not needed at all to achieve replication and conflict resolution. Central servers can provide a global order, which allows all clients to apply operations in the same order. This is an alternate approach to ensure convergence, as opposed to CRDTs, which rely on commutativity for convergence. Therefore, using a central server for global ordering dismisses the need and, thus, the complexity of CRDTs. For this reason, Figma also does not use true CRDTs but uses a central server [23]. We discuss this method further in a later section.

The Local-First paper describes also the "closed-laptop" problem. This problem occurs in a peer-to-peer setting, where users collaborate without using a central server. When a user updates the state of the application on a laptop and then closes the laptop to resume work on a phone, the phone is not able to connect to the laptop to get the latest state. While the laptop and phone would still converge if they were to both reconnect at a later stage, it is not an acceptable user experience for the phone to miss the work on the laptop, even though they were both connected to the internet, just not at the same time. To solve this problem, the paper proposes "cloud peers," which are always-online servers that do not act as a central authority but as a place where the latest stage of the application can always be stored. The laptop can upload the latest state to this cloud peer and then shut off, while the phone can still see the changes made on the laptop.

To summarize, CRDTs are especially relevant in the context of Local-First and have been used in production software for many different use cases. There are challenges when using CRDTs however, and CRDTs may not be needed for centralized Offline-First software. The decentralized nature of CRDTs imposes an overhead in performance, disk-, and memory usage. Existing CRDTs do not always correctly preserve user intent, and pose challenges when invariants should be maintained under concurrency, which is more challenging when composing CRDTs together. When existing CRDT implementations do not fit a specific use case, it is difficult to extend or implement a custom CRDT due to implementation

[6]`https://github.com/yjs/yjs`

complexity and the difficulties of ensuring correctness and convergence.

The network topology CRDTs are designed for, peer-to-peer, also poses challenges. The peer-to-peer functionality in the browser, WebRTC, suffers from unreliable NAT traversal and the complexity of setting up new connections. Furthermore, peer-to-peer suffers from the "closed-laptop" problem, which is solved by using cloud peers, introducing some complexity and centralization. On the other hand, peer-to-peer networks might be needed to fully satisfy all local-first ideals.

Last Write Wins

As seen in the existing frameworks, last write wins is a popular conflict resolution mechanism. As we know, Last Write Wins is also a type of CRDT. However, there is a difference in how these frameworks use Last Write Wins and a Last Write Wins CRDT. Where Last Write Wins CRDTs use a commutative operation to resolve conflicts without centralization; these frameworks use a centralized server to decide on the final value. Using a centralized server, Last Write Wins conflict resolution can be simplified, removing the overhead required for a Last Write Wins CRDT.

Last Write Wins is a very popular conflict resolution mechanism in the industry, being used in production by popular Local-First and collaborative applications such Figma [23], Linear [17] and Notion [45]. Figma has written about their choice to use centralized LWW after first exploring an Operational Transformation-based and CRDT-based solution. Operational Transformation was discarded due to the inherent complexity and implementation difficulty. Figma also chose not to use CRDTs, as they use a centralized server and could thus simplify the solution by removing the overhead and complexity associated with CRDTs. In the end, Figma's conflict resolution is based on Last Write Wins, with the simplification that timestamps do not need to be stored, as the centralized server can define the order of events. Figma additionally highlights a problem we previously discussed with CRDTs: invariants. Recall the problem with a tree data structure CRDT, in which two concurrent reparenting operations create a cycle. Using a central authoritative server, the server can reject reparenting operations that cause cycles, enforcing the invariant.

Last Write Wins is one of the simplest conflict resolution mechanisms, both conceptually and implementation-wise. Simplicity is an important factor in building software, as highlighted by Figma [23]. However, the simplicity of Last Write Wins can also lead to problems when working offline or asynchronously [31, 45]. If two connected users concurrently change a property that is using LWW, they will quickly realise that another user is also changing this property, especially if the software shows the presence of users on a page. However, if one of the two users is offline, they will not notice this, and one user will lose their changes unintentionally. This may not be a big problem for smaller properties, such as a title of a todo item. However, this becomes a problem for Last Write Wins properties of a bigger size. Imagine a Google Document where the entire text of the document is stored in one Last Write Wins register; losing all edits is not an acceptable user experience.

Other literature provides evidence that this problem is possibly not as big as it seems. The phenomenon of "social locking" is described by [33]. Social locking means that people intuitively coordinate together to avoid conflicts. People often discuss who is working on what to avoid modifying the same document together. This occurrence was also observed in the Local-First paper [1], where users displayed social locking in the validation of the case studies. The authors found that users encountered conflicts when collaborating surprisingly rarely, providing evidence that last write wins and other generic conflict resolution mechanisms may work well enough in reality.

Locks

One way to deal with conflicts is to ensure there never are any conflicts. This approach is explored by Knill, which proposes using locking as a mechanism to ensure two concurrent users can never change the same data at the same time, therefore avoiding conflicts altogether. This approach is also used by Fan and Sun [46], which uses locking as a mechanism to enable collaborative programming. Some applications dealing with conflicts this way are Google Sheets[7], Google Slides[8], Miro[9] and Figma[10].

Implementing locking to enable collaboration requires some additional features to ensure a good user experience. Users need to be able to see whether they can edit a property or if another collaborator

[7]`https://sheets.google.com`
[8]`https://slides.google.com`
[9]`https://miro.com/`
[10]`https://figma.com`

currently locks it. Furthermore, locks must encompass as little data as needed to ensure other users are not locked out editing unnecessarily.

With these features implemented, there are still concerns over user experience. For example, if a user locks a property of a document and forgets to unlock it, other users are locked out unnecessarily [45].

Locking also requires a central server to keep track of the locks or a distributed consensus protocol such as Paxos [47] or Raft [48], which significantly increases complexity. When a user is offline, they will not be able to acquire locks and thus will never be able to make changes without regaining connectivity.

Manual Conflict Resolution

All conflict resolution methods we have reviewed are automatic conflict resolution. However, not all conflicts can be correctly resolved without user input. One example of a conflict that requires user input is a merge conflict in a version control such as Git[11]. Git will tell the user that there is a conflict that must be solved and allow the user to specify how it should be solved. Manual conflict resolution gives users much control over how a conflict should be solved, but manual resolution is also tedious. If automatic conflict resolution that correctly preserves user intent is possible, the software should not require the user to solve it themselves.

Manual conflict resolution is further discussed by Schiefer *et al.* [49]. A synchronization model "forking histories" is proposed. When a conflict cannot automatically be resolved while preserving user intent, the forking histories model will create a "fork" of the application state. Users can then decide when to manually resolve the conflict at a later point in time. A downside of this model is increased mental overhead for users. Users now have to be able to reason about multiple histories. Furthermore, application developers must also add support for forked histories in their database and create a UI that allows users to later merge histories.

Application-Specific Conflict Resolution

Bayou is an eventually consistent, replicated storage system designed for mobile computing [50]. Bayou is designed around application-specific conflict detection and resolution. The reasoning behind application-specific conflict resolution is that applications should be involved in conflict detection and resolution, as this depends on the application's semantics.

An example given in the Bayou paper is a meeting room scheduler application. In this application, users can reserve rooms at a specific time, even while offline. A reservation made offline is not guaranteed to be fulfilled, as it can conflict with reservations made by other users. Using a general-purpose conflict resolution mechanism such as Last Write Wins would not be appropriate for the semantics of this application, as users would then be able to override confirmed reservations.

Application-specific conflict resolution in Bayou is implemented through two techniques: dependency checks and merge procedures. Dependency checks allow an application to run a query and compare this against an expected result. If the result of the query is the expected result, there is no conflict. If this is not the case, a conflict is detected, and a merge procedure is executed. For the operation of reserving a room, the dependency check would be if there is already a reservation for that room that overlaps with the new reservation. In the merge procedure, applications can specify behaviour to resolve the conflict. For the reservation operation, the merge procedure can try reserving a specified alternative.

Furthermore, applications can use an "error log" to enable manual conflict resolution, if automatic conflict resolution can not succeed for a specific operation. If reserving the room detects a conflict, and no valid alternatives can be found in the resolution of this conflict, this failure can be reported to an error log. The application can then show this conflict to the user to gain manual input on how to resolve the issue.

Bayou is a framework that developers can extend to create custom conflict resolution that incorporates their application's semantics. Semantic conflict resolution, specifically for file systems, was already implemented in previous systems such as Locus [51], Fiscus [52], and Coda [53]. These systems show that semantic conflict resolution can be used over general-purpose conflict resolution to more effectively preserve user intent, and avoid having to perform manual conflict resolution.

The Global Sequence Protocol (GSP) is an abstract model for eventually consistent data replication [39]. GSP also makes use of application-specific conflict resolution but simplifies this process relative to Bayou. Instead of requiring explicit dependency checks and merge procedures, conflicts are resolved by creating a global order.

[11]`https://git-scm.com/`

The eventually consistent Dynamo database offers developers multiple options for conflict resolution [54]. Developers can opt into a generic resolution mechanism such as Last Write Wins, but applications can also decide to use custom application-specific conflict resolution, allowing for increased user experience. On the other hand, Dynamo states that developers may not want to write custom conflict resolution mechanisms, therefore allowing developers to easily opt into Last Write Wins. This hybrid approach offers developers flexibility when needed and offers simplicity by default. This hybrid approach is also implemented in Offline-First frameworks PowerSync and WatermelonDB.

CouchDB employs revision-based conflict resolution [55]. In this model, every conflict is resolved by choosing a deterministic arbitrary winner. CouchDB then flags the document as conflicting. The next time the application replicates with CouchDB, it can detect that a document is conflicting, review the two conflicting versions of the document, resolve the conflict in the application code, and upload the resolved version to CouchDB again. This strategy is also the strategy PouchDB uses, as PouchDB is just a CouchDB client, and the RxDB framework.

Bayou uses a peer-to-peer network topology. Therefore, the conflict resolution code is present in every client. GSP, SQLSync and Replicache use a client-server network topology. When using a client-server topology, an additional choice must be made of where to put the conflict-specific code. In the GSP model, this code is present on both the client and the server. In the client, the code is present to enable optimistic application of updates, so users get immediate feedback when making a change while the operation is sent to the server in the background. The server serves as an authoritative truth and must contain the application-specific conflict resolution code to apply the update authoritatively.

Application-specific conflict resolution offers more flexibility than other conflict resolution methods. However, implementation is less trivial. Application-specific conflict resolution requires a total order of updates. GSP, Replicache and SQLSync all depend on a central authoritative server to order updates. Bayou is designed to work fully peer-to-peer, using anti-entropy [56] to propagate updates to other servers. To define a total order of updates, a primary commit scheme [57] is used. In this scheme, one Bayou peer is designated as the primary peer, and the primary peer defines the total order.

Conflict Avoidance

We have explored locking as a method to avoid conflicts, but there is another approach. Conflicts only occur when concurrent writes to the same record are made in multiple data stores [58]. Thus, the application can only write to a single data store to avoid conflicts.

Conflict avoidance is used in the replication protocol of many relational databases [59–61]. In this protocol, one database is designated as the master database. One master database can have many replicated follower databases. All writes made to the database can only be accepted at the master database, while reads are available from any database. For relational databases, this improves redundancy, scalability and availability. For read operations, there is no single point of failure anymore. The database read-load can be spread out across follower databases to improve read performance and master write performance. However, unlike other distributed databases, if a single system cannot handle the throughput of write operations, a single master database will not work.

We have reviewed Figma's infrastructure, which uses Last Write Wins as a conflict resolution mechanism [23]. However, Figma also makes use of a single data store per Figma document, therefore there are not actually any conflicts as there is only one authoritative data store. Figma's Last Write Wins refers to how properties are overridden by later writes to the same property, however there are never two values for the same property at the same time, as there is only one authoritative data store. This is the case for the Figma server; however, on the Figma client, there can be conflicts. When a client changes a property from A to B, then receives a C from the server, there is a conflict between an unacknowledged B, and the authoritative C. In this case, Figma clients keep B as the property value because the client knows the server will soon override the authoritative C to B.

Figma shows how to use conflict avoidance on the server to avoid complexity of multiple authoritative data stores, while using conflict resolution on the clients to allow for optimistic mutations in the client. Two frameworks, Replicache and SQLSync also rely on a single authoritative server to avoid conflicts. However, unlike Figma the client also does not need any conflict resolution, but it still has optimistic writes.

These frameworks use server reconciliation, which is a technique commonly found in real-time multiplayer games [62, 63]. Server reconciliation is also designed for a client-server model, in which there is one authoritative server and several clients that follow that server. Server reconciliation is often combined with client-side prediction [64]. Client-side prediction is a term often used in multiplayer games, however in web development, the same concept can be found under the name optimistic UI. Server reconciliation and client-side prediction work as follows. When a client intends to do something, such as creating a new to-do item, it sends an operation to the server, which describes the change the client wants to make. While the client sends the operation to the server, the client optimistically applies the operation locally. It tries to predict what the authoritative server will do. In this case, the to-do item will thus be instantly renamed on the client. The server then receives the operation and applies it, returning the new application state to the client. The client then takes on the authoritative application state.

Replicache has written about the benefits of conflict avoidance by taking a replicated counter as an example and comparing this to how it would be implemented using a CRDT [36]. Using the most popular CRDT library Yjs[12], the recommended method of storing a counter is an array of numbers, which sum is the counter. This works but is not efficient or obvious for developers. As we have seen, implementing a counter using Last Write Wins is not possible, as there is no increase operation. Replicache then shows that with application-specific conflict resolution, creating a counter is trivial and obvious. Other conflict resolution mechanisms require the developer to have a mental model of conflicts that can occur and how to resolve them; server reconciliation does not require this, which is a big benefit.

3.4.2 Number of Leaders

We have reviewed many conflict resolution strategies and have already seen that an important property in the selection of these conflict resolution algorithms is the network topology.

Single-Leader

In a single-leader or single-master system, one node is designated as the leader [65]. This leader is solely responsible for accepting operations from all other nodes. The leader also propagates all updates to other nodes; the nodes are not all directly connected to every other node. Nodes that are not leaders are followers; these followers are read-only as writes must go through the authoritative leader.

The leader creates a total order of operations. With a total order of operations, conflicts are entirely avoided. This has many benefits, as discussed in subsection 3.4.1. Of all strategies, single-leader is the simplest to implement. Most relational databases use a single-leader architecture [59–61].

The single-leader architecture suffers from decreased availability. If nodes cannot connect to the leader, they cannot write new updates or receive updated information. If the amount of writes is bigger than a single system can handle, the architecture falls apart. Furthermore, outages of the network or the leader node itself will result in the outage of the entire system, which is why the Dynamo database opted for a decentralized system without a single leader [54].

In single-leader systems, follower nodes are read-only and thus cannot accept local or offline writes. This severely restricts offline application usage, as offline users can only read previously synchronized information but are not able to make updates.

Multi-Leader

A multi-leader system is an extension of a single-leader system. Multi-leader systems allow multiple nodes to act as leaders and thus accept writes; therefore, any leader node can still write to the data store while offline [65]. These writes, or operations, are then propagated to other leader nodes. As multi-leader systems can receive operations on multiple nodes, multi-leader systems gain increased availability. Besides offline usage, writes in a multi-leader system are also much faster, as writes are immediately applied locally and do not suffer from network latency.

All reviewed Local-First frameworks are multi-leader systems, as local and offline writes are required for the Local-First ideals of performance and offline capability. However, the complexity of multi-leader systems is significantly increased compared to single-leader systems. Multi-leader systems require operations to be ordered using a consensus protocol or require conflict resolution in order to converge to the same state eventually.

3.4.3 Operation Scheduling

We have seen that one way to deal with conflicts is to avoid conflicts altogether. In a single-leader system, all operations arrive at a single node, which can create an order of operations, avoiding the need for conflict resolution.

[12] `https://github.com/yjs/yjs`

In a multi-leader system, the order of operations is not defined. If all leaders apply operations in a different order, the state of the nodes could converge, and eventual consistency would not be achievable. Therefore, multi-leader systems need to decide on a common order of operations or use a conflict resolution mechanism like CRDTs, which guarantees convergence even when operations are not ordered. To settle on a common order of operations, multi-leader systems require consensus over what this order is.

One way to achieve consensus in an eventually consistent system is quorum-based consensus [66]. In a quorum-based consensus algorithm, a predetermined amount of nodes must agree on a decision before the decision is considered final. For example, a node can propose to add a new operation to the operation order. When the necessary amount of nodes agree, consensus is reached, and the new order is propagated to all other nodes. Quorum-based consensus can be reached using the Raft [48] or Paxos [47] consensus protocol. Examples of users of quorum-based consensus are the eventually consistent Dynamo database [54].

Another way to achieve consensus is to select one node as the primary, which is solely responsible for ordering operations. While this strategy sounds comparable to the single-leader architecture, in a single-leader architecture only one node can apply operations. In a multi-leader architecture with a primary commit architecture, all leaders can also tentatively apply operations locally.

Primary-based commit is a popular protocol in the industry, as it applies very well to the client-server model. In the client-server model, the server can act as the primary. This architecture is used by Figma [23], Linear [17] and Notion [45]. Offline-First frameworks Replicache, PowerSync, and SQLSync also use this commit scheme and a client-server network model.

Primary-based commit is also used in peer-to-peer topologies, such as the Bayou system [50]. Bayou discusses the reason for their choice of primary commit over a quorum-based two-phase commit protocol. In a case where data is replicated over laptops that are often disconnected (shutdown), requiring the majority of laptops to be connected poses issues.

As a primary-based commit relies on a single primary node, it suffers from the same problems as the single-leader architecture. When the primary node is not available, consensus cannot be reached. If the throughput of writes is larger than a single machine can handle, primary commit cannot be used.

3.4.4 Network Architecture

In the review of Offline-First frameworks, we have seen that all frameworks are client-server based, with one framework also supporting peer-to-peer.

Peer-to-Peer

Peer-to-peer sees less usage in Offline-First applications, however, Local-First applications are often designed around peer-to-peer. In the validation of the Local-First case studies, two problems related to the peer-to-peer network topology were found [1]. The first problem was NAT traversal, which proved unreliable depending on the router and network topology of the user. Lastly, the "closed-laptop" problem was discussed, which occurs when a user wants to resume work performed on a disconnected device on another newly connected device. To solve the closed laptop problem, the Local-First paper proposes a "cloud peer", which is a device or server which is always online, so there is always one peer with the latest state. To satisfy the ideal of longevity, managed cloud peers cannot be used, and users would have to set up these cloud peers themselves.

Another problem associated with a web application that has a network architecture without a centralized server is the volatility of browser storage. In a Local-First application where the primary copy of data is stored in the web browser, there is a risk of losing data. Browsers are not expected to store critical data, such as data which would be stored in Local-First applications. For example, safari clears browser data, such as data stored in IndexedDB, after 7 days of inactivity [67]. Due to this, data should still be stored in a persistent place [68], such as a backup in the cloud.

Some of these issues are specifically related to peer-to-peer software in the browser. The Bayou system [50] is entirely designed around a peer-to-peer network architecture with epidemic propagation. Issues such as the lack of durable persistent storage is not a problem in the Bayou system due to the storage being outside the browser. Other issues, such as the closed-laptop problem, are still present.

Client-Server

The peer-to-peer architecture without an authoritative server does not fit very well with a Software as a Service (SaaS) architecture, as centralized control is important to monetize the service. In the Local-First paper, some benefits of SaaS applications are highlighted: users can access their data from anywhere and on any device and collaborate with other users. Further benefits of SaaS software for businesses and customers are discussed by Waters. Among the benefits are lower internal IT requirements, reduced capital investment, faster implementation, and contractually guaranteed reliability and security. Customers do not have to be concerned about the installation, implementation, configuration, and maintenance of hardware and software.

The Local-First paper scopes the applications that the paper designs to only software for creating documents or files, as well as personal data repositories. Software such as "banking services, e-commerce, social networking, ride-sharing, or similar services" is explicitly not what the Local-First paper addresses, as these types of applications are better served by centralized systems.

Existing software that highlights many of the benefits of Local-First, such as Notion, Figma, and Linear, are all SaaS software and adopt a centralised client-server architecture. All existing frameworks are also designed to support a client-server network architecture.

Client-server web applications can still be decentralized. For example, decentralized software such as Mastodon[13] allows users to set up their own server, which provides a browser interface that is connected to the server using a client-server architecture.

3.5 Framework API Design

In this section we discuss several subjects related to the API design of the framework. We discuss literature related to Developer Experience. Then, we discuss literature that concerns type safety and static typing. Next, we discuss literature regarding full-stack frameworks. Finally, we examine sources discussing general software API design.

3.5.1 Developer Experience

User Experience refers to how users interact with and perceive products, systems and services. User Experience from a developer's point of view is called Developer Experience [70]. Developer Experience captures the feelings of a Developer when conducting activities within a working environment. Developers use a multitude of development tools, such as Integrated Development Environments (IDEs) and other tools that are used to create software, such as programming languages and software libraries [71]. Besides tools, Developer Experience is also affected by aspects such as development processes, working environment and team culture.

There are several reasons why Developer Experience is an important factor. Developer Experience is closely related to developer happiness and satisfaction [72]. Besides happiness and satisfaction, Developer happiness has been shown to positively affect productivity. In another study, the relation between Developer Experience and code bugginess was compared, showing that increased Developer Experience is correlated with reduced bugs [73]. Higher developer satisfaction has also been shown to increase the quality of products [74]. For companies, Developer Experience is also significantly related to developer retention and attraction [72, 75].

Multiple studies have investigated methods to improve Developer Experience. There are many methods related to the work environment to improve Developer Experience, however in this thesis, we will focus on Developer Experience related to tools. One method is to shorten the feedback loops of the tools [72, 74]. Developers use many different types of tools that deliver feedback for the developer, such as whether or not the code can compile, or whether the tests have succeeded and failed. Reducing the time to feedback allows developers to perform corrective actions when needed more rapidly. Shorter feedback loops reduce friction and interruptions; therefore, shortening feedback loops increases developer satisfaction and Developer Experience.

Due to software development's complexity, developers face a significant cognitive load. Developer tools may increase cognitive load, or reduce it. To reduce cognitive load, tools may help by removing unnecessary hurdles in the developer process [74]. Tools should be easy to use and streamline development. Reducing cognitive load further increases developer productivity and experience.

[13]`https://joinmastodon.org/`

The health of the codebase is another factor that is related to Developer Experience. A codebase of a higher quality, better maintainability, and ease of development impacts developer productivity and satisfaction positively [72].

As shown, developer tools such as IDEs and software libraries can significantly impact the developer experience. Therefore, libraries should be designed to improve it by reducing feedback loops, cognitive load and contributing to the quality of the codebase.

3.5.2 Type Safety

Type safety is the degree to which a program is free from type errors. A type-safe program does not contain operations that could possibly lead to runtime type errors during the execution of the program. In type-safe, or statically typed languages such as Java and C, type errors will be caught by a static type checker at compile time. If a program passes the static type checker, it is guaranteed to satisfy some set of safety properties at runtime.

Type safety helps prevent common programming errors, such as passing an integer to a function which expects a string. The benefits of static typing are discussed in many sources. Static typing helps developers detect errors earlier in the development process [76]. Therefore, static typing also shortens the feedback cycle of detecting errors, in turn improving Developer Experience. Static typing is also said to help developers read and understand code, as it serves as a form of documentation [77, 78]. Other literature shows that static typing is beneficial for the maintainability of software systems [79].

JavaScript is the programming language of the web. However, JavaScript is not a statically typed language, but a dynamically typed language. To bring the benefits of static typing to JavaScript, TypeScript was invented. TypeScript[14] compiles to JavaScript, and can therefore be used anywhere that JavaScript can be used. TypeScript is a very popular method to create modern web applications; in the annual Stack Overflow developer survey, TypeScript is the third most loved programming language [80]. Furthermore, in the State of JS report, 93% of TypeScript developers stated they would use TypeScript again.

One especially significant benefit of statically typed languages we have not yet discussed is the possibility of tight editor integration. Statically typed programs have a structure that allows code editors or IDEs to provide programmers with code completion but dynamically typed programs typically do not have enough information to provide robust code completion. Code completion allows developers to explore APIs without having to context switch to external documentation [81]. One source investigated the impact of code completion on developer productivity by comparing JavaScript to TypeScript with code completion, showing a marginal increase in productivity for TypeScript developers [82].

TypeScript also offers type inference, which allows developers to omit types in some positions, and having TypeScript automatically infer the corresponding type. This allows developers to gain the benefits of static typing without having to write additional code.

As discussed, static typing and TypeScript offer many benefits. Regarding Developer Experience, static typing improves maintainability, impacting satisfaction and productivity. Static typing shortens the feedback loop to detect errors earlier in the process, improving satisfaction. Finally, static typing has a tight integration with the editor, improving Developer Experience through code completion.

3.5.3 Full-Stack Frameworks

The NodeJS[15] JavaScript runtime allows for the execution of JavaScript outside a web browser. This possibility of running JavaScript on the backend and frontend offers interesting benefits and opportunities

The benefits of full-stack JavaScript have been discussed in some literature. Full-stack Javascript allows developers to use one programming language on the front and backend. This allows developers to reuse code and knowledge between the client and the server[83]. Developers do not have to context switch between different languages, reducing mental overhead.

The possibilities of full-stack JavaScript have fostered a wave of new frameworks: Next.js[16], Remix[17], Astro[18], SvelteKit[19], TRPC[20] and others. These frameworks consolidate the client and server, allowing

[14]`https://www.typescriptlang.org/`
[15]`https://nodejs.org/en`
[16]`https://nextjs.org/`
[17]`https://remix.run/`
[18]`https://astro.build/`
[19]`https://kit.svelte.dev/`
[20]`https://trpc.io/`

developers to write comprehensive web applications more efficiently.

Full stack JavaScript already offers significant benefits, however, full-stack TypeScript expands on those benefits even further. Full-stack TypeScript enables end-to-end type safety, which ensures type safety across the network boundary. In full-stack projects without a common language for client and server, type safety is often only guaranteed within a single environment. Sending data across the network required manually writing code to handle serialization and deserialization, leading to possible errors due to inconsistencies between the client and the server. In full-stack TypeScript, frameworks streamline this process, providing mechanisms for type-safe data transmission between the client and the server. Reducing potential runtime errors and improving efficiency by eliminating the need to manually write code responsible for sending data over the network.

End-to-end type safety improves Developer Experience by simplifying networking code and bringing type safety to a place where it was traditionally absent: the client-server network boundary.

3.5.4 API Design

When creating a framework, the design of the framework API is important. API design affects API users in many ways [84]. It has a big impact on developers but can also affect end-users. API design affects multiple quality attributes; it affects the usability of the design, which is composed of learnability, productivity, error-prevention, simplicity, consistency and matching mental models [84, 85]. Furthermore, API design involves API power; this aspect consists of expressiveness, extensibility, evolvability, performance and robustness [84]. The importance of API design is reiterated by Bloch [86].

Many sources present case studies on specific API design patterns, which are helpful when debating this specific pattern, but too specific to be generally applicable. Other sources offer general API design guidelines and suggestions to guide API designers in their task [86, 87]. These sources are often in the form of expert opinion and thus may not be grounded in evidence; however, they may serve as more general recommendations.

An important API design consideration in TypeScript libraries seems to be designing for type inference [88, 89]. Type inference allows developers to omit type annotations that can be calculated from program context while preserving type safety. By using type inference, developers can write more concise code. Type inference reduces cumbersome maintenance of type signatures and unnecessary information in the code. To achieve type inference in TypeScript, generics are a powerful tool [88].

3.6 Gaps in Related Work

We have reviewed and discussed many existing Offline-First frameworks. In the literature review, we have discussed many technologies and algorithms these frameworks use. Despite the large number of existing frameworks and the features they offer, several gaps remain unaddressed:

Most frameworks provide generic conflict resolution strategies such as Last Write Wins conflict resolution or CRDTs. This limitation significantly reduces the types of applications that can be created with this framework. Some applications require more context in order to handle conflicts while preserving user intent. For example, the meeting room scheduler application could not be created using frameworks that offer only generic conflict resolution. This is an important limitation that should be addressed in the novel framework.

The majority of existing frameworks use document storage as a data store for the application. As seen in the literature, many real-world applications have a relational data model. Using document storage can pose challenges and friction when used with a relational data model that includes complex relations. Document stores do not offer the same level of support for enforcing relational integrity and querying relational data. The new framework should address this gap by integrating a data store that natively supports relational data models, streamlining the use of complex relationships and reducing friction for developers working with relational data.

An important factor of developer experience is type safety. Ensuring type safety can prevent runtime errors and improve the reliablity of the application. Furthermore, type safety improves developer experience through better integration with the IDE and autocomplete. The absence of type safety in existing Offline-First frameworks is another gap that should be addressed.

Finally, some existing frameworks are tightly coupled to a backend service provided by the framework, and a corresponding database that is required to be used with the framework. This reduces the flexibility for developers to choose the optimal database for their context. Furthermore, these backend services offer only limited customization, which results in constraints on how developers can tailor the backend to the application's specific needs. On the other hand, some frameworks allow for the use of an arbitrary backend, which conforms to a specific protocol. This approach offers greater flexibility by allowing developers to use a custom backend and gives developers ultimate control over the backend. However, this approach also requires a significant effort from developers to implement the custom backend. To address these gaps, a balance between flexibility and ease of implementation should be offered.

In conclusion, we address several gaps in existing work. The proposed framework will offer application-specific conflict resolution, integrate native support for relations within the data model, and ensure type safety to enhance reliability and developer experience. Additionally, the framework will allow the use of custom backends with minimal effort while avoiding the constraints of tightly coupled services, balancing flexibility and ease of implementation.

Chapter 4

Design

In this chapter, we present and discuss the design for the Relic framework. The chapter is split up into four sections: one section discussing the user stories and requirements, one section specifying the design of the frameworks' storage management, one section presenting the replication system of the framework, and the final section presenting the API of the framework.

4.1 User Stories & Requirements

The goal of the framework is to offer developers a system that can be used to create Offline-First applications. To make sure the framework addresses the needs of end-users, we will extract the requirements of the framework from user stories.

4.1.1 Offline-First

The framework has two main types of end-users: developers building applications using the framework, and users using applications created with the framework. User stories related to the application users are primarily related to the Offline-First paradigm; these are:

- **As a** user
  I want to read and write data when the network is inconsistent or unavailable
  So that I can continue working in all environments without interruptions.
- **As a** user
  I want my interactions with the application to be fast and consistent
  So that my experience is smooth and responsive.
- **As a** user
  I want to collaborate with other users in real-time
  So that I can efficiently work together simultaneously.
- **As a** user
  I want to access my work from multiple devices
  So that I can seamlessly switch between devices and continue my work.

From these user stories, we extract the following requirements: The framework must

- **R1:** Allow for offline read and write operations when the network is inconsistent or unavailable
- **R2:** Handle concurrent updates with built-in flexible conflict resolution
- **R3:** Guarantee eventual consistency
- **R4:** Synchronize offline changes when the network connection is recovered
- **R5:** Have consistent, responsive interactions which are not limited by network latency
- **R6:** Allow for seamless multi-device access
- **R7:** Allow multiple users to do work simultaneously and see updates in realtime
- **R8:** Be able to handle a large amount of data

4.1.2 Developer Experience

In the related work section, we have discussed the importance of Developer Experience, and its benefits regarding developer satisfaction and performance. The following user stories are related to providing developers with a good Developer Experience:

- **As a** developer
  I want the framework to enforce type safety
  So that I avoid runtime errors, improve maintainability and experience great editor integration.
- **As a** developer
  I want the framework to support type inference
  So that I do not have to specify types myself, reducing repetition and increasing readability.

These user stories form the following requirements: The framework must

- **R9:** Enforce type safety on the client and server
- **R10:** Offer end-to-end type safety between the client and server, spanning the network boundary
- **R11:** Be full stack, covering the client and the server with one language
- **R12:** Be designed to allow for type inference where possible

4.1.3 Modularity & Flexibility

In the existing Offline-First frameworks, we have seen that several frameworks create restrictions on the backend infrastructure. Many frameworks are tightly integrated with a specific database for the backend. While this allows the framework to optimize for this specific database, it poses a big restriction on the developer, which now must use that database in order to use the framework. Many frameworks also supply an existing backend service for the developer to run, this means the developer does not have to write their own backend, which can save time and reduce the time to get started. On the other hand, this greatly reduces flexibility. This results in frameworks where developers have to write permissions in a SQL Domain Specific Language[1], situations where the subset of data to be synchronized is specified in YAML[2], or situations where conflict resolution cannot be customized to fit the needs of the application[3]. Therefore, our final set of user stories are related to modularity. These are:

- **As a** developer
  I want the flexibility to decide which database I use
  So that I can pick the best technology for my application's and team's needs.
- **As a** developer
  I want to have control over my backend
  So that my application logic is not limited by the framework.

Which give us the final requirements for the framework: The framework must

- **R13:** Allow developers to determine their own type of backend database
- **R14:** Give developers the power to write their own flexible backend

4.2 Storage

In this section, we will discuss the design of the Offline-First storage solution on client devices. We have stated several requirements which relate to the storage solution:

- The framework must be able to handle a large amount of data
- The framework must be type-safe

These requirements relate to the first research question: What is an effective way to structure, store and query large amounts of data in the frontend that supports many different types of applications?

[1]`https://electric-sql.com/docs/usage/data-modelling/permissions`
[2]`https://docs.powersync.com/usage/sync-rules/example-global-data`
[3]`https://www.mongodb.com/docs/atlas/app-services/sync/details/conflict-resolution/#custom-conflict-resolution`

4.2.1 Related Work Summary

In reviewing the existing frameworks, we have seen several methods of tackling this issue. Most frameworks store data in a document store, other frameworks use SQLite, and one framework uses a graph store. In the literature review, we have also discussed Linear, which uses a traditional state manager in place of a database.

While document databases are prevalent in the existing frameworks, a large portion of the literature review does not focus on document databases, instead focussing on SQL and graph databases. A recurring theme of literature was the importance of relations. Many data models include relations, however, document databases do not inherently include support for relations. Therefore, the inability to use relations in document databases poses a large downside. On the other hand, document databases are typically schemaless. Not requiring a schema is seen as an advantage in some literature.

The literature discusses the benefits of SQL and SQLite. SQL is a powerful query language, and therefore provides developers with a lot of power. Besides the benefits of SQL, the advantages of SQLite have also been discussed, such as indexes, constraints, triggers, constraints, foreign keys, full-text search and automatic query optimization.

A recurring theme in literature was the disadvantages of SQL and the technical negative aspects of using SQLite in the browser. Regarding SQL, SQL is not able to produce tree-shaped results, which graph-based query languages are able to do. Furthermore, querying relations requires verbose syntax and fragments of queries are not easily reused. Finally, SQL is not type safe and thus error prone. Using SQLite in the browser requires the use of WASM. These binaries are around 400kb in size compressed, and thus use a lot of bandwidth. Finally, SQL requires a schema to use, which can result in a longer time to get started.

Graph databases were proposed to solve the downsides of the SQL query language, and can be implemented in native JavaScript, therefore circumventing the use of WASM, and the associated issues. Graph databases have support for relations, and unlike SQL, recursive nested relations. Literature reviewed the use of the Datalog query language, however found that it presents a high learning curve, and does not translate well to JavaScript objects. Therefore, the proposed solution is writing a new graph query language that can be efficiently represented in JavaScript.

4.2.2 Other Insights

While document databases do not inherently support relations, and therefore features such as joins typically do not exist in document databases. On the other hand, joins can still be made in the application code. A wrapper could be built around a document database to provide such features. While some sources see the schemaless nature of document databases as an upside, a requirement for the storage solution is that it must be type-safe. To be type-safe, the types of the document storage must be specified, therefore, a schema is required. This does not disqualify a document database as a solution, as it is possible to create a schemafull model around a schemaless document database. This is shown by the Mongoose[4] library, which extends the MongoDB database with a schema.

The literature exploring graph databases is based on three benefits that are not present in SQL with SQLite: type safety, recursive nested relations, and a more lightweight implementation that does not require a large WASM binary. On the other hand, the standard query language Datalog was found not to be a good fit. Therefore, another query language would have to be designed, and implemented by the framework. Ideally, this language would support everything SQL supports, such as GROUP BY, JOINS, aggregations such as COUNT and SUM, and furthermore, recursive JOINS. This is not a trivial task, furthermore, one of the biggest benefits of SQL is the ubiquitousness of the language. Many users are already experienced with SQL, having to learn a new query language specifically for the framework is a big burden and would significantly reduce the ease-to-use of the framework. This problem is also present when extending document databases with relations, as this also requires a new query language.

Using SQL would reduce the barrier of entry to the framework, and is furthermore a very flexible query language, creating a query language as flexible as SQL is not a trivial task. On the other hand, several issues have been found with SQL. The inability to produce tree-shaped results, which graph-based languages can do. The verbose syntax for relations, the problem of reusing fragments of queries, and the absence of type safety. However, none of the literature sources proposed using an Object Relational Mapper (ORM) or query builder around SQL.

An ORM would solve several problems, ORMs can simplify verbose syntax for relations, ORMs allow

[4]`https://mongoosejs.com/`

developers to reuse fragments of queries, as queries are specified in JavaScript. Finally, an ORM would solve the problem of SQL's lack of type safety. There are some downsides to ORMs, not every user that knows SQL would know the ORM. Therefore it has some of the same problems of designing new query languages. However, an ORM or especially query builders are typically closely aligned to SQL, and thus should not pose such a high barrier of entry for developers that know SQL but do not know the ORM. Unfortunately, ORMs do not solve the problem of recursive nested relations. However, this is a problem that might be tolerated. In the cases where recursive nested relations are needed, they can still be made in application code.

The literature presented many benefits of SQLite: indexes, constraints, triggers, constraints, foreign keys, full-text search and automatic query optimization. The major downside of SQLite was the 400kb WASM binary that increases the bundle of the website. Bundle sizes are often a major concern when creating websites. For example, some companies reported seeing a 1% increase in conversion rate for every 100ms the website loads faster [90]. On the other hand, these websites are not the intended use-case for Offline-First applications, Offline-First targets typically Long-Lived web applications. Offline-First is more concerned with reducing latency than reducing the bandwidth by 400kb. The first startup time of an application is increased, such that every subsequent interaction is significantly faster. For this reason, the 400kb increase in bandwidth might not be as big as a problem as the authors feared. Furthermore, after loading the WASM binary once, it can be cached by the browser for subsequent navigation to the website.

Finally, the Riffle framework presents reactive relational queries. Reactive queries fit perfectly in the interactive nature of modern frontend frameworks such as React[5], Vue[6], Solid[7] and others. When a change to the database is made, all queries which are currently displayed in the frontend should be invalidated, to always show the latest information.

4.2.3 Solution

We present our solution to the storage requirements and the first research question: What is an effective way to structure, store and query large amounts of data in the frontend that supports many different types of applications?

SQLite is designed and proven to store large amounts of data. It provides many features that empower developers. The familiar SQL query language allows developers to effectively and flexibly query data. The relational design of SQL allows SQLite to support many different types of applications that have relations in their data schema. While some applications which use recursive nested relations are not optimally supported by SQL, they can still implement recursive nested relations through application code. The inclusion of an ORM will satisfy the requirement of type safety and make working with the database more effective. Finally, a reactive relational architecture will allow for real-time updates in the UI.

4.3 Replication

We will now discuss the design of the solution to provide eventually consistent replication. Requirements that relate to this section are:
The framework must

- Allow for offline read and write operations when the network is inconsistent or unavailable
- Have consistent, fast interactions which are not limited by network latency
- Allow for multi-device access so work is not limited by one device
- Allow multiple users to do work simultaneously and see updates in real-time
- Be able to handle a large amount of data
- Be full stack, covering the client and the server with one language
- Be type-safe
- Offer end-to-end type safety, spanning the network boundary
- Be designed to allow for type inference where possible
- Allow developers to determine their own type of backend database
- Give developers the power to write their own flexible backend

[5]`https://react.dev/`
[6]`https://vuejs.org/`
[7]`https://www.solidjs.com/`

With these requirements, we provide an answer to the first research question: How to efficiently synchronize updates between clients while ensuring eventual consistency of state and minimizing architectural and coding constraints on the application?

4.3.1 Number of Leaders

In the literature review, we have seen two types of systems, single-leader and multi-leader systems. Single-leader systems are simpler to implement and are widely used by applications such as relational databases. However, in a single-leader system, only the leader can process write operations. This is not an acceptable user experience and does not satisfy the requirements of the framework. For this reason, the framework will use a multi-leader architecture. Implementation-wise, the complexity is increased. However, this complexity does not necessarily have to leak to the developer using the framework, as the framework could deal with the complexity and thus hide it from the developer.

4.3.2 Network Architecture

When reviewing the network architecture for Offline-First frameworks, applications and related literature, we have seen two primary architectures: peer-to-peer and client-server.

In Local-First (not Offline-First) applications, the focus lies within moving ownership from cloud companies to the client, therefore, Local-First applications are best suited with a decentralized peer-to-peer architecture. However, we have seen several challenges in peer-to-peer systems, NAT traversal, the ”closed-laptop” problem, the volatile nature of browser storage. Besides this, some applications are unsuited to use a Local-First architecture: ”banking services, e-commerce, social networking, ride-sharing, or similar services”. Many applications work best in a centralized environment. Furthermore, we have seen that centralized architectures are often simpler to implement and reason about than decentralized architectures. Finally, decentralization also has a big impact on software business models. SaaS applications are typically client-server so the business can use subscriptions to monetize the service.

This thesis does not aim to solve the issues Local-First presents. We require that the framework can be used to create many types of applications, not only decentralized applications. Furthermore, peer-to-peer presents issues, and especially peer-to-peer in a browser environment. Business-wise, peer-to-peer architectures impose a significant impact. Finally, most developers are already accustomed to a client-server model, thus creating a peer-to-peer framework will increase the barrier to entry. For these reasons, the framework will use a client-server network architecture.

4.3.3 Conflict Resolution

One of the most impactful choices a framework has to make is the conflict resolution algorithm. This algorithm has a large impact on how developers implement applications using the framework and can also significantly impact the end-user experience if conflicts are not resolved according to user intent.

In the existing framework and literature review, we have seen a multitude of different techniques to tackle conflict resolution. We will now discuss these techniques.

The conceptually simplest conflict resolution mechanism is manual conflict resolution. In this model, the user specifies exactly how conflicts are resolved, therefore user intent is optimally preserved. However, technically, creating an interface that users can understand and use to perform conflict resolution is challenging and time-consuming. More importantly, having users always manually merge conflicts that could also be solved automatically while preserving user intent results in a very bad user experience. Therefore, relying only on manual conflict resolution is not an option.

We have also discussed locking as a way to avoid conflicts, in which case conflicts never occur, making conflict resolution unnecessary. While many applications use locking in production, the user experience can suffer. Critically, locking also does not allow offline users to perform write operations; thus, this solution would not meet the requirements.

In the concurrent text editing domain, one of the most used algorithms is Operational Transformation. It has been used in production by the most popular text-editing applications, and has also seen use outside text-editing. Operational Transformation is a complicated algorithm, implementing Operational Tranformation correctly is difficult and error-prone. Operational Transformation has been implemented for data such as JSON types, which gives developers a way to use OT without having to implement and prove the correctness themselves. However, with this type of implementation, only a predefined set of general operations is possible. A general Operational Transformation implementation for arbitrary data types can therefore not support application-specific conflict resolution. If application-specific conflict resolution is needed for an application, developers would have to implement the OT operations themselves. The complexity of OT would leak to developers, which would mean that developers would then have to understand and reason about Operational Transformation. Therefore, creating a framework around Operational Transformation requires general-purpose conflict resolution. Otherwise, it would not alleviate the burden of OT of developers.

Another candidate for conflict resolution are CRDTs. CRDTs have properties which allow CRDTs to ensure there never are any conflicts. CRDTs are designed for peer-to-peer systems, and thus have been proposed to be the conflict resolution mechanism for Local-First (not Offline-First) software, as they work fully decentralized. Because CRDTs are designed for decentralization, they impose a performance, disk- and memory usage overhead. As the framework will use a client-server architecture, this overhead which is only present because of the decentralized peer-to-peer design of CRDTs would be theoretically unnecessary for the framework. Implementing CRDTs correctly and efficiently is difficult. For this reason there are many CRDT libraries which offer implementations of popular CRDT data types. However, these CRDT data types are general-purpose. Therefore, like Operational Transformation, these general-purpose data types cannot be used for application-specific conflict resolution. And implementing custom CRDTs requires users of the framework, and developers to understand CRDTs and ensure their correctness and convergence. Additionally, CRDTs pose challenges when the application requires certain invariants to be upheld, as there is no central authority to reject invariants.

A big topic regarding conflict resolution in the literature and existing frameworks was to keep it simple. Many frameworks provide Last Write Wins conflict resolution. Together with a centralized server, Last Write Wins conflict resolution is especially simple to implement, as the last value sent to the server can override previous values. Last Write Wins conflict resolution is one of the simplest conflict resolution mechanisms. This can present issues in some situations due to not correctly preserving user intent. On the other hand, literature describes that this does not often present a problem in reality. The phenomenon of "social locking" means that people intuitively coordinate together in order to avoid conflicts. Therefore, a simple Last Write Wins conflict resolution, which does not preserve user intent in some cases, may be acceptable for many applications.

However, using Last Write Wins as a general-purpose conflict resolution algorithm is not possible in all applications. The Bayou paper presents a use case where general conflict resolution algorithms do not suffice: a meeting room scheduler where users can reserve rooms while offline. When a user reserves a room, the application needs to check if the time and room overlap with another reservation. This specific logic cannot be implemented using Last Write Wins, and other general-purpose conflict resolution algorithms, such as specific OT and CRDT implementations, also struggle with this problem. For this problem, we need application-specific conflict resolution. Supporting application-specific conflict resolution allows the framework to be adapted to more types of applications that cover advanced use cases, such as the meeting room scheduler.

One method for application-specific conflict resolution involves storing both the conflicting documents in the database, this is revision-based conflict resolution. When reading data, the database returns the two conflicting versions of the document; the application can then choose which one is the winning version, resolving the conflict. Alternatively, the server can return a conflict when writing data that has been updated concurrently, in which case the client can resolve the conflict before writing to the server. Revision-based conflict resolution gives developers the flexibility of application-specific conflict resolution. However, note that conflict detection in this mode is limited, as conflicts are only detected per single document. In the case of the room scheduler, two reservations are stored in different documents but still conflict. No conflict would be found using revision-based conflict resolution.

The final conflict resolution mechanism discussed in the review is conflict avoidance. By maintaining a global order of operations, conflicts are essentially avoided. A global order can be created in a peer-to-peer architecture by using a consensus algorithm, such as majority quorum-based consensus, where the majority of nodes must be available to make a decision, or by primary-based consensus, where a single node is appointed as the authority to define the order. In a client-server architecture, maintaining a global order can be achieved by using a single server to define the order, much like primary-based consensus in a peer-to-peer system. The easiest way to define an order on the server is to order the operations by time of arrival.

Clients can send operations to the primary, which will order the operations and return the order to the clients. However, if the client cannot connect to the primary, there is no authoritative operations order of new operations, thus the client does not know what order the new operation should have. Furthermore, we want the operations to be applied on the client without waiting for network latency of the order of the operations. To enable this, we can use optimistic operations (also called speculative or tentative operations). The client can immediately apply the operation locally, while asynchronously sending the operation to the server. The server then returns the actual order to the client; for example, another node could have created an operation before the client's operation. The client then rolls back the optimistically applied operations, and applies them in the order received from the primary. Bayou uses this technique, Bayou is based on a peer-to-peer system.

Clients send operations to the primary, and receive back the order of operations. These operations must be deterministic so every client applies operations the same and the state of each client converges. In a decentralized system such as Bayou's peer-to-peer system, this makes sense. In a client-server system, the idea of deterministic operations to ensure convergence is a big requirement, however. Alternatively, clients can send operations to the server, and instead of the server responding with an operation order, the server can respond with the new state of the client. This technique is frequently used by real-time games such as Quake under the name client-side predication and server reconciliation. Furthermore, it is used in Offline-First frameworks such as Replicache and SQLSync. This means that operations can be implemented differently on the client and server, while the state of the client will always converge as the state is authoritatively decided by the server.

Looking back at the meeting room scheduler, we can now implement this application. The server receives all reservation operations in order of arrival, when processing each operation, it can check whether there is already a conflicting reservation in the database. If this is not the case, the server responds back the new state, which includes the new reservation. If the server detects a conflict when processing the operation, the server decides not to store the reservation in the state, and returns a state to the client in which the reservation is not included. Interestingly, when not performing conflict detection, such as simply updating the name of one reservation, this technique provides Last Write Wins semantics by default.

4.3.4 Solution

We now answer the second research question: How to efficiently synchronize updates between clients while ensuring eventual consistency of state and minimizing architectural and coding constraints on the application?

The replication solution involves several design decisions. The first design decision was the number of leaders. To satisfy the requirements, the framework should support multi-leader replication as single-leader replication does not support offline operation. Furthermore, the framework will use a traditional client-server architecture. Finally, we have seen that general-purpose conflict resolution solutions imply constraints on the application. Letting developers implement custom CRDTs or OT requires developers to understand these techniques, requires complex implementation for the developer and complicated proof of correctness and convergence. To avoid these issues, we propose server reconciliation with client-side prediction. This solution fits the client-server architecture well and can thus avoid the overhead associated with CRDTs. By having the server respond with state, the developer does not need to worry about convergence. Furthermore, it has been shown that the solution is very flexible regarding application-specific conflict resolution. This solution answers our research question and satisfies the requirements.

4.4 System Architecture

We now present the overall system architecture of applications created using the framework. The architecture is presented in Figure 4.1.

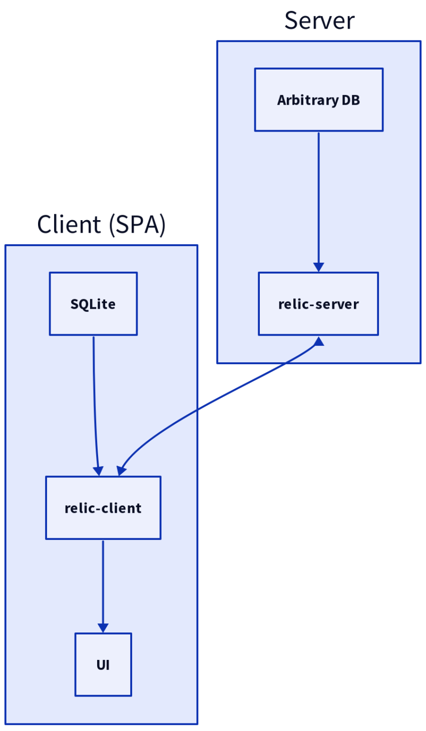


Figure 4.1: Overall system architecture of applications created using the Relic framework

As chosen, the architecture is a classical client-server architecture. Multiple clients are connected to the server.

Clients use a Single-Page Application architecture (SPA). The SPA model fits Offline-First applications more than a Multi-Page Application (MPA) architecture. In the MPA architecture, every page is rendered by the server. Thus, when offline, an MPA application cannot load new pages, as the server is responsible for rendering the pages. In a Single-Page Application, pages are rendered locally and can thus function offline after loading the full initial application once.

The client consists of three main components: the SQLite database, the relic-client and the user interface. The relic-client consists of the framework runtime, together with application-specific code written by the developer. The relic-client wraps the SQLite database, and enables reactivity so the UI updates when a mutation is executed. In the relic-client, the developer has specified the SQLite schema, and implemented mutations which are used to write to the SQLite database.

On the server-side, the relic-server again consists of the framework runtime and application specific code. Besides the relic-server, the developer can connect an arbitrary database. The relic-client and relic-server run the replication algorithm over the network. Like the relic-client, the server contains the application-specific code written by the developer to run mutations, and to select the data that clients may access from the database.

4.5 Replication Algorithm

In this section, we describe the replication algorithm, describing how the client and server interact in order to achieve eventual consistency. There is not one main implementation or definition of server reconciliation and client-side prediction. The design chosen is inspired by several sources of related work using these techniques. These related works are the Replicache framework [91], the SQLSync framework [13], the Global Sequence Protocol [39], the Fast-Paced Multiplayer article Gambetta and the Quake3 network protocol [63].

First, we will define some terms used by the replication. It is important that these terms are chosen thoughtfully, as they will also be used in the API exposed to the developer. The server reconciliation and client-side prediction heavily revolves around write operations. In popular modern libraries such as Tanstack Query and TRPC, these are often called mutations. This term is also used by the Replicache framework. We will adopt this term to describe write operations, as developers will be familiar with the terminology. This will help achieve a shared mental-model, which improves Developer Experience.

The Global Sequence Protocol and Replicache split synchronization up in two parts, synchronizing the client state with the latest state of server, and writing data, or mutating data. Splitting this process up in two parts is beneficial, as both parts may want to be done in different stages of replication. The act of synchronizing state is called a ”pull”, while the act of writing mutations to the server is a ”push”. This terminology mirrors the Git VCS, which uses the two terms to achieve the same.

We will now describe the behaviour of some processes of replication. First, we describe what happens when an entirely new client connects to the server. We then describe the process of read operations (queries) and write operations (mutations), both available without involving the server. Finally, the pull and push processes relate to synchronizing with the server. Push synchronizes changes from the client to the server. Pull synchronizes changes from the server to the client.

4.5.1 Initial Connection

The initial connection is the simplest behaviour of the system. It is presented in Figure 4.2.

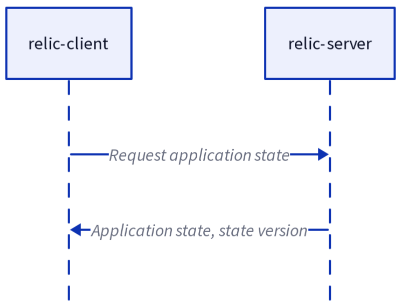


Figure 4.2: The behaviour of the client and server when a client first connects to a server

When a Relic client is first started, it sends a request to the server to get the authoritative application state. The relic server then runs some code written by the developer; this code is responsible for retrieving all information that the client is authorized to see. This information could be very large. The server sends this information back to the client, and with this information, the version of this state. This version will be used in subsequent pulls. When receiving this information, the client stores the state, and the version of the state in the SQLite database. The client can now make queries to its local state.

4.5.2 Queries

After the initial connection, the database on the client is populated and the client is now able to make queries and execute mutations locally. In this process, the server does not have to be involved at all, therefore avoiding any network latency on queries and mutations. Importantly, developers must also be able to make subscriptions to queries. When the underlying data that the query has returned changes, the subscription must be fired. This will provide reactivity in modern UI frameworks based on reactivity.

4.5.3 Mutations

Write operations, or mutations, can also be performed entirely locally. First, we describe how mutations are represented in the system.

We define a mutation as a function to be called and the arguments to that function. For example, a mutation could target the function "add-todo" with the arguments "name": "Finish thesis" .

The mutation defines the action to be performed. To perform this action, two steps are involved. First, the mutation is executed optimistically on the client. This step is crucial, as it shows the user the result of the mutation instantly. Besides executing the mutation optimistically, the mutation is stored in a mutation queue in the relic-client. This mutation queue is later used in the push operation to synchronize locally applied mutations with the server. If the user is offline for a few days, the mutation queue may store many operations, which must all be synchronized with the server.

4.5.4 Push

The process of synchronizing mutations with the server is presented in figure 4.3.

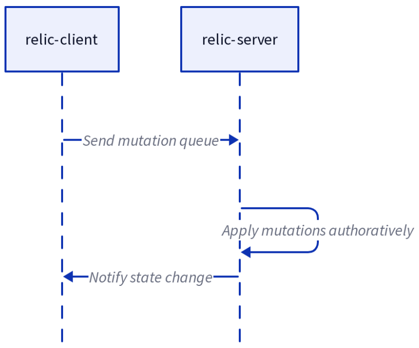


Figure 4.3: The push process that is responsible for synchronizing local changes with the server

The client sends its entire mutation queue to the server. The server then applies all mutations authoritatively. After the mutations are applied on the server, the server notifies all clients, not only the client that invoked the mutation, that the authoritative state has changed. When receiving this notification, the clients invoke the pull procedure to retrieve the updated state.

4.5.5 Pull

The pull procedure allows an existing client to get up to date with the server state. The server state may change when that client has executed a push operation or when another client performs a pull operation that changes the state the two clients share. Additionally, the server state may also be changed by the server itself, not initiated by a push from a client. The pull process is presented in figure 4.4.

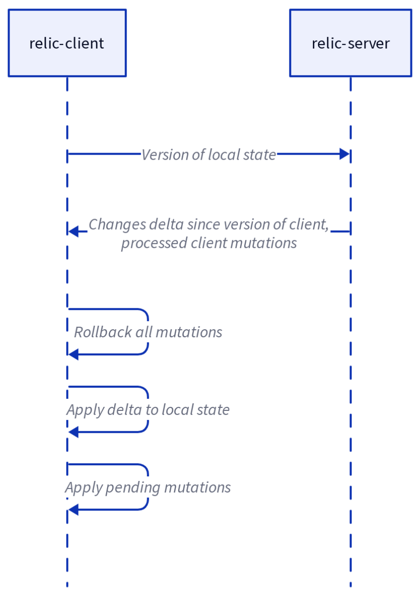

relic-client
relic-server
Version of local state
Changes delta since version of client, processed client mutations
Rollback all mutations
Apply delta to local state
Apply pending mutations

Figure 4.4: The pull process which allows a client to converge with the server state

After the initial connection, clients already have a large part of the state. Therefore, the server only sends the difference between the state of the server and the client. This delta message allows the client to converge to the server state without having to re-download the entire server state. This highly reduces the required network bandwidth for each pull.

The process starts with the client sending the version of the local state to the server. With this version, the server will calculate a delta between the client and server state. The server will also send the client the mutations of that client that have been processed by the server.

Upon receiving the delta from the server, the client rolls back all mutations that have been optimistically applied locally. Then, the client is able to apply the delta to the local state, after which the client is up to speed with the server state. The client may also have applied mutations locally, that have not been processed by the server yet. These mutations are re-applied to the local state so no mutations are lost, ensuring the user does not observe synchronization artefacts.

With the pull and push processes, we can now replicate data between clients and the server.

4.6 Framework API

This section provides an overview of the framework's storage and replication aspects and presents its developer-facing API. We will first define the terminology used by the framework. Then, we introduce building blocks that developers use to build Offline-First applications. Finally, we present the overall framework architecture.

4.6.1 Terminology

Schema

A key part of any Offline-First framework is the client-side database. Some frameworks take a schemaless approach, allowing developers to use an implicit schema instead of an explicit schema. Relic will use an explicit schema, as this is needed to ensure type safety.

The client-side database schema is not only important on the client, but the server also needs to know about the schema of the client-side database. The server is responsible for populating the client-side database when the client first connects and updating this data when the client pulls for updated data. Therefore, the server also needs to know about the schema to allow type safety on pulls.

Requiring users to define and maintain a special schema that Relic understands would increase development time. As the Relic framework includes an ORM, which will also require a schema, relic can use this schema to ensure that users only need to define their schema once. This is also advantageous as developers do not need to learn a new DSL to define their Relic schema.

Mutations & Mutators

The replication algorithm's essence is mutations and mutators. A mutation is a record that describes a write operation. Mutations can be executed with a mutator, which describes how the write operation is carried out. Mutators run in two environments: on the client during client-side prediction and on the server authoritatively. Mutations are stored on the client and transferred to the server.

A mutation can be represented in JSON as follows:

```json
{ "id": 1, "name": "addTodo", "input": { "name": "Buy groceries" } }
```

As shown, a mutation consists of three parts. First, the mutation id, which is used on both the server and client to identify a single mutation. Then, we have the name of the mutator which is responsible for executing that mutation. Finally, we have the input of the mutator, which could also be called the mutator arguments or parameters.

To execute this mutation, it needs to be handled by the corresponding mutator, identified by the name property. Like the schema, mutators are defined by developers using the framework. A mutator is a simple TypeScript function which takes the mutation's input as an argument. For example, a mutator that can execute the previous mutation could be defined by the developer as follows:

```typescript
function addTodo(todo: { name: string }) {
    // Add todo to the database.
}
```

As described, mutators are executed on both the client and the server. Therefore, the name and input of mutators are shared between the client and server. Crucially, the function body, or the handler of a mutator does not have to be shared between the client and server. For example, addTodo on the client will insert the to-do in the SQLite database. The server may wish to use MySQL, and thus the addTodo mutator inserts the to-do into a MySQL database. The server may also wish to do additional things, such as validate that the user may actually create a new todo. The client may also wish to perform other logic than the server, such as show a toast message that a new todo has been created. On the other hand, having to implement each write operation twice is a time-consuming development task. For that reason, mutators should also be able to be written once and shared across the server and client. The implications of this feature are discussed in a later section.

Context

A mutator that only has access to the input of the mutator alone is not useful. Besides mutation input, mutators typically require more information and objects. A mutator would need a database connection to make changes, furthermore, a mutator may also want to know the user that is associated to this request.

The additional necessities for a mutator are dependent on the application. Therefore, developers should be able to choose these dependencies based on the requirements of the application and mutator. We store these dependencies in a context object. Developers can create their own context, which will be made available by the framework in any mutator.

4.6.2 Relic Classes

The main building blocks of the Relic framework are the RelicDefinition, RelicClient and RelicServer classes. Developers will use these classes to specify their schema, mutators and contexts, and the RelicClient and RelicServer classes will be embedded in Relic applications at runtime.

4.6.3 RelicDefinition

The RelicDefinition class's main purpose is to provide end-to-end type safety. Using this class, developers will register the schema, the types of mutators, and the context that is shared between the client and the server. The RelicDefinition represents the contract between the client and the server. The contract specifies the mutators both environments can invoke, and the schema of the data that the client may receive from the server.

In the RelicDefinition, mutators do not have to specify a handler. The mutators which are registered in the RelicDefinition may only consist of a name and the input type, excluding the function body.

4.6.4 RelicClient

The RelicClient is the class responsible for managing the client-side behaviour of the Relic framework. It is instantiated with a RelicDefinition. In the RelicClient, developers must implement the client-side mutator handlers that are specified in the RelicDefinition to fulfil the contract. Before developers implement the client-side mutators, they may wish to specify a context type, which defines the objects that can be used in the mutators.

A RelicClient that is implemented by the developer, containing the database schema, the mutator context, and the mutators and their handler, can then be instantiated. To instantiate a RelicClient, several arguments are needed.

- The URL, which specifies how the RelicClient can reach the RelicServer
- The SQLite database adapter, which conforms to a RelicSQLiteDB interface
- A poke adapter, that defines how the client can receive notifications from the server
- The context object, that satisfies the context type required by mutators

The SQLite database adapter defines how the RelicClient may interact with its SQLite database. To implement the interface, a single exec method, which takes a SQL string and SQL parameters, and returns the rows must be implemented. Using a database adapter on the client has important benefits. JavaScript runs in many environments; naturally, JavaScript runs in the browser. However, in recent years, JavaScript can also run on the server, using NodeJS[8], JavaScript can run as a desktop application using Electron[9] or Tauri[10]. Furthermore, using React Native[11], JavaScript can also be used on smartphones. A specific SQLite implementation may be designed for the browser but cannot be run in a desktop or mobile environment. By allowing the RelicClient to accept an adapter, Relic becomes a cross-platform framework, allowing developers to use Relic to create web applications, desktop applications and native mobile apps. They can all use the same RelicClient; only the database adapter has to be switched.

An instantiated RelicClient exposes several methods. The *query* method accepts an ORM query, which returns the results of the query. Using this method, developers can make any database query from anywhere in the application. Besides returning the results of any query once, the RelicClient also exposes a method that allows developers to reactively subscribe to a query, returning new results when updated.

For each mutator, the instantiated client exposes a method with the same name as the mutator. The method accepts the input of the mutator as an argument. With these methods, developers can invoke the mutators.

4.6.5 RelicServer

The RelicServer class has the same responsibility as the RelicClient, but for the server. In this class, developers register the mutator handlers for the server. Again, developers can specify a context type that is available in mutators.

Besides mutators, the server has two other components that developers must implement for their application. A puller, and a poker. The puller is a function that specifies the data the server makes accessible to the client. The data the puller returns must be in the form of the database schema of the client, when the response of a pull is received on the client, the data is directly inserted into the local database. For this reason, the schema is defined in the RelicDefinition. Because of this, implementing the puller is entirely type-safe. The handler of the puller, specified by the developer, also has access to the context of the RelicServer.

Developers can also specify a poker, a function that defines how the client is notified when the server state changes. This enables clients to receive real-time updates. Allowing the developer to implement

[8]`https://nodejs.org/en`
[9]`https://www.electronjs.org/`
[10]`https://tauri.app/`
[11]`https://reactnative.dev/`

this provides them with more flexibility to choose a medium to send these notifications.

After the mutators, pusher, and poker have been implemented, developers can use the RelicServer to respond to HTTP requests. To do this, the framework exposes a *handleRelicRequest* function. This function takes a RelicServer, an HTTP request, the context object, and a server-side database adapter. The handleRelicRequest function returns an HTTP response that can be served using an HTTP server of the developer's choice.

The server-side database adapter is more extensible than the client-side database adapter. The client adapter only supports SQLite, but the server-side adapter can be used with more types of databases.

4.6.6 Overall Framework Architecture

We have individually discussed the components that make up the framework. We now present the overall architecture of the framework API in figure 4.5.

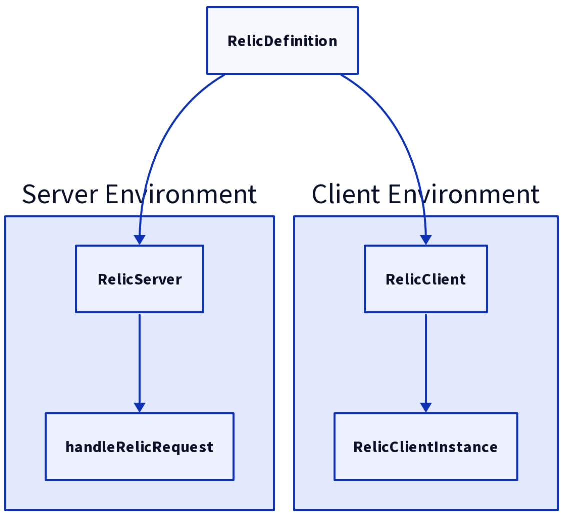


Figure 4.5: The architecture of the Relic framework

The RelicDefinition is the starting point of any Relic application. The RelicDefinition defines the client-side database schema and the mutators that are part of the Relic application. It is used in both the server and the client runtime environment. On the server and the client, developers define the RelicServer and RelicClient, implementing the mutator handlers for each environment. On the server, a RelicServer can be used in the handleRelicRequest handler, which takes an HTTP response and returns an HTTP request. On the client, a RelicClient can be instantiated to create a RelicClient instance, which is then used to query and invoke mutations throughout the client application.

4.6.7 Shared Mutators

An important design decision of the framework is to allow clients and servers to use independent mutators. The addTodo mutator on the client is different than the one on the server. This is needed, because the server can have an entirely different database than the client. Therefore, an addTodo mutator on the client, executing an SQLite query could not be reused on the server, which uses Postgres for example. Allowing different mutators allows for much flexibility, for example, the server could choose to do more validation logic than the client. On the other hand, this also means that developers have to implement each write operation twice, once for the server environment and once for the client environment.

Normally, RelicDefinition only contains the mutator name and mutator input, not the handler, as these have to be defined in the client and the server. To reduce this burden, we introduce another feature, allowing developers to implement mutator handlers in the RelicDefinition. This allows developers to write a mutator once and share it across the client and the server. This would save a lot of time. There are two main options to make this possible. The client and server must use the same database, thus SQLite, as the client is already constrained to SQLite. The other option is to write ”Repository” interfaces, and add those to the shared context in the RelicDefinition. The mutator handlers in the RelicDefinition can then make use of those interfaces to implement all application logic only once. However, this does mean that the repository interface has to be implemented on the client and the server. The options: implement mutators twice, use SQLite on the server and client, or create repository interfaces, all involve tradeoffs and there both have downsides. This choice is one the application developer can make, depending on the application’s requirements.

Chapter 5

Implementation

In this chapter, we will discuss the implementation of the Relic framework. Recall that the goal of the framework is to provide developers with a full-stack, type-safe web framework to build Offline-First applications. In chapter 4, we have described the system architecture, the storage system, and the replication protocol. With this design, we can now begin implementing the framework.

This chapter first discusses the implementation of the storage system. Then, the implementation of the replication protocol is presented. Finally, we discuss the implementation of the framework API.

5.1 Storage

In chapter 4, we have presented the design of the storage system that can structure, store and query large amounts of data in the frontend, supporting many types of applications. We have reviewed multiple data stores, and selected SQLite with an ORM to fulfill the research question and the requirements. In this section, we discuss the available libraries that enable SQLite to run on the frontend. Then, we will discuss how to persist this data in the browser. We will then highlight how reactivity is achieved. Finally, we will discuss choices for ORMs and how to integrate the SQLite system with the ORM.

5.1.1 SQLite Libraries

SQLite in the browser has an extensive history. This history began with the proposal of the Web SQL specification in 2009 [92]. Before standardization, the specification was abandoned, as the specific SQLite implementation was not deemed to be a suitable standardization. In 2012, a new project was started: SQL.js [93], which showed that SQLite can be compiled to WebAssembly. SQL.js allows for the creation and query of SQLite databases in the browser. SQL.js, however, does not support the persistence of this database; it is entirely stored in memory. In 2021, the absurd-sql project was created to add persistence to sql.js [93]. Absurd-sql uses the persistent indexeddb browser API [94], to store blocks of data in small chunks. In the same year, the wa-sqlite library was created, with the intent of experimenting with multiple virtual filesystems for SQLite. In sql.js combined with absurd-sql, the data is still stored in memory and persisted to the disk. Wa-sqlite enables the usage of SQLite directly from the disk without loading the entire database into memory. Like absurd-sql, wa-sqlite also supports IndexedDB as a file system. Besides IndexedDB, absurd-sql also includes support for the newer Origin Private File System (OPFS) browser API [95]. The OPFS API was specifically designed for high-performance in-place file access in browsers. The Origin Prive File System API spawned interest from the official SQLite project, and thus, the sqlite3 WASM/JS Subproject was created [96]. The SQLite WASM project thus provides an official implementation of SQLite in the browser from the organization that maintains SQLite. Regarding persistence, IndexedDB is not available, only OPFS.

5.1.2 Persistance

We have seen many libraries that provide SQLite in the browser, not all libraries support the same persistence targets. To make a choice of which library to use, we must first decide which API we want to use to persist this data.

The first option to persist data in web browsers is the IndexedDB API, which was created in 2015 [94]. The IndexedDB provides developers with a key-value store that can store structured data. The

IndexedDB API can be used without an additional library as a database for developers to use. It provides asynchronous access to this data and can be used from the main thread.

The Origin Prive File System (OPFS) [95] has a different goal than IndexedDB. IndexedDB aims to give developers a ready-to-use database with transactions and indexes. The OPFS API aims to give developers low-level, highly performant access to raw files and directories. It was standardized in 2020 and has been implemented in all major browsers in 2023. Like IndexedDB, OPFS is also only accessible through asynchronous methods. One important aspect of the OPFS API, is that some functions of the interface are only available in the context of a Web Worker. Therefore, using OPFS to persist SQLite requires the use of Web Workers.

Long discusses the implementation of absurd-sql, which uses IndexedDB to persist SQLite [68]. The reason the project is named absurd-sql is because of the absurdity of storing a database (SQLite) in another database (IndexedDB). IndexedDB was never really designed for this use case. OPFS was designed for this use case, which is the reason why the official SQLite WASM project only supports OPFS and not IndexedDB. In benchmarks of the wa-sqlite project which has support for both an IndexedDB VFS and an OPFS VFS, we can compare the performance of the two persistence solutions. These benchmarks show that OPFS is faster than IndexedDB for many operations [97]. This is also noticed by RxDB, which states that OPFS is 3 to 4 times faster than IndexedDB [98].

OPFS is faster than IndexedDB for SQLite databases. Furthermore, OPFS was designed for this use, where using IndexedDB for storing SQLite data is possible but hacky. For these reasons, we will use OPFS to persist the SQLite database.

This leaves two SQLite libraries that support OPFS, the official SQLite WASM project, and the wa-sqlite project. Wa-sqlite is described as "WebAssembly SQLite with experimental support for browser storage extensions". The library is also not published on the Node Package Manager registry, and the experimental nature and motivation of the library can be seen as the downsides. Furthermore, the library is not stable yet as of 30-04-2024. On the other hand, the official SQLite WASM project was created with the motivation of making WASM a first-class member of SQLite. We will therefore use the SQLite WASM library to enable storage for the web framework.

5.1.3 OPFS VFS

SQLite WASM will persist through OPFS; SQLite WASM provides two different OPFS Virtual File Systems (VFS) to interface with OPFS, which have trade-offs impacting developers using the framework.

The first VFS is the default OPFS VFS. This strategy requires developers hosting applications using SQLite WASM to set two HTTP headers, the Cross-Origin-Opener-Policy (COOP) and Cross-Origin-Embedder-Policy (COEP) response headers. These headers are required by OPFS for the functions used by this strategy. Requiring developers to set these headers when deploying the application is not ideal.

The other VFS is the OPFS SyncAccessHandle Pool VFS (SAH-Pool). This implementation has certain advantages. First of all, the developer does not have to set COOP and COEP headers. Additionally, this implementation is shown to improve performance compared to the default OPFS VFS. The major downside of this VFS is that it does not allow for multiple connections to the database. Therefore, if the user opens two tabs, these tabs cannot share the same database. The framework should still be able to handle multiple tabs, as allowing users to only open one tab is not a good User Experience. Multiple tabs with the SAH-Pool VFS are still possible by using cross-tab coordination and routing all access to the database through one tab, thus ensuring there is always only one database connection open.

The SAH-Pool VFS has some important benefits. The increased performance is important to keep the application responsive. Furthermore, not requiring additional headers reduces friction when deploying the framework. The major downside can be circumvented by using cross-tab coordination. Accordingly, we will use the SAH-Pool VFS in the framework.

5.1.4 Reactivity

In the Riffle system, we have seen the importance of reactivity to let developers use declarative transformations of data [11]. The Riffle system uses SKDB [99], which is an alternative SQL engine that can also run in the browser. One of the primary features of SKDB is the support for incremental updates. If a query is dependent on a table, and later a row is added to this table, the result of the query is updated without having to recalculate the entire query. However, incremental updates on a fine-grained level such as SKDB are not trivially added to an existing library such as SQLite WASM.

As reactivity is not included in SQLite WASM, we must create a reactivity layer as an extension to

SQLite WASM. The layer must expose two methods: one method to execute statements changing the database and one method to subscribe to a query. The subscribe to query method should have a callback that is invoked when the underlying data is changed.

The reactivity layer will keep track of all current subscriptions. The most trivial implementation of reactivity is to re-run every subscribed query when any update happens. This is easy to implement and is correct; however, re-running every subscribed query can be expensive when many queries are subscribed.

We could also implement a more fine-grained reactivity system, which re-runs only the queries that depend on the table that is updated. This is not as fine-grained as SKDB which is used by the Riffle reactive syste reactive siystem. However, it is definitely more performant than running every query. On the other hand, it significantly increases implementation complexity. Every query will have to be analyzed to determine which tables are affected by the query. To improve the performance of the application, we will implement this type of fine-grained reactivity.

There are several ways to do this. We could require users to define the tables that are affected by the query themselves, such as React's useEffect hook, which requires users to manually specify the reactive dependencies [100]. This is not a great User Experience. However, the User Experience could be improved by adding a linter plugin which can automatically parse the query and insert the dependencies as runtime, which is the approach taken by React. The downside of this approach is that developers must install a linter and add a custom plugin to this linter. We can also create a plugin that can calculate the dependent tables at compile time, however this plugin will be tightly coupled to the bundler that is used, which means that many plugins would have to be written for different bundlers, or the framework would only work with one specific bundler. Lastly, the dependencies (tables) could be calculated during runtime. This requires having an SQLite-compatible parser that parses the AST of the sqlite statement for every query. This incurs some performance loss during runtime but does not result in friction associated with manual dependency tracking or the problems of compile-time parsing. Furthermore, it allows for more dynamism in queries. For this reason, we will parse the queries at runtime to enable the reactivity for the framework.

In summary, we build a reactive layer on top of SQLite, exposing two methods: one method that every update to the database must go through and one method that can be used to subscribe to a query. Subscription callbacks for a query will be invoked if any table of the query is included in an update statement, in order to improve performance.

5.1.5 Object–Relational Mapping

Another important decision for the storage of the framework, is which ORM library to include. While there are definitive advantages and disadvantages of some ORMs over others, deciding which ORM to choose often involves personal preference. In theory, the framework could be ORM agnostic, which would allow developers to choose their own ORM. However, in the interest of saving time, we will choose one ORM and build the framework around that.

There are many JavaScript ORMs. However, the main reason for using an ORM is to provide type safety. In recent times, three main ORMs have seen increased adoption and popularity: Prisma[1], Drizzle[2] and Kysely[3].

Of the three ORM, Prisma sees the most usage by far with almost 10 times as many downloads than the other packages[4]. However, Prisma currently only works in a Node.js environment. Therefore, Prisma cannot be used as an ORM on the front end. That leaves Drizzle and Kysely. Drizzle and Kysely both take a very different approach to ORMs. In Drizzle, developers specify their schema in a way that makes the schema an object available at runtime. Kysely schemas are entirely defined in TypeScript types, TypeScript types are only used in compile time, therefore the schema defined in Kysely is not available in runtime. For a framework, being able to read the schema in runtime is valuable, as this allows for runtime validation and is needed for the framework to insert JSON data in the database. Furthermore, Drizzle currently has more than twice the GitHub stars of Kysely, and has recently overtaken Kysely in weekly downloads. For this reason, we will use Drizzle in the Relic framework.

In future work, making the framework ORM agnostic and allowing developers to specify their own ORM would be a clear improvement.

[1]`https://www.prisma.io/`
[2]`https://drizzle.team/`
[3]`https://kysely.dev/`
[4]`https://npmtrends.com/drizzle-orm-vs-kysely-vs-prisma`

5.2 Rollbacks

The chosen replication algorithm: server reconciliation and client-side prediction, requires that the storage system can rollback to previous points in time. We will now discuss methods to implement rollbacks in SQLite.

The first possible solution is to use SQLite transactions. SQLite transactions allow developers to use the "ROLLBACK" statement to undo all changes since the transaction has begun. An important property of SQLite transactions is that transactions are managed within the SQLite process. Thus, SQLite transactions are automatically rolled back when the browser is closed or the device is turned off. This presents problems with offline usage. If a user performs work offline, and shuts down their device, all work performed is lost as the data changed within the transaction is not saved. Therefore, SQLite transactions are not suitable for offline operation.

Another solution would be the official SQLite session extension [101]. This extension introduces the concept of a changeset, changesets capture all changes made to the database since the creation of the changeset. Changesets are typically used to replicate changes of one sqlite database, to another sqlite database. Besides this use-case, changesets can also be inverted, which can be used to undo all changes in the changeset, effectively rolling back all changes until the start of the changeset. Unfortunately, changesets suffer from a similar problem seen with transactions. Where transactions are rolled back when shutting down the SQLite process, changesets are lost. Therefore, the changes made to the database are stored, but we lose the ability to roll back these changes.

One way to perform a rollback is to store every opposite query for every query made and, when rolling back, execute every opposite query. An INSERT statement would be inverted to a DELETE statement, and the other way around. These queries could then be stored persistently in the database. When the rollback function is invoked, we can execute these queries in reverse order to get back to the initial state. We could require that developers write an opposite query, for every write query they use. Of course, manual tracking of queries and opposite queries is not a good experience at all.

SQLite has a demonstration of this technique to enable automatic undo and redo functionality [102]. This example uses SQLite triggers, to automatically save the opposite query of every write query that is invoked on the database to a specific SQLite table called an "undolog". The use of triggers requires no extra effort from developers, and the undo log table ensures that we can persist the changes that must be rolled back when SQLite is restarted. Therefore, we adapt the SQLite example to implement rollbacks that will be used later, in the replication algorithm.

5.3 Replication

The high-level replication algorithms were presented in figure 4.2, 4.4 and 4.3. In this section, we will discuss several implementation details of these algorithms, starting with the mutation queue. Then we will discuss how the framework deals with delta messages, tracking changes between versions of the database. Finally, we discuss realtime updates.

5.3.1 Mutation Queue

When applying a mutation locally, the mutation must be stored somewhere, as the mutation must be sent to the server at a later stage, and as mutations may possibly be replayed. The location where these mutations are stored must offer persistance. Mutations can be created offline, and after restarting the application, the mutation must still be able to be sent to the server, otherwise synchronization would not be possible and local changes would be lost.

The framework already includes a place where data is persisted, namely the SQLite database. It makes sense to store the mutation inputs here as well, in a special *relic_mutation_queue* table. This table contains the columns mutation id, mutation name and mutation input.

5.3.2 Client-Side Mutations

When a client first begins a mutation, for example, by clicking the "Add To-Do" button, two things happen. The mutation is stored in the mutation queue. The corresponding user-defined mutation handler, the mutator, is called with the inputs of the mutation. The update to the mutation queue and the execution of the mutation handler must happen atomically. If the mutation handler fails, the mutation should not be stored in the mutation queue. For this reason, we use SQLite transactions to automatically rollback all changes if the mutation handler fails.

5.3.3 Server-Side Mutations

The application of mutations on the server looks very different from the process on the client. The server does not store a mutation queue. However, the server has some other requirements regarding the application of mutations. Mutations must be idempotent, as we do not want mutations to be invoked twice on the server if the client sends the same mutation twice or if they did not receive an acknowledgement of the success of the first mutation. To ensure idempotency, the server must track which mutations it has already applied from which client. To do so, the server stores the latest processed mutation id of each client. Mutation ids are monotonically incremented numbers, therefore the server can know if a mutation is already applied by looking if that mutation id is lower than the latest processed mutation id from that client, effectively ensuring idempotency. We could also achieve idempotency by assigning a random ID to each mutation and storing those, but using monotonically incremented mutation IDs significantly saves storage space.

Thus, when applying a mutation on the server, the server will first check whether the mutation is already combined by comparing the mutation id to the last processed mutation id of that client. If it is applied, the mutation can be skipped. If not, the server will invoke the user-defined handler of the mutation. After invoking the handler, the server will set the last processed mutation id to the new mutation id. These actions, comparing the mutation id, executing the mutation and storing the mutation id, must also be applied atomically in order to ensure consistency. Therefore, these actions must also executed in a transaction.

Note that by requiring the server to use transactions, we do impose a restriction on the arbitrary server database. However, this restriction is needed to ensure idempotency of mutations, and keep the state of the application consistent.

5.4 Delta Messages

In the design of the pull process, as defined in figure 4.4, an important aspect is that the server sends delta messages to the client, describing what state has changed since the latest client pull. There is a multitude of different methods to implement the capture of a delta between two states of the database, which we will discuss in this section.

5.4.1 No Delta Messages

Before we discuss the possible options for delta tracking, we will first discuss the option of not sending delta messages at all. Instead of sending a delta message, the server could also send the entire application state on every pull. The client can then reset the local database and apply this state. This option is very simple and does not impose any requirements on the server when used.

This option may be an attractive option for small datasets. If the application state is small, retrieving and sending the entire state is not a problem. However, this option is very inefficient, which results in a large overhead, especially for larger application states. For every pull, the server database must do a lot of work to load the entire application state. The entire application state must be transferred over the network, costing a lot of network bandwidth. Finally, the client database must delete all data and reinsert all rows, reducing client performance.

For these reasons, sending no delta messages would not be a valid option for applications that do not have a very small application state. Therefore, other options must be considered so as not to exclude larger applications from using the framework.

5.4.2 Logical Replication

A popular method for replication of data and change data capture is using logical replication from the database to track changes. Both ElectricSQL and PowerSync use Postgres logical replication to track the changes made to the database. When a client pulls, the server reads replication log, to see what database operations have been processed since the previous pull. However, as seen in PowerSync and Postgres, using a replication log requires the framework to be tightly coupled to a database. Possibly, a replication log adapter could be made for multiple databases, so users could integrate Postgresql and

MySQL, however this still requires databases to export a replication log. Furthermore, a replication log requires keeping track of a large history of operations. While the size of this log can be reduced using log compaction, there is still an inherent overhead in disk usage.

5.4.3 Triggers

The problem of database flexibility, as seen in the logical replication route, can be partially solved by using database triggers instead of logical replication. Users could set up triggers which track all operations to the database to a special changelog table. This is related to the strategy we use to implement rollbacks. However, setting up these triggers can be a hassle for users of the framework, and not every datastore will support triggers. Furthermore, the problems associated with keeping a history of operations, such as size are still present. Finally, triggers also incur their own performance overhead, as every database write will result in two writes.

5.4.4 Timestamp

The previous options, logical replication and triggers involved keeping a log of operations in the database. An alternative would be to track changes within the state. This is the method employed by WatermelonDB. Using timestamp change data capture, every row in the database is required to have a last modified column. With this column, we can now query all data that has changed since the last pull by selecting all rows that have a last modified value that is bigger than the timestamp of the last pull. By embedding this data in the rows, we do not need to keep a log of operations, reducing size and complexity. Furthermore, this method is flexible for databases that do not have a replication log or triggers.

There are also some downsides to this method. Using timestamp change data capture requires users to change the schema of their database to use the framework. While this is not a very invasive change, it is still a requirement imposed on the user. Furthermore, this method does not allow for database deletes but requires the use of soft deletes. Soft deletes involve that records that are ”deleted” are not actually deleted from the database, but soft deleted by using a boolean column that stores whether the row is ”deleted”. Then, every database query should have a where statement that filters out all soft deleted rows. This is a requirement, as performing normal database deletes will not show up when querying all changes since a timestamp. Otherwise deletes would then never be replicated to the client, which would not allow for convergence.

There are also some inherent problems with timestamps and consistency. Not all databases have monotonically increasing timestamps, which results in missed updates. For example, imagine the following scenario. A user pulls all changes, the server clock decreases due to NTP or leap seconds, a row is modified and has a last modified timestamp that is lower than the timestamp associated with the user pull. The user then pulls again, but will never receive the row that was changed. This inconsistency is not acceptable, therefore a monotonically increasing timestamp would be required.

Replicache highlights another problem associated with the use of last-modified timestamps within transactions [103]. In this scenario, a transaction is first started in the transaction, a row is changed at timestamp t1. Before the transaction finishes, a user pulls data at timestamp t2. The effects of this transaction are not visible to this pull yet, as the transaction has not been committed. The transaction then finishes, publishing the row to the database. The user pulls again but will not see the changed row, as the row has a last modified timestamp that is smaller than the last pull. The user will then never receive the change of this row. The framework requires that mutations on the server are atomically applied with the update and compared with the last processed mutation id. Therefore, this problem makes timestamp-based change data capture not suitable.

5.4.5 Version Numbers

Instead of using timestamps on rows, monotonically increasing version numbers can be used, or lamport clocks [104]. This solves the issues associated with timestamps but introduces other challenges. There are several alternatives to consider when using versioned rows.

One possibility is storing a ”database version” in a special table. When a row is updated, increment the database version and set the row version to the database version. When calculating the delta message, all rows with a version number greater than the database version when the client last pulled, are selected. The selection of rows can be very efficient if there is an index on the version column. However, the incrementation of the database version and the updated row must happen in a transaction to avoid consistency issues. This effectively makes the database version function as a global lock, as only one transaction can get and update this number at a time.

There are several methods to reduce this concurrency bottleneck. One example involves keeping a version for each table, instead of each database. When updating a row, the version of that table is incremented, and the version of the rows in that table refers to the table version instead of the database version. Instead of the client sending their database version, it sends the version of each of the server's tables, as seen in the client's previous pull.

Another row version method is proposed by Replicache, which is suitable for multi-tenant applications [105]. In this method, each tenant has its own version. All rows associated with that tenant correspond to the version of that tenant. This reduces the bottleneck to transactions within each tenant, instead of the database or each table. However, not every application is multi-tenant, and many multi-tenant applications still allow for sharing of data between tenants.

Changing the version tracking from database-wide to table-wide is also possible. This makes the locking more fine-grained and thus reduces concurrency issues. But also increases the size of the version object that represents the version of the database at one point.

We can go one step further, and use row version tracking. In this method, each row has an independent version number. This effectively removes the concurrency bottleneck, as each row can be updated without locking a version that is shared across other rows. While removing the write concurrency issue, this option has some consequences for reading the delta. Where the table and database version lookups where a comparison between each rows version and the database version, using independent row versions requires each row to be compared with its previous value. This decreases the performance of reads. Furthermore, while a single database version can easily be shared across the network, and the versions of all tables would also not be a very big size, the versions of all individual rows a client has access can be of a much larger size. For large datasets, this would quickly result in a network transfer bottleneck. Therefore, Replicache proposes storing this data on the server instead of sending it back and forth to the client [106]. This option also has a significantly higher implementation complexity than the other options.

The individual row versioning option has some other attractive properties. Imagine a system where authorization for specific rows is dynamic, for example, a user can only view a to-do item when the user is part of a special group. When the user is added to the group, they will not automatically pull the to-do items that they have newly gained access to, as these to-do items may not be modified since the last pull. In an individual row versioning scenario, these new row items will automatically be pulled, even though they have not been changed. This makes it much easier for developers to implement; it is a side effect of the increased read cost.

5.4.6 Choosing an Option

As seen, all different methods to resolve the delta message have their own benefits and disadvantages. We did not identify one single optimal method. Some applications will prefer the increased read performance of database, table or multi-tenant based row versioning. Other applications may value increased write concurrency and easier dynamic views and will, therefore, want to use individual row versioning.

For this reason, we can design an abstraction allowing developers to choose their own delta message resolution logic. This would allow developers to make this decision themselves, instead of requiring them to use a suboptimal solution. Additionally, the framework will provide several prebuilt adapters to resolve the delta messages according to the strategies outlined above. These adapters will allow developers to choose an existing option and will increase the ease of integration, reducing development time.

5.5 Realtime Updates

We will now discuss the implementation of real-time updates within the framework. In the push design of figure 4.3, we see that the server notifies clients when a push is applied. In a web application, there are several ways to send information from the server to clients, which we will explore now.

5.5.1 Polling

One way to receive updates from the server is to periodically check whether the server has new updates. This is a very simple architecture, as it follows the default web request-response model. However, there are some obvious downsides to this model. The client will not immediately know when an update from the server is available; depending on the polling interval, the client may have to wait several seconds to receive updates. The polling interval can be reduced; however, this increases the network bandwidth and the load on the server. Polling is a very simple model, and can easily be integrated into web applications, but it is not optimal.

5.5.2 Server-Sent Events

Server-sent events (SSE) is a web standard that provides servers with a unidirectional channel to send messages to clients [107]. Web servers expose a special endpoint which exposes an event stream. Clients can subscribe to this endpoint, receiving events when the server publishes new events. SSE has several benefits over polling. Clients will immediately be notified when an event is published, instead of having to wait for the polling interval to receive new information. Furthermore, server load is reduced, as clients only invoke the server when needed.

There is one main downside regarding server-sent events, SSE is a stateful protocol. Stateful protocols have several implications, such as increased complexity in maintaining a connection state, fault tolerance, and most importantly, scalability issues.

5.5.3 WebSockets

WebSockets are similar to SSE, but come with one main difference: WebSockets allow clients to send messages to the server, enabling bi-directional communication [108]. Furthermore, while SSE is based on HTTP, WebSockets use a custom WebSocket protocol.

For our use-case, the bi-directional nature of WebSockets does not offer important benefits, as we are looking for a method to notify the client from the server. Theoretically, we could use the bi-directional nature of WebSockets to send push messages from the client to the server through WebSockets instead of HTTP. However, we would gain no significant benefits by using this. Therefore, WebSockets would be used unidirectionally. WebSockets would not provide functionality that SSE does not already offer while being simpler to use and reason about.

Like SSE, WebSockets are also stateful. Another problem of the stateful nature of these two technologies is the use in serverless environments. Serverless environments, such as AWS Lambda[5], Azure Functions[6], Google Cloud Functions[7] or Cloudflare Workers[8], provide stateless environments where each function invocation is isolated and does not maintain state between executions. Furthermore, functions are often limited in terms of execution time. Therefore, the long-lived connections required by SSE and WebSockets are challenging in this environment.

5.5.4 A Stateless Solution

As discussed, the stateful nature of real-time communication brings many challenges. Requiring users of the framework to use SSE or WebSockets makes the entire framework stateful. It would, therefore, expose the framework to scalability issues and restrictions on runtime environments such as serverless. On the other hand, a stateless solution such as polling is also not ideal.

Therefore, we can extract the stateful part of the framework, the realtime notifications from the framework itself. This allows users to dictate how to manage real-time notifications without being forced in one direction by the framework. It allows the framework to remain stateless and, therefore, allows for usage with serverless functions. Users can then decide what real-time solution makes the most sense for their use case; they can choose to use SSE in a single server that the framework runs on, keeping it simple. They can choose to use a pub/sub service such as MQTT[9], Soketi[10], Redis[11] or others, that they maintain and scale themselves to enable real-time updates. Users can choose to use a managed real-time service, such as Pusher[12], Ably[13] or AWS IoT Core[14] to manage scaling for them.

The framework allows the server to dictate how notifications are published. To make integration easier, the framework can once again provide adapters to make working with the above protocols easier.

[5]`https://aws.amazon.com/lambda/`
[6]`https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview`
[7]`https://cloud.google.com/functions`
[8]`https://workers.cloudflare.com/`
[9]`https://mqtt.org/`
[10]`https://soketi.app/`
[11]`https://redis.io/blog/how-to-create-notification-services-with-redis-websockets-and-vue-js/`
[12]`https://pusher.com/`
[13]`https://ably.com/`
[14]`https://aws.amazon.com/iot-core/`

5.6 Framework API

We have discussed the implementation details of the storage and replication aspect of the framework. We will now discuss the implementation of the framework API. The framework is implemented across multiple packages, this enables consumers of the framework to only import the necessary dependencies. The three main packages are @relic/core, @relic/client and @relic/server. In this section, we will discuss the API exposed by each package.

5.6.1 @relic/core

The @relic/core package contains the Relic constructs that are shared between the client and the server, such as the RelicDefinition, and RelicMutation class.

RelicDefinitionBuilder

To construct a RelicDefinition, we introduce another class, the RelicDefinitionBuilder. This class is used by the developer to create a new RelicDefinition for their application. The RelicDefinitionBuilder class uses the "builder" design pattern to provide a way to construct a RelicDefinition [109]. Using a builder pattern has two main advantages.

The RelicDefinition class has several generics that are required to ensure type safety: the type of the context, the type of the schema, and the type of the mutations. Creating the RelicDefinition class requires the schema object, the mutators object and the type of the context. In TypeScript, there is currently no partial type inference [110]. Therefore, a constructor or function can either infer all generic types, or all generic types must be manually specified. Using the builder pattern, we can split up the registration of these three main components into different functions, allowing the types of objects to be automatically inferred and the user to specify the type of context.

The biggest advantage of the builder pattern is that it allows the definition of the mutators to be split across multiple files. Using the builder pattern, we can first create a RelicDefinitionBuilder, and register the context and schema. Then, using that type, we can extract a way to define mutators that are correctly typed. These mutators can be implemented in multiple files. For example, one file could contain mutators that are related to to-do items; the mutators related to users can be implemented in another file, without losing the type information used in the mutator.

5.6.2 @relic/client

The @relic/client package contains all code that runs on client devices using a Relic application. The package contains the RelicClient, RelicClientBuilder and RelicClientInstance classes. A RelicClient-Builder is constructed by passing a RelicDefinition, in which all type information is stored. The builder pattern is used for the same reasons as the RelicDefinitionBuilder.

RelicClientInstance

Developers use the RelicClientInstance to query data from the database and invoke mutations to update it. Since querying in Relic is asynchronous, developers have to deal with loading states, errors, and data invalidation. To reduce this burden, Relic features an integration with the popular Tanstack Query library [111]. Tanstack Query is an asynchronous state management solution. It features a declarative API that handles many challenges related to asynchronous state. Furthermore, Tanstack Query is framework agnostic, by integrating with Tanstack Query, Relic immediately can be used in React, Vue, Angular, Solid and Svelte applications. With the Tanstack Query integration, queries are invalidated, queries are cached, and common queries in different locations in the API are deduplicated. Tanstack Query also manages part of the reactive aspect of the query layer for us. When a mutation is invoked, Relic will invalidate the corresponding queries, and Tanstack Query will make sure they are immediately updated again.

Adapters

To create a RelicClientInstance, developers need to pass a database adapter. We have discussed this interface in the design chapter, and developers can trivially implement this interface for their custom SQLite library. To make this easier, Relic exposes an SQLite-wasm adapter in the @relic/sqlite-wasm package. In the future, more adapters can be created for sqlite libraries that are designed for desktop and mobile environments.

5.6.3 @relic/server

Like the client package, @relic/server contains code that runs on a server. The main constructs exposed by this package are RelicServer, RelicServerBuilder and the handleRelicRequest function. The RelicServerBuilder is used to construct a RelicServer in the same way as the definition and client.

handleRelicRequest

On the server-side, developers use the handleRelicRequest function to transform an HTTP request into an HTTP response. HandleRelicRequest extracts a JSON pull or push request from the request body. The push request contains mutations, which contain unparsed mutation inputs. These inputs must be parsed and validated to ensure the runtime object always corresponds to the mutation input type, otherwise a malicious user might take advantage of that. To do this, we use the Zod library [112]. Zod is a schema validation library that extends TypeScript types with runtime validation. Using Zod, developers specify their mutation input in Zod objects. This allows mutation inputs to always be validated by the framework.

The handleRelicRequest function takes an HTTP request and returns an HTTP response. Specifically, this function takes and returns Request and Response objects from the web-native Fetch API standard [113]. Using a web standard has many benefits related to interoperability. WinterCG is a group that focuses on interoperability of web APIs across non-browser runtimes [114]. Using the Request and Response objects from the Fetch API as outlined in WinterCG, the framework will automatically be interoperable with runtimes such as Cloudflare Workers, Deno, Bun, Vercel, AWS Lambda, and many other libraries that work with these standardized objects.

Adapters

On the server-side, a database adapter must be specified by the developer which manages data the Relic framework requires to function. The adapter interface contains the following methods:

```
export type RelicServerDatabase<TTx = any> = {
    transaction: <T>(callback: (tx: TTx) => Promise<T>) => Promise<T>;
    getClient: (
        tx: TTx,
        clientId: string
    ) => Promise<RelicServerDatabaseClient | undefined>;
    createClient: (tx: TTx, clientId: string) => Promise<void>;
    updateClient: (
        tx: TTx,
        clientId: string,
        mutationId: number
    ) => Promise<void>;
};
```

In the transaction method, developers register a function that executes a callback within a database transaction. This transaction is used by the storage layer to update metadata atomically with the mutation effects. The get-, create- and updateClient functions are used to store the metadata such as the last processed mutation id of a specific client. By implementing this interface, developers have the power to use any database they may want to use. The biggest restriction on what databases can be used is the requirement for transaction support.

While the interface is very small, Relic can make it even easier for developers to get started by providing packages that implement this interface for specific databases or ORMs. Relic provides an @relic/adapter-drizzle package, providing an adapter for databases using the Drizzle ORM.

5.7 Code

The code for the implementation of the relic framework can be found at `https://github.com/vigovlugt/relic`. The code for the @relic/* packages can be found in the packages directory. Three example applications can be found in the examples directory: a minimal todo application, an issue tracker using Tanstack Router for routing, and the meeting room scheduler from experiment 1.

Chapter 6

Experiments

To validate the Offline-First framework, we conduct three experiments. In the first experiment, we implement an example application using the framework. In the second experiment, we benchmark the frontend of this application. Finally, we benchmark the backend of the application in the third and final experiment.

6.1 Meeting Room Scheduler Case Study

6.1.1 Design

The first experiment aims to evaluate the practical effectiveness and developer experience of using the proposed framework to create an Offline-First application. Furthermore, we aim to show that the requirements of the framework have been fulfilled using this case study.

We chose to implement a meeting room scheduler application for this experiment. The main reason for this choice is that a meeting room scheduler application requires application-specific conflict resolution, which we show is possible with the framework.

To focus on the framework's features, we keep the application simple. The application will feature a data model of two tables: rooms and reservations. To showcase the relational capabilities of the model, this data model features a relation: a one-to-many relation from rooms to reservations. Every room has many reservations, and every reservation is for one room. The relational data model in SQL is shown in listing 1.

```sql
CREATE TABLE rooms (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE reservations (
    id TEXT PRIMARY KEY,
    room_id TEXT NOT NULL,
    owner TEXT NOT NULL,
    start INTEGER NOT NULL,
    end INTEGER NOT NULL,
    FOREIGN KEY (room_id) REFERENCES rooms (id)
);
```

Listing 1: The relational data model of the meeting room scheduler

Rooms are identified by an UUID and have a room name. Reservations have an UUID, and a room_id of the reserved room. Furthermore, reservations have an owner, normally this would probably relate to the id of the user that created the reservation. To keep the data model simple, the owner column will just contain the name of the person who made the reservation. Finally, a reservation includes a start and end time for which the reservation holds.

The application will include three operations: create, update and delete a reservation. To keep it simple, we do not allow users to update or create rooms.

On the server, we must choose what database to use to store the rooms and reservations. We may choose any database that supports transactions. For the case study, we will use a Postgres database to store this data.

6.1.2 Results

Creating a Relic application involves three main steps. First, developers create a RelicDefinition that describes the contract between the client and the server. Then, developers implement the client and server of the application, satisfying the RelicDefinition.

The experiment's result is split into four parts. First, we show the implementation of the logic shared between the client and the server, the RelicDefinition. Then, we show the implementation of the application's backend. Next, we show the implementation of the front end. Finally, we showcase the resulting application.

The application's three components, the client, server, and shared component can be structured in multiple ways. Developers can put the components in the same package or split them up into multiple packages in a monorepo. That is what we will do for this case study.

Shared

In this package, developers create the RelicDefinition. Recall that the RelicDefinition is an object which stores the schema of the client, and the types of mutators that the client and server must implement.

First, we show how the schema for the meeting room scheduler is implemented. Developers define the schema using the Drizzle ORM. The implementation of the schema for the application is shown in listing 2.

```
export const rooms = sqliteTable("rooms", {
    id: text("id").primaryKey(),
    name: text("name").notNull(),
});

export const reservations = sqliteTable("reservations", {
    id: text("id").primaryKey(),
    roomId: text("room_id").notNull().references(() => rooms.id),
    owner: text("owner").notNull(),
    start: integer("start", { mode: "timestamp" }).notNull(),
    end: integer("end", { mode: "timestamp" }).notNull(),
});

export const roomsRelations = relations(rooms, ({ many }) => ({
    reservations: many(reservations),
}));

export const reservationsRelations = relations(reservations, ({ one }) => ({
    room: one(rooms, { fields: [reservations.roomId], references: [rooms.id] }),
}));
```

Listing 2: The Drizzle schema of the application

The *rooms* and *reservations* variables define the rooms and reservations tables as designed in listing 1. Table declarations in Drizzle are closely related to the SQL counterpart, so even if you have never used Drizzle, you may understand how to implement new tables.

Next, we define the *roomsRelations* and *reservationsRelations* variables. These Drizzle constructs are used to define the relations in the application's data model. We define a room as having many reservations and a reservation as having one room. We also define the foreign key and the reverse foreign key for this relation.

With this schema, we now have a single source of truth for the data model on the client and server. On the client, we can use this schema to easily and type-safely query and mutate local data. On the server, we know exactly what data model the client expects from the server and can provide this data in a type-safe manner.

Next, we define the mutators of the application. Mutators are defined using the Zod input parsing library to allow the server to validate that requests from the client actually conform to the type. We have specified that the application must support three operations: create, update, and delete reservations.

To define mutators, we first need to initialize a new RelicDefinition, and specify the schema. By specifying the mutators, we complete the RelicDefinition. The creation of the RelicDefinition is shown in listing 3.

```
const schema = { rooms, reservations };
const d = initRelicDefinition().schema(schema);

export const relicDefinition = d.mutations({
    createReservation: d.mutation.input(
        z.object({
            id: z.string(),
            roomId: z.string(),
            start: z.coerce.date(),
            end: z.coerce.date(),
        })
    ),
    updateReservation: d.mutation.input(
        z.object({
            id: z.string(),
            start: z.coerce.date().optional(),
            end: z.coerce.date().optional(),
        })
    ),
    deleteReservation: d.mutation.input(z.string()),
});
```

Listing 3: The definitions of the mutators of the application

A new RelicDefinition is initialized using *initRelicDefinition*, using the builder pattern to register the schema. Then, we register the mutations by calling the mutations method of the RelicDefinition. The mutations object contains the three operations we want to support in the application.

The individual mutations are created using a method on the RelicDefinitionBuilder. The input of this mutation is specified using the input method, which takes in a zod schema. For the createReservation mutation, we want the input to be an object with the properties id and roomId of type string, as well as the properties of start and end of type date. For update reservation, we expect the id of the reservation to be updated, and optionally a new start or end time.

We could specify mutation handlers in our RelicDefinition but will choose not to. We will later implement these handlers individually on the server and client.

We have now created a RelicDefinition. This RelicDefinition will be imported into the client and server. Using the RelicDefinition, we will ensure that the implementation on the client and server is in accordance with the contract specified by the RelicDefinition.

Server

Developing the server part of a Relic application generally consists of four steps: setting up the server database, implementing the mutators, implementing the puller and poker, and exposing the server.

The server-side database does not need to have the same schema as the client. The server may wish to store data the client may not see, such as passwords. Furthermore, the server database must store some metadata for Relic to work.

To implement the server, we will also use Drizzle as an ORM for Postgres. We could also use raw SQL, but then we would lose type-safety for database queries. Besides the Drizzle schema, we will now also instantiate the drizzle database object that can be used to interface with the database. The code for the schema and logic to set up the database is shown in listing 4.

```
export const rooms = pgTable("rooms", {
    id: uuid("id").primaryKey(),
    name: text("name").notNull(),
    version: integer("version").notNull().$defaultFn(() => 0).$onUpdateFn(() => sql`version + 1`);
});
export const reservations = pgTable("reservations", {
    id: uuid("id").primaryKey(),
    roomId: uuid("room_id").notNull().references(() => rooms.id),
    owner: text("owner").notNull(),
    start: timestamp("start").notNull(),
    end: timestamp("end").notNull(),
    version: integer("version").notNull().$defaultFn(() => 0).$onUpdateFn(() => sql`version + 1`);
});
export const roomsRelations = relations(rooms, ({ many }) => ({
    reservations: many(reservations),
}));
export const reservationsRelations = relations(reservations, ({ one }) => ({
    room: one(rooms, { fields: [reservations.roomId], references: [rooms.id] }),
}));

export const clients = pgTable("relic_clients", {
    id: uuid("id").primaryKey(),
    mutationId: integer("mutation_id").notNull(),
});

export const clientViews = pgTable("relic_client_views", {
    id: uuid("id").primaryKey(),
    createdAt: timestamp("created_at"),
    data: json("data").notNull(),
});

export const pool = new Pool({ connectionString: process.env.POSTGRES_CONNECTION_STRING });
export const db = drizzle(pool, {
    schema: { rooms, reservations, roomsRelations, reservationsRelations, clients, clientViews }
});
```

Listing 4: The schema and setup logic for the server database

As shown, the rooms and reservations tables and their relations from the client database schema are almost copied. The only addition is the version column in these tables, which allows the server to generate delta messages.

Additionally, we have defined *relic_clients* and *relic_client_views* tables. This data is required for Relic to keep track of metadata related to each connected RelicClient.

Finally, the Postgres connection pool is initialized, and the Drizzle interface to use the ORM is set up.

With the logic for using the server database, we will initialize a new RelicServer, after which we can implement the mutators defined in the RelicDefinition. The initialization of a new RelicServer requires a RelicDefinition that the server will implement. This snippet is shown in listing 5.

```
const s = initRelicServer(relicDefinition)
    .transaction<ExtractTransaction<typeof db>>()
    .context<{ user: string; }>();
```

Listing 5: The initialization of the RelicServer

First, we call initRelicServer with our relicDefinition. Then, we register the type of the transaction object for our database. The transaction object will be used in mutators to access the database. It allows the Relic framework to run mutators and update metadata atomically. Furthermore, we define a context that will also be used in mutators. For the meeting room scheduler, we define that every mutator has access to the name of the user who calls the mutator.

With the initialized RelicServer, we have now registered all data that is needed to implement the mutators. The implementation of the mutators is shown in listing 6.

```
const deleteReservation = s.mutation.deleteReservation.mutate(
    async ({ tx, input }) => {
        await tx.delete(reservations).where(eq(reservations.id, input));
    }
);

const updateReservation = s.mutation.updateReservation.mutate(
    async ({ tx, input }) => {
        await tx.update(reservations).set(input).where(eq(reservations.id, input.id));
    }
);

const createReservation = s.mutation.createReservation.mutate(
    async ({ tx, input, ctx }) => {
        const conflicts = await getReservationConflicts(tx, input.roomId, input.start, input.end);
        if (conflicts.length) return;

        await tx.insert(reservations).values({ ...input, owner: ctx.user });
    }
);
```

Listing 6: The implementation of the mutators as defined in the RelicDefinition for the RelicServer

We use the mutation property from the RelicServerBuilder to access mutations defined in our RelicDefinition. In the IDE, the LSP will autocomplete the defined mutators after typing s.mutation. We implement a mutator by passing a handler to the mutate function, after which the mutator is finished. In this handler, we have access to three objects. The first object is tx, the transaction of the type which we have defined earlier. Then, we have the input object, which is the type of the input that was defined for that mutator in the RelicDefinition. Finally, we have the ctx (Context) object of the type `{ user: string; }`, as we have defined it to be when initializing the relic server.

In the deleteReservation mutation, we use Drizzle to delete a reservation within a transaction. In this mutator, we have defined the input to be of type string. In the updateReservation mutation, we update an existing reservation. For this mutator, the input was defined as `{ id: string; start?: Date; end?: Date; }`. Finally, in the createReservation mutation, we have implemented custom application-specific conflict resolution. The getReservationConflicts function is removed for brevity. It returns all reservations that overlap with a specific room, start time and end time. If there are any conflicting reservations, we do not create a new reservation for that room. In this mutator, we also use the context. We specify the owner of a new reservation as the name of the user from the mutation context.

For the size of this application, it is not necessary, but it is important to highlight that these mutations can be extracted into other files. More importantly, this entire piece of code is type-safe. It is difficult to show this using simple code highlighting on paper, but hovering over the input object, ctx object will tell the developer the type of the object. If a mutation name has a spelling error, the IDE will report this to the developer and the build would fail at compile time. Furthermore, developers get autocomplete when using all of these objects; typing `tx.` will show developers all possible options, such as delete, insert and update.

We will now implement the puller and poker of our server. The puller is responsible for creating a delta message with all updates to the client database since a previous point in time. To implement this, we will use the version per row method as outlined in chapter 5. As discussed, the framework includes an adapter for developers to use this method. The implementation is listed in listing 7

```
const puller = s.puller.pull(
    rowVersion(
        rowVersionPostgresAdapter(),
        async ({ tx }) => {
            return {
                reservations:
                    await tx.select({ id: reservations.id, version: reservations.version })
                            .from(reservations),
                rooms:
                    await tx.select({ id: rooms.id, version: rooms.version }).from(rooms),
            };
        },
        async ({ tx, entities }) => {
            return {
                reservations:
                    await tx.select().from(reservations)
                            .where(inArray(reservations.id, entities.reservations)),
                rooms:
                    await tx.select().from(rooms).where(inArray(rooms.id, entities.rooms)),
            };
        }
    )
);
```

Listing 7: The implementation of the puller of the application

We create a new row version puller by passing the rowVersion adapter when creating a new puller. The adapter has three arguments: one interface to communicate with the database, a method to allow developers to select which rows a client has access to, and one method to resolve only the changed rows.

The database interface is required for the framework to store data related to previous client pulls. In the handler to specify the view, we tell the framework all rows which a RelicClient has access to. We only select the id and version, not the entire row to reduce database network transfer. In this case, we simply select all rooms and all reservations. We could also select only the rooms and reservations of the user invoking the puller to enforce authorization.

In the second handler, we have access to an entities object. This object contains all entities that have changed between the previous pull and the current pull. Now, we select the entire row instead of only the id and version to return to the client.

Again, all handlers and return types are type-safe. The rooms and reservations have been specified by the client schema. If these handlers do not return data that fits in the client schema, developers will get a type error, and the build will fail.

Next, we define the poker. The poker is used to send real-time notification to clients and is called by the RelicServer when the data changes and clients may wish to pull to receive new data.

```
const pokeEmittor = new EventEmitter();
const poker = s.poker.poke(async () => pokeEmitter.emit("poke"));
```

Listing 8: The implementation of the poker of the application

As shown in 8, the poker emits a poke event on a NodeJS EventEmitter. This EventEmitter will later be used to relay notifications over server-sent events. Every connected client will receive a notification if any client makes a change. In multi-tenant applications, the logic to select which clients should be poked would be implemented here.

We have now implemented all the server application logic. We will now put everything together in a RelicServer object, and expose this RelicServer using a HTTP server.

```
export const relicServer = s.pull(puller).poke(poker).mutations({
    createReservation,
    deleteReservation,
    updateReservation,
});
```

Listing 9: The assembly of the RelicServer object

In listing 9, we compose all parts we have created up to this point, we build the RelicServer by specifying the puller, the poker and all mutations. If we forgot a mutation that was defined in the RelicDefinition, we would get a type error.

```
const app = new Hono();
app.post("/relic/*", (c) => {
    const user = c.req.query("user");
    if (!user) return new Response("Unauthorized", { status: 401 });

    return handleRelicRequest({
        relicServer,
        req: c.req.raw,
        context: {
            user,
        },
        database: postgresAdapter(db),
    });
});
app.get("/relic/poke", (c) => {
    return streamSSE(c, async (stream) => {
        pokeEmitter.on("poke", () => stream.writeSSE({ data: "" }));

        await new Promise((resolve) => pokeEmitter.on("end", resolve));
    });
});
serve({ fetch: app.fetch, port: 3000 });
```

Listing 10: Creating an HTTP server to handle all relic requests

In listing 10, we show the code needed to serve the RelicServer using the Hono HTTP server library. On every POST request on path /relic/*, we get the user's name from the query string. In a production server with real authorization, you would extract the sessionId from the cookies or the JWT token from the authorization header. Then, we call the handleRelicRequest function from the Relic framework. We pass this function with our relicServer, the Request object, our custom context, and the database adapter for Relic to interface with our database, which is included in the framework.

Additionally, we expose another endpoint, /relic/poke. This endpoint responds with a long-lived server-sent events stream. When the pokeEmitter receives a "poke" event, as we have specified in our poker, an event is sent over the SSE stream.

This concludes the server's implementation. In summary, to create a server using Relic, we first initialize a new RelicServerBuilder with an existing RelicDefinition. We register the types of objects needed in our handlers: the transaction type and the context type. We then implement our handlers, implementing the mutator handler for every mutator as defined in our RelicDefinition and the puller and poker handlers. Finally, we build the RelicServer and expose this using an HTTP server using the handleRelicRequest function.

Client

With the server in place, we create the final part of our application, the client. For the front end, we use the Vite bundler and the React UI framework. The process for implementing the client-side part of Relic is generally the same as the backend part. We start by initializing a new RelicClient by passing the RelicDefinition.

```
const c = initRelicClient(relicDefinition).context<{ user: string }>();
```

Listing 11: Initializing a new RelicClient

In listing 11, we also pass a context; this context can be different and, in many cases, would be different than the server. But in this case, we define `{ user: string }` to be the context type again.

The next step is to implement the client-side mutators. For this application, the client-side mutator handlers implement the same logic as what is shown in listing 6. Instead of using Postgres, the client-side mutators use SQLite.

```
const deleteReservation = c.mutation.deleteReservation.mutate(
    async ({ tx, input }) => {
        await tx.delete(reservations).where(eq(reservations.id, input));
    }
);

const updateReservation = c.mutation.updateReservation.mutate(
    async ({ tx, input }) => {
        await tx.update(reservations).set(input).where(eq(reservations.id, input.id));
    }
);

const createReservation = c.mutation.createReservation.mutate(
    async ({ tx, input, ctx }) => {
        const conflicts = await getReservationConflicts(tx, input.roomId, input.start, input.end);
        if (conflicts.length) return;

        await tx.insert(reservations).values({ ...input, owner: ctx.user });
    }
);
```

Listing 12: The implementation of the mutators as defined in the RelicDefinition for the RelicClient

As you can see, the code in the client-side mutators in listing 12 is almost exactly the same as the code for the server. However, the types of tx, reservations, and rooms are different, as these objects refer to the client-side SQLite transaction and the rooms and reservations tables defined in the client-side schema. The code does not have to be the same, but for this application, it will be.

Next, we assemble the RelicClient in listing 13

```
export const relicClient = c.mutations({
    createReservation,
    updateReservation,
    deleteReservation,
});
```

Listing 13: The creation of the RelicClient

This RelicClient cannot yet be used. The RelicClient needs some objects that are required at runtime. This is the following step and shown in listing 14.

```
export const queryClient = new QueryClient();

export const sqlite = await createSqliteWasmDb(new SqliteWasmWorker());
export const db = drizzle(sqlite, { schema: drizzleSchema });

const user = "John Doe";

const url = "http://localhost:3000/relic?user=" + encodeURIComponent(user);

export const relic = await createRelicClient({
    relicClient,
    queryClient,
    db,
    sqlite,
    url,
    context: {
        user,
    },
    poker: ssePoker(),
});
```

Listing 14: Creating a RelicClientInstance that can be used in the application

A lot happens in this snippet, and we will go through it one by one. The first thing we do is create a QueryClient, this is an object from TanStack Query that is responsible for asynchronous state management and caching. Then, we create a sqlite connection using the @relic/sqlite-wasm library. This object implements the RelicSQLiteDB interface and can thus be used as the client database for Relic. Then, just like the server, we create the drizzle db object, which allows us to communicate with the database in a type-safe manner. Next, we specify the URL where the client can reach the RelicServer to synchronize. We also set a user variable as we chose to require one in the context of our mutators. For simplicity, we define it to be ”John Doe”. Finally, we specify that the clients can receive notifications using the built-in server-sent events poke adapter.

Now, we have completed all setup needed to initialize Relic. We can now use the relic object throughout the application to query data and invoke mutations. The application UI will feature a form where users can create new reservations, a table which shows the current reservations, and a button to delete existing reservations.

To create a new reservation, we call the relic.mutate.createReservation function, as shown in listing 15.

```
relic.mutate.createReservation({
    id: crypto.randomUUID(),
    roomId,
    start,
    end,
});
```

Listing 15: Creating a new reservatiosn

This function expects the mutator input as an argument. We pass a new UUID as the reservation ID, a roomId, a start and end variable that describes the ID of the room we want to reserve, and the start and end times.

The code to delete a reservation in listing 16 is equally simple.

```
relic.mutate.deleteReservation(reservationId);
```

Listing 16: Deleting an existing reservation

Here, we pass a string containing the reservationId to the deleteReservation mutator, which expects a string, as we have defined in the mutator input.

Under the hood, a lot of things happen when invoking createReservation or deleteReservation. First, the mutation is applied locally. All UI updates to reflect the updated data. Then, the mutation is sent to the server, where it is applied authoritatively. The server then notifies all clients of a state change. The clients initialize a new pull request to get the authoritative state of the server. The locally applied mutation is rolled back, and the authoritative state is applied. Finally, the UI is updated again to reflect the authoritative state. Crucially, developers do not have to think about this at all when creating or deleting a reservation; they just call the function, and the Relic framework takes care of the rest. Developers also do not have to worry about runtime errors, as everything is type-safe.

Now, we will show how to query data. In the application, a table is shown which presents the reservations in the system. We retrieve the data for this table in 17.

```
const dbQuery = db.query.reservations.findMany({with: {rooms: true}});
const { data } = useSuspenseQuery(relic.query(dbQuery));
```

Listing 17: Reading the reservations data

The dbQuery variable represents a Drizzle database query. This query finds all reservations from our reservations table, and includes the room for each reservation, as we have defined in our relations. This query is then passed to the `relic.query` method, which turns the database query into a Tanstack Query query. Using useSuspenseQuery, we use React Suspense to wait for this data to load. The result of the database query is then available in the data variable.

Crucially, this query is reactive. If data in the database changes, the query is re-run, and React will rerender the data. Furthermore, we could dynamically change the database query, which would also reactively cause a rerender. Unlike mutations, queries do not have to be pre-defined; therefore, dynamic queries can be made anywhere in the application. This allows developers to co-locate data fetching in components.

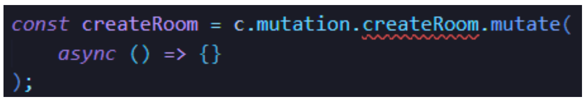
We have shown how Relic on the client is set up for the meeting room scheduler application. Furthermore, we presented how the application invokes mutations and can read data.

Type Safety

We now show the type safety of the application implementation. To show type safety, we include images from the Visual Studio Code code editor, which uses the TypeScript LSP to show inline errors. We will explicitly focus on showing type safety regarding types which are implicitly defined by the user.

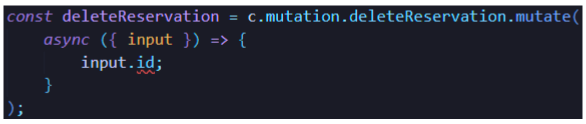
The construction of the RelicDefinition serves as the source of truth for the client and server implementation. Therefore, no advanced safety mechanisms are used when creating the RelicDefinition.

On the client, we must ensure that the developer correctly implements the contract as specified in the RelicDefinition. As such, type safety is enforced in many places.



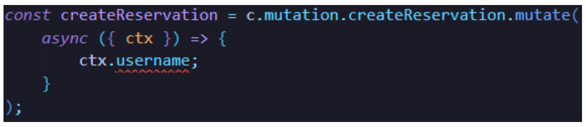
```
const createRoom = c.mutation.createRoom.mutate(
    async () => {}
);
```

(a) Error when creating mutator that is not defined in definition



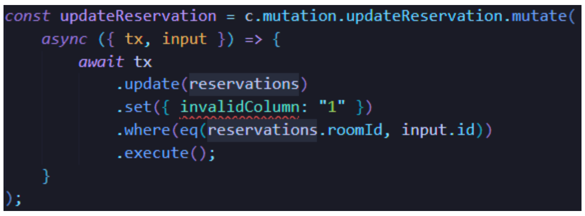
```
const deleteReservation = c.mutation.deleteReservation.mutate(
    async ({ input }) => {
        input.id;
    }
);
```

(b) Error when using an input that was not defined in the definition



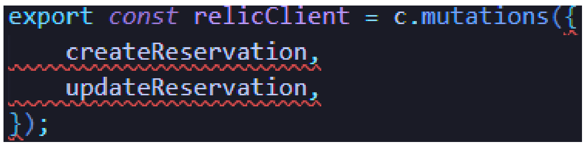
```
const createReservation = c.mutation.createReservation.mutate(
    async ({ ctx }) => {
        ctx.username;
    }
);
```

(c) Error when using a property that does not exist on defined context type



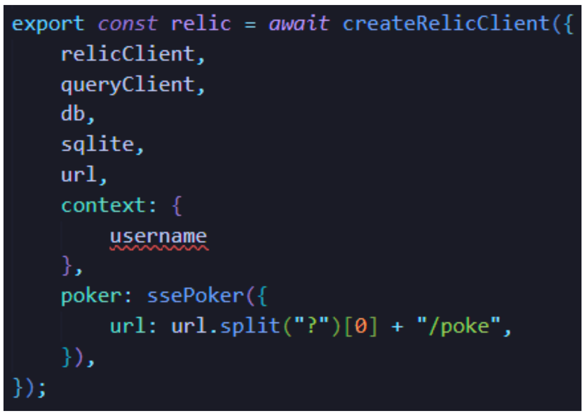
```
const updateReservation = c.mutation.updateReservation.mutate(
    async ({ tx, input }) => {
        await tx
            .update(reservations)
            .set({ invalidColumn: "1" })
            .where(eq(reservations.roomId, input.id))
            .execute();
    }
);
```

(d) Error when using the database incorrectly



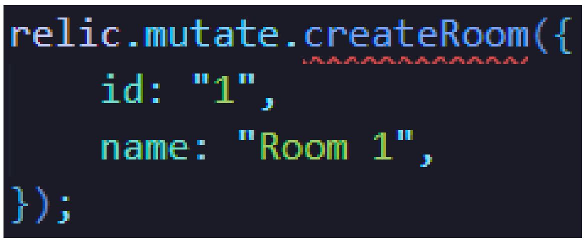
```
export const relicClient = c.mutations({
    createReservation,
    updateReservation,
});
```

(e) Error forgetting to implement a mutation



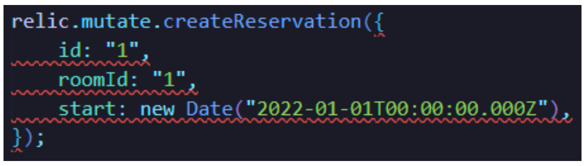
```
export const relic = await createRelicClient({
    relicClient,
    queryClient,
    db,
    sqlite,
    url,
    context: {
        username
    },
    poker: ssePoker({
        url: url.split("?")[0] + "/poke",
    }),
});
```

(f) Error when passing a context object that does not satisfy the defined type



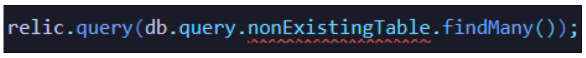
```
relic.mutate.createRoom({
    id: "1",
    name: "Room 1",
});
```

(g) Error when calling a non-existing mutation



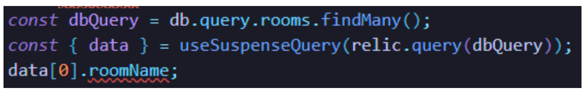
```
relic.mutate.createReservation({
    id: "1",
    roomId: "1",
    start: new Date("2022-01-01T00:00:00.000Z"),
});
```

(h) Error when calling a mutation with an incorrect input



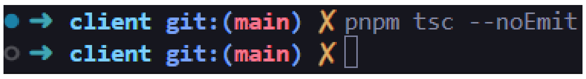
```
relic.query(db.query.nonExistingTable.findMany());
```

(i) Error when querying a table that does not exist



```
const dbQuery = db.query.rooms.findMany();
const { data } = useSuspenseQuery(relic.query(dbQuery));
data[0].roomName;
```

(j) Error when using a non-existing column of a table



```
● ➜ client git:(main) ✗ pnpm tsc --noEmit
○ ➜ client git:(main) ✗
```

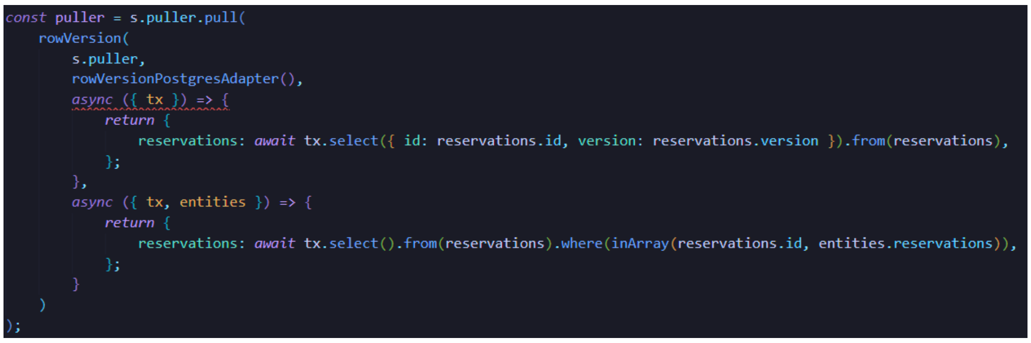
(k) When compiling the typescript of the correct implementation of the application, there are no errors

Figure 6.1: Client-side type safety of relic framework

As shown in figure 6.1, there are many types of errors that the framework catches for the developer.

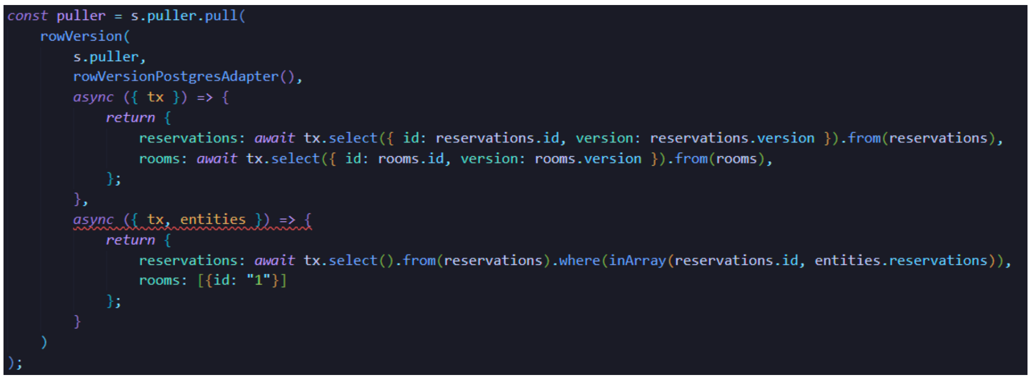
In 6.1a, an error is reported when the developer tries to implement a mutator that is not defined in the RelicDefinition. In 6.1b, the input is of type `string`, as defined in the RelicDefinition, therefore it has no id property. Figure 6.1c shows the developer accessing a property that was not defined in the context type, it should be `ctx.user`. In 6.1d, the developer tries updating a column of the reservations table. When registering the mutator implementations in 6.1e, the developer is notified when they have forgotten a mutator implementation. When creating the relic object, the developer passes an incorrect context object in 6.1f. In 6.1g, the developer tries to invoke a mutator that was not specified in the RelicDefinition. In 6.1h, the developer forgets to pass an end date to the createReservation mutator. Figure 6.1i, the developer is notified when they try to query a table that was not registered in the schema. In 6.1j, the developer queries all rooms, but tries to access a column that does not exist. Finally, we show that if all these errors are fixed, the compilation of typescript succeeds without errors in figure 6.1k.

We will now show the type safety on the server-side. We will not repeat the errors related to implementing mutations and contexts as shown in figure 6.1a to figure 6.1f, as they are the same as on the client. Besides implementing mutators and the context object, developers mainly benefit from type safety on the server-side in the puller.



```
const puller = s.puller.pull(
    rowVersion(
        s.puller,
        rowVersionPostgresAdapter(),
        async ({ tx }) => {
            return {
                reservations: await tx.select({ id: reservations.id, version: reservations.version }).from(reservations),
            };
        },
        async ({ tx, entities }) => {
            return {
                reservations: await tx.select().from(reservations).where(inArray(reservations.id, entities.reservations)),
            };
        }
    )
);
```

(a) Error when creating mutator that is not defined in definition



```
const puller = s.puller.pull(
    rowVersion(
        s.puller,
        rowVersionPostgresAdapter(),
        async ({ tx }) => {
            return {
                reservations: await tx.select({ id: reservations.id, version: reservations.version }).from(reservations),
                rooms: await tx.select({ id: rooms.id, version: rooms.version }).from(rooms),
            };
        },
        async ({ tx, entities }) => {
            return {
                reservations: await tx.select().from(reservations).where(inArray(reservations.id, entities.reservations)),
                rooms: [{id: "1"}]
            };
        }
    )
);
```

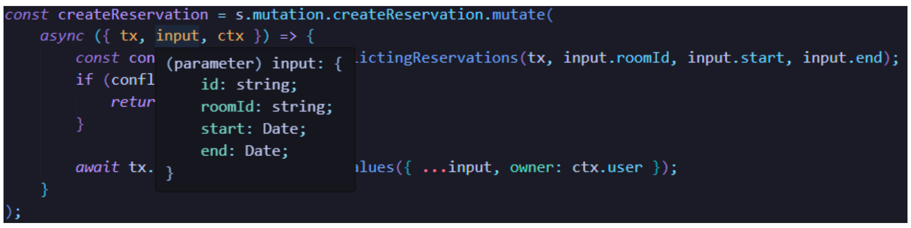
(b) Error when forgetting to return the name column of the rooms table

Figure 6.2: Server-side puller type safety of relic framework

In figure 6.2, we show two situations where a developer made a mistake implementing the puller. In figure 6.2a, the developer forgot to return the data for the rooms table as defined in the client-side schema. In 6.2b, the developer forgot to return a column from the rooms table, the name column.

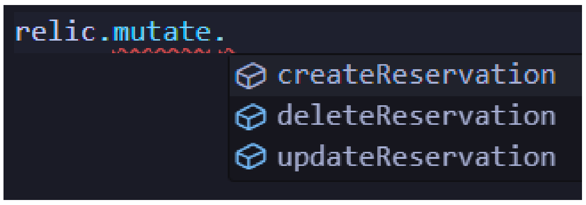
As can be seen, both the server and client are type-safe. Furthermore, by using the RelicDefinition, we have achieved end-to-end type safety as the data that is sent over the network is defined in this RelicDefinition, which the client and server must correctly implement to be able to compile without errors.

Finally, we highlight some of the IDE benefits that developers additionally gain from the type safety of the relic framework in figure 6.3



```
const createReservation = s.mutation.createReservation.mutate(
    async ({ tx, input, ctx }) => {
        const con (parameter) input: {  .ictingReservations(tx, input.roomId, input.start, input.end);
        if (confl     id: string;
            retur     roomId: string;
        }             start: Date;
                      end: Date;
        await tx.  }                     ılues({ ...input, owner: ctx.user });
    }
);
```

(a) When hovering over any variable in Relic, the IDE shows its type



```
relic.mutate.
             createReservation
             deleteReservation
             updateReservation
```

(b) Autocomplete support when writing code using Relic

```
relic.mutate.deleteReservation("1");
```

(c) When the cursor is on an object, clicking F12 (Go to definition) brings the developer to the definition

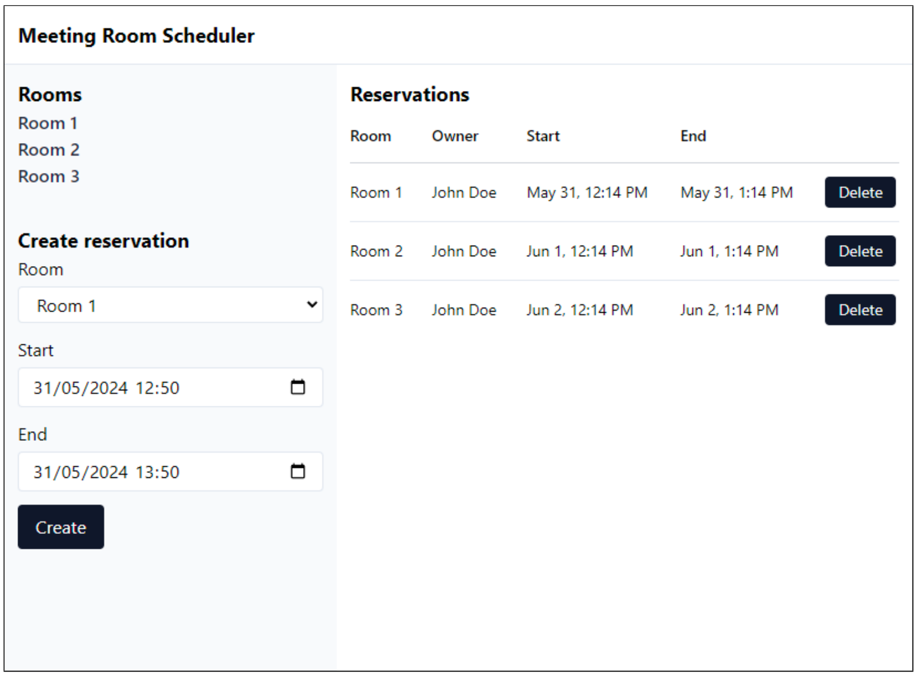
Figure 6.3: Additional IDE benefits resulting from type safety of Relic

When hovering over any object in relic, such as the input, the ctx, the relic object, the transaction or database object, developers can see the inferred type without having to look up how that object is defined. This is shown in 6.3a for the input object in the createReservation mutator. In figure 6.3b, we show that developers get autocomplete for the properties of all objects. As shown, typing `relic.mutate` shows the available mutations. Other objects like `db.query` show the possible tables to query. Finally, when using the go to defintion action definition on objects, developers will navigate to the definition. When using go to definition on the deleteReservation method as shown in 6.3c, the developer will be navigated to the implementation of the deleteReservation mutator. Go to definition on any table in a database query will bring developers to the schema of the table.

Demonstration

We will now demonstrate the implementation of the meeting room scheduler application. In this demonstration, we will first show the application itself. Then, we will highlight the Offline-First aspects of the website. Finally, we show a situation that highlights the custom application-specific conflict resolution.

The application features two pages. On the homepage, users can see all reservations and create new reservations. Users can click a room to view the reservations associated with that room on the room's page.



Meeting Room Scheduler

Rooms

Room 1
Room 2
Room 3

Create reservation

Room: Room 1

Start: 31/05/2024 12:50

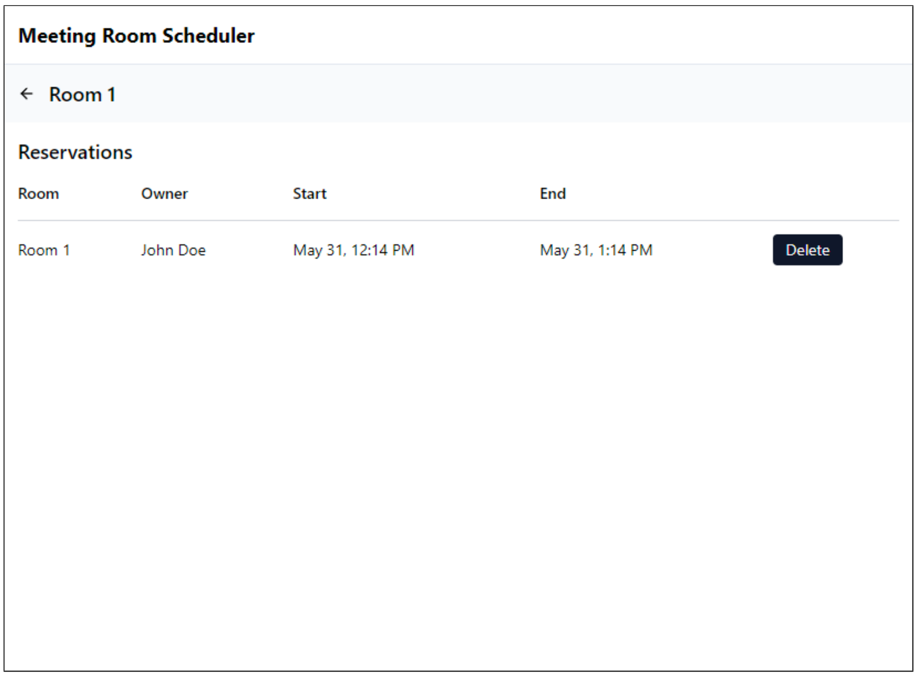
End: 31/05/2024 13:50

Create

Reservations

Room	Owner	Start	End	
Room 1	John Doe	May 31, 12:14 PM	May 31, 1:14 PM	Delete
Room 2	John Doe	Jun 1, 12:14 PM	Jun 1, 1:14 PM	Delete
Room 3	John Doe	Jun 2, 12:14 PM	Jun 2, 1:14 PM	Delete

Figure 6.4: The homepage of the meeting room scheduler application



Meeting Room Scheduler

← Room 1

Reservations

Room	Owner	Start	End	
Room 1	John Doe	May 31, 12:14 PM	May 31, 1:14 PM	Delete

Figure 6.5: The page where users can see all reservations for a specific room

The homepage in figure 6.4 features a sidebar where you can go to the page of a room and create a new reservation. Furthermore, a table containing data of all reservations is shown. In 6.5, the room page is shown for room 1.

The website itself is simple, and UI-wise, it is not special. However, the website has special properties as it is Offline-First. This is difficult to show in images; therefore, we show the following interactions in videos. The sources of the videos are shown in table 6.1.

In video 1, the speed and responsiveness of the application are highlighted. The first time the site is loaded, it will take some time to load all data from the server. Subsequent loads, however, load instantaneously. In the video, we use Chrome's dev tools to simulate real-world network latency. Even with this latency, all reads and writes of the application are not affected. Switching between pages

Video	Source
1	`https://github.com/vigovlugt/relic-thesis-assets/raw/main/speed.mkv`
2	`https://github.com/vigovlugt/relic-thesis-assets/raw/main/offline.mkv`
3	`https://github.com/vigovlugt/relic-thesis-assets/raw/main/realtime.mkv`
4	`https://github.com/vigovlugt/relic-thesis-assets/raw/main/conflict.mkv`

Table 6.1: The demonstration videos showing the meeting room scheduler's speed, offline capabilities, real-time collaboration, and conflict resolution

between the homepage and the rooms page is immediate. Furthermore, creating new reservations and deleting existing reservations are also instantaneous.

In the next video, video 2, we show the offline capabilities of the application. First, we use the Chrome dev tools to simulate an offline environment where the browser is not connected to the internet. We show that the application still has read and write abilities.

Next, in video 3, we demonstrate real-time collaboration. Two independent browsers interact with the application, and both users can see each other's changes in real time.

In the final video, video 4, we show the custom conflict resolution. In the meeting room scheduler application, we have created custom conflict resolution where users may not create a reservation for a room which already has a reservation at that time. To show this conflict resolution in the application, we show two offline browsers. Both users create a reservation for the same room at the same time while offline. After the first user reconnects, the reservation is made authoritatively on the server. When the second user reconnects, their reservation is sent to the server. The server sees a conflict, and thus, that reservation is not made. The second user's reservation is removed; they see only the reservation from the first user.

Note that the application does not provide a good user experience in the final video. The reservation of user 1 is deleted due to a conflict without the user being notified of this deletion. Furthermore, we would like users to be aware of which reservations are tentative, and which reservations are committed by the server. For this last issue, we can use the `relic.pendingMutations()` function. This function returns the current mutations that are only locally applied. The state of each reservation can then reflect whether that mutation is only locally created. For the notification issue, developers can implement an ”errorlog” approach, as presented in Bayou [50]. With this approach, an error log or notification table is created, which is appended with a notification when a conflict happens. The frontend can then respond to new notifications by showing them to the user. Furthermore, the server could send an email notifying the server of the deletion of the reservation on conflict.

Requirements

We will now show the requirements that are satisfied by this experiment. The first set of requirements related to Offline-First are primarily demonstrated in the video demonstrations.

- **R5:** The framework has consistent, responsive interactions which are not limited by network latency, as demonstrated in video 1
- **R1:** The framework allows for offline read and write operations when the network is inconsistent or unavailable, as shown in video 2
- **R4:** The framework synchronizes offline changes when the network connection is recovered, as shown in video 4
- **R6:** The framework allows for seamless multi-device access, as demonstrated in video 2 and 4
- **R7:** The framework allows multiple users to do work simultaneously and see updates in realtime, as shown in video 3
- **R2:** The framework handles concurrent updates with built-in flexible conflict resolution, as demonstrated in video 4 and shown in listing 6

The requirements regarding Developer Experience are demonstrated to be satisfied in the implementation of the case study.

- **R9:** The framework enforces type safety on the client and server, as demonstrated in figure 6.1 and 6.2
- **R10:** The framework offers end-to-end type safety between the client and server, spanning the network boundary, is satisfied by using the RelicDefinition to ensure type safety on the client and

server in 6.1 and 6.2.
- **R11:** The framework is full stack, covering the client and the server with one language, as shown in the server-side implementation in section 6.1.2 and the client-side implementation in section 6.1.2
- **R12:** The framework is designed to allow for type inference where possible, as the only place that the developer must manually specify types is to register a context type and transaction type. This is the only place where types are needed before the object is instantiated. Besides this, as seen in the implementation, everything in the framework supports type inference.

Finally, the requirements regarding modularity and flexibility are demonstrated in this experiment:

- **R13:** The framework allows developers to determine their own type of backend database. This requirement is demonstrated in 6.1.2, where we use an adapter to integrate a specific database with the backend. However, this requirement is only partially satisfied. Not every type of database can be used, the database must support transactions. Without transactions, we cannot ensure consistency, which would implicate **R5**
- **R14:** The framework gives developers the power to write their own flexible backend, which is also shown in 6.1.2. We show that developers implement write operations themselves and write code to specify what users can read

The final requirement that was not demonstrated in this experiment is **R8**: The framework must be able to handle a large amount of data. This requirement will be validated in the next two experiments.

6.1.3 Discussion

In this experiment, we have shown that the created application successfully satisfies the Offline-First principles, resulting in a great User Experience through offline usage, responsiveness and speed due to not being affected by network latency in interactions, providing easy user collaboration and seamless multi-device access.

We have also highlighted the Developer Experience of Relic, which is a crucial aspect of designing a framework that will be used by developers. We show that developers can create fully type-safe applications using the framework, providing type safety on the client and on the server and, crucially, end-to-end type safety over the network boundary. Type safety helps developers catch errors more quickly and reduces runtime errors. Furthermore, type safety allows IDEs to provide superior assistance than dynamically typed frameworks.

End-to-end type safety is made possible without using code generation by using a full-stack approach where the client and server are both written in TypeScript. Other end-to-end type-safe client-server libraries often rely on code generation, such as gRPC[1] and GraphQL[2]. By removing the code generation step, we improve Developer Experience by providing a shorter feedback cycle and ensuring less cognitive overhead.

We have designed the framework to allow for application-specific conflict resolution and have shown the implementation and demonstration in the experiment. Providing application-specific conflict resolution allows developers to maximally preserve user intent. Therefore, the framework makes it possible to provide users with a better User Experience than other Offline-First frameworks, which only support general-purpose conflict resolution. Furthermore, the client-side prediction and server reconciliation conflict resolution algorithm used by Relic offers developers a simple mental model. All mutations are ordered by the server and then executed, developers only have to reason about server-side and client-side mutations. The conflict resolution of Relic has several other properties that other frameworks do not provide, such as allowing developers to write invariants. As seen, with Relic we can write an invariant to stop users from making two reservations at the same time, which is not possible with many of the existing Offline-First frameworks.

We have also shown the flexibility and control that Relic provides developers. Where some other frameworks lock you in with a specific database, Relic allows developers to choose the best database for their use case. Furthermore, on the server side, developers have total control over write and read operations, allowing developers to define access control and any custom logic in code instead of using limited configurations such as other frameworks. On the other hand, we see that implementing a server in Relic is much simpler than other frameworks that require developers to implement a server from scratch. Relic simplifies and accelerates backend development without sacrificing flexibility and control.

On the client side, Relic also provides several important benefits. By having all accessible data locally,

[1]`https://grpc.io/`
[2]`https://graphql.org/`

developers can use a very powerful tool to create reactive and remarkably dynamic user interfaces that are much more flexible than only being able to access data in the form provided by a traditional REST API. Using embedded SQLite also brings benefits over other frameworks using a document store. The relational data model allows developers to optimally utilize relational data. Furthermore, we see that the inclusion of an ORM in the SQLite database solves many of the downsides of the SQL language, as described in the literature. The ORM allows for type safety, easy querying of relations, and reusing query fragments. Crucially, developers can still opt-in to use raw SQL when it makes sense.

6.2 Client-Side Performance

In this experiment, we aim to measure the client-side performance of read and write operations of the Relic framework.

6.2.1 Method

To measure client-side performance, we use the meeting room scheduler application as created in the case study. The benchmarks run without a server to remove network latency and server-side performance from the results to only measure the time for local data access and data writes. The server is benchmarked in the next experiment. We also do not measure the time for the data to be rendered using the browser; we only measure the time for the data to be available.

We measure several metrics of the application:

- **Initial load**: The time it takes for Relic to set up and load all data into the database when accessing the site for the first time. This load time does not include the time for the JavaScript bundle to load, only the time for the Relic code to run.
- **Subsequent load**: The time it takes for Relic to setup when all data is already loaded in the database on subsequent visits
- **Select N rows**: The latency for Relic to return the results of the query:
  `SELECT * FROM reservations LIMIT N`
- **Select 10 joined**: The latency for Relic to return the results of the query:
  `SELECT * FROM reservations LEFT JOIN rooms ON reservations.room_id = rooms.id LIMIT 10`
- **Create reservation**: The latency for the `createReservation` mutation to be completed
- **Update reservation**: The latency for the `updateReservation` mutation to be completed
- **Delete reservation**: The latency for the `deleteReservation` mutation to be completed

To measure the scalability of Relic on the client-side, we measure the impact of the total number of rows in the Relic database on the metrics. We repeat the metrics for the following row counts: 1, 10, 100, 1000, 10000, 100000. To get a reliable result, the metrics are repeated at least 10 times for every row count.

The measurements will be compared to the response time guidelines as outlined in Miller [115]. In this paper, three magnitudes for response times are described:

- 0.1 second is the limit for users to feel that responses are instantaneous.
- 1 second is the limit for the user to keep their flow of thought, but they will notice the delay.
- 10 seconds is the limit for the user's attention to be kept.

Importantly, these numbers are upper bounds, so responses over 0.1 seconds do not feel instantaneous for all or most users; lower response times may still not feel instantaneous for some users. Furthermore, we compare the response times with the frame times of 60FPS and 144FPS monitors: 16.67ms and 6.94ms.

The benchmark is run in the Google Chrome browser on a system with the following specifications:

- **CPU:** AMD Ryzen 7 5800X 8-Core 3.80GHz
- **RAM:** 16 GB
- **OS:** Windows 11 Home 64-bit Version 23H2
- **Browser:** Google Chrome Version 125.0.6422.142 (Official Build) (64-bit)

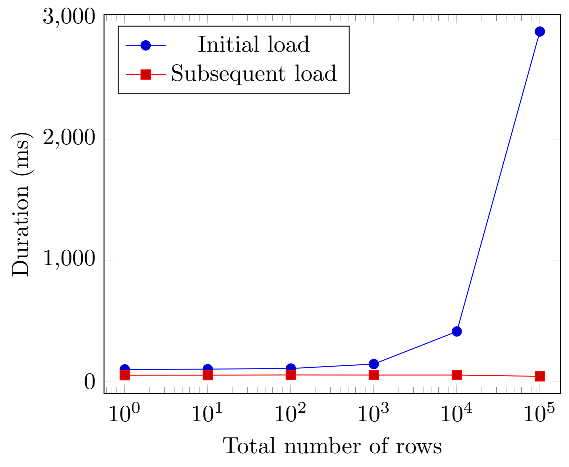
The code for the benchmark can be found at: `https://github.com/vigovlugt/relic/tree/main/benchmarks/client`.

6.2.2 Results

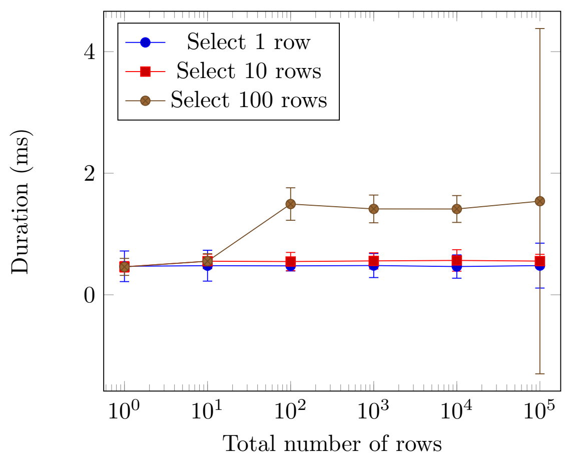
The results of the benchmark are summarized in table 6.2 and visualized in the graphs in figure 6.6:

Rows	**1**	**10**	**100**	**1000**	**10000**	**100000**
Initial load	96.7 ± 2.38	98.67 ± 5.52	103.77 ± 2.72	140.63 ± 3.67	409.73 ± 5.09	2888.3 ± 0
Subsequent load	48.58 ± 1.12	48.95 ± 1.41	50.51 ± 3.4	49.8 ± 2.69	49.87 ± 1.32	39.09 ± 0.74
Select 1 rows	0.47 ± 0.25	0.48 ± 0.25	0.48 ± 0.08	0.48 ± 0.2	0.46 ± 0.19	0.48 ± 0.37
Select 10 rows	0.46 ± 0.09	0.55 ± 0.12	0.55 ± 0.15	0.56 ± 0.13	0.57 ± 0.18	0.55 ± 0.11
Select 100 rows	0.46 ± 0.14	0.55 ± 0.12	1.49 ± 0.27	1.41 ± 0.23	1.41 ± 0.22	1.54 ± 2.84
Select 10 joined	0.5 ± 0.09	0.61 ± 0.13	0.64 ± 0.24	0.64 ± 0.15	0.64 ± 0.12	0.64 ± 0.13
Update reservation	10 ± 1.18	10.37 ± 1.03	11.06 ± 0.84	11.17 ± 1.12	11.28 ± 1.04	11.49 ± 1.62
Create reservation	12.15 ± 1.38	11.6 ± 2.19	10.38 ± 2.28	10.46 ± 2.09	11.18 ± 2.24	18.87 ± 2.41
Delete reservation	13.05 ± 0.75	13.16 ± 0.81	13.37 ± 1.16	13.04 ± 0.73	13.19 ± 1.31	12.64 ± 0.58

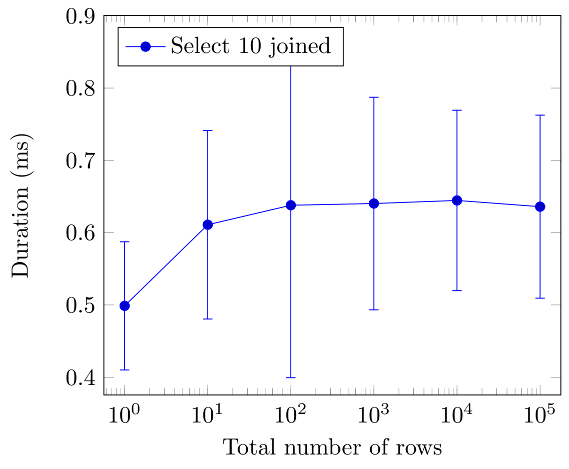
Table 6.2: Average duration in milliseconds of client-side operations for databases with different number of rows. (N ≥ 10)



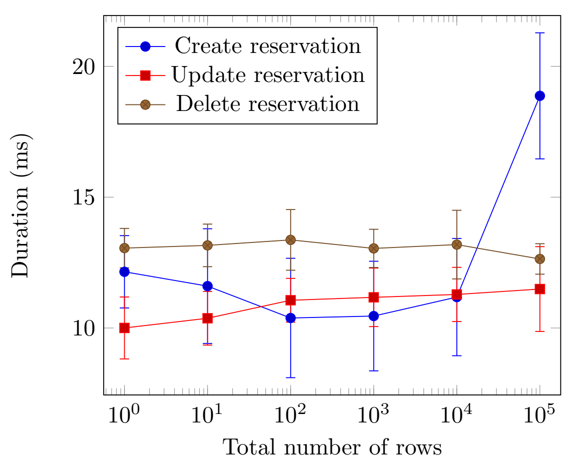
(a) Average duration in milliseconds of setting up and loading the data for a new and existing Relic instance by total number of rows. (N ≥ 1)



(b) Average duration in milliseconds of selecting 1, 10 and 100 rows by total number of rows in the database. (N ≥ 650)



(c) Average duration in milliseconds of selecting 10 rows using a join by total number of rows in the database. (N ≥ 1552)



(d) Average duration in milliseconds to completion of mutations by total number of rows in the database. (N ≥ 54)

Figure 6.6: Average latency of various client-side Relic operations for the meeting room scheduler

In figure 6.6a, we see the load times for setting up the Relic client. The first step in setting up Relic is creating a new OPFS SQLite WASM connection. This step takes up the majority of the time when setting up the Relic client in a subsequent page load. On initial page loads, the majority of time is spent inserting the data from the server into the database. We see that loading Relic in subsequent website visits, where the data is already inserted in the database takes around 50ms. For initial page visits, where there is no data in the database, the load times increase with the database size, taking 3 seconds to load a database with 100000 rows.

In figure 6.6b, we see the latency of simple SQL queries. Selecting 1 row takes around 0.5ms, independent of the total rows in the database. Selecting more rows, 10 or 100, increases the latency to 0.5-1.5ms. Selecting 10 rows in a database with 1 row is the same as selecting 1 row, or selecting 100 rows in a database with 10 rows is the same as selecting 10 rows, which explains the sudden increases that flatten out over time. Furthermore, we see that the latency of a select of 10 rows with a join in figure 6.6c takes around 0.6ms.

Figure 6.6d, we see the latency of mutations. For row counts under 100000, updating mutations and creating mutations take around 12ms, and deleting reservations takes around 13ms. For a database with 100000 rows, the latency of creating a reservation increases to 19ms.

6.2.3 Discussion

The load times for initializing Relic highly depend on whether the website has already been launched once before, which impacts whether the application data is already inserted into the database. We see that when loading Relic in a subsequent visit, the setup times are around 50ms, which is constant over the size of the database. 50ms is under the limit of instantaneous response perception of 100ms. End-users who will use Relic applications would likely not feel the impact of this latency. Regarding the loading times for the initial visit, the loading time is around 100ms for databases with less than 100 rows. While this will not feel instant, and users will notice the delay, this time is still acceptable. For databases with rows under 10000, the loading time is 400ms, which is still very acceptable for users to wait until the site first loads. Loading 100000 rows incurs a loading time of 2.9 seconds. This is much more than users would expect from non Offline-First websites, but for only the first setup it should be acceptable, staying under the 10 second limit.

Importantly, these loading times do not paint the whole picture. Besides the time for Relic to set up locally on the client, loading times in real applications are increased by other factors. Clients have to load the JavaScript bundle and the SQLite WASM binary and wait for the server and network transfer to retrieve the rows to insert in the database.

For the select rows metrics, we see latencies of 0.5ms for 1 row to 1.5ms for 100 rows. In an application, one page will likely display the data of multiple queries. With these latencies, applications will be able to make multiple queries and still stay under the frame time of a 144hz monitor. In this experiment, we have only measured the time for the data to become available in JavaScript; other sources, such as the UI library, the browser renderer and the latency in the monitor, will increase the interaction latency further. However, this is true for any application. The times discussed show the overhead of the Relic database. Furthermore, the query times heavily depend on the specific SQL statement, the database indexes, and the type of application.

For mutations, we see latencies of around 10-13ms. The create reservation mutation's latency increases with more rows, as the create reservation mutation also checks for conflicts with an SQL query that must scan rows. For all mutations, latencies exceed the frame time of a 144hz monitor. However, they can still be calculated within one frame for 60hz monitors. Together with the time of executing the mutations, mutations invalidate the queries that are affected by that mutation. Therefore, users will also have to wait for these queries to be re-queried, adding to the latency of the mutation.

In creating these benchmarks, we implemented several optimizations to decrease these latencies, which we will discuss now.

To reduce the overhead of Web Worker messages, we have implemented an additional method on the SQLiteDB interface: `execBatch`, which allows the framework to batch multiple SQL statements together, sending it in one Web Worker message. This has resulted in a -33% latency decrease in the initial load for databases with 100000 rows.

Additionally, we have created an optimization to temporarily deactivate the rollback table triggers when we insert the data from the server. Before this optimization, the triggers resulted in additional overhead by writing each insert to the database twice. This optimization resulted in an additional -25% latency decrease for the initial load of 100,000 rows databases.

There are several other possible optimizations that can slightly increase the performance of the Relic framework. However, by far, the biggest bottleneck is the performance of the SQLite database. Increasing the performance of SQLite-WASM would increase the performance of the Relic framework in all operations. Unfortunately, SQLite in the browser is currently (as of 12-06-2024) not close to the same performance level as SQLite in desktop or even mobile environments. We see this particularly in the write operations in each of the mutations and the initial load. SQLite on the web cannot use WAL mode[3], which would heavily increase the performance of all write operations. Possibly, WAL mode SQLite wasm can be achieved in the future, and therefore, highly improve the latencies of SQLite-WASM and, therefore, Relic applications.

6.3 Server-Side Performance

In this experiment, we will measure the server-side performance of the Relic framework.

6.3.1 Method

Like the client-side experiment, we will benchmark the server-side performance using the meeting room scheduler application. For every operation, we will measure the maximum throughput in requests per second, and the average latency for the client to receive a response. The following operations will be measured:

- **Initial pull**: Serving the pull data for a client that requests the data from the server for the first time. This response will include all rows to which the client has access.
- **Pull N changes**: Serving the pull data for a client that already has a local state. The client has already completed an initial pull. The server will then make N changes to the database. Then, we measure the time it takes for the client to receive the delta message, which includes these N changes, to get the client up to date with the server.
- **Create reservation**: In this operation, the client makes a push request with a single createReservation mutation.
- **Update reservation**: For this operation, the client makes a push request with a single updateReservation mutation.
- **Delete reservation**: The client makes a push request with a single deleteReservation mutation.

Again, we will measure the scalability of the server. To do this, we repeat these measurements for different client-side row counts: 1, 10, 100, 1000, 10000, 100000. Importantly, these row counts do not represent the total rows in the server database but the amount of rows that a single client has access to. For example, the server-side database may have 2 million rows, of which the client may access 100. We measure the impact of the client-side row count as this has the most impact on the pull endpoint. While the total server-side row count will also have an impact on the performance of the server, that can be optimized outside of the Relic server by using indexes, for example, and has a much lower impact than the total amount of rows that are sent to the client.

The Relic server uses the row version delta message adapter. As we have discussed, this adapter enables developers with a lot of flexibility and a high push throughput. We expect the pull throughput to be lower than that of other adapters. Other adapters, such as a global version or version per table, are not measured in this experiment.

To get insight into the Relic server performance, we will compare the performance to the performance of a normal non Offline-First REST server. While there are important distinctions between an Offline-First server and a normal server, this can provide some insight into the overhead of the Relic framework and Offline-First strategies. The REST server serves the following endpoints:

- **GET** /reservations
  Getting 10 reservations
- **POST** /reservations
  The createReservation mutation
- **PUT** /reservations/:id
  The updateReservation mutation
- **DELETE** /reservations/:id
  The deleteReservation mutation

[3]`https://www.sqlite.org/wal.html`

We will compare the performance of the POST, PUT and DELETE endpoints with the push endpoint of the Relic server. We will also discuss the performance difference between the GET endpoints and the Relic pull endpoint, although these endpoints are much harder to compare due to the characteristics of Offline-First.

The Relic and REST servers are benchmarked using the K6[4] benchmarking framework. Each test is run with 100 clients over 60 seconds. The benchmarking client and server are run on different machines on the same network to reduce network latency. Both machines are Hetzner CCX13 servers with dedicated vCPUs with the following specifications:

- **CPU:** AMD Milan EPYC 7003 2-Core 2.40GHz
- **RAM:** 8 GB
- **OS:** Ubuntu 24.04

The REST and Relic servers are run on the latest LTS version of Node.js, Node.js v20.15.0, as of 24/06/2024. The node process is run with `export NODE_OPTIONS=--max-old-space-size=7500` to ensure the server does not run out of memory. Both servers serve HTTP over the Hono web framework, with version v4.4.4. Finally, both servers use Postgres database version 16.3-1 in a docker container on the same machine.

The benchmarking code can be found at: `https://github.com/vigovlugt/relic/tree/main/benchmarks/server`

6.3.2 Results

The results of the Relic server-side benchmark are summarized in table 6.3, which shows the throughput, table 6.4 which shows the latency, and visualized in the graphs in figure 6.7.

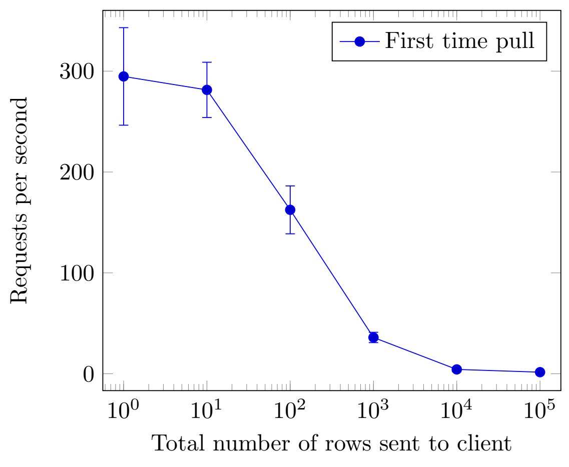
Rows	**1**	**10**	**100**	**1000**	**10000**	**100000**
Initial pull	294.74 ± 48.34	281.39 ± 27.41	162.48 ± 23.74	35.98 ± 5.13	4.3 ± 2.17	1.57 ± 1.16
Pull 1 change	341.37 ± 51.64	340.34 ± 43.74	279.71 ± 36.27	118.73 ± 28.39	11.76 ± 6.25	1.39 ± 0.85
Pull 10 changes	310.97 ± 40.85	312.21 ± 37.56	252.52 ± 43.35	114.2 ± 27.43	12.21 ± 6.61	1.39 ± 0.75
Pull 100 changes	179.13 ± 18.47	178.61 ± 16.95	159.37 ± 22.52	89.95 ± 18.99	11.88 ± 6.26	1.39 ± 0.92
Create reservation	290.34 ± 56.88	283.03 ± 58.17	289.11 ± 58.15	275.24 ± 66.07	240.51 ± 49.09	82.88 ± 15.65
Update reservation	489.08 ± 69.23	483.39 ± 66.19	495.26 ± 69.69	503.59 ± 104.61	502.71 ± 72.38	491.16 ± 92.29
Delete reservation	504 ± 72.47	507.47 ± 73.44	501.74 ± 75.56	538.34 ± 79.11	535.61 ± 88.43	508.84 ± 98.73

Table 6.3: Througput in requests per second for server-side operations per number of rows in client state. (N $\geq$ 61)

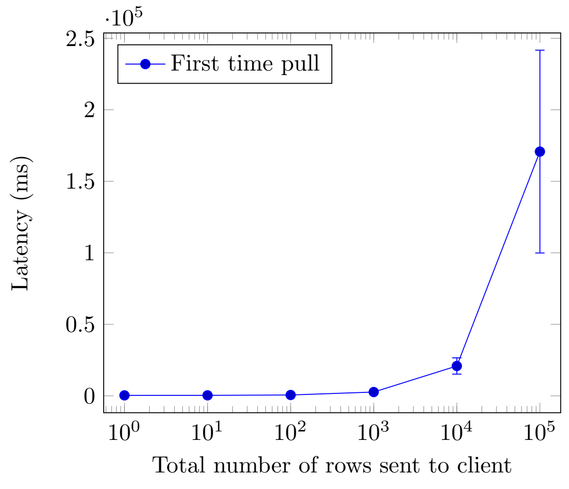
Rows	**1**	**10**	**100**	**1000**	**10000**	**100000**
Initial pull	334.08 ± 55.22	350.06 ± 20.4	597.88 ± 38.37	2660.75 ± 295.04	20868.12 ± 5698.77	170758.67 ± 70889.48
Pull 1 change	283.71 ± 30.27	284.5 ± 26.96	346.47 ± 32.85	770.61 ± 99.23	5509.3 ± 1883.01	25353.85 ± 25488.61
Pull 10 changes	311.5 ± 28.68	310.27 ± 30.02	378.13 ± 38.96	801.39 ± 105.21	5282.64 ± 1763.5	25679.8 ± 24802.24
Pull 100 changes	541.92 ± 67.3	543.69 ± 65.73	599.74 ± 77.3	1019.14 ± 151.45	5482.64 ± 1867.87	25431.9 ± 24876.09
Create reservation	333.23 ± 62.07	341.67 ± 68.88	334.47 ± 63.88	345.95 ± 65.3	396.02 ± 58.84	1121.89 ± 180.05
Update reservation	197.75 ± 18.94	200.08 ± 19.54	195.24 ± 18.92	188.98 ± 17.04	192.38 ± 19.78	193.81 ± 20.23
Delete reservation	191.95 ± 20.96	190.66 ± 19.29	192.78 ± 24.6	179.63 ± 15.72	180.49 ± 16.69	187.09 ± 20.94

Table 6.4: Average latency in milliseconds for server-side operations per number of rows in client state. (N $\geq$ 105)

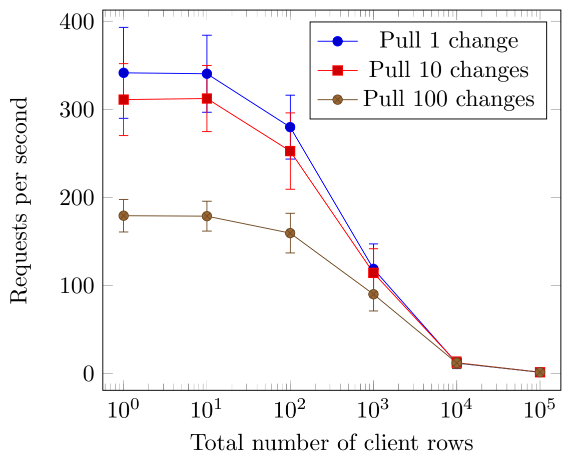
[4] `https://k6.io/`



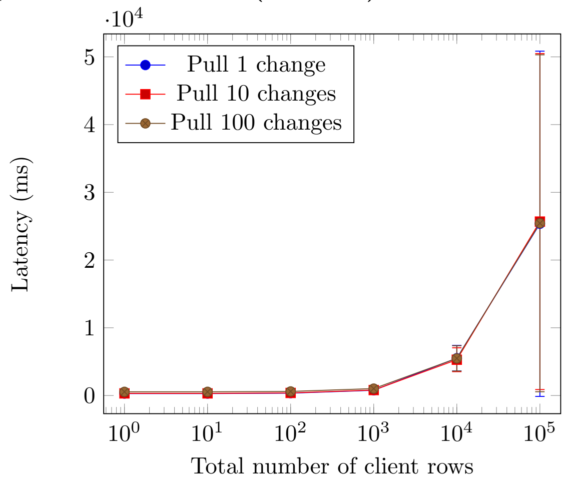
(a) Throughput in requests per second for the first pull of a new client. (N $\geq$ 61)



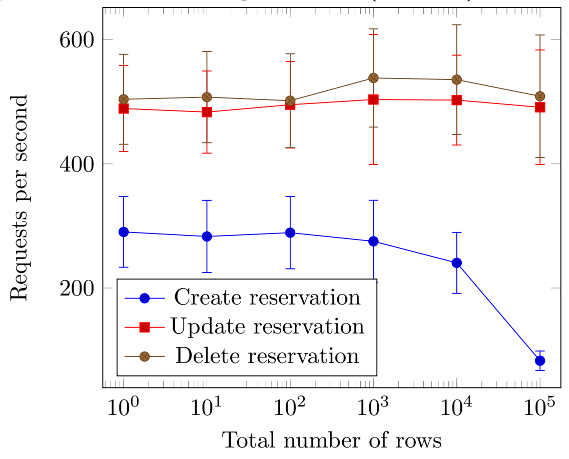
(b) Average latency in milliseconds for the first pull of a new client. (N $\geq$ 105)



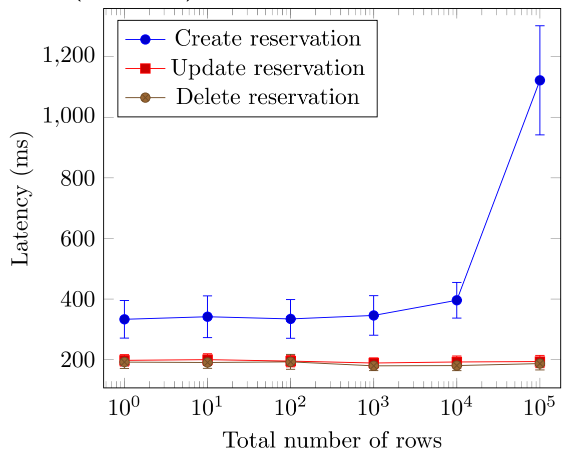
(c) Throughput for subsequent pulls of clients per amount of changed rows. (N $\geq$ 62)



(d) Average latency in milliseconds for subsequent pulls of clients per amount of changed rows. (N $\geq$ 229)



(e) Throughput for pushing mutations to the server. (N $\geq$ 62)



(f) Average latency in milliseconds for pushing mutations to the server. (N $\geq$ 5387)

Figure 6.7: Throughput of server for handling pull and push requests by total number of rows in client state.

In figure 6.7a and 6.7b, the performance of a new user loading the data for the first time is shown. For databases with one row, the benchmark server can serve around 300 requests per second. When the client has access to more rows, the throughput declines quickly, with the server being able to serve 35 requests per second. For 100000 rows, the server is only able to serve 1.6 requests per second.

In 6.7c and 6.7d, the performance is shown for users that already have data locally. The amount of changes since the initial pull is shown in the legend. We see that the performance does not improve much as opposed to the performance of the first pull, even degrading when pulling 100 changes. This operation also suffers from large performance degradation for larger datasets, handling 1.4 requests per second when clients have access to 100000 rows.

Finally, we see the performance of the mutations in 6.7e and 6.7f. This performance is relatively constant over the amount of rows clients may access, except for the create reservation mutation. This mutation scans for overlapping reservations, which costs more performance if it must scan more rows in the database.

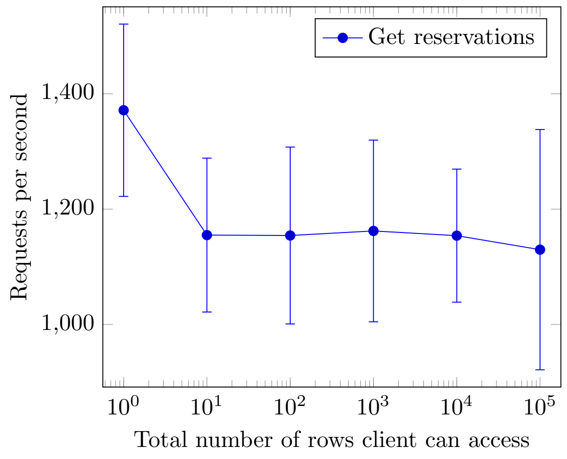
In table 6.3, the throughput of the REST server is shown. The latency is shown in table 6.6. Finally, the results are visualized in figure 6.8.

Rows	**1**	**10**	**100**	**1000**	**10000**	**100000**
Get reservations	1371.52 ± 149.38	1155 ± 133.42	1154.3 ± 153.38	1162.2 ± 157.51	1154.03 ± 115.36	1129.76 ± 208.34
Create reservation	352.11 ± 82	357.69 ± 105.94	350.03 ± 86.56	343.48 ± 77.93	290.57 ± 47.69	96.4 ± 11.12
Update reservation	1292.48 ± 133.72	1279.7 ± 142.74	1285.23 ± 135.55	1274.75 ± 165.59	1270.39 ± 122.79	1263.15 ± 135.24
Delete reservation	1976.39 ± 219.82	1934.67 ± 225.42	1987.25 ± 230.65	1995.34 ± 211.19	1950.54 ± 190.99	1915.64 ± 216.79

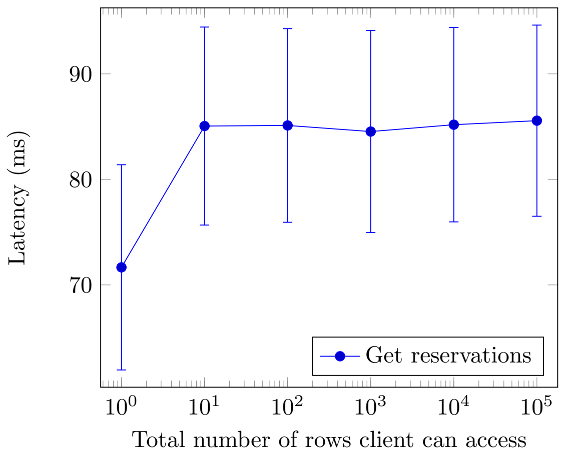
Table 6.5: Througput in requests per second for server-side operations per number of rows client may access for the REST server. (N $\geq$ 61)

Rows	**1**	**10**	**100**	**1000**	**10000**	**100000**
Get reservations	71.66 ± 9.73	85.06 ± 9.38	85.11 ± 9.18	84.53 ± 9.57	85.18 ± 9.21	85.57 ± 9.06
Create reservation	279.18 ± 64.58	270.39 ± 67.23	280.94 ± 68.28	286.19 ± 63.69	338.57 ± 52.03	1010.48 ± 75.03
Update reservation	76.02 ± 16.07	76.78 ± 14.24	76.44 ± 11.52	77.09 ± 8.5	77.33 ± 7.97	77.77 ± 15.08
Delete reservation	49.7 ± 25.75	50.77 ± 13.31	49.43 ± 31.9	49.22 ± 30.34	50.36 ± 14.33	51.28 ± 15.8

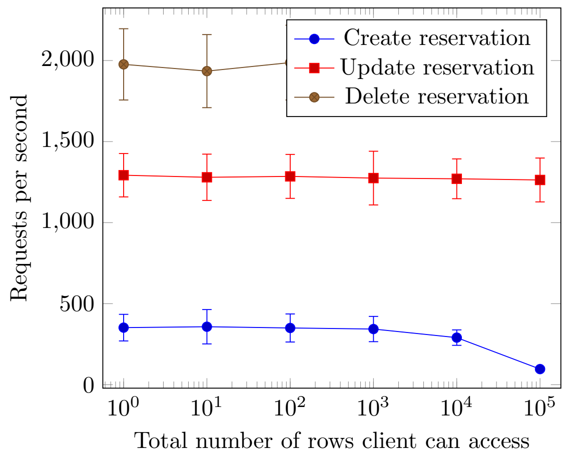
Table 6.6: Average latency in milliseconds for server-side operations per number of rows client may access for the REST server. (N $\geq$ 5977)



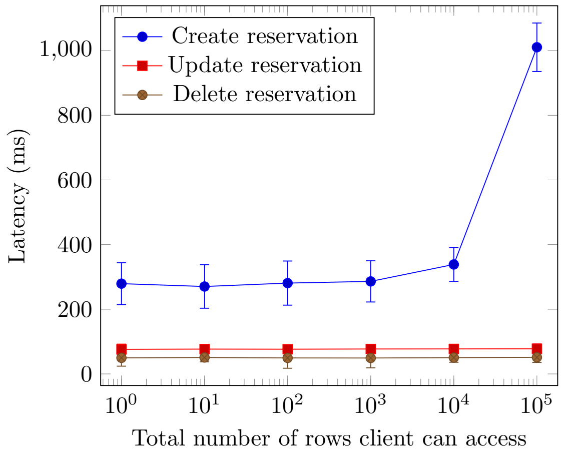
(a) Throughput in requests per second for getting reservations. (N $\geq$ 61)



(b) Average latency in milliseconds for getting reservations. (N $\geq$ 70045)



(c) Throughput for creating, updating or deleting reservations. (N $\geq$ 61)



(d) Average latency in milliseconds for creating, updating or deleting reservations. (N $\geq$ 5977)

Figure 6.8: Throughput for serving get, create, update and delete requests by number of rows clients may access for the REST server.

In figure 6.8a and 6.8b, we see that the throughput and latency are almost constant over the different total number of accessible rows per client. Only for 1 row, we see that the latency is lower and the throughput is higher; this is because this endpoint returns the first 10 rows. In a database with 1 row, it only returns one, which results in a better performance. For more than 10 rows, the server can serve around 1150 requests per second.

For the write endpoints in 6.8c and 6.8d, we also see the same performance over different amounts of rows. Only the create reservation operation's performance declines with more rows as it scans for overlapping reservations.

6.3.3 Discussion

Comparing the create, update and delete reservation performance of the REST server to the Relic server, we see that the Relic server has a large amount of overhead. For example, the REST server allows for 130% more updates per second. The reason for this performance discrepancy is that Relic ensures that every mutation is executed idempotently. This is not the case for the REST server. To enable this idempotency, the Relic server creates a transaction, queries the RelicClient, checks whether this updateReservation mutation has already been processed, then updates the reservation, updates the RelicClient and commits the transaction. For the REST server, the server simply updates the reservation and returns, which results in much greater performance, but the idempotency is lost. In REST servers, this idempotency may not be necessary as users can immediately see the result of their actions. In an Offline-First application, however, users can store theoretically infinite pending mutations offline; when the user reconnects, they will send these mutations, and the server will execute them. If these mutations fire twice, this would result in a very bad user experience, and therefore, mutations must be idempotent in an Offline-First scenario.

We cannot easily compare the REST GET reservations endpoint to the pull performance of the Relic server. A REST GET endpoint would be called more in a traditional REST application than the pull operation, as every page transition requires a new get request. On the other hand, the pull operation may also be called often when multiple users are collaborating. We cannot directly compare the REST endpoint to the Relic pull operation, but the REST endpoint can serve as a baseline for how performant Relic requests could be with the common HTTP framework and JavaScript runtime. For applications where the user does not have access to much data, the pull endpoint serves between 300-35 requests per second for 1-1000 rows on the benchmarking machine. For 100000 rows, the server is only able to serve one and a half requests per second.

The main reason for this low performance is the type of pulling we have implemented for the example application. The row versioning adapter gives developers a lot of flexibility as it allows them to define exactly which rows are accessible by any client at any time. However, every row versioning pull performs one query that returns two columns for every row the client may access and one query that gets the full rows which have been updated. Furthermore, calculating which rows are updated since the previous pull is also a heavy computation, especially in a language such as JavaScript.

To unlock higher performance, especially in applications with big client states, developers should use another adapter for pulling, such as the version per tenant, table or global adapters. These options only need to query the database once, speeding up the first-time pull and secondary pulls. More importantly, pulling the changes since the previous pull is much cheaper by being able to efficiently query only the new data from the database instead of calculating this in JavaScript. On the other hand, these pulling mechanisms are also less trivial to use when users dynamically have access to some rows but not others. Furthermore, these pulling mechanisms also impose a global lock on the push throughput.

In future work, an adapter for managing pulls through the database's logical replication mechanism could be designed. This would have a large impact on some aspects of Relic, as it requires a stateful service to listen to the logical replication log and would, therefore, not work in a serverless environment. However, logical replication would unlock much higher performance.

Chapter 7

Discussion

In this chapter, we discuss the results of the experiments in chapter 6.

In the first experiment, we presented a case study in which we used the Relic framework to create an Offline-First application. We showed the implementation, demonstrated its usage and shown that the Relic framework satisfies the requirements as defined in chapter 4. In the second and third experiments, we have shown the performance characteristics of the server-side and client-side parts of the Relic framework.

7.1 Limitations

In the first experiment, we see some of the limitations of the current implementation of Relic. Firstly, setting up Relic, the client and the server, still takes a notable amount of work. While Relic allows developers to not start from scratch, the flexibility that Relic provides still requires developers to wire the flexible parts together, such as the HTTP server on the server-side and the setup of the SQLite database on the client-side. We could abstract away these parts but that would significantly reduce flexiblity. Therefore, we propose the solution that many other TypeScript frameworks, such as Next.js[1], T3 app[2], and Remix[3] also provide: a create-relic-app CLI tool. This CLI tool would allow developers to select from some popular databases such as Postgres, MySQL and SQLite, a server-side HTTP server such as Hono or Fastify, and select a client-side UI library such as React, Vue or Solid. The tool would then setup a new Relic application with the technologies of the developer's choice. Developers could then still easily swap out technologies for different ones.

Another limitation is the need for transactions in the server-side database. This restricts the framework for use-cases where databases without transactions, such as document databases. would be used. Transactionality is required to ensure consistency for all clients. Furthermore, transactions are required to make sure that all mutators are executed exactly once. Regarding client-side consistency, without transactions, clients could receive data that would eliminate referential integrity. This would not be acceptable in a relational client-side database. Therefore, when developers want to use databases without transactions in the backend, such as a document database, they should probably also use a document database on the frontend.

An issue that is not found in normal REST APIs and other Offline-First frameworks that use state transfer instead of operation transfer is that mutations in Relic are typically implemented twice. Once on the server, and once on the client. This is required to use client-side prediction, which is an essential component of the Offline-First replication algorithm. SQLSync, which uses the same replication algorithm, solves this by running SQLite on the server and the client. This allows the backend and the frontend to both have the same database, therefore mutations can be re-used on the client and server. In Relic, this is not the case, as the backend database can be very different from the frontend database. However, it would be possible to create an abstraction that allows developers to implement mutations using a common subset of SQLite and other relational databases such as Postgres, MySQL and SQL Server. This would allow developers to implement the mutation once using the common subset of functionality and share that mutation with the client and server. On the other hand, implementing a simple create,

[1]`https://nextjs.org/docs/pages/api-reference/create-next-app`
[2]`https://create.t3.gg/`
[3]`https://remix.run/docs/en/main/other-api/create-remix`

update and delete mutation for each table that can be updated on the client and on the server may not be a big issue.

Finally, a limitation that developers may find with the Relic framework is that it requires using TypeScript on the backend. The productivity and Developer Experience of full-stack TypeScript frameworks is a big advantage, as can be seen by the adoption of other full-stack frameworks such as Next.js. However, not every developer team will want to use TypeScript server-side. Technically, it is perfectly fine to use Relic only on the client-side and implement the Relic protocol in a server implemented in any other language. However, this server would have to be created from scratch, and developers would lose end-to-end type safety. To allow Relic to be easily used with other backend languages and preserve type safety, we propose using code generation and defining the RelicDefinition in a language-independent specification, for example, using TypeSpec[4]. For each backend language, a Relic library should be created which allows developers to create a RelicServer as easily as they can in TypeScript. With this solution, developers can create Relic applications in any supported backend language by using code generation.

[4]`https://typespec.io/`

Chapter 8

Conclusion

In this thesis, we have presented the Relic Offline-First web framework. Offline-first applications bring important benefits to users. They are offline capable, very responsive, and support real-time collaboration. However, creating Offline-First applications is complex: Developers need to handle device storage and data replication. To address these issues, we have designed and implemented a framework that handles this complexity so developers can focus on creating applications.

Our first research question: "How to structure, store and query large amounts of data in the frontend that supports many different types of applications?" was addressed by embedding an SQLite database in the frontend combined with an ORM. This combination allows developers to persistently store data in the browser using SQLite, query data effectively by using a relational data model and SQL, and use an ORM to provide type safety and streamline relational queries.

The second research question: "How to synchronize updates between clients while ensuring eventual consistency of state and minimizing architectural and coding constraints on the application?" was handled using a combination of client-side prediction and server reconciliation. Using these techniques, developers can use application-specific conflict resolution to ensure optimal preservation of user intent and custom arbitrary invariants.

The third and final research question: "How to design an Offline-First framework API to facilitate developer productivity, ease-of-use, modularity, and safety?" was addressed by creating a framework that handles both the client and the server. Developers first specify the data model and the operations on this data model and implement this contract on the client and server side. This ensures type safety on the client and the server, and end-to-end type safety between the client and server. By providing type safety, Developer Experience is increased through IDE integration and by catching errors immediately. The API of the framework is designed for type inference, enabling developers to write less code without sacrificing type safety. Finally, modularity is provided by allowing developers to implement adapters for their own storage implementation.

With the answers to the sub-questions, we provide an answer to the main research question: "How to design a web framework for Offline-First applications?" in the form of the design and implementation of the Relic framework. We have conducted three experiments, one validating the Developer Experience of using the framework, and the end-user experience of using the Offline-First applications created using the framework. The last two experiments validate and showcase the framework's client-side and server-side performance.

The framework solves the gaps identified in existing Offline-First frameworks by providing an effective relational data model and allowing developers to take advantage of application-specific conflict resolution. Furthermore, the framework does not constrain developers to use a single backend database and gives developers full control over the backend, without requiring developers to start from scratch. Relic allows developers to effectively create Offline-First applications, bringing Offline-First benefits to more users.

Chapter 9

Future Work

In this chapter, we discuss several options for future work. There are two main features to be implemented for Relic to be production-ready.

- Currently, Relic does not support cross-tab usage. The SQLite-WASM OPFS SAH-pool database used in the Relic framework for the browser does not support concurrent access from two tabs, therefore initializing Relic in a separate tab will error. We propose to solve this using leader election, in which one tab contains the main Relic instance, and other tabs communicate with this tab to mutate data and subscribe to queries.
- One challenge of Offline-First is that we also need to migrate clients when the data model of the application is updated. To solve this we could simply clear the frontend database and just update the data model, but migrations can also be implemented in a way where the existing client-side database is updated using migration scripts defined by the developer.

We have discussed several of the current limitations of Relic in depth in chapter 7. These also provide opportunities for future work.

- A create-relic-app CLI tool can be created to greatly simplify and speed up the setup of new Relic applications.
- Adapters for using SQLite in a desktop JavaScript environment and React Native can be created for developers to create Relic clients outside the browser.
- To solve the issue of each mutator having to be implemented twice, an abstraction could be created that allows developers to implement a mutator once and can be used for both the client and server side. Currently, developers can do this themselves by implementing mutations in the RelicDefinition, but an abstraction provided by the framework could greatly simplify this process.
- Currently, Relic can only be used with a TypeScript backend. To remove this limitation, we could define the RelicDefinition in a language-independent specification such as TypeSpec[1], and use code generation to compile this specification to a RelicDefinition in any other language.
- Higher performance could be unlocked by creating adapters for using database Logical Replication mechanisms for Real-Time updates.

[1]`https://typespec.io/`

Bibliography

[1] M. Kleppmann, A. Wiggins, P. Van Hardenberg, and M. McGranaghan, “Local-first software: You own your data, in spite of the cloud,” en, in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Athens Greece: ACM, Oct. 2019, pp. 154–178, ISBN: 978-1-4503-6995-4. DOI: `10.1145/3359591.3359737`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3359591.3359737` (visited on 01/09/2024).

[2] O. First. “Offline first.” (), [Online]. Available: `https://offlinefirst.org/` (visited on 04/22/2024).

[3] RxDB. “Local first / offline first.” (), [Online]. Available: `https://rxdb.info/offline-first.html` (visited on 04/21/2024).

[4] M. Gamble. “Offline-first: A mindset for developing faster, more reliable mobile apps.” (), [Online]. Available: `https://www.couchbase.com/blog/offline-first-more-reliable-mobile-apps/` (visited on 04/22/2024).

[5] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, ISSN: 0163-5700. DOI: `10.1145/564585.564601`. [Online]. Available: `https://doi.org/10.1145/564585.564601` (visited on 01/11/2024).

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-Free Replicated Data Types,” en, in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 386–400, ISBN: 978-3-642-24550-3. DOI: `10.1007/978-3-642-24550-3_29`.

[7] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of convergent and commutative replicated data types,” Ph.D. dissertation, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[8] C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems,” in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’89, Portland, Oregon, USA: Association for Computing Machinery, 1989, pp. 399–407, ISBN: 0897913175. DOI: `10.1145/67544.66963`. [Online]. Available: `https://doi.org/10.1145/67544.66963`.

[9] “Local-first web development.” (2023), [Online]. Available: `https://localfirstweb.dev/` (visited on 04/04/2024).

[10] “Postgresql: Write-ahead logging (wal).” (), [Online]. Available: `https://www.postgresql.org/docs/current/wal-intro.html` (visited on 04/05/2024).

[11] G. Litt, N. Schiefer, J. Schickling, and D. Jackson, “Riffle: Reactive Relational State for Local-First Applications,” en, in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, San Francisco CA USA: ACM, Oct. 2023, pp. 1–16, ISBN: 9798400701320. DOI: `10.1145/3586183.3606801`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3586183.3606801` (visited on 01/09/2024).

[12] S. Parunashvili. “Database in the browser, a spec.” (), [Online]. Available: `https://stopa.io/post/279` (visited on 04/05/2024).

[13] C. Sverre. “Stop building databases.” (), [Online]. Available: `https://sqlsync.dev/posts/stop-building-databases/` (visited on 04/05/2024).

[14] S. Parunashvili. “A graph-based firebase.” (), [Online]. Available: `https://stopa.io/post/296` (visited on 04/05/2024).

[15] N. Prokopov. "The web after tomorrow." (), [Online]. Available: `https://tonsky.me/blog/the-web-after-tomorrow/` (visited on 04/08/2024).

[16] J. Rusher. "Rhetorical device: Triple store." (), [Online]. Available: `https://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html` (visited on 04/08/2024).

[17] T. Artman. "Linear realtime sync engine." (), [Online]. Available: `https://www.youtube.com/watch?v=WxK11RsLqp4&t=2175s` (visited on 04/11/2024).

[18] J. Day-Richter. "What's different about the new google docs: Making collaboration fast." (), [Online]. Available: `https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html` (visited on 04/09/2024).

[19] A. M. David Wang. "Google wave operational transform." (), [Online]. Available: `https://web.archive.org/web/20090531063923/http://www.waveprotocol.org/whitepapers/operational-transform` (visited on 04/09/2024).

[20] M. Kleppmann. "Crdts and the quest for distributed consistency." (), [Online]. Available: `https://www.youtube.com/watch?v=B5NULPSiOGw` (visited on 04/09/2024).

[21] A. Imine, M. Rusinowitch, G. Oster, and P. Molli, "Formal design and verification of operational transformation algorithms for copies convergence," *Theoretical Computer Science*, vol. 351, no. 2, pp. 167–183, 2006.

[22] D. Li and R. Li, "An admissibility-based operational transformation framework for collaborative editing systems," *Computer Supported Cooperative Work (CSCW)*, vol. 19, pp. 1–43, 2010.

[23] E. Wallace. "How figma's multiplayer technology works." (), [Online]. Available: `https://www.figma.com/blog/how-figmas-multiplayer-technology-works/` (visited on 04/09/2024).

[24] V. B. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency in distributed systems," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[25] K. Jahns. "How we made jupyter notebooks collaborative with yjs." (), [Online]. Available: `https://blog.jupyter.org/how-we-made-jupyter-notebooks-collaborative-with-yjs-b8dff6a9d8af` (visited on 04/09/2024).

[26] Redis. "Active-active geo-distribution (crdts-based)." (), [Online]. Available: `https://redis.io/active-active/` (visited on 04/09/2024).

[27] Riak. "Concept data types." (), [Online]. Available: `https://docs.riak.com/riak/kv/2.2.3/learn/concepts/crdts/index.html` (visited on 04/09/2024).

[28] R. Nehme. "Azure cosmosdb build 2018: The catalyst for next generation apps." (), [Online]. Available: `https://azure.microsoft.com/en-us/blog/azure-cosmosdb-build-2018-the-catalyst-for-next-generation-apps/?_lrsc=996193e3-e253-4a89-a50a-0b105862c2e6` (visited on 04/09/2024).

[29] M. Ptaszek. "Chat service architecture: Persistance." (), [Online]. Available: `https://technology.riotgames.com/news/chat-service-architecture-persistence` (visited on 04/09/2024).

[30] M. S. Martin Kleppmann Annette Bieniusa. "General-purpose crdt libraries." (), [Online]. Available: `https://crdt.tech/implementations` (visited on 04/09/2024).

[31] Z. Knill. "You don't need crdts for collaborative experiences." (), [Online]. Available: `https://zknill.io/posts/collaboration-no-crdts/` (visited on 04/08/2024).

[32] M. Kleppmann, V. B. Gomes, D. P. Mulligan, and A. R. Beresford, "Interleaving anomalies in collaborative text editors," in *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, 2019, pp. 1–7.

[33] P. Butler. "You might not need a crdt." (), [Online]. Available: `https://driftingin.space/posts/you-might-not-need-a-crdt` (visited on 04/08/2024).

[34] G. Litt, S. Lim, M. Kleppmann, and P. Van Hardenberg, "Peritext: A crdt for collaborative rich text editing," *Proceedings of the ACM on Human-Computer Interaction*, vol. 6, no. CSCW2, pp. 1–36, 2022.

[35] R. Vaillant, D. Vasilas, M. Shapiro, and T. L. Nguyen, "Crdts for truly concurrent file systems," in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, 2021, pp. 35–41.

[36] Rocicorp. "Ready player two - bringing game-style state synchronization to the web." (), [Online]. Available: `https://rocicorp.dev/blog/ready-player-two` (visited on 04/10/2024).

[37] V. Balegas. "Introducing rich-crdts." (), [Online]. Available: `https://electric-sql.com/blog/2022/05/03/introducing-rich-crdts` (visited on 04/10/2024).

[38] V. Balegas *et al.*, "Extending eventually consistent cloud databases for enforcing numeric invariants," in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, IEEE, 2015, pp. 31–36.

[39] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich, "Global sequence protocol: A robust abstraction for replicated shared state," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[40] VLCN. "A gentle introduction to crdts." (), [Online]. Available: `https://vlcn.io/blog/intro-to-crdts` (visited on 04/11/2024).

[41] W. Yu and C.-L. Ignat, "Conflict-free replicated relations for multi-synchronous database management at edge," in *2020 IEEE International Conference on Smart Data Services (SMDS)*, IEEE, 2020, pp. 113–121.

[42] J. Lazaroff. "An interactive intro to crdts." (), [Online]. Available: `https://jakelazaroff.com/words/an-interactive-intro-to-crdts/` (visited on 04/10/2024).

[43] N. W. Group. "Session traversal utilities for nat (stun)." (), [Online]. Available: `https://www.rfc-editor.org/rfc/rfc5389.html` (visited on 04/10/2024).

[44] M. developer network. "Signaling and video callign." (), [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling` (visited on 04/10/2024).

[45] J. Teton-Landis. "Notion last write wins." (), [Online]. Available: `https://news.ycombinator.com/item?id=38289327` (visited on 04/11/2024).

[46] H. Fan and C. Sun, "Supporting semantic conflict prevention in real-time collaborative programming environments," *ACM SIGAPP Applied Computing Review*, vol. 12, no. 2, pp. 39–52, 2012.

[47] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[48] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[49] N. Schiefer, G. Litt, and D. Jackson, "Merge what you can, fork what you can't: Managing data integrity in local-first software," in *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, 2022, pp. 24–32.

[50] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, ser. SOSP '95, New York, NY, USA: Association for Computing Machinery, Dec. 1995, pp. 172–182, ISBN: 978-0-89791-715-5. DOI: `10.1145/224056.224070`. [Online]. Available: `https://dl.acm.org/doi/10.1145/224056.224070` (visited on 01/09/2024).

[51] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The locus distributed operating system," *ACM SIGOPS Operating Systems Review*, vol. 17, no. 5, pp. 49–70, 1983.

[52] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek, "Resolving file conflicts in the ficus file system.," in *USENIX Summer*, 1994, pp. 183–195.

[53] P. Kumar and M. Satyanarayanan, "Log-based directory resolution in the coda file system," in *[1993] Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, IEEE, 1993, pp. 202–213.

[54] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[55] CouchDB. "Conflict management." (), [Online]. Available: `https://guide.couchdb.org/draft/conflicts.html` (visited on 04/15/2024).

[56] A. Demers *et al.*, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 1–12.

[57] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed ingres," *IEEE Transactions on software Engineering*, no. 3, pp. 188–194, 1979.

[58] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017.

[59] Postgres. "Postgresql: Documentation: 16: 20.6. replication." (), [Online]. Available: `https://www.postgresql.org/docs/current/runtime-config-replication.html` (visited on 04/15/2024).

[60] MySQL. "Mysql 8.0 reference manual :: Chapter 19 replication." (), [Online]. Available: `https://dev.mysql.com/doc/refman/8.0/en/replication.html` (visited on 04/15/2024).

[61] Microsoft. "Sql server replication." (), [Online]. Available: `https://learn.microsoft.com/en-us/sql/relational-databases/replication/sql-server-replication?view=sql-server-ver16` (visited on 04/15/2024).

[62] G. Gambetta. "Fast-paced multiplayer (part ii): Client-side prediction and server reconciliation." (), [Online]. Available: `https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html` (visited on 04/16/2024).

[63] Enz, *The Quake3 Networking Model.* [Online]. Available: `https://fabiensanglard.net/quake3/./The%20Quake3%20Networking%20Mode_files/The%20Quake3%20Networking%20Mode.html` (visited on 01/17/2024).

[64] Y. W. Bernier, "Latency compensating methods in client/server in-game protocol design and optimization," in *Game Developers Conference*, vol. 98033, 2001.

[65] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005.

[66] P. J. Keleher, "Decentralized replicated-object protocols," in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, 1999, pp. 143–151.

[67] J. Wilander. "Full third-party cookie blocking and more." (), [Online]. Available: `https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/` (visited on 04/18/2024).

[68] J. Long. "A future for sql on the web." (), [Online]. Available: `https://jlongster.com/future-sql-web` (visited on 04/18/2024).

[69] B. Waters, "Software as a service: A look at the customer benefits," *Journal of digital asset management*, vol. 1, pp. 32–39, 2005.

[70] F. Fagerholm and J. Münch, "Developer experience: Concept and definition," in *2012 international conference on software and system process (ICSSP)*, IEEE, 2012, pp. 73–77.

[71] T. Mikkonen, "Flow, intrinsic motivation, and developer experience in software engineering," *Agile processes in software engineering and extreme programming*, vol. 104, 2016.

[72] M. Greiler, M.-A. Storey, and A. Noda, "An actionable framework for understanding and improving developer experience," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1411–1425, 2022.

[73] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" In *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 153–162.

[74] A. Noda, M.-A. Storey, N. Forsgren, and M. Greiler, "Devex: What actually drives productivity: The developer-centric approach to measuring and improving productivity," *Queue*, vol. 21, no. 2, pp. 35–53, 2023.

[75] T. Delaet. "Why your it organization should prioritize developer experience." (), [Online]. Available: `https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/tech-forward/why-your-it-organization-should-prioritize-developer-experience` (visited on 04/19/2024).

[76] R. Cartwright and M. Fagan, "Soft typing," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991, pp. 278–292.

[77] B. C. Pierce, *Types and programming languages.* MIT press, 2002.

[78] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, "An empirical study of the influence of static type systems on the usability of undocumented software," *ACM Sigplan Notices*, vol. 47, no. 10, pp. 683–702, 2012.

[79] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? an empirical study," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, IEEE, 2012, pp. 153–162.

[80] S. Overflow. "Stack overflow developer survey 2023." (), [Online]. Available: `https://survey.stackoverflow.co/2023/#technology-admired-and-desired` (visited on 04/19/2024).

[81] C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 859–869.

[82] L. Fischer and S. Hanenberg, "An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio," *ACM SIGPLAN Notices*, vol. 51, no. 2, pp. 154–167, 2015.

[83] A. Mardan, A. Mardan, and Corrigan, *Full Stack JavaScript*. Springer, 2018.

[84] J. Stylos and B. Myers, "Mapping the space of api design decisions," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, IEEE, 2007, pp. 50–60.

[85] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, IEEE, 2013, pp. 5–14.

[86] J. Bloch, "How to design a good api and why it matters," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 506–507.

[87] K. Cwalina, J. Barton, and B. Abrams, *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Addison-Wesley Professional, 2020.

[88] S. Cassiere. "Decisions on developer experience." (), [Online]. Available: `https://tanstack.com/router/latest/docs/framework/react/decisions-on-dx` (visited on 04/22/2024).

[89] tRPC. "Trpc - move fast and break nothing." (), [Online]. Available: `https://trpc.io/` (visited on 04/22/2024).

[90] CloudFlare. "How website performance affects conversion rates." (), [Online]. Available: `https://www.cloudflare.com/learning/performance/more/website-performance-conversion-rates/` (visited on 04/23/2024).

[91] Replicache. "How replicache works." (), [Online]. Available: `https://doc.replicache.dev/concepts/how-it-works` (visited on 04/26/2024).

[92] I. Hickson. "Web sql database w3c working draft 22 december 2009." (), [Online]. Available: `https://www.w3.org/TR/2009/WD-webdatabase-20091222/` (visited on 04/30/2024).

[93] SQL.js. "Sqlite compiled to javascript." (), [Online]. Available: `https://sql.js.org/#/` (visited on 04/30/2024).

[94] J. Bell. "Indexed database api 3.0." (), [Online]. Available: `https://w3c.github.io/IndexedDB/` (visited on 04/30/2024).

[95] MDN. "Origin private file system." (), [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/API/File_System_API/Origin_private_file_system` (visited on 04/30/2024).

[96] SQLite. "Sqlite3 webassembly & javascript documentation index." (), [Online]. Available: `https://sqlite.org/wasm/doc/trunk/index.md` (visited on 04/30/2024).

[97] R. Hashimoto. "Wa-sqlite benchmarks." (), [Online]. Available: `https://rhashimoto.github.io/wa-sqlite/demo/benchmarks.html` (visited on 04/30/2024).

[98] RxDB. "Origin private file system (opfs) database with the rxdb opfs-rxstorage." (), [Online]. Available: `https://rxdb.info/rx-storage-opfs.html` (visited on 04/30/2024).

[99] SkipLabs. "Skdb - the fastest way to build real-time features." (), [Online]. Available: `https://www.skdb.io/` (visited on 04/30/2024).

[100] React. "Useeffect - react." (), [Online]. Available: `https://react.dev/reference/react/useEffect` (visited on 04/30/2024).

[101] SQLite. “The session extension.” (), [Online]. Available: https://sqlite.org/sessionintro.html (visited on 05/01/2024).

[102] SQLite. “Automatic undo/redo using sqlite.” (), [Online]. Available: https://www.sqlite.org/undoredo.html (visited on 05/01/2024).

[103] Replicache. “Why not use last-modified.” (), [Online]. Available: https://doc.replicache.dev/strategies/global-version#why-not-use-last-modified (visited on 05/13/2024).

[104] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” en, vol. 21, no. 7, 1978.

[105] Replicache. “Per-space version strategy.” (), [Online]. Available: https://doc.replicache.dev/strategies/per-space-version (visited on 05/14/2024).

[106] Replicache. “The row version strategy.” (), [Online]. Available: https://doc.replicache.dev/strategies/row-version (visited on 05/14/2024).

[107] Mozilla. “Using server-sent events.” (), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events (visited on 05/16/2024).

[108] Mozilla. “The websocket api.” (), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (visited on 05/16/2024).

[109] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design patterns: Abstraction and reuse of object-oriented design,” in *ECOOP’93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7*, Springer, 1993, pp. 406–431.

[110] W. Wigham. “Proposal: Partial type argument inference.” (), [Online]. Available: https://github.com/microsoft/TypeScript/issues/26242 (visited on 05/23/2024).

[111] Tanstack. “Tanstack query.” (), [Online]. Available: https://github.com/microsoft/TypeScript/issues/26242 (visited on 05/23/2024).

[112] Zod. “Zod documentation.” (), [Online]. Available: https://zod.dev/ (visited on 05/24/2024).

[113] M. D. Network. “Fetch api — web apis.” (), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (visited on 05/24/2024).

[114] web-interoperable runtimes community group. “Wintercg.” (), [Online]. Available: https://wintercg.org/ (visited on 05/24/2025).

[115] R. B. Miller, “Response time in man-computer conversational transactions,” in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, 1968, pp. 267–277.

Appendix A

Non-crucial information

The source code of the Relic framework and the experiments can be found at `https://github.com/vigovlugt/relic`. Source code for generation of diagrams and charts can be found at `https://github.com/vigovlugt/relic-diagrams`