UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Continious Synchronization of Conflict-Free Replicated Relations

Gustav Heide Iversen

INF-3990: Master's Thesis in Computer Science - May 2022

Abstract

Local-first software is an attempt to use the benefits of cloud service while reducing its drawbacks[1]. Local-first software gives the clients ownership and control of their data and makes the service always available. It is achieved by having the primary copy of the service at the client. The most common way to implement local-first software is by utilizing Conflict-free Replicated Datatypes[2] or CRDTs, which are conflict-free by design. Conflict-free Replicated Relations or CRR are CRDT applied to SQL databases[3]. CRR systems require some form of communication middleware to propagate its state so that each site can converge to a common state. Earlier CRR systems have used SSH File Transfer Protocol to propagate states or SFTP, which means writing to disk many times.

This thesis focuses mainly on the communication portion of a CRR system applied to an SQLite database called SynQLite[4]. SynQLite is a service that can augment an SQLite database with CRR-support. It also includes the possibility to clone and synchronize with remote sites.

The previous version of SynQLite allowed only to pull states from a remote site. We propose a solution where users can synchronize continuously with more than two sites simultaneously. The solution involves creating a centralized leader that other sites can connect to with TCP connections, and the other sites synchronize with the leader.

The thesis includes an evaluation process including many experiments. These experiments are used to evaluate how well SynQLite supports local-first properties. This includes testing how SynQLite affects the offline database and how well SynQLite supports synchronizing sites.

Acknowledgements

My thesis supervisor Weihai Yu deserves acknowledgment and thanks. I am grateful for his assistance and guidance throughout this project. We worked well together, bouncing ideas back and forth.

I also want to thank him for introducing me and allowing me to work on an exciting topic.

Thanks to all the friends I made through my education. Thanks for all the shared lunch and coffee breaks.

I want to thank my friends and family back home for supporting me throughout my education. Making all of you proud was always a great motivator.

Lastly, I want to give my oldest brother Simon special thanks. He stepped up as a bigger brother and helped me settle in a new city. Without him, I would have never started my education in Tromsø.

Contents

List of Figures

1 Introduction

Cloud computing services have become a standard for modern applications. Because cloud computing allows for seamless collaborative work, sharing of computer resources, and sharing data. However, this comes with drawbacks like data ownership and service availability[1]. Because data is stored at the service owner's servers when using their application. Cloud applications are unavailable when the servers are down, or the user is not connected. Furthermore, it is unclear who owns the information when stored on the service owners' servers[1]. These are problems that did not occur when using old applications that were downloaded locally, giving clients complete control and ownership of their applications.

Clients expect services to have low latency and be available at all times. One way to reduce latency is to have the service located closer to the users to reduce the travel distance for the data. However, what if a service has clients at many different locations globally? A solution is to have the service replicated to multiple locations globally, giving the users a better option to connect to the service. Replication also helps with availability since a replica can take over if one server is down[5]. A problem with replication is that it introduces problems with consistency.

The CAP theorem says that a distributed system can only have two out of three properties[6]. The properties are Consistency, which is how consistent data are over multiple replicas; Availability, which indicates how accessible data are at any given time; and Partitioning, which describes how tolerant the system is to network partitions. Developers consider which properties to prioritize when developing a distributed system.

Local-first software is a set of principles that tries to utilize the benefits of cloud services and limit its drawbacks by having the primary copy of the application on the client's computer and sharing the results/data when the client wants to[1]. This allows the client to use the application locally and later publish the result if the client wants to share and utilize the benefits of cloud computing. The application is always available since it is downloaded locally and does not use remote servers. Still, this will loosen the consistency between replicas because they can make concurrent updates to their local copy. In other words, local-first software prioritizes availability and partition tolerance over consistency. Local-first software is made possible through Conflict-free Replicated Data Types or CRDTs, which are data types that are by definition conflict-free[2]. There are two types of CRDTs: state-based and operational-based.

Conflict-free Replicated Relations or CRR are CRDTs applied to relational databases[3], providing the relational database with local first properties. In earlier implementations, the approach has either used object-relational mappers[3] or SSH File Transfer Protocol(SFTP)to send data[4]. These have some drawbacks; ORM requires all users of the CRR to use the same ORM making it less general, and SFTP requires writing to disk, which can be pretty slow and cause stress to the hard drive. Previous CRR systems focused only on synchronization between two sites.

This thesis will extend a CRR implementation written in python called SynQLite. SynQLite is a system that can augment an already existing SQLite database to one with CRR support. SynQLite can clone the CRR SQLite database to other locations. Users can then conduct concurrent updates and later send pull requests to sync. SynQLite used SFTP to communicate between nodes. We propose two new ways to communicate between nodes. The first way is by using an interactive shell with SSH for discrete pull and push. The second way is a centralized topology for continuous synchronization using TCP connection between nodes and a leader. Then all communication is done through TCP calls instead of SFTP and SSH. The nodes can choose to connect to a leader or pull or push requests at their own will.

The primary motivation for using TCP is to give the users a more seamless experience when using the database while still upholding Local-first properties.

1.1 Requirements and specifications

The main requirement of this thesis is to find out if it is possible to create a system that continuously synchronizes CRDTs SQL databases and find out if it is feasible to do it with more than two sites. The main specification is that the system must uphold local-first properties. To achieve these goals, we will divide the goal into several minor problems, similar to iterative development methods[7].

When considering the CAP theorem, SynQLite lowers consistency constraints to have constant availability and tolerance to internet partitioning. SynQLite should uphold strong eventual consistency or SEC, not requiring coordination between replicas to resolve conflicts. The database must always be available for the service to be called local first, even when not connected to the internet. The user should not be required to install additional applications to use the database, but the user would have to have the correct libraries to synchronize with other users. SynQLite should support pushing and pulling to and from different sites. The users should have the ability to opt in and out of continuous synchronization whenever they want, or just discrete pull and push.

1.2 Outline

- **Technical background** describes technologies that the user should understand before reading the rest of the thesis.
- **Related work** explains other solutions that are related to CRDT synchronization.
- **Methodology** describes the methodology used for the research in this thesis.
- **Approach** justifies the technologies and design choices used for this thesis.
- **Design and implementation** describe the design and implementation of the current state of SynQLite.
- **Contribution** outline the features contributed from this thesis and what was introduced in the previous implementation.
- **Experiments** describes all the experiments performed on SynQLite.

- **Discussion** evaluates the SynQLite in terms of the requirement specified in Section 1.1
- **Conclusion** summarized the thesis.

2 Technical Background

2.1 Local first software

Cloud apps have made it easier to do collaborative work over the internet. Applications such as Trello[8] and Google Docs[9] make it possible to work with people on the same service almost as seamlessly as working on the same computer. Centralized cloud services come with advantages, but the trade-off is that data produced by the services is not under the user's control. In other words, they lose ownership. Another disadvantage is that the service is only accessible via the internet. The primary copy of the data is stored on the service owner's servers in regular cloud applications, whereas data stored on the user's device is only for caching (faster access). From the service's perspective, data stored only on the user's computer and not on the servers does not exist.

Local-first software is services that are primarily on the users' computer, and then when the user wants to share their results, they can publish them to the cloud. Local-first software tries to swap the roles, where the user stores the primary copy, and the server holds data for faster access for other devices[1]. Local-first software gives the user ownership of their data while utilizing the benefits of cloud computing. The user can also use the services without connecting to the cloud if the service does not require it.

Local-first software increases availability but loosens consistency because each replica might have diverging states. Local-first software is eventually consistent.

2.2 Eventual consistency

In modern applications, the state of the service is often replicated over several replicas to improve its availability and latency, making it more robust and closer to the end-user. The rise of cloud computing and mobile apps has made distributed services more relevant than ever. However, this forces the developer to develop apps that support distribution, the same way that the rise of parallel computing forced developers to utilize hardware more efficiently. Distributed systems are dependent on latency and scalability[10].

Several nodes with different hardware communicate over channels with different speeds within a service. This discrepancy forces developers to reduce communication between nodes preventing problems that occur from latency issues. Cloud services should also be scalable, utilizing parallelism and better hardware to stop the bottleneck with centralized services.

When considering scalability and replication is where consistency models are relevant. From an application programmer's perspective, a strongly consistent system is ideal. It makes the application easier to develop and prevents concurrency issues. However, having strong consistency in a distributed system introduces other problems:

1. The performance will suffer since a node must propagate an update to all replicas before committing the update to the database.

2. If there is a partition in the network, some nodes cannot communicate, thereby reducing the availability of the service.

3. Since clients cannot use the service unless all nodes are consistent, the service would be unavailable until someone fixed the partition.

Eventual constancy models guarantee that when a node performs an update in a distributed system, the node will eventually propagate the update to all replicas in the system. The service should be available even if all nodes are inconsistent, but it does mean nodes might have different states at certain times.

2.2.1 Strong eventual consistency

The drawback of eventual consistency models is that conflicts can occur since updates are executed asynchronously and might occur in different orders for each replica. Conflict occurs when multiple sites update the same value, which means there is no straightforward way to decide what the value should be. Normal solutions for conflict resolution are to use consensus algorithms like paxos[11] or perform rollback. Neither is ideal since conflict resolutions are complicated and may reduce service availability, and rollback will also reduce service availability. Optimistic solutions are error-prone and are difficult to develop. Strong eventual consistency models or SEC models solve these issues.

SEC is a consistency model that does not require any consensus or rollback to resolve conflicts because SEC is by design conflict-free[2]. SEC uses mathematical properties to make conflicts impossible. These properties include commutativity and monotonicity in a semilattice. A replicated counter that can only increment and decrement is an example of a data type that achieves SEC. Since addition and subtraction can be done in any order, the counter would be SEC following the commutative laws. Since a conflict cannot occur, no consensus or rollback would be required to resolve it.

2.3 Conflict-free replicated data types

Conflict-free replicated data types, or CRDTs, are abstract data types designed for replication with SEC. CRDTs can be updated individually and merged without any coordination between the replicas. One of the main properties of CRDT is that all replicas will reach the same state if they perform the same updates. CRDTs guarantee convergence for all states if they propagate all updates to replicas. Their synchronization model chooses how one CRDT replica merges with other replicas. There are two groups of synchronization models for CRDTs, operational- and state-based CRDTs.

2.3.1 Operation-based CRDT

In operational-based CRDTs, replicas converge by propagating local operations to all other replicas[12]. In other words, when a replica performs an update, it tells the other replicas to perform the same operation. Operational-based CRDTs require that all replicas receive all updates, and some require that the operations are performed in a specific order, often causal order. There are two main functions in all operational-based CRDTs, generator and effector. First, the generator checks the operation and the involved states and produces an effector function. Then this effector function is propagated and executed in all replicas. The states of the replicas will converge if the replicas deliver the messages in causal order or if the operations are commutative. However, it is common for operational-based CRDTs to require that operations are performed precisely once in a causal order. Reliable multicast communication subsystems are used in operational-based CRDTs to share updates to all replicas reliably.

2.3.2 State-based CRDT

With state-based CRDTs, replicas share their local state. A replica that receives a state performs a join operation on the received state with its local state. The design of CRDTs must include a merge function to integrate a remote replica's state with its own. There are some principles that state-based CRDTs must follow to ensure convergence[2]. All possible states from all replicas must be partially ordered; a join of any two states must always exist, forming a join-semilattice. Updates must be inflationary; the most recent state supersedes the previous state.

2.4 Example CRDTs

There exist many different types of state-based CRDTs. In this section, we will describe some of them. They will be grouped in two subsections: set-based or register.

2.4.1 Set CRDTs

A set CRDT is a state-based CRDT that upholds the set rules. A set has some properties, there are no copies, and one can only add or delete to/from a set. Users cannot change a value within a set.

Grow-only set

Grow-only set or G-set is one of the simplest forms of state-based CRDT. As the name implies, G-set is a set that is only growing[13]. Elements can only be added, not removed. The G-set will always be conflict-free since the element could either be there or never have been there when merging two sets. When adding elements to the set, we create a set with the elements we want to add and perform a union operation with the G-set. Merging two different G-sets is done by performing a union operation between both sets. The problem with G-sets is that we cannot remove elements from the set, which is often a necessary functionality for a set.

Two-phase Set

An improvement to G-sets is the two-phase set or 2p-set, which can represent deleted elements[13]. 2P-sets are composed of two G-sets: one for inserted elements and another for deleted elements. To check if an element "exists," you have to check if it exists in the "add-set" but not in the "remove-set." Adding elements is done by adding them to the "add-set." Removing elements is carried out by checking that the element exists in the "add-set" before adding it to the "remove-set." Merging two sets is done by performing union operations on both sets. The main limitation of 2P-sets is that elements cannot be added back after an element has been removed(added to the "remove-set").

Last write wins element set

Last write wins element set or LWW-element-set is a version of 2p-set, meaning it has two sets, one for added elements and another for deleted elements. It improves the 2p-set by assigning a timestamp to each element; each element in both sets are tuples. The timestamp is the time when the element was inserted. To see if an element "exists," you have to see if it is in the "add-set," then see if the timestamp is newer than any timestamp with the same element in the removed set. If a newer pair exists in the removed set, the element does not exist. Adding, removing, and merging are performed similarly to the 2p-set. Even though this structure addresses the issue of the 2p-set, it makes additional ones. The set depends on the accuracy of the clock of the computer. Another issue is that the elements are not removed from the sets; instead, newer elements are added, which means the sets can include many ghost elements.

Causal-length set

CL-set is a CRDT set that only contains one set with all the information needed to know the element's state, without the need for timestamps or another set. Each element in the CL-set includes an integer that describes its state called causal length[3]. To see if an element exists, first check if the value exists in the set and if the causal length is even or odd. If the causal length is odd, then the element exists. If it is even, then it does not exist. To add an element, first, check if the element exists. If it does and the causal length is even, the causal length is incremented by one. If it does not exist, the replica adds the element with a causal length of 1. Nothing happens if the element exists with an odd causal length because it already exists. When removing elements, the opposite happens. Checking if it exists, then checking if the causal length is even or odd, increment if the value is odd, doing nothing if the element is even. Merge is done by adding all elements that only exist in one of the replicas to one set, then adding the element with the most significant causal length for all other elements. In contrast to the 2P-sets, CL-sets aim to decide the element's state by changing the causal length, making CL-set one of the more flexible state-based CRDTs and addressing ghost elements. Another benefit of the CL-set is that it does not require unreliable computer clocks.

2.4.2 Delta-state CRDT

A drawback of the original state-based CRDTs is the communication overhead from sending the entire state between replicas when propagating, even if only one element has changed. On the one hand, operational CRDTs have smaller messages but have different drawbacks. They need a middle layer to ensure reliable causal broadcast, which requires a logging system to keep track of the operation performed at each replica. Another drawback is that operational CRDTs execute the operations one by one, which can be slow.

Delta-state CRDTs are an optimization done to state-based CRDTs, where delta-mutators return delta states[14]; updated states after last merge. Delta-mutator is used on the state to limit the size of the message when propagating. Since fewer rows will be included in the message, the size of the message will decrease. Delta-mutators decrease the messages' size, increasing the speed of sending the messages and merging. However, each replica must keep track of the last time each element was updated and the last time the replicas have propagated the element to other replicas. Keeping track of which replicas have what can get complicated when many replicas are connected. In the rest of the thesis, we will write delta as δ.

2.4.3 Register CRDTs

Register CRDTs is a memory cell that supports operations to update the value of an already existing element within that memory cell[13]. Register CRDTs need a way to determine which update is correct since CRDTs are supposed to be conflict-free, which is done by deciding that newer updates overwrite previous updates. If updates are not concurrent, then sequential order is preserved. However, concurrent updates will cause problems unless the replica performing the update takes precautions. The issue is how to determine one assignment to be older than another. There are two main approaches to solving this issue: last writer wins and multi-value register.

Last write wins register

Last-writer wins register or LWW-register associates all updates with a timestamp. This timestamp is used to determine which update came last. As mentioned before, the latest update is the correct one. The timestamp is assumed to be unique, totally ordered, and consistent with causal order[13]. Each replica adds unique information to the time stamp, ensuring that there are not two equal time stamps. The unique information could, for example, be the mac address of the replica.

2.5 Relational Database Model

The relational database model is the most used database model. It is a database model that uses tables to group entities into relations. The columns in the table are the attributes that describe the entity, while the rows are the entries. Each table should have a unique identifier called the candidate key to identify each element in the table. The candidate key could be the whole row or only one primary key. Foreign keys are attributes related to other tables, a candidate key for another table. For example, we have a person and a family table. The candidate key for the person table could be the first name plus the surname, while the family table could use the surname as its candidate key. In this example, the reference from a person to the family would be the surname. In other words, the surname is the foreign key to a family. The most common way to query a relational database is through Structured Query Language, SQL. It uses a structured script to query a relational database.

2.5.1 Sqlite

SQLite is a lightweight SQL database management system. The main difference between SQLite and other SQL databases is that an SQLite database is just one file. It does not require a separate server process to handle queries; instead, it reads and writes like a regular disk file. According to the SQLite homepage, [15] SQLite is the most used database engine globally. All mobile phones, almost all computers, and many applications use SQLite.

The main factors for using SQLite in this project are that it is lightweight, cross-platform, widely used, and has SQL triggers. Triggers are required for Synqlite to work. Triggers are stored procedures that invoke queries when a specified condition has been met. For example, we have a people table and a family table, and a new person is born. A trigger could check the family name of the newly added person and increment the number of family members. Triggers are helpful when we have multiple connected tables, as this project has.

Rollback journal

SQLite uses a rollback journal to uphold ACID properties. The journal is a separate file that ensures integrity if an error occurs during a database write operation[16]. When performing a write, the database copies the pages updated to the journal so that the database can rollback the affected pages to the old stable state. The main focus is that a transaction should either be fully executed or not at all, even if the database is interrupted in the middle of a transaction. Interrupts might occur from a program crash, operating system crash, or power failure.

SQLite has different journal modes; DELETE, TRUNCATE, PERSIST, MEMORY, WAL, and off. DELETE, TRUNCATE, and PERSIST are the normal journal modes and only differ in how they invalidate journals. Invalidating journals is when a transaction has been committed to the database, meaning the database state is stable. The journal is stored in RAM instead of a disk file when using MEMORY mode. MEMORY mode can, in some cases, increase the performance by a significant amount. The downside is that if the computer crashes, the journal is lost and can cause corruption in the database.

Write ahead-log or WAL mode is a newer version of journal rollback. With WAL, transactions are first written to a cache merged with the database when it exceeds a specific number of pages[17]. The WAL cache can show the state of the database at different times. One of the main benefits of WAL is that applications can read from the main SQLite database file simultaneously, as another process is updating the WAL file.

2.6 Conflict-free replicated relations

Conflict-free replicated relations, or CRR, apply CRDTs to relational databases to support multi-synchronous data access and allow for offline work[3]. CRR has two layers, one application relation layer(AR-layer) and one conflict-free replication relation layer(CRR-layer). AR-layer represents the users' typical database with their defined schemas and tables. Users only interact with the AR-layer as they would normally. A CRR-layer augments the AR-layer's relations to achieve CRDT properties and is abstracted from users. Operations trickle down from the AR-layer to the CRR-layer to reflect the correct state in both layers. How the operations trickle down is up to the implementation.

The CRR-layer combines CL set CRDT, LWW register, and delta-state CRDT. Each row in the CRR layer includes a causal length attribute to determine whether the row belongs in the AR layer, a timestamp to support attribute assignments, and a Universally Unique Identifier (UUID) for each row. The UUID is used to identify rows in the CRR-layer and is the same at all sites. Combining the LWW register and CL set makes it possible to perform updates within the rows required for relational databases. When merging subsets, the sender sends join-irreducible sets to improve performance, making it a delta-state CRDT.

Insertion and deletions are similar to CL-set explained in section 2.4.1. When a user performs an AR-layer insertion, an operation trickles down to the CRR-layer that cheks the causal length of the element and increments it if it is even. If the element does not exist, a new entry is added to the CRR-layer with a causal length set to one and a unique UUID representing the new element. When a user performs an AR-layer deletion, an operation checks the CRR-layer to analyze the causal length and increments it if it is odd.

Updates are done similar to LWW-register explained in 2.4.3. When the user performs an AR-layer update, an operation trickled down to the CRR-layer to update the timestamp of the attribute that was updated.

Connected replicas merge by performing an anti-entropy operation on all known replicas through their CRR-layer. When a replica wants to merge, it receives a delta set from another site, including CRR-layer rows. The receiving replica merges the rows by; comparing the causal length for each row and comparing the timestamp for each attribute within the row. The row with the most significant causal length decides the row's state, whether the row should exist or not. The timestamps decide what value each attribute should have.

2.6.1 CRR for Multi-Synchronous Database Management at Edge

[3] was the first to introduce CRR. It first introduces CL-set CRDT, which is particularly suited for relational databases. The paper also explains how to augment an existing relational database with CRR in a two-layer architecture. Lastly, they describe an implementation of CRR using an object-relation mapper or ORM called Ecto. The ORM handles the CRR-tables in this implementation, and sites communicate through this ORM layer. One drawback of this implementation is that all applications must use the same ORM.

2.6.2 Augmenting SQLite for Local-First Software

Later [4] introduces an implementation of CRR called SynQLite, which can augment an SQLite database with CRR support for local-first software. The database can be used as a typical SQLite database and does not require SynQLite to work. With one command, users can augment existing SQLite databases with CRR support. The augmentation is done by adding tables and triggers to the database. This database can be cloned to another location and later merged with other sites by executing simple command-line operations. The location of the sites is kept in a "site_table" so that they can perform requests on each other. By using the command line, sites can perform a pull request. The remote site then generates a delta-set and sends it back, after which the local site merges delta with its own. All connected sites communicate using SSH and SCP.

2.7 SSH Tunneling

SSH Port Forwarding or SSH Tunneling is a mechanism that uses SSH to tunnel TCP/IP ports from a server to a client or the other way[18]. Allowing one to connect to a remote port as if it is local, encrypting the transferred data so that a third party cannot read it. The main applications for SSH tunneling are to add encryption to legacy applications, create connections through a firewall, and create a backdoor to internal networks so that one can work from home. Malicious users can abuse SSH tunneling to access internal networks. That is why users should be careful when giving people access.

3 Related work

3.1 Efficient Synchronization of State-based CRDTs

[19] states that current synchronization methods for delta are redundant and perform worse than expected and, in some cases, even worse than normal state-based CRDT.

There were two observations done on delta synchronization: The first observation is that when a replica merges with another replica, it might send the state back because it looks at the state as a new update. They give an example where we have two replicas, A and B. A adds a to its set, and B adds b to its set. And then B propagates its δ-buffer {b} to A giving it {a,b} as its state. When A is propagating back {a,b} will be in the δ-buffer because it sees b as a new update. B already had b making this redundant. Another thing is if B performs an insertion c before merging with A, B will send {a,b,c} back to A at the following propagation. If this pattern continues, delta synchronization will not be much better than regular state-based. The solution to this problem is to track which replica made the update so it would not be sent back to the source at a propagation point. They call this "avoid back-propagation of δ-groups," BP optimization for short.

The second observation is that the δ-buffer might have the same element multiple times and propagate it multiple times to the same replica. This problem occurs if an element is propagated from two different sources but with other elements packed in. They call this "remove redundant state in received δ-groups," RR optimization for short.

The solution proposed from this thesis will only use BP optimization since it is a centralized solution with a leader or a one-to-one connection. They mention that acyclic topologies or topologies without cycle would only require BP optimization for the best result.

3.2 OWebSync

[20] proposes OWebSync, which is a middleware for synchronizing web-based applications. It allows the user to work offline and synchronize when they want to, or automatically synchronize when online. The paper proposes a new CvRDT data model used to replicate JSON structures quickly. The data model is a combination of state-based CRDT and Merkle trees. They use Merkle trees to detect data changes when merging with other clients quickly. The main difference between OWebSync and SynQLite is that SynQLite is for applications that use SQLite databases, while OWebSync uses a key-value store.

3.3 Data replication on the cloud/edge

[21] present a replication system for fast data access at the edge. The proposed solution uses a multi-master structure for replication, meaning any replica can submit a result immediately after performing a write. The replicas can propagate the result asynchronous at a later point.

The convergence algorithm achieves eventual consistency through Operational Transformation and CRDTs techniques. It uses the MongoDB data model, which is a JSON-like document store.

There are two types of replicas; cloud and edge replica. The cloud replica is the replica that receives messages and sends them to other replicas. In contrast, the edge replica handles requests directly from the clients. Each replica performs conflict resolution when a conflict occurs. The middleware looks at conflicting operations and chooses the newest one for each attribute. They based their conflict resolution on Last Writer Wins.

The main difference between their and our solution is that we use a relational model as our data model and CL-set CRDT to synchronize. Instead of operational transformation and MongoDB.

MongoDB is a NoSQL database designed for a heterogeneous distributed system, giving users the flexibility to add and remove attributes for each entry. However, we wanted to use SQLite since it is the most used database and lightweight and easy to clone.

4 Methodology

In contrast to regular science, computer science works on computational processes, not physical objects[22]. Often research in computer science is done by creating and testing models to prove a concept. After testing and verifying the concept, one can abstract it, allowing others to apply the concept to other cases.

[23] proposed three paradigms for computer science research. Computer science is a broad field with different goals, and these goals decide what kind of abstraction and structure the research should have. The paradigms that they proposed aid researchers in structuring their approach. Each paradigm includes multiple steps. However, the steps are meant to be used as a guideline, not an end-all-be-all template. The paradigms are:

Theory: is rooted in mathematics and consists of four steps, followed by developing a valid and coherent theory:

1. Define objects of study
2. Hypothesize relationships between the objects.
3. Find out if the relationships are true.
4. Analyse the results

Abstraction: is based on the experimental scientific method and involves four steps:

1. Construct a hypothesis
2. Develop a model and predict results based on the model.
3. Create experiments and collect results.
4. Analyse the results

Design: is based on the engineering process. It includes four steps to construct a system to achieve specific goals. The steps include:

1. State requirements
2. State specifications
3. Design and implement the system
4. Test the system

We decided to use design as our template for constructing the thesis. In Section 1.1 we define requirements and specifications for the system. In Chapter 6 we describe the design and implementation of SynQLite. In Chapter 8 we ran an experiment to test the system, and in Chapter 9 we discussed the results from the experiments.

While the design paradigm is a rough outline of how we structured the work, the method for developing the system was an iterative development method, similar to scrum [7]. We divided requirements and specifications into goals which were achieved by solving more minor problems. We use unit tests to prove that the problem is solved. We had weekly meetings where we discussed work done in the previous week and what should be done the following week. In these meetings is where we discovered all the requirements.

One issue with distributed systems is that they are hard to test because they are unpredictable and can grant different results because they are often non-deterministic. Bugs that happen in a distributed system because of specific ordering are called distributed concurrency bugs[24]. The developer must go through many permutations of events to find these bugs. That is why thorough testing is crucial in our case. The testing process includes locating bugs and reproducing them in a non-deterministic fashion.

We performed experiments to test the performance of the system. Even though performance is not necessarily essential to prove the concept, it indicates how usable and feasible the system is in the real world.

5 Approach

This chapter will discuss why we have chosen certain technologies and design choices in the context of our requirements. We aim to create and explore a continuous synchronization system for replicated SQL databases while upholding local-first properties with more than two sites connected.

5.1 Technology choices

5.1.1 Python

The programming done in this thesis is written in python. The main reason for this is that both the researcher and the supervisor are familiar with the programming language. Regardless, SQLite and python are similar because they require minimal setup and only consist of files. This means we can copy the code the same way as we clone the database, using SFTP. Python is an interpreted language meaning it does not need to be compiled to run commands, and we will ask remote sites to perform python scripts through SSH shells.

Python also has an easy-to-use SQLite library, shipped as a default python library[25]. One of pythons strengths is its many libraries; one we will use for interactive SSH shell is Paramiko. Paramiko supports both clients and servers over the SSHv2 protocol[26]. It is recommended for common client use cases, such as remote shell commands or transferring files because it provides the foundation for Fabric, a high-level SSH library.

5.1.2 CRDT

In [1] they state that CRDTs have the potential to be the foundation of local-first software, which is not surprising when considering what it allows and what local first software requires. CRDTs allow for concurrent updates on shared data, with the possibility for synchronization at a later point, which is what local first software requires.

5.1.3 SQLite

SQLite is a lightweight database management system, making it a clear candidate for a CRR system. SQLite is an embedded database accessible as any standard computer file, and this means it does not require any specialized processes to access the database. Other database management systems such as Postgresql[27] and MySQL[28] use a separate process to relay requests, which is not desirable since it requires a complicated setup. However, PostgreSQL and MySQL support concurrent access better than SQLite, and one requirement that we wanted to explore was SynQLite with more than two sites. This area is where SQLite might perform worse. Another issue with SQLite is that it does not have a built-in locking mechanism which could be helpful during the merging process.

5.1.4 TCP connection through SSH tunnels

We think establishing a TCP connection is the best solution for continuous synchronization. TCP has complete reliability[29], meaning TCP guarantees successful delivery of packages, which would prevent diverting states between sites or other side effects. We imagine users could run a simple script to connect to other sites, and then they perform pull and push requests to and from each other. SynQLite uses SSH to communicate, which means sites already have their SSH address. That is why we use SSH tunnels to establish TCP connections. Not using SSH tunnels would require users to connect using IP addresses, which would require configuration for security beyond normal users' capability. It is much easier to establish SSH connections.

5.2 Design choices

5.2.1 Push

We have to consider what is required to achieve these goals. Continuous synchronization would require the sites to be able to push and pull to and from each other unless they both perform pull. We want as little coordination between the sites as possible, meaning requiring all sites to perform pull is not desirable. The first thing we will implement is a push mechanism since the previous implementation of SynQLite did not have this mechanism. So sites can perform push and pull as a continuous synchronization.

5.2.2 Leader role

When considering multiple sites connecting, the main issues are SQLite concurrency limitations. Furthermore, different edge cases would occur when two sites synchronize simultaneously. We will introduce another role leader within the network topology. This leader is what the other sites can connect to and synchronize concurrently. This means that the distributed concurrency bugs would occur only at the leader. To prevent them, we will use locks.

6 Design and Implementation

This thesis is an extension of earlier work published on [4]. This section will describe the design and implementation of the system and its features. In section Chapter 7, we will describe what was already introduced in [4], distinguishing this work from earlier work.

6.1 Overview

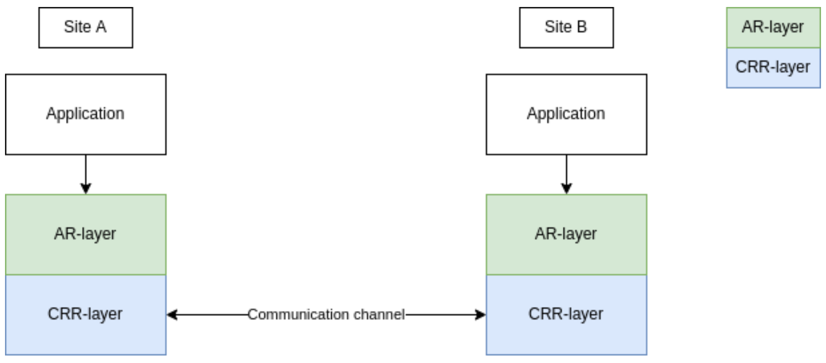

Figure 6.1: SynQLite overview

This system, SynQLite, enables SQLite databases to be augmented with CRR support. After giving the database CRR support, the user can clone the database to other locations. SQLite is used for this very reason because SQLite databases are only made up of one SQLite file, meaning sites can clone their entire database with SFTP. Following cloning, both sites can perform concurrent updates. The sites can sync with each other as often as they want, as long as the other sites are online.

SynQLite is written in Python3, using TCP and SSH to communicate between sites.

There are two main ways to synchronize with other sites: continuous synchronization through TCP connection and discrete push and pull when the sites want to through SSH. Both methods have different benefits and drawbacks. This section will first describe the general design and architecture; then, we will describe the two different methods in detail. As seen in Figure 6.1 the software and data elements with green color are in the application relation layer, and the software and data elements in blue are in the CRR-layer. We will use this color scheme throughout the thesis.

6.2 Database Architecture

The user is not required to change anything in their schema to use SynQLite, but SynQLite will add tables and triggers to the existing database. The added tables and triggers represent the CRR-layer, while the existing tables with which the user interacts represent the AR-layer. It is crucial that the user only directly queries the AR-layer and not the CRR-layer. Directly querying the CRR-layer can cause unwanted side effects because CRR-layer is supposed to reflect the state of the AR-layer.

6.2.1 Tables

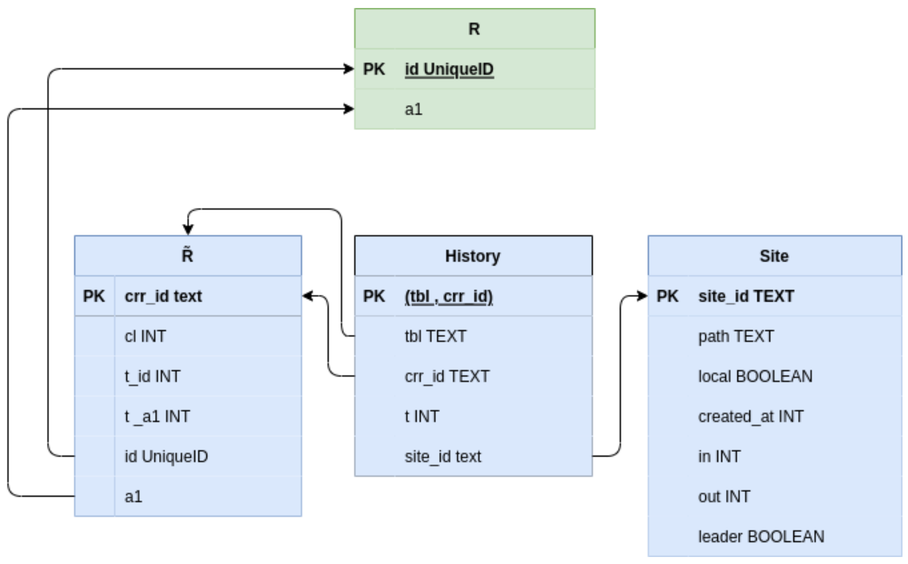

Figure 6.2: SynQLite schema overview

Augmenting a database with schema R(a1, a2,...), a new table R̃(crr_id, cl, t_a1, t_a2,...a1, a2,...) is created where R̃(crr_id) is a unique identifier, R̃(cl) is the causal length of the row, and R̃(ti) is the time stamp for R̃(ai). Tuple (crr_id, cl) represents the CLSet CRDT, and (t_ai, ai) represents the LWW register CRDT within the row. Figure 6.2 displays what happens when augmenting relation R with CRR support.

SynQLite also creates a site table S(site_id, path, local, created_at, in, out, leader) that keeps the information about other sites. S(site_id) is a unique identifier and the primary key. S(path) is the path to the database. It can either be a personal file or an SSH path, depending on where the site is located compared to the local machine. S(local) is whether this site is the local site. S(created_at) is when the database was augmented with CRR support, either by being cloned or initialized. S(in) is the timestamp of when the local site transferred δ from the local site to the remote site. Either the remote site pulled the state from the local site, or the local site pushed its state to the remote site. S(out) is the opposite; the timestamp of the remote site transferring δ to the local site. S(leader) is to mark if the site is a leader. SynQLite uses S(leader) for continuous synchronization, which we will describe in Section 6.7.

SynQLite will also create a history table HR(tbl, crr_id, t, site_id) that holds when each row was last updated in R̃. HR(tbl) is the table's name where the changed row is, HR(crr_id) is the unique identifier for that row within R̃, HR(t) is the timestamp of when the row was changed, and HR(site_id) is the id for the site performing the update referencing one entry in the site table. Tuple (tbl, crr_id) is the candidate key for the table. New rows replace old rows if they have the same candidate key so that each row in R̃ is only represented by one row in HR. HR needs to include HR(t), so SynQLite can have multiple R̃ without looping through all R̃ tables to find an entry that matches HR(crr_id).

Sites use S(in), S(out), and HR to generate the correct δ from R̃ before synchronizing it to other sites. We can now know when we last sent δ or received δ between sites with the S(in) and S(out), which helps us generate smaller deltas. If we dropped the HR table, we would have a normal state-based CRDTS. When synchronizing, sites would have to send every entry in every R̃ table since they would not have any information on the other sites' states.

6.2.2 Triggers

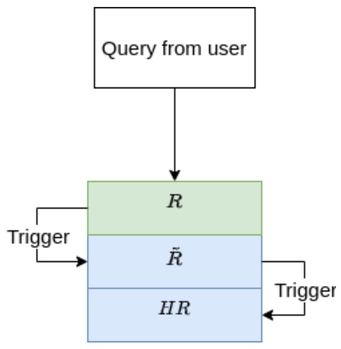

Figure 6.3: Query plan

Triggers link the AR-layer with the CRR-layer and are essential for CRR to work in a relational database. There are three triggers in R̃ table; add, delete and update triggers. These will propagate these SQL operations from the AR-layer to the CRR-layer. Add and delete triggers will increment the causal length in R̃ for that element with the same primary key following the CL-set CRDTs rules when a user adds or delete an element in R. When the user updates R(ai), the trigger updates R̃(t_ai) following the LWW-register CRDT rules.

There are also two other triggers for the history table HR, which will trickle down updates from R̃ to HR. When the add trigger adds a new entry to R̃, a new entry is added to HR with a new timestamp as HR(t) and HR(crr_id) as the same crr_id as in the R̃. The last trigger is for updates done in R̃; this will update the HR(t) and set HR(site_id) to the site_id, which did the update. Update trigger would fire when a delete occurs in the AR-layer since the delete trigger updates the HR table. Update trigger would also fire when a deleted element was reinserted to the AR-layer since this would perform an update to the $\tilde{R}$.

6.2.3 Indexes

$\tilde{R}$(id), $\tilde{R}$(crr_id) and HR(crr_id) are indexes for their tables. $\tilde{R}$(id) is the primary key for R. This is very important to prevent an exponential time increase of SQL operations on the database because the triggers query both tables. When an application performs inserts, deletes, or updates to R, it must first check the unique constraint for id within R and $\tilde{R}$, then check (tbl, crr_id) in HR. If we do not have the indexes, the chain of triggers will cause an exponential drop in performance when performing insert, update, or delete in R.

6.3 Clone

Cloning is a necessity to allow replication. When cloning, the original site copies the whole database file to another location using SFTP. We use SQLite since the whole database consists of one file making it easy to transfer to other locations with SFTP. After copying, the only table that needs to be updated is the site table since the local table for the new site would be the site with its path. All entries in other tables would be the same as in the other database. The HR triggers had to be changed to use the correct site_id to note which site performed the update.

6.4 Communication

Understanding how the different sites communicate is crucial before explaining the synchronization process. The site communicates through SSH or TCP connections through an SSH tunnel. These channels use the SSH path from S(path) to communicate. SynQLite uses SSH when the user does not want continuous synchronization but rather just discrete merges through push and pull requests. TCP is used to connect to a centralized leader API for continuous synchronization. When a site wants to connect to the leader, it uses the SSH path to connect with the API through a TCP channel. Using an SSH tunnel reduces the metadata needed to connect to the TCP channel, and it is more straightforward since they already know each other's SSH path.

6.5 Synchronization

Sites synchronize by either pushing or pulling, and both include some similar steps; generating δ, transferring δ, and merging received δ with the local state. The difference between pulling and pushing is whether the data goes in or out. The following sections will first explain how sites create δ and how they merge δ. After that, we will describe how sites transfer δ to other sites because there are two main communication methods.

For simplicity, we will use an example where a site with id α generates δ and sends it to a site with id β. Then site β merges δ to its state.

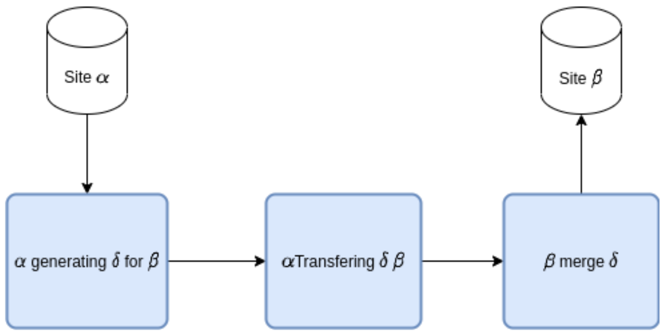
Site α
Site β
α generating δ for β
αTransfering δ β
β merge δ

Figure 6.4: δ from α to β

6.5.1 Generating Delta

```
SELECT *
FROM ~R as crr
     inner join HR as hist
     on (crr.crr_id == hist.crr_id)
WHERE hist.t >
      (SELECT out FROM site
       WHERE site.site_id={site_id})
      AND hist.site_id != {site_id}
```

Figure 6.5: Generate δ

site_id is the id of the site that the δ is for

When generating δ, α looks at all entries in the history table(HR) with a later timestamp than S(out) in β; we call this δHR. Then α joins tables δHR(tbl) with δHR on δHR(crr_id), creating δ. α also filters out all entries with site id equal to βs site id since this means β produced it. This filtering process is similar to BP optimization from [19], giving us a more optimal δ. In more technical terms, α loops through all R̃ tables and join them with HR using the query in Figure 6.5

6.5.2 Merging Delta

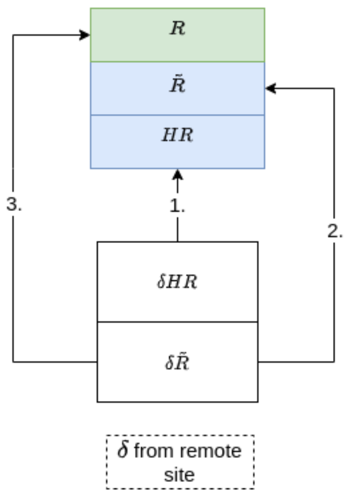

Figure 6.6: Merging process

β splits δ into δHR and δR̃ parts. When merging delta, the rows trickle up from the CRR-layer to the AR-layer. This means we add to the CRR-layer, then to AR-layer to know what state each row should have. All triggers are disabled before merging; this prevents entries from trickling down to CRR-layer after adding to AR-layer. We created another table called merge lock to disable triggers with only one entry; up. The triggers check if up is true in the where clause. By disabling triggers, we mean setting up to False.

First, β loops through δHR and adds it to HR. β uses the table name inside the inserted row to know where the δR̃ with the same crr_id should be inserted. Secondly, β merges δR̃ row with its R̃, following the CRR rules. If an entry already exists in R̃ with the same crr_id, β compares the causal lengths; the one with the highest causal length determines if it should exist. After that, β compares the t_ai for each attribute, the most recent timestamp for each attribute wins and decides which attribute β should choose. Thirdly, based on the updated row in R̃, it is trickled up to the AR-layer. The triggers are enabled when β has merged all entries from δ. To limit the divergence of clocks at each site, the most recent timestamp from the δ is compared with the local clock. β clock is updated if δ clock is newer than the local clock.

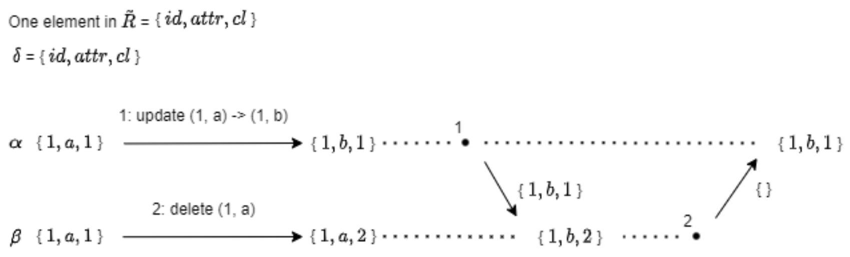

Figure 6.7: Merge edge case

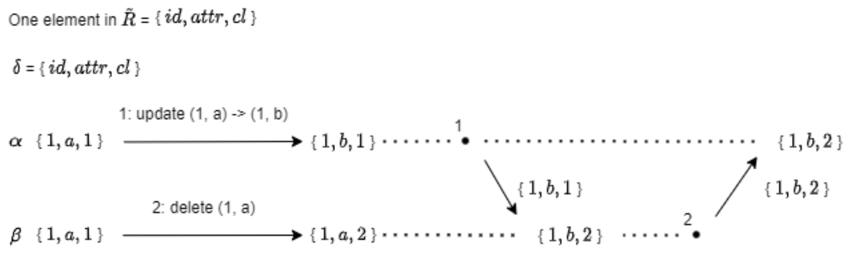

Figure 6.8: Fixed merge edge case

We have not addressed one edge case, depicted in Figure 6.7. The edge case is when one site deletes a row simultaneously as another site updates the row. In Figure 6.7, both sites α and β start with the same element { 1, a, 1 }. Site α updates the row to { 1, b, 1 } at the same time site β deletes the row. In point 1, α pushes its state to β , making βs state { 1, b ,2 }. In point 2, β pushes to α .βs δ to α is empty even though βs element has a more significant causal length than αs. This is because β writes in HR that α performed the last update on the element, then in point 2, β does not add the element to the δ.

We fix this by checking when merging if the old row has newer information than the incoming δ. If the old row has newer information than the δ, the new history entry is changed to the merging site id and updated the timestamp. Figure 6.8 displays what happens with this fix. In point 2, β pushes its state because what α pushed in point 1 had older information than β.

The following sections will describe two different methods of using SynQLite: discrete push and pull vs. continuous synchronization.

6.6 Discrete push and pull

All communication is done through an SSH shell when using discrete push and pull. In this thesis, we wanted to limit the time we write and read to disk for several reasons. The main reason is that disk I/O is very slow, and many seeks to the disk will wear it out. An earlier version of SynQLite created an SQLite file for delta, and then the δ was copied over to receiving site by using SFTP. The old version wrote to disk three times, while now it only writes ones when transferring delta.

6.6.1 Pull

When α wants to pull from β, site α sends the query in Figure 6.5 to β through a SSH shell command. Returning from this query is δ, and α merges it with its database. β then updates S(out) in row $S(\alpha)$, and α updates S(in) in $S(\beta)$. So that the next push or pull does not send redundant data.

6.6.2 Push

Push is the opposite of pull. When α wants to push to β, α generates δ and asks β to merge it to its database. α tells β to execute a python script with δ as an argument. The script merges the δ argument to the local database. After that α updates S(out) in $S(\beta)$ and β updates S(out) $S(\alpha)$ for the same reason mentioned above.

6.7 Continuous synchronisation

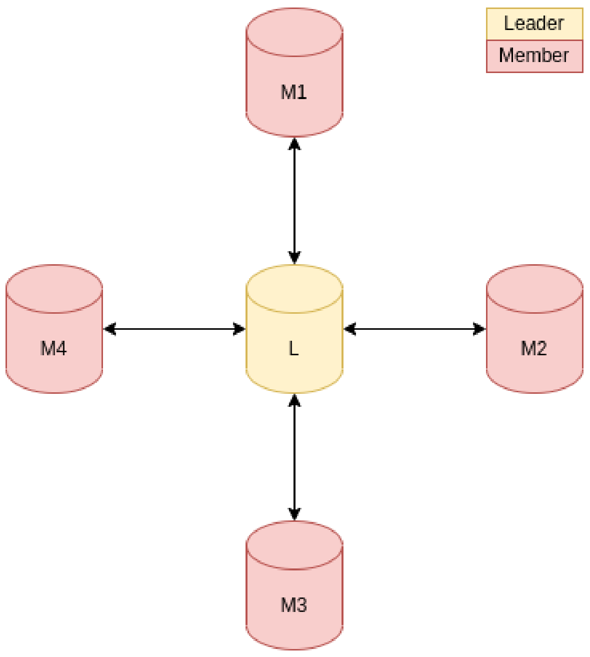

Figure 6.9: Leader network topology

One of the main contributions of this thesis is continuous synchronization through a TCP connection using an SSH tunnel. This version uses a centralized network topology where one site is a leader. The site is the leader by assigning S(leader) to true. All other sites are only connected to the leader, and these sites are called members. Figure 6.9 displays the network topology. We assume that all members know about the leader.

The leader starts an API server that listens to a specific port and waits for members to connect. When a member connects, the server starts a thread that handles requests from that member. We call that thread a client thread. The client thread takes three types of requests from the connected member: push, pull, and verify.

From the members' perspective, there are only two sites in the network, the leader and itself.

6.7.1 Verify

After establishing a connection, the client thread expects a verify request as the first request. After connecting the member, calls verify with the member's site id as a parameter. The client thread checks if it is in the leader site table. The client thread returns an acknowledgment package if it exists. Client thread saves the members id to be able to generate deltas and update S(out) and S(in) in the site table and HR(site_id) in the history table.

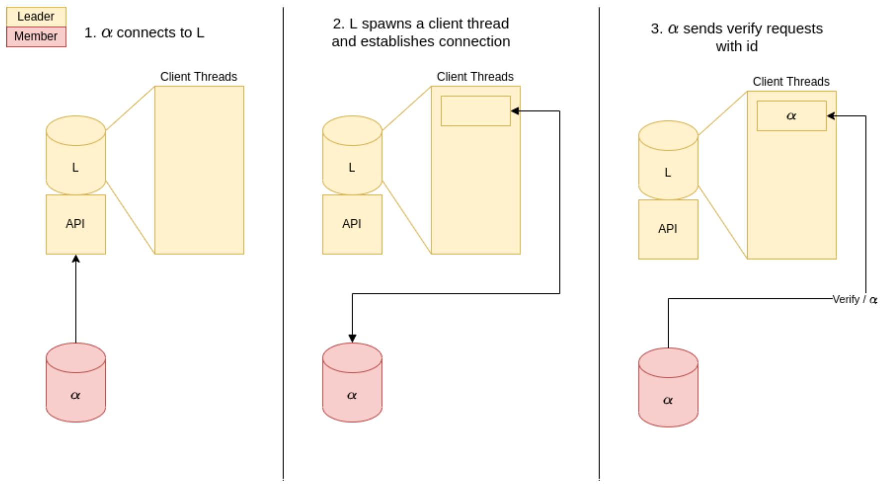
Leader
Member
1. α connects to L
Client Threads
L
API
α
2. L spawns a client thread and establishes connection
Client Threads
L
API
α
3. α sends verify requests with id
Client Threads
α
L
API
Verify / α
α

Figure 6.10: Member α connecting to leader L

Figure 6.10 displays member α connecting to the leader L. In point 1, α connects to L using the API. In point 2, L spawns a member thread and establishes a connection with α. In point 3, α sends a verify request with α site id.

6.7.2 Push

When a member calls push, the client thread locks the database and notifies the site that it is ready to receive δ. Then the member sends δ. Before merging it to the leader database, the client thread modifies δHR(t) to when the API received δ, different from the discrete push and pull method with no leader. Updating δHR(t) is done so that it looks like the leader did the update, fixing a problem shown in Figure 6.11. After this, δ is merged to the leader database the same way mentioned above. Lastly, the leader notifies the member that it is done merging.

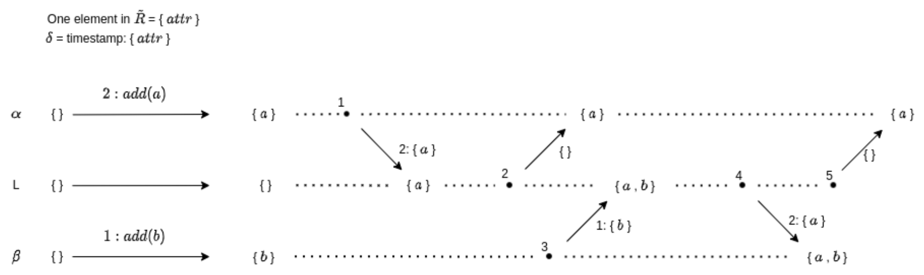

Figure 6.11: Bug when pushing to leader

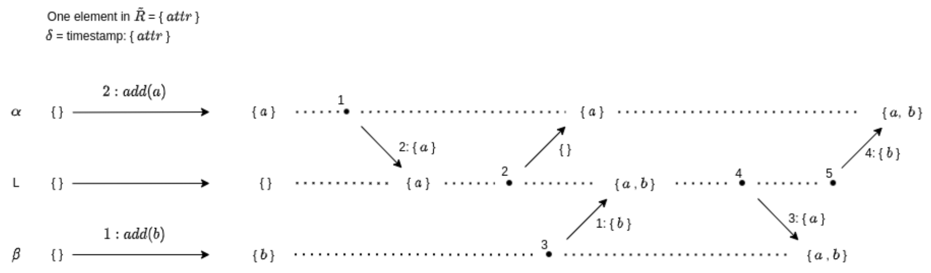

Figure 6.12: Bug fixed when pushing to leader

Figure 6.11 displays the leader L synchronizing with two members, α and β . All sites start with an empty state. α decides to add { a } simultaneously as β add { b } . In point 1, α pushes its state to L, updating S(out) for α in L to 2. In point 2, α pulls from L, which returns nothing. In point 3, β pushes its state to L, making Ls state { a, b }. In point 4, β pulls Ls state, L generates 1: { a } as the δ for β, and β merges it to its state. In point 5, α pulls from L again, but L returns nothing. This is because when β pushed, δ had a timestamp of 1; as mentioned before, L noted S(out) for α as 2. This means that when L generates δ for α, it thinks that α has gotten that element. The row in Ls HR for b have a earlier time stamp than S(out) for α. Figure 6.12 shows what happens in the same scenario, but the leader sets the time stamp when it gets a new state. The big difference is that in point 3, the leader gives the δ from β a later timestamp than the S(out) for α. L then generates { b } in point 5, giving α the correct state.

6.7.3 Pull

When a member calls pull, the leader generates δ and changes the δHR(site_id) to the leader's id, and this is so that the member does not send the data back to the leader at a later point, BP optimization [19]. The size of δ is sent to the leader so that the member knows how much to receive. The leader then sends

δ to the member, and the member merges it into its database.

6.7.4 Autoincrement primary key

SynQLite supports primary keys that are auto-incrementing integers, which are primary keys that, if not specified for a new entry, give the entry an id with one id higher than the highest in the table. When merging a new row from a remote site, the id is not merged and added to the AR layer. Instead, the local site gives the row a new autoincremented id. This means each site has its primary key for that row. The primary key is saved in the local $\tilde{R}$ table so that it is added back if the row is deleted and added back.

6.8 Command-line

To use SynQLite, the user run command through a command-line script. We have taken inspiration from git commands when naming the commands. The command-line script includes:

- init: augmenting existing SQLite database with CRR support.
- clone: clones SynQLite database.
- push: push δ to a remote database.
- pull: pull δ from a remote database.
- api: starts a centralized leader, which members can connect to for continuous synchronization. A leader can only use it.
- tunnel: connect to the centralized leader. It will only work if the leader has run api.
- upload: upload SynQLite code to a remote location.

6.9 Using SynQLite for local-first software

SynQLite is not a process that is required to run for the user to interact with the database. SynQLite is a service that users can use to communicate with other sites, augment existing SQLite databases, and clone CRR SQLite databases. Applications using a database augmented with CRR are not required to download extra libraries or dependencies. SynQLite acts as a middle layer that applications can use to synchronize with other sites. Applications use simple commands described in Section 6.8 to use SynQLite, and implementation details are abstracted away from the application.

If the application wants to synchronize continuously with other sites, one of the sites must run the API command, and the other sites must run the tunnel command.

6.10 Discrete vs Continuous synchronisation

Now that we have described the two ways to use SynQLite, we want to describe the benefits/drawbacks of both. We wanted to support two use cases for SynQLite; one more individual where one works alone a lot and irregularly synchronizes, and another where one is continuously synchronizing like one would with a regular cloud app.

The discrete push and pull model fits the first use case because one does not need to establish a connection each time one wants to synchronize. While the continuous model is more fitting for the second case, this gives the user a more seamless experience. We use TCP connections for the continuous model because it is faster and has less overhead than SSH shell calls after establishing a connection. Another benefit of the discrete model is that sites can push to other sites that are not leaders, and the other sites do not need to run a script to accept requests.

We feel that having the flexibility of how one share their data is what local first software is all about.

6.11 Implementation process

As mentioned Chapter 4 we used an iterative method. However, we wanted to explain this process in more detail. This section will explain the test environment and what type of technologies we used. Then we will explain the process of implementing a new feature, from the requirement elicitation to finishing a feature. Lastly, we will use an example of one feature to show the process in more detail.

6.11.1 Test environment

We used pytest[30] to create tests. Pytest is a testing framework that makes it easy to create small readable tests. To simulate a distributed system with remote sites, we used docker[31]. We used Pytest because Pytest has test fixtures that help with setting up the test environment. Fixtures made the setup process seamless, especially with the docker image. Before running the test, we create a docker image, and then the fixtures would initialize a local CRR database and clone it to the image using SynQLite.

6.11.2 One iteration

Each week we had a meeting where we discussed potential features or bugs. We then discussed how this could be implemented and what kind of technologies are required to solve it. Then I would create unit tests using Pytest, which highlights the feature. Then I would try to implement the feature needed to pass the test. I would note problems and other findings I found so that we could discuss them at the next meeting. This process would continue until I was finished with the feature or fixed the bug. Then I would push the current implementation to our git repository. Then the process would start again.

Being able to bounce ideas with someone that knew the system helped me unravel many edge cases similar to Figure 6.12. Having many thorough tests made the implementation process much easier because when I made a change, I could run all tests to see if something else in the system had been affected. This allowed me to experiment and change things freely without the fear of ruining previous work.

6.11.3 Fixing generating delta bug

This subsection uses an example to show how the implementation processes worked. In a weekly meeting, my supervisor wanted SynQLite to generate δ in a more optimized way. Because at that point, SynQLite worked as a state-based CRDT, not a δ CRDT. We discussed what would be required to fix that; we came up with adding S(out) in S to generate δ by comparing S(out) with HR(t). Figure 6.13 is the test created to highlight the problem. In the test, local_crr adds 'doc1' then pushes to remote_crr, then local_crr adds 'doc2' and generates a new δ for remote_crr. For the test to pass, the new δ must only have 'doc2,' and S(out) for remote_crr must be larger than before pushing.

```
def test_push_delta(local_db, remote_db):
  local_crr = Crr(local_db)
  remote_crr = Crr(remote_db)

  pre_last_push = ldb.query(
    f"SELECT out from site_tbl WHERE path = {rdb}"
  )
  ldb.insert(
    "insert into AR_tbl(attr) VALUES(doc1)"
  )

  crr.push_all_sites()
  ldb.insert(
    "insert into AR_tbl(attr) VALUES(doc2)"
  )

  r_id = r_crr.site_id()
  crr_delta, hr_delta = l_crr.generate_delta(r_id)
  last_push = ldb.query(
    f"SELECT out from site_tbl WHERE path = {rdb}"
  )

    assert last_push > pre_last_push
    assert crr_delta == 'doc2'
```

Figure 6.13: Test δ generating bug

This issue has two parts;

1. Create S(out) and update S(out) when data is going out of the site.
2. Compare S(out) with HR(t) when generating delta.

The first part includes adding S(out) to S and then updating it when data goes out of the site. This means that S(out) must be updated when α pushes to β and when β pulls from α. The second part includes changing the generate δ script, where we check that HR(t) is more significant than S(out), as seen in Figure 6.5

After implementing, I run the test to see if it is done. Then I run all the other tests before pushing the implementation to our git repository. The supervisor then looks over it, discussing it further in the weekly meeting.

7 Contributions

This section will describe the state of implementation before starting this thesis to separate contributions from this thesis from earlier work[4].

7.1 Features from previous implementation

In [4], the main focus was augmenting SQLite databases with CRR support and synchronization between two sites. The paper introduced HR and $\tilde{R}$ to be able to generate delta. Sites could synchronize through pull requests, but sites could not perform push requests. When a site performed a pull request, the pushing site generated delta and placed it in a separate SQLite file. This file was transferred through SFTP. The pulling sites attached the file to its database and merged it.

7.2 Features introduced from this thesis

7.2.1 Main contributions

One of the main goals of this thesis was to improve SynQLite to give the users a more seamless experience. That is why we added automatic synchronization. Another goal was to support more than two sites synchronizing simultaneously.

We achieved both goals by implementing the centralized TCP API.

However, this introduced new problems from many sites concurrently performing operations. Testing and detecting edge cases got harder because reproducing them was a challenge. These edge cases are like the ones discussed in 6.7.2 and 6.5.2. We have used much time to try to detect most of these edge cases, but there is no way to know if we have found every bug.

7.2.2 Other contributions

Other features and contributions added from this thesis:

- Two methods for synchronization; discrete and continuous.
- Fixing delta generator by adding S(out) and comparing with HR(t) when generating δ
- BP optimization to δ by adding HR(site_id) and not sending it back to the creator of the data element.
- Limit writing to disk to one when synchronization. Using SSH shell or TCP for propagation instead of SFTP.
- Evaluation of SynQLite's current implementation using experiments and the requirements stated in the introduction.

7.3 Local first properties

The rest of the thesis will focus on evaluating the current state of SynQLite. Before starting this process, we will define what metrics we will use to evaluate. SynQLite is supposed to support local-first software. When we think of local-first software, the user should have the primary copy of the application. The application should be available at all times since it is accessible on the user's local machine. Local-first software should allow the users to share data and utilize the many benefits of cloud computing. Such as sending data to perform heavy computational work to a data center.

By this definition of local first properties, we split it into two parts.

- Offline experience

- Online experience

By offline experience, we mean how much the local first software affects usability when offline. For example, how much the software slows down the application. In our case, this would be how much SynQLite slows down SQL operations on a database augmented with CRR support and how much the database grows in size.

The online experience is how much the software affects the online experience. For example, how long it takes to send data to another site. In our case, this will include checking how fast sites can synchronize.

8 Experiments

This chapter will evaluate SynQLite's current implementation by running experiments. The primary purpose of the experiments is to see how well SynQLite works as a service giving users local-first properties, meaning having the possibility to work offline and then later synchronize.

All experiments will be running on the same computers with specifications noted in Section 8.1. All experiment uses the same schema as the base relation, Person(id autoincrement INT, name text NULL, age INT). We created a set of queries that the experiment uses noted in Section 8.2, which means that when two experiments perform insertions, it is the same set of insertions on both. We will execute every experiment at least five times and use the average. All experiments except the journal mode experiment in Section 8.5 will use MEMORY mode as the journal mode.

In the first part, we want to see how much SynQLite influences the offline experience by comparing the time of performing different types of SQL operations. These SQL operations will include insert, update, delete, and select. We will also evaluate the size increase of an augmented database, which should increase by a significant amount.

In the second part, we will look at the online experience. The online experience includes both continuous synchronization and discrete push and pull. We will look at how fast these different methods are and how they compare. These experiments will include different use cases of SynQLite, from users who often synchronize to users who synchronize rarely. We want to see if there are correct ways to use SynQLite for specific use cases.

To finish the online experiments, we will look at multiple sites connected to one leader. Here we want to see how increasing the number of sites affects the synchronization time.

In the third and last part, we will look at the performance of SQL operations when using different journal modes. The result from this experiment will be used to discuss the potential of using journal mode more actively in later implementation.

8.1 Hardware specifications

The experiments were performed on two computers, one of the computers represents a local machine, and the other represents a remote machine. The computers have SSH access to each other.

Local machine:

- CPU: Intel i5-6400T (4) @ 2.800GHz
- Disk: SAMSUNG MZ7LN512HMJP-000L1, 512GB SSD
- Memory: 2x 8192 MB

Remote machine:

- CPU: 2 x Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
- Disk: 4 x 2000GB HDD i hardware RAID5
- Memory: 128 GB RAM

The remote machine is only used for experiments involving synchronization. In the synchronization experiment, we take time from the perspective of the local machine, meaning we send requests from the local machine and wait until the remote machine responds. The rest of the experiments are only conducted on the local machine.

8.2 Experiment setup

We generate the SQL operations the same way for all experiments so that the results are comparable to each other. SQL operations performed on the databases are: update, insert, select, and delete. All experiments except the experiments conducted in Subsection 8.3.1 use only insert operations to fill the database of the local machine. The schema used for all experiments is Person(id autoincrement INT, name text NULL, age INT). When inserting each site runs the script in Figure 8.1, with a unique tag to prevent unique constraints errors.

```
def insert(db, num_queries, tag=""):
  for i in range(num_queries):
    db.insert(
      f'
        insert into person(name, id, age)
        values({str(i)+tag},{i},{i})
      '
    )
```

Figure 8.1: Insert persons to db

```
def delete(db, num_queries):
  for i in range(num_queries):
    db.delete(
      f'
        DELETE from person where id={i}
      '
    )
```

Figure 8.2: Delete all elements in person from db

```
def select(db, num_queries):
  for i in range(num_queries):
    db.select(
      f'
        SELECT * from person where id={i}
      '
    )
```

Figure 8.3: Select each element in person from db

```
def update(db, num_queries):
  for i in range(num_queries):
    db.update(
      f'
        UPDATE person set name="i{i}" where id={i}
      '
    )
```

Figure 8.4: Update name in person from db

8.3 Offline experiments

8.3.1 Speed

We start with testing the difference in performance between databases with SynQLite augmentation and not. The experiment includes taking the time to insert people using the script in Figure 8.1, updating their name using the script in Figure 8.4, searching for the people's names using the scrip in Figure 8.3, and lastly deleting all the people using the script in Figure 8.2.

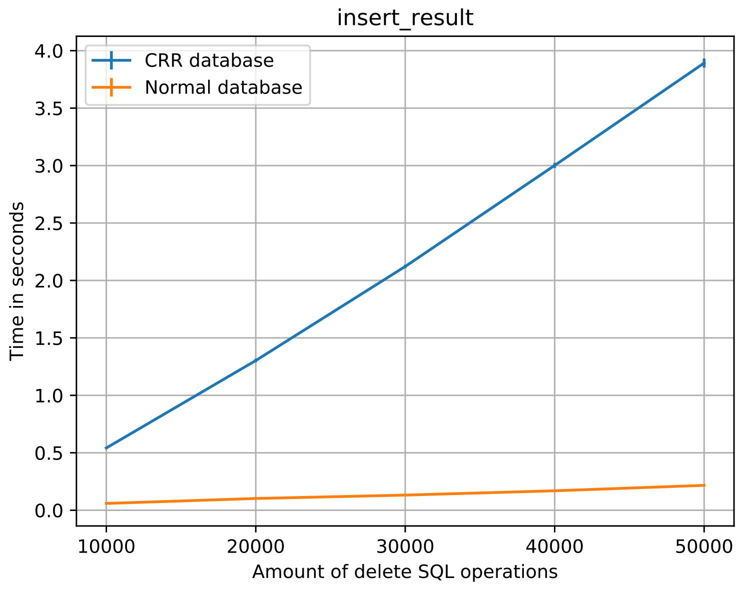

Figure 8.5: Insert result

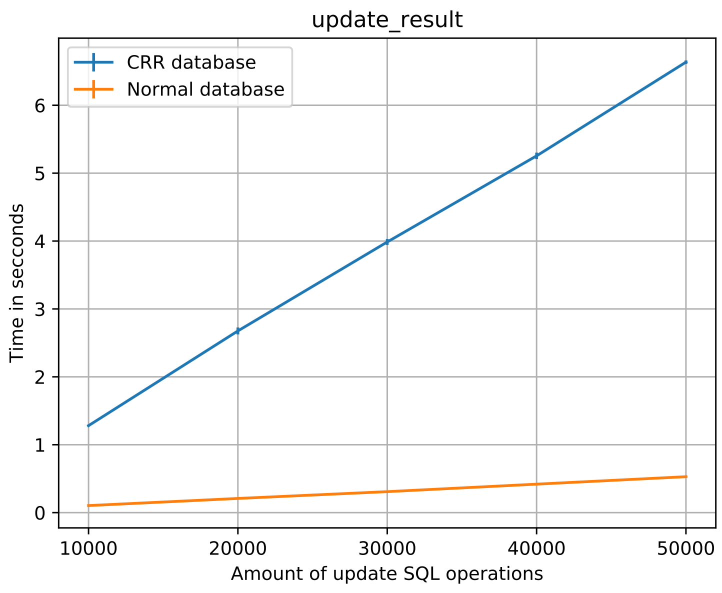

Figure 8.6: Update result

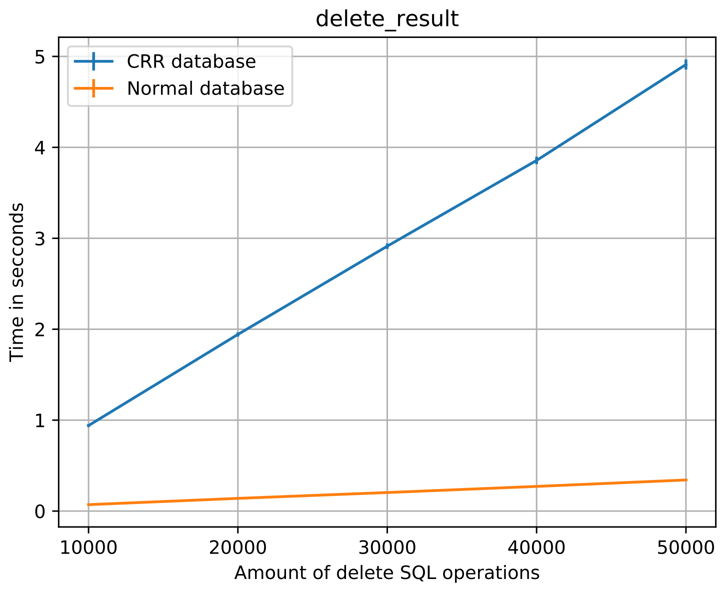

Figure 8.7: Delete result

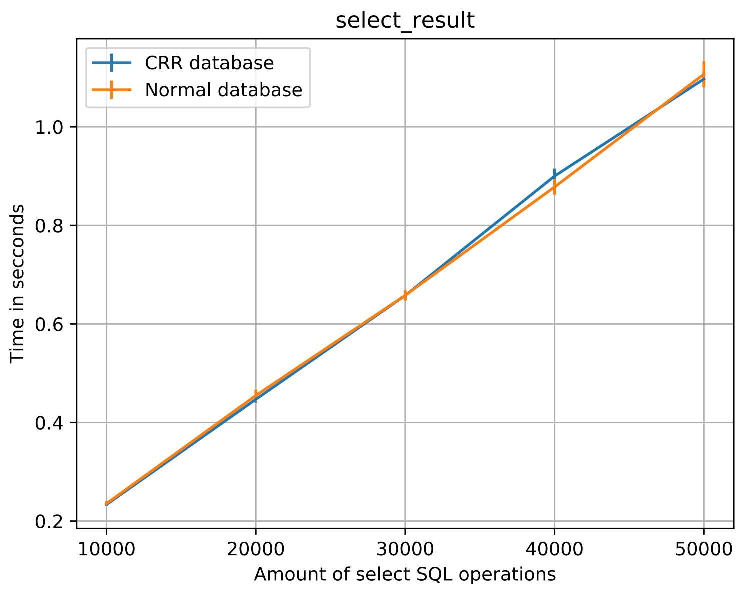

Figure 8.8: Select result

The results show that operations that perform write operations perform significantly worse when the database is augmented with CRR support. Inserts are 18 times slower with CRR support, while delete and update are 12 times slower. Querying is the same with or without CRR support.

8.3.2 Space

In this experiment, we test how much augmenting the database with SynQLite increases the size of the database. We want to see how much the size increases by adding tables and triggers. Then, see the size increase by inserting rows to the AR-layer, which will trickle down to the CRR-layer.

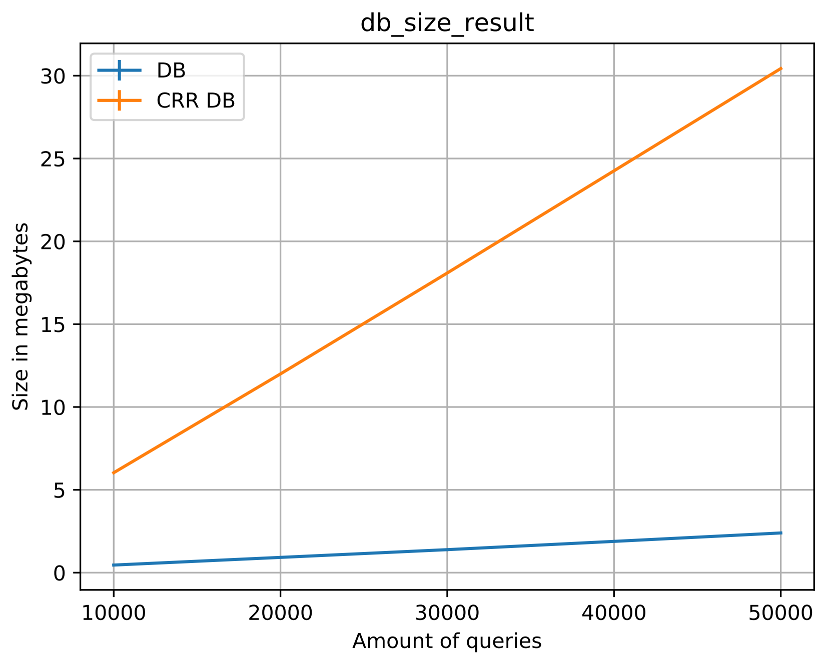

Figure 8.9: db size result

Figure 8.9 shows that the database with CRR support is significantly larger than the one without CRR support. The difference is not surprising because when one row is added to the Person another is added to $\tilde{Person}$ and HR.

8.4 Online experiments

In this section, the focus is on the online experience when using SynQLite. Looking at how fast sites synchronize. There are two use cases for SynQLite, discrete push and pull and continuous synchronization. We want to see if the two different models work better for the mentioned use cases.

We also want to test how the system works with multiple connected sites to one leader. Looking at how fast the sites synchronize in different cases. When the sites perform many more minor changes, and when the sites perform few more considerable changes. We also want to see what happens when we increase the number of connected sites. We expect a bottleneck at the leader at some point since the leader locks its database.

8.4.1 Merging Delta

Synchronization is a big part of the online experience, and δ merging is a big part of the synchronization process. How much time and resources are used at each site to be consistent will correlate directly with the seamless experience that we want. This section will perform experiments to measure the time to merge δ to one local database. To locate potential bottlenecks, we will divide the synchronization process into smaller processes and measure each step's time. These steps include; generating δ, fill δ to a temporary database, merge δ from the temporary database.

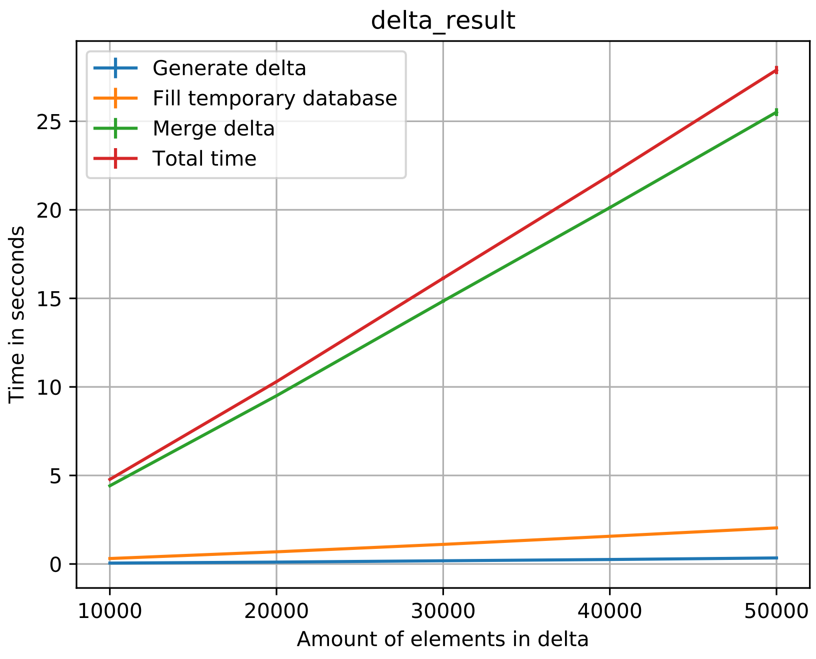

Figure 8.10: Delta result

Figure 8.10 shows that merging δ from the temporary database is what takes the most time. Other than that, we can see that the merging process scales linearly with the size of δ.

8.4.2 Discrete vs Continuous synchronization

We want to evaluate the speed of using either the discrete or continuous method. We set up two sites and took the time to pull for both methods. The sites were at two different machines, the local machine, and the remote machine. The local machine performed requests to the remote machine.

Pull

We ran two experiments to evaluate the two different methods; first, a user synchronizes a few times with an extensive data set, and second, a user pulls often but with smaller data sets.

In the first experiment, the remote machine inserted several rows into its AR layer, and then the local machine pulled from the remote machine. We changed the number of rows inserted to see if there was a certain threshold that changed the result significantly. The result is shown in Figure 8.11.

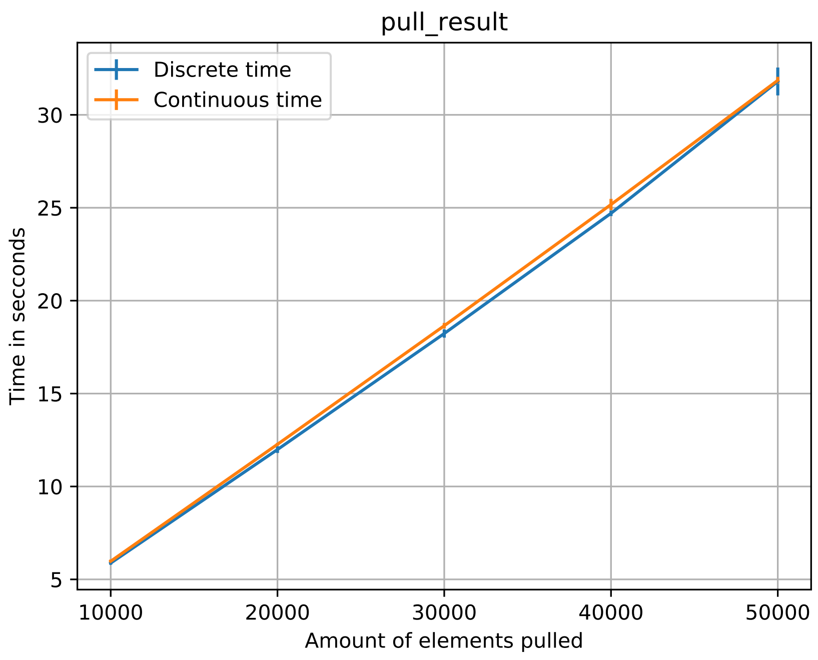

Figure 8.11: pull result

Figure 8.11 shows that there is no significant difference between the two models when only pulling one time, even if the dataset gets very extensive.

In the second experiment, the remote machine inserted one row, and then the local machine pulled from the remote machine. Figure 8.12 displays how long it took for a site to pull a certain number of times. One issue with this experiment was that the inserting into the remote machine was included when taking the time. To get the correct result, we removed the time for inserting it into the database.

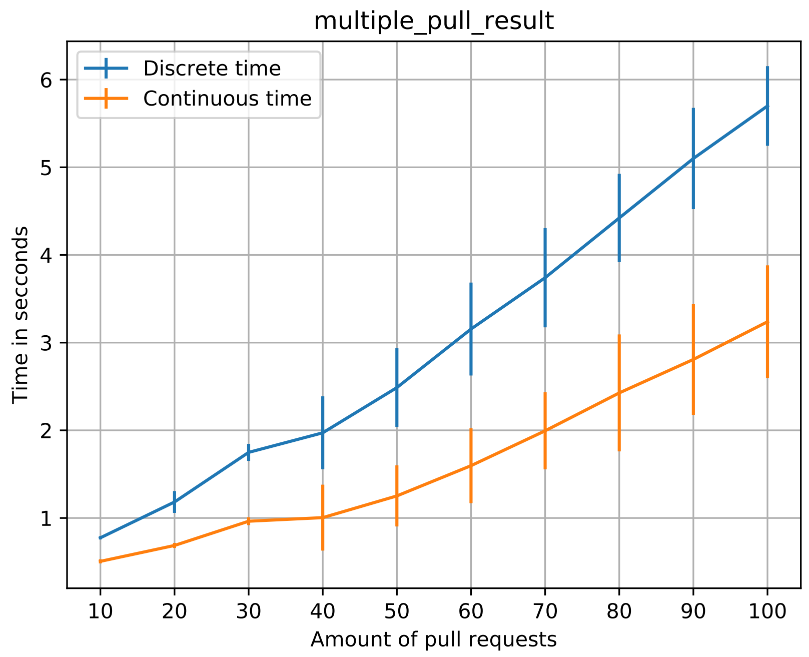

Figure 8.12: multipe pull result

Figure 8.12 shows that there is a significant difference between the discrete model and the continuous model. The continuous model is around 1.5 to 2 times faster than the discrete model.

8.4.3 Leader

In the following experiments, we test how well the centralized leader implementation scale with the number of sites connected. The leader is the remote machine, and the members are from the local machine. One problem with this experiment is that all the members use the local machine's hardware. This means all members will write and read from the same disk, which is not how users should use the system. We will not write or read from the local machine's databases to prevent this issue but instead prepare data in memory to push to the leader.

We test how well the leader handles multiple sites pushing to the leader. We prepared the data before pushing it to the leader, as mentioned above. All sites added 10000 rows to their database and performed push requests simultaneously.

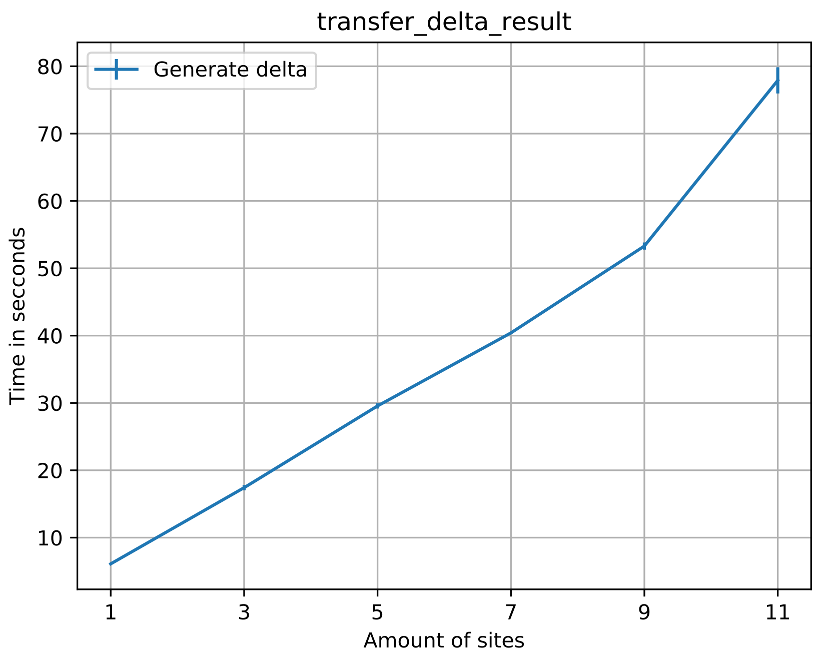

Figure 8.13: Push delta result

Figure 8.13 shows that the time to push to the leader scales linearly with the number of members until it reaches 11 members. It should be mentioned that the leader has a CPU with ten cores shown in Section 8.1.

8.5 Journal mode

Lastly, we wanted to test how different journal modes affect the system's performance. Journal mode is interesting for SynQLite since it gives the database system atomic commit with rollback capabilities, and distributed systems are volatile and have many sources for crashes. Rollback capabilities help databases keep a consistent state in case of a crash.

Write-ahead-log is even more attractive for SynQLite because of two reasons. The first reason is that WAL mode stores multiple transactions in the rollback file, which allows the site to roll back to a state multiple steps behind. The second reason is that when an SQLite database uses WAL mode, a process can simultaneously write to the journal file as another process reads from the main SQLite file. Furthermore, one of the main issues with the given solution is that we have multiple processes concurrently reading and writing from the same file. We think WAL mode could be used to fix this concurrency issue.

The experiment includes inserting different amounts of rows into a CRR database with all the journal modes; write-ahead-log(WAL), TRUNCATE, PERSISTS, MEMORY, DELETE, and off mode.

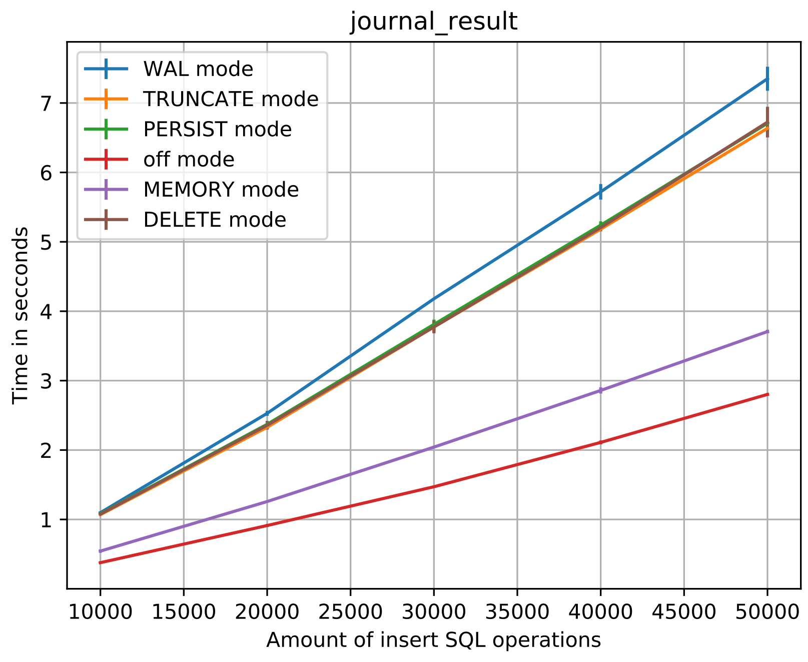

Figure 8.14: Journal mode result

Figure 8.14 shows that the database with WAL mode is the slowest. TRUNCATE, PERSISTS, and DELETE mode perform the same but faster than WAL mode. MEMORY mode is the second-fastest. Lastly, off mode or no journal mode is the fastest.

9 Discussion

This chapter will discuss the work done in this thesis. The discussion will focus on how well our work achieved the goal stated in the introduction of this thesis. We will discuss what advantages and disadvantages the solution has and potential improvements that could be made in a later thesis.

Major requirements:

- Automatic synchronization CRDTs SQL databases or CRR databases.
- CRR system with multiple members simultaneously synchronizing.

Minor requirements:

- User should be able to push and pull to and from other sites.
- SynQLite should support SEC
- Uphold local-first properties; a user should be able to use the database when not connected to the internet.
- User should not need to download additional applications to use the database.
- User should be able to opt-in and out of automatic synchronization.

- User should be able to discrete push and pull when he/she wants.

We will evaluate how well we achieved the goals in the upcoming sections. The evaluation process will use the experiments and analysis of the design.

9.1 Major goals

SynQLite's current implementation supports automatic synchronization with more than two sites through TCP connections. We solved this by assigning a site as a leader, which start-up an API. The leader has multiple threads handling requests from multiple sites. All of the threads perform operations on the same database. Having multiple threads operating on the same database concurrently introduced edge cases that were hard to locate and reproduce. We spent much time locating problems and recreating them through tests since the behavior of the threads is not deterministic. An example of edge case is Figure 6.11.

If the leader is not online, the other sites will have to start finding a new leader or discrete push and pull toward each other to synchronize. We also recognize that the current implementation is a pseudo-centralized implementation, where the leader will become a bottleneck if there are too many members.

There are clear improvements that could be made, such as electing a new leader through an election algorithm such as the bully algorithm[32]. Then again, this requires some form of coordination between the sites, which was something we wanted to limit. We could also have chosen another network topology, for example, a ring topology where sites are required to propagate information at a specific interval. The information would be spread in a pulse-like fashion and require a longer time to converge. However, the bottleneck at the leader would disappear. Ring topology would be much more complicated, to cover all edge cases would require extensive testing, even more than we did in this thesis.

9.2 Supporting push

As mentioned in the contribution chapter [4] did not support pushing for synchronization. Pushing was the first thing that we implemented for the new solution. It was an excellent way of getting to know SynQLite. SynQLite's current implementation supports pushing through an SSH shell or TCP connection. Unfortunately, the SSH shell does not work if δ gets too big. The problem is that shell has a character limit, limiting the size of messages that can be sent. There is, however, an easy fix because pull does not have the same problem. Pull does not have the same problem because the shell does not have a character limit when returning. Meaning we could tell the site that is being pushed to, too perform a pull request from the pushing site. Pushing using the continuous method does not have the same problem since when pushing with TCP, it divides the data into smaller packages sent one by one. Furthermore, as seen in Figure 8.11, SSH pull has similar results as TCP, and there is no reason it would be different for pushing.

9.3 Local-first properties

Local-first software properties that we considered are the ability to use a service when not connected to the internet and later having the possibility to synchronize when the user wants to. We evaluate this by splitting the user experience into the offline experience and online experience.

9.3.1 Offline experience

SynQLite lets the user use their database as expected when offline. All data in the database is accessible and can be changed concurrently by each site. That does not mean that there is not a trade-off when considering performance.

As we can see in Section 8.3, there is a significant difference in performance when executing insert, update, and delete. Inserts with CRR support are around 18 times slower, while delete and update are around 12 times slower. The main reason for this is that these SQL operations fire triggers. The triggers include multiple queries; updating the clock, checking if the trigger is disabled, creating new entries, and updating entries in $\tilde{R}$.

Checking if the trigger is locked is done by performing a query to a "lock" table, which is a table with one entry that can only be true or false. The only other way to disable triggers is to drop them before merging and creating them when done with merging, which would increase the speed for offline use, but this would slow the merging process.

The part of the triggers that takes the longest time is searching through $\tilde{R}$ and HR to update the row with the same id in $\tilde{R}$, incrementing causal length for adding and deleting in $\tilde{R}$, updating attributes in $\tilde{R}$ when updating, and lastly updating the HR(t) and HR(site_id) in HR. The results show that the most significant discrepancy is when performing insert queries. Using SQLite explain query plan command[33], we can see that the database performs one more search through $\tilde{R}$ when performing insert. Insert trigger checks if a row with the same id exists in $\tilde{R}$ before inserting it to $\tilde{R}$.

Figure 8.8 show that the CRR database performs the same for select queries. We expected this since select queries do not interact with the CRR-layer.

Offline experience is essential for local-first properties. Furthermore, how fast the database is accessible is vital for the offline experience, so inserting, updating, and deleting taking 13 to 18 times more with CRR support is significant.

The main cost comes from the chains of triggers. One way to improve the offline experience is to drop the history table when working offline and then generate it when going online. However, this would slow down the synchronization process because the history table must be generated before creating δ.

Many have done research considering database optimization[34]. Some optimizations could fit this system; for example, we could have stored commonly accessed tables in memory. Storing $\tilde{R}$ and HR in memory would significantly increase the speed of insert, update, and delete queries.

9.3.2 Online experience

When considering the online experience, we think about how well the two different synchronization methods worked.

Synchronization

The synchronization process is a big part of the online experience. In Figure 8.10 we compared generating, filling temp database, and merging δ with the local database. Looking at Figure 8.10, we can see that what takes the longest time is merging δ to the database. This is not surprising since it involved many more steps. The main takeaway from these results is that none of these processes takes exponentially more time when merging more rows, and the merging process has the most significant potential for improvement.

Discrete vs continious

We compared the discrete and the continuous model to see if one is significantly better than the other in certain use cases. The first thing we compared was pulling a large set of data from a remote site. Figure 8.11 shows that they give similar results. The only difference is that the error bar shows that the discrete model was less consistent.

This result was surprising, the expected result was that the discrete model would work better for smaller packages, and the continuous model would be faster with larger packages. Because when using the continuous model, the one pulling needs to establish a TCP connection through a three-way handshake[35]. We thought that the time to establish a connection would be significant for smaller packages and less significant when the packages got larger. The result, however, shows that the overhead from both models is similar. When using the continuous model, the sender must do a couple more steps since the receiver must know the size of δ before receiving when using TCP. When considering the goals, this shows that SynQLite works fine for both the discrete and the continuous model. This means the user can decide how he/she wants to synchronize without suffering performance issues, which supports local-first properties.

Second, we compared them by pulling small datasets many times. This is to see which model worked best for applications that want to synchronize often. This is to evaluate SynQLite as a service to help with collaborative works, applications like google docs, and Trello. Figure 8.12 shows that the continuous model is around 1.5 to 2 times faster than the discrete model, which was what we expected. The reason the continuous model performs better in Figure 8.12 compared to Figure 8.11 is that the sites do not need to establish the TCP connection between each pull in Figure 8.12. This shows that sending data through TCP connections is faster than SSH shells' responses.

Figure 8.13 shows that the performance of the continuous model scales linearly with the number of members connected until we exceed ten sites, which is not surprising since the computer that hosted the leader uses a CPU with ten cores. The centralized model's performance depends on how many CPU cores the leader has.

Availability

Another thing that could be tested is the availability of the database when synchronizing. This experiment would include connecting a site to a leader for continuous synchronization and then taking the time to perform a set of SQL operations. Then we could compare the results to the experiment conducted in Subsection 8.3.1.

When a site merges δ, the application or user cannot access the database. The availability of the database would definitively be affected because there can only be one connection to an SQLite database. The availability is definitively a part of the online experience.

9.4 Strong eventual consistency

Applications or users that use SynQLite will have SEC. This is because SynQLite is based on CRDT, which means sites do not require coordination between each other to be consistent. SynQLite uses CL-set to decide if an element should exist or not and LWW to decide which value each attribute in a row should be.

9.5 SQLite

Before, SQLite was the obvious choice for a CRR system because SQLite is an embedded lightweight DBMS, making it easy to clone and set up. However, one of the main limitations of SynQLite is that SQLite is not designed for concurrent use. SQLite locks all tables when someone performs a write transaction. Earlier implementations of SynQLite focused on peer-to-peer connections between two sites, where both sites have one connection to their database. This thesis explored more than two sites connected to a centralized site, which required multiple threads to access the database. We handled the concurrency issue by only letting one thread perform a pull or push request simultaneously, using locks. Other SQL DBMS might perform better when considering merging and performing operations on the database simultaneously as it is synchronizing.

WAL mode is worth considering in the future; it would allow applications to query the database simultaneously as SynQLite synchronizes. WAL mode could allow sites to pull from a leader simultaneously as another site pushes. An issue with this is that S(out) must be updated when a site pulls, which means the leader would have to lock the database when updating it. Figure 8.14 shows that WAL mode reduces the insertion performance by a significant amount. The database has to write to multiple files using WAL instead of only one.

9.6 Combining different DBMS

When we look at the centralized solution, we find that the main problem is that SQLite does not work well with concurrency. However, PostgreSQL works as a client-server DBMS, which can handle multiple requests concurrently. One idea is to utilize both DBMS since we want to use SQLite since it is easy to set up and clone. A solution could be to use both where the leader is a PostgreSQL DBMS, and the members are SQLite databases, where the leader and the members have the same tables and triggers. This could work since both databases are SQL-based, which means data could be sent directly from one database to another without much editing.

PostgresSQL uses a Multiversion Concurrency Control or MVCC for concurrent data access[27]. This means concurrent SQL statements return data from a snapshot of the underlying data, regardless of the current underlying data. PostgresSQL has different transaction isolation levels:

1. Read committed: under a transaction, it will be able to read data committed before the transaction and data change from the current transaction.

2. Repeatable read: under a transaction, allows for reads only on data committed before the transaction.

3. Serializable: emulates serializable transactions.

Read committed is the most interesting for SynQLite, allowing multiple sites to simultaneously push and pull to a site. It will most likely be faster than the current implementation. Another thing that might be interesting is that with PostgresSQL, users can lock specific tables and have a built-in mechanism to disable triggers. So let us say we have a database with multiple tables, and only one table is changed when merged. This table could be the only table that is locked.

PostgreSQL is DBMS made for client-server architecture, which is what the continuous model effectively is. Furthermore, since the leader is more static than other sites, it would not be so significant that the leader is more heavyweight than the members.

However, this would require using SQL operations accessible for PostgreSQL and SQLite. SQLite operations such as RETURNING could not be used. Alternatively, the developer must write specialized code for the member and the leader, doubling the work needed. Implementing this could be an interesting thesis, where they could conduct similar experiences to see if it handles concurrency better than this solution.

9.7 Lessons learned

The main lesson learned while working on this thesis is how to work using an agile development model. Me and my supervisor worked as a team to find new features and ideas that I could implement, bouncing ideas between each other to find out what was feasible and desirable within the timeframe.

One thing that should be mentioned is that working on distributed systems with multiple threads working on the same data is difficult. Creating a good testing environment is crucial. Using the testing environment we created made it easy to tweak clocks and permutations to recreate edge cases that could occur, but before we could create these tests, we had to find these edge cases. Locating edge cases was also a big part of the work done in this thesis because how can we find them if we do not know they are there? To cause edge cases to happen, we would set up a leader and connect multiple sites that would synchronize within short time intervals simultaneously as they were updating their database. Then after knowing that there were bugs, we would have to analyze the databases to find out what was wrong. For example, do the timestamps in HR, $\tilde{R}$, and S make sense? And then start drawing timelines to see what could be wrong. That is when having someone to bounce ideas with is very important. Working with distributed systems requires excellent testing and development environment.

10 Conclusion

SynQLite giving users the ability to synchronize SQLite databases with more than two sites continuously was the primary goal for this thesis. We split the goal into two parts: first continuous synchronization, and second more than two sites synchronizing simultaneously. SynQLite should also uphold local first properties, which entails that the users can use the database after augmenting as usual, even if not connected. Then at a later point, the user can synchronize with other sites as the user want.

We used TCP connections to achieve continuous synchronization. The constant connection between two sites granted better results for multiple push and pull requests than SSH shells. This is shown in Figure 8.12. We implemented a centralized solution for multiple sites synchronizing at the same time. We introduced a new role leader for the system, which other sites can connect to for synchronization. This meant the network topology was many peer-to-peer connections to the leader. To allow synchronizing with other sites than the leader SynQLite supports push and pull using another communication method, where sites send the data through an SSH shell.

We split local-first properties into the online and offline experience when evaluating the implementation. Offline experience is evaluated by how much SynQLite affects the performance of the AR layer in the database.

The experiments show that SynQLite slows down write operations after augmenting a database with CRR support. The main reason is that SynQLite adds triggers to the databases. These triggers fire when write operations are done to the AR layer, and the triggers perform queries and other operations to the CRR metatables. Slowing down the operations affects the offline experience of the user and goes against local first properties. However, triggers will always impact the performance required for CRDTs systems on a SQL database, but some improvements could be made. Such as adding $\tilde{R}$ and HR to memory since they are queried within the triggers.

When considering the online experience, we evaluated the speed of synchronizing sites. We wanted to compare the two different synchronization models, discrete and continuous. The result shows they are very similar when considering pulling large data sets one time. However, the continuous model performs better when the sites pull small sets many times. This is because the continuous model uses TCP connections, and when pulling multiple times, the sites do not need to establish new connections.

Multiple sites synchronizing at the same time introduced concurrency issues. We handled this issue by only allowing one thread at the leader to respond to pull or push requests at the time. This caused a linear performance drop when connecting more members. We discussed improvements to the centralized solution, either utilizing WAL journal mode or implementing the leader as a PostgreSQL database.

Bibliography

[1] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: You own your data, in spite of the cloud," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, (New York, NY, USA), p. 154–178, Association for Computing Machinery, 2019.

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems* (X. Défago, F. Petit, and V. Villain, eds.), (Berlin, Heidelberg), pp. 386–400, Springer Berlin Heidelberg, 2011.

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

[4] I. T. Tomter and W. Yu, "Augmenting sqlite for local-first software," in *New Trends in Database and Information Systems* (L. Bellatreche, M. Dumas, P. Karras, R. Matulevičius, A. Awad, M. Weidlich, M. Ivanović, and O. Hartig, eds.), (Cham), pp. 247–257, Springer International Publishing, 2021.

[5] M. Cristian, "Database replication," *Database Systems Journal*, vol. 1, p. 33–38, Feb 2010.

[6] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," Jun 2002.

[7] K. Schwaber, "Scrum development process," in *Business Object Design and Implementation* (J. Sutherland, C. Casanave, J. Miller, P. Patel, and G. Hollowell, eds.), (London), pp. 117–134, Springer London, 1997.

[8] "About trello." `https://trello.com/en/about`. [Online; accessed 10-March-2022].

[9] "Share files from google drive - computer - docs editors help," 2018.

[10] S. Burckhardt, "Principles of eventual consistency," Oct 2018.

[11] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[12] N. Preguiça, "Conflict-free replicated data types: An overview," June 2018.

[13] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," Research Report 7506, INRIA, Jan. 2011.

[14] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 162–173, Jan. 2018.

[15] "Sqlite home page." `https://sqlite.org/index.html`. [Online; accessed: 2021-24-10].

[16] P. Sanderson, *Chapter 4 SQLite rollback journals*, p. 103–130. Paul Sanderson, 2018.

[17] P. Sanderson, *Chapter 5 Write-Ahead Logs*, p. 131–161. Paul Sanderson, 2018.

[18] "Ssh tunnel." `https://www.ssh.com/academy/ssh/tunneling`. [Online; accessed: 2021-24-10].

[19] V. Enes, P. S. Almeida, C. Baquero, and J. a. Leitão, "Efficient synchronization of state-based CRDTs," in *35th IEEE International Conference on Data Engineering*, ICDE 2019, pp. 148–159, Apr. 2019.

[20] K. Jannes, B. Lagaisse, and W. Joosen, "OWebSync: Seamless synchronization of distributed web clients," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, pp. 2338–2351, Sept. 2021.

[21] D. Mealha, N. Preguiça, M. C. Gomes, and J. Leitão, "Data replication on the cloud/edge," *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data - PaPoC '19*, Mar 2019.

[22] T. Colburn, "Methodology of computer science," *The Blackwell Guide to the Philosophy of Computing and Information*, p. 318–326, 2004.

[23] P. Denning, D. Gries, D. Comer, M. Mulder, A. Tucker, A. Turner, and P. Young, "Computing as a discipline," *Computer*, vol. 22, p. 63–70, 1989.

[24] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and et al., "What bugs live in the cloud? a study of 3000+ issues in cloud systems," *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[25] "Sqlite3 - db-api 2.0 interface for sqlite databases¶." `"https://docs.python.org/3/library/sqlite3.html"`, Apr 2022.

[26] J. Forcier, "Welcome to paramiko!¶." `"https://www.paramiko.org/"`, 2022. [Online; accessed 1-April-2022].

[27] "Postgresql homepage." `"https://www.postgresql.org/"`, 2022. [Online; accessed 26-April-2022].

[28] "Mysql homepage." `"https://www.mysql.com/"`, 2022. [Online; accessed 26-April-2022].

[29] D. E. Comer and D. E. Comer, *TCP: Reliable Transport Service*, p. 309–319. Prentice Hall, 2 ed., 1999.

[30] H. Krekel, "Pytest: helps you write better programs." `"https://docs.pytest.org/"`, 2015. [Online; accessed 10-April-2022].

[31] "Docker homepage." `"https://www.docker.com/"`, Apr 2022. [Online; accessed 10-April-2022].

[32] G. Coulouris, A. K. Bhattacharjee, and S. Mukherjee, *Time and coordination*, p. 287–310. Addison-Wesley, 2 ed., 2012.

[33] "The explain query plan command." `https://sqlite.com/eqp.html#the_explain_query_plan_command`, 2021. [Online; accessed 15-April-2022].

[34] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "Query optimization through the looking glass, and what we found running the join order benchmark," *The VLDB Journal*, vol. 27, no. 5, p. 643–668, 2017.

[35] C. A. Sunshine and Y. K. Datal, "Connection management in transport protocols," *Computer Networks (1976)*, vol. 2, p. 454–473, Dec 1978.