UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Towards a General Database Management System of Conflict-Free Replicated Relations

Iver Toft Tomter

INF-3981 Master's Thesis in Computer Science - June 2021

Abstract

Local-first [1] databases have advantages relating to the CAP-theorem [2]. Moving away from a client-server architecture and into a local-first one where the server is within the local device allows for constant availability. By storing the data in Conflict-Free Replicated Datatypes, one can merge multiple local-first devices to achieve a consistent state as needed [3]. Local-first software allows a system with strong eventual consistency that is always available.

A general database of conflict-free replicated relations would allow any application that relies on a database to achieve local-first properties. Using only SQLite [4], we accomplish this, as SQLite is a popular database system applied in millions of devices. Generality makes this project differ from the previous CRR[5], which uses a relational mapper [6].

We propose a two-layer database design using Causal Lenght-set [7] for rows in the relations and Last Write Wins-registers [3] for attributes in the rows. Achieving this means implementing methods to propagate data between layers, storing necessary metadata that automatically upholds CRDT functionality. In addition, every LWW-register attribute needs an accurate timestamp. Since SQLite only offers millisecond time resolution, we emulate nanosecond time accuracy using randomized values. SQLite does not support communication, so the system must partially rely on external SSH [8] functionality to make all relations appear local.

Merges are performed entirely in SQLite. Relations across the system merge using a left-join operator with the foreign table as the left one, allowing access to all rows that are not in the local table and all rows in both tables. CL-set functionality decides the state of each row, and LWW-registers determine the values of each attribute. In merges, the SQL script resolves reference integrity constraints.

We base experiments on data and client-centric consistency [9] and availability as latency [10]. The experiments highlight the strong eventual consistency across two merge methods. Full merges are when the entire system merges periodically, which grants the system complete consistency on demand. Partial merges simulate that even though nodes are not always available, the system can uphold a consistency threshold. Availability is a trade-off compared to a local datastore when measured as transaction latency. However, the system's scalability outperforms other fairer comparisons like Paxos [11] and Raft [12].

This thesis aims to create a local-first database of conflict-free replicated whose principles are generally applicable to any SQLite database.

Preface

A thing to note, is that I choose to use *tables* and *relations* somewhat interchangeably during the project. As a rule of thumb, I try to use *relations* when writing about conflict-free replicated relations and databases in general terms and *tables* when talking about specific SQL tables. This project ties the two together so the usage of *table* versus *relation* become more obtuse as the thesis goes on. I tested changing all instances of *table* to *relation*, but deemed it to provide a less clear reader experience.

Acknowledgements

I want to acknowledge and thank my thesis advisor Weihai Yu. He has helped and guided me throughout this project. Thanks to him, I have found myself in a situation where I could work on an exciting thesis. Even towards the last couple of weeks of this project, I am still passionate and continue to try new and interesting solutions that have come up in discussion with my advisor.

I want to thank my family for supporting me and Luna, who has helped inspire and motivate me. Without their support, this thesis would not have been possible. Knowing that they had my back and that I always had somewhere to look for motivation helped me tremendously during this long journey.

I want to also issue special thanks to people who helped look over and proofread my thesis: William, Iselin, and Jørn. Finally, I thank the student association Imladris and all its members for good company and coffee.

Contents

List of Figures

1 Introduction

1.1 Motivation

With the constant and ever-increasing push towards cloud and Internet of Things solutions in computer science, new and never-before-seen issues arise. As the diversity of architecture grows from one to multiple computers, so does the complexity of data management. The CAP-theorem [2] states that one can achieve two out of three when it comes to Consistency, Availability, and tolerating Partitioning in a distributed system. When developing a system for data storage, the question is often one of which of the three to prioritize.

Databases have become synonymous with data storage in online environments, and in turn, SQL has arisen as the standard of easy relational data management. Databases commonly rely on a client-server architecture, where the client issues requests to a central server. One quickly sees the problem of such an architecture, any device depending on the database's data also relies on connection to an external server. As an example, we look at two architectures, a single server and a replicated server one. For the single server, availability relies on the server being active, and data is unavailable when the server is down. As a solution, one might employ multiple replicated servers; as one is down, another one picks up. Ignoring the new consistency requirements that arise with multiple servers, one might still say that the availability is imperfect. Availability still relies on the client having access to the servers. When network connection breaks, for example through loss of internet, the data becomes unavailable.

The solution is local-first [1] architecture where edge devices are both client and server. The data is moved from an external server store to a local one, which gives constant availability. However, one sacrifices consistency, as the local data might not match that of the server at all times. The issues are now effectively and consistently merging the data from the local to the external store to ensure eventual consistency.

Conflict-Free Replicated Data Types (CRDT) [3] are structures of data in which multiple replicated instances can be combined into a single consistent value. In the case of local-first, one client might have one value while the server has another. On a consistency conflict like this, CRDTs must ensure that both parts update their internal datastore to the matching values. CRDTs are designed [3] and verified [13] as eventually consistent as after a hiatus from communication, the data converges on a single consistent value. Applying CRDTs to database relations result in a database of conflict-free replicated relations (CRR), as done in [5].

Local-First Software and CRDTs go hand-in-hand [14], and these principles applied to database technology exist [5] and lay the fundamentals for this research. However, the simplicity offered by a simple SQL-server architecture as in the previous examples is not one found in current solutions. They are either no-SQL [15] or require a specific intermediary between application and database using Object Relation Mappers (ORMs) [5]. Limitations like these identify a need for a general relational SQL database system that can be applied easily to any database application, even an existing one, without the bloat of extra software layers (ORM) or dependencies.

Data ownership [16] is also an arising issue as cloud storage becomes common [17]. Data ownership refers to how and where data is stored and the level of trust between storage and client. Continuing with the two client-server examples, let us suppose that the client's data is sensitive, personal, or otherwise uncompromisable when it comes to trust. In the client-server example, one quickly sees ownership issues, as the client's data is stored and accessible by a server to which the client has limited insight. But in the Local-First approach, the client has their own replica of the data. Furthermore, the client might decide they want to delete the data. For the first example, the client has no choice but to trust that the server deletes the data. For the second example, the client knows what specific data it needs to delete and can more easily hold the server accountable for failure to deliver their deletion request.

1.2 Challenges

Some databases use CRDTs for consistency, but one's tools are severely limited when looking to implement a general solution. A general relational DBMS will need to rely on SQL, as it has been the most common programming language across relational database systems for years [18]. SQL is by no means a complete programming language, as it is only meant for interactions with data and structures in databases. And while dialects such as SQLite [4] are built on the C programming language [19] with the possibility of extensions, there are still limiting factors imposed when extending existing code. Therefore, the feasibility of possible solutions is an issue in its own right, and how we work around the severe limiting factors.

We recognize the following issues relating to a general SQL database manager supporting CRDTs, ranging from highest to lowest priority. Goals in the following sections will be more general, while these are potential issues with trying to reach the goal:

- **Pure-SQL:** Ensuring that the database manager is usable in any system that supports SQL by using only SQL to implement conflict-free replicated relations.
- **Timestamps:** Consistency relies on accurate timestamps, will SQL provide accurate enough time?
- **SQL Functionality Limitations:** Is it possible to implement without the functionality of a general programming language?
- **ACID**: Can CRDTs ensure Atomic, Consistent, Isolated, and Durable transactions?
- **Embeddedness [20]**: It should be lightweight enough to run as part of other applications. Just as SQLite [21].

1.3 Goals

- **Research Issue:** Is it possible to create a general Database Management System of Conflict-Free Replicated Relations?

Arising from the motivation of building such a system, we further divide the main issue into the following goals and requirements:

- **CAP Properties:** The system should have strong eventual consistency from CRDTs, constant availability from having a locally replicated database while being replicated across multiple computers.

- **Generality/General Applicability:** The easiest way to ensure that a system is general is that it relies only on SQL. Augmentations can be applied to any SQL application, new or preexisting, that upgrades it to a CRDT-DB without impeding the current functionality.

- **Performance:** While not a primary goal for now, different performance metrics will be measured to make the current state of implementation comparable to other database management systems.

1.4 Methodology

Commonly in computer science, one applies theory as a method to prove a concept [22]. In the case of CRDT-powered local-first databases, this has been done already [5]. However, as this project's motivation relies only on SQL and not other programming languages, the bridge between mathematical theory and program is more obtuse. SQL is, after all, a query language with no variables, definable functions, or objects. Therefore, the previous concepts may not apply. We work our way from the initial underlying math out to the programming language SQL again, taking into account the limitations of SQL compared to general purpose programming languages. The limitations must be overcome to prove the concept, and new solutions applied.

There is a clear focus to the project: we will apply experiments to test that the implementation of the mathematical concepts is correct and highlight the strengths and limitations of CRDTs. As there is uncertainty when using known concepts, in this case CRR [5], in a new way, one must ensure that experiments still provide the same results as assurance. While there is no system like this one, performance compared to other similar systems might give an idea for use cases, as some applications might value certain advantages. This might also show if the application is even feasible as a usable system at the current time and give a picture of further improvements.

1.5 Use-Case: Local-First and Generality

A specific use-case that highlights the strengths and needs for a local-first database is an environmental scientist's research application that relies on a database. In this example, the researcher's application relies on data from multiple arctic ships and other remote expeditions. With the data-gathering agents being far from civilization, their network connectivity is unreliable. However, a local-first database would allow each ship to store data locally and synchronize with a central server or other ships once they reach port and connect to a network.

Generality in the scope of the researcher's application would mean that the researcher can treat the database like any other database or even utilize a user application developed without local-first in mind. Each ship interacts with the database using SQL as it would any other database. However, beneath the usual SQL instructions, the underlying database ensures that the data can be easily synchronized with other replicas. The database functions without the application needing to adjust; as far as the application is concerned, it is a standard relational database.

2 Technical Background

2.1 Local-First Software

Local-first software moves computing and storage away from the cloud layer into the user device. As seen in **figure 2.1**, a local-first database updates its internal database first and merges it with an external one later, in this case, a server database. Applications like this one would be well-suited in:

- Environments where network connectivity is limited, such as small devices in an unstable network.
- More significant devices running in places without constant connection, like underground or off-shore.
- Devices trying to minimize bandwidth usage through packaging communication into a larger merge instead of multiple small requests, all while continuing operations.
- Internet of Things, small edge devices that might be periodically disconnected from the network, can regain consistency by merging with other edge nodes as they see fit without halting the network with their absence or needing it to operate.

Local-First design prioritizes availability over constant consistency. There is no way of knowing the exact state of other replicas without continual communi-

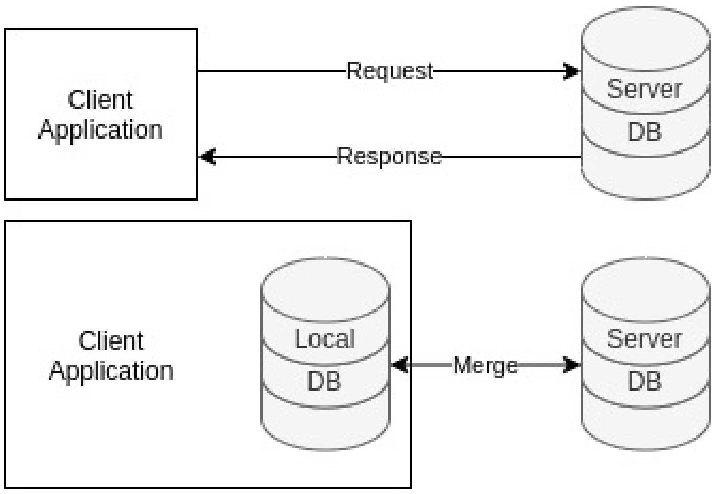

Figure 2.1: Comparison of a standard server-client architecture and local-first

cation. Instead, local-first software focus on strong eventual consistency [1], often through the use of CRDTs.

2.2 Relational Database Model

This project explores both relational databases as well as the underlying mathematical theory. We distinguish between mathematical tuples, hereby referred to as tuples, and relational database tuples hereby referred to as rows.

In mathematics, set relations refer to a relationship between two sets where each element in a set of tuples contains a tuple with the element of another set [23]. Building on this, relational databases have relations/tables instead of sets. Like having another set's elements in one's tuples, a row in a relation refers to other rows in other relations through a foreign key. The Relational Database Model can result in hierarchies and structures of relations built on rows referring to other rows in other relations.

Relational databases remain an industry standard, even with the rise of non-relational databases. As seen in the popular repository service GitHub, the SQL topic has 16,654 repositories, versus the 1,688 repositories with the NoSQL topic [24]. The Relational Database Model is general and widespread, and SQL is the most popular way to interact with relational databases.

2.3 Conflict-Free Replicated Data Types

This section focuses on different types of CRDTs. As seen in **figure 2.2**, there is a branching relationship between the different types of CRDTs. Applying the CRDTs to databases, as done in CRR, requires an understanding of multiple CRDTs, as some properties differ between them, so do the use cases of the different ones.

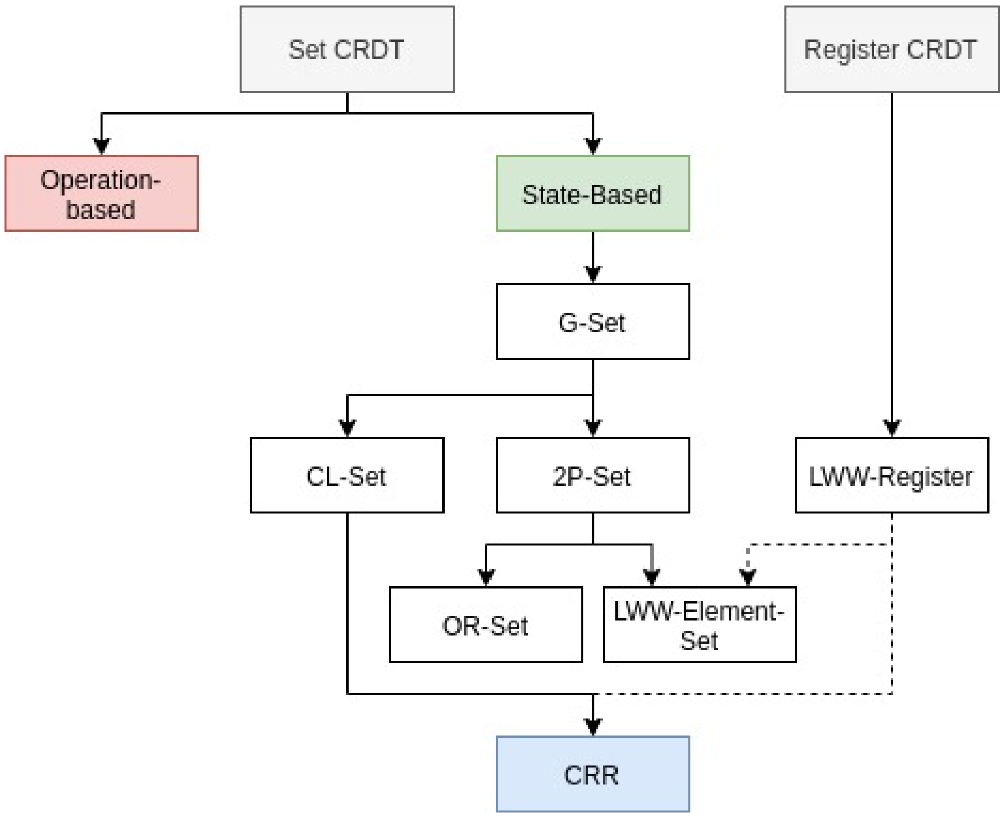

Figure 2.2: Relationship between different types of CRDTs

Set CRDT refers to the underlying data type being a set and differs from a counter CRDT, where the underlying data type is an integer-like counter that can be updated in ways that the CRDT structure allows. CRDTs as a whole are defined as data structures that can decide the state without the use of voting and consensus algorithms such as Paxos [11], and Raft [12], all while maintaining strong eventual consistency.

2.3.1 State-Based versus Operation-Based CRDTs

There are two main branches of CRDTs, state-based and operation-based [3]. For this project, state-based CRDTs are the main focus because they apply to CRR. Operation-based is better suited for distributed networks with a strong multicast guarantee. For this section, counter CRDTs will be used for examples, as they more simply highlight the differences between state-based and operation-based CRDTs.

Operation-based CRDTs work by broadcasting any operation to all other nodes in the network [25]. Operation-based CRDTs have some challenges, as the updates shared to other nodes are potentially unordered. Unordered operations mean operations have to be commutative, so applying them in any order does not change the finalized result. Given a system of multiple nodes all performing different concurrent updates, each node must broadcast its update to the other nodes and perform foreign updates given by other nodes. Operation-based CRDTs must result in an equal state after applying all the updates on any node in the system locally in any order. ACID transactions are needed, as the system must ensure to apply each update at least once and no more than once.

State-based CRDTs only apply changes to their local state. At any point in time, the system's total state should be attainable by combining all local states. For example, for m nodes n with local state n(s) and a merge function f there is an implied total state $total(s) = f(n_1(s), n_2(s), ...n_m(s))$. The total system state is attainable by merging the local states, where any states can be combined through a join-semilattice [26], where the underlying CRDT defines the join method, referred to as merge. All this results in an eventually consistent system, where there is always a consistent state as long as one can merge with the other nodes to apply it locally.

While issues with operation-based CRDTs are related to communication, state-based CRDTs have no such issues, as merging two states more than once should cause no changes when applied more than once without local updates in-between. State-based CRDTs instead need a more complex underlying data structure to ensure that the system is syncable.

We explore State-based CRDTs through five different set CRDTs: Grow-Only Set, Two-Phase Set, Last Write Wins -Set, Observe-Remove Set as they appear in [3], and Causal-Length Set [7]. As seen in **figure 2.2**, these data structures commonly borrow from each other, and one can paint a clear line from one CRDT to the next. Finally, one sees how combining the principles applies to a relational database like in CRR [5].

2.3.2 Grow-Only Set: G-Set

G-Set is the most simplistic form of state-based set CRDT, and as the name suggests, it only grows. No elements can be removed from the set once added. The effect of this design is that no consistency conflict can occur when there is no uncertainty of whether a set element has been deleted or was just never added.

Listing 2.1: G-Set

```
set A := Ø

G.has (element e) : boolean b
    $b = (e \in G.A)$
G.add (element e)
    $G.A := G.A \cup \{e\}$
G.merge (S)
    $G.A = G.A \cup S.A$
```

To see whether an element exists a G-set, as seen in **Listing 2.1**, is done by checking whether the element is in the G-set. Adding new elements is done by creating a subset containing the element and adding it to the G-set through a union operation. A G-set merges by performing a union of two G-sets.

G-set has no remove operation, as simply removing an element from the set would cause consistency issues on a merge. For elements only existing in one set before a merge, there would be no way to tell if it existed in both sets and one of them removed it or existed in no sets and one of them added it.

G-sets simplicity ensures that it is conflict-free. There can be no conflicts when only adding elements to a set since there is no question why the element is there. However, only adding elements also means that that G-set has limited use-cases. As removing elements is often essential functionality for a set to have. Other set-CRDTs aims to include this functionality in different ways.

2.3.3 Two-Phase Set: 2P-Set

To upgrade the functionality of G-Set, 2P-Set looks to add the ability to remove elements. It accomplishes this by combining two G-Sets. A 2P-set is therefore not a single set but an abstraction of two sets: the added-set and the removed-set.

Listing 2.2: 2P-Set

```
 set A := Ø
 set R := Ø

2P.has (element e) : boolean b
    $b = (e \in 2P.A \land e \notin 2P.R)$
2P.add (element e)
    $2P.A := 2P.A \cup \{e\}$
2P.remove (element e)
    if has(e)
        $2P.R := 2P.R \cup \{e\}$
2P.merge (S)
    $2P.A := 2P.A \cup S.A$
    $2P.R := 2P.A \cup S.R$
```

Elements can be both added and removed from a 2P-Set, as seen in **listing 2.2**. An add operation adds the element to the added-set, while a remove operation firstly ensures that the element exists in the set and then adds it to the removed-set.

Looking up elements is done by checking that the element exists in the added-set and not the removed-set. A removed element will exist in both sets, and an existing element will only exist in the added-set.

A 2P-set merges by taking the union of both the added and removed-sets from two 2P-sets. As the two sets within a 2P-set are grow-only, there are no conflicts on a merge.

The main limitation of a 2P-Set is that it can not add back elements once removed. This limitation is evident when one views a 2P-Set as a fusion of two G-Sets, allowing for the removal of elements from one of its either the adder or removed -sets would cause the same consistency issues seen with a G-Set. LWW-set, OR-set, and CL-set aim to combat this issue in three different ways.

2.3.4 Last Write Wins Element Set: LWW-Element-Set

Like a 2P-set, an LWW-Set is also an abstraction of two subsets, an added-set and a removed-set. It further enhances the abilities of a 2P-set by tying a timestamp when adding to either of its two subsets.

Listing 2.3: LWW-Element-set

```
 set A := ∅
 set R := ∅

L.has (element e) : boolean b
    $b = \exists t, \forall t' > t : (e, t) \in L.A \wedge (e, t') \notin L.R$
L.add (element e)
    $t = now()$
    $L.A := L.A \cup \{(e, t)\}$
L.remove (element e)
    $t = now()$
    if has(e)
        $L.R := L.R \cup \{(e, t)\}$
L.merge (S)
    $L.A := L.A \cup S.A$
    $L.R := L.A \cup S.R$
```

Elements in both sets within an LWW-element-set are 2-tuples. These tuples contain an element and a timestamp. The timestamp is the current time given when an element is added or removed from the set.

The has-operation, as seen in **listing 2.3**, of an LWW-element-set work by checking a given element exists in the added-set. If it does, the set also has to make sure that the element in the added-set has the most up-to-date timestamp tied to it. If a newer timestamp is paired with an element in the removed-set, the element does not exist. In short, the element exists if it is in the added-set, and there exists no newer version in the removed-set.

Aside from the has-operation, this set is similar to 2P-set in most regards. Even with the simplicity of the changes, brand new issues become apparent. Firstly, relying on timers compromises consistency. Now, the sets are only as accurate as the underlying clock hardware. When it comes to simultaneous updates across nodes, timer drift might decide whether an element exists or not [27]. Secondly, the set does not delete old values containing an element. It just adds new ones, resulting in ghost elements bloating the sets. A more elegant solution would be to remove obsolete elements.

2.3.5 Observe Remove Set: OR-Set

As a response to the issue of each operation increasing the space needed on disk or in memory, the OR-Set limits growth through the removal of obsolete elements. While this project does not use OR-set, it is the closest in functionality to the CRDT applied, CL-set, and therefore a good point of comparison. Another point of the OR-set is that it functions without timers, though there are even more advanced versions of the OR-set that use timestamps to limit growth even further [28].

Listing 2.4: OR-Set

```
 set A := Ø
 set R := Ø

OR.has (element e) : boolean b
    $b = (\exists n : (e, n) \in OR.A)$
OR.add (element e)
    $n = unique()$
    $OR.A := OR.A \cup \{(e, n)\} \setminus OR.R$
OR.remove (element e)
    $temp := \{(e, n) | \exists n : (e, n) \in OR.A\}$
    $OR.R := OR.R \cup \ temp$
    $OR.A := OR.A \setminus temp$
OR.merge (S)
    $OR.A := (S.A \setminus OR.R \cup OR.A \setminus S.R)$
    $OR.R := S.R \cup OR.R$
```

Similar to LWW-set, each element is a pair. However, this time the tuple contains an element and a unique identifier. OR-set works by shuffling 2-tuples around between its sets. Adding an element generates a new unique identifier, adds the pair, and then removes all elements in the removed-set from the added-set.

For removals, the OR-set moves pairs containing the element from the added-set to the removed-set in three steps. Firstly, it generates a subset of all pairs containing the element in the added-set. Secondly, it adds the subset to the removed-set. Finally, it removes the subset from the added-set, effectively removing all instances of the element from the added-set.

Furthermore, a merge of two instances of OR-sets is done by taking each added-set and removing the opposing removed-set, and then merging the two added-sets. The OR-set joins removed-sets as usual. Merging OR-sets results in a system where gaining an element through a merge, and then deleting it locally, a subsequent merge will result in the node that initially added the element to also delete it, as the set ties each element to a unique identifier.

One can identify that OR-set is conflict-free through how it handles deletions. Moving all current instances of an element-id pair into the removed-set ensures that these values are deleted from any other set on a merge. However, suppose another set has added a new instance of the element. In that case, this is unaffected as it generates a new unique identifier not matching that of any of the removed elements.

2.3.6 Causal-Length Set: CL-set

Vastly differing from LWW-set and OR-set, CL-set forgoes the use of the two sets architecture in favor of a single set containing all the needed information for the same functionality as an OR-set without the use of timestamps like the LWW-set. Slightly improving upon the OR-sets storage-saving method of removing elements once re-added, this method aims never to generate a new instance of an element, merely change its state.

Listing 2.5: CL-Set

```
set A := $\emptyset$


CL.has (element e) : boolean b
    $b := (\exists cl : (e, cl) \in A \wedge odd(cl))$
CL.add (element e)
    if $(\exists cl : (e, cl) \in A \wedge even(cl)$
        $CL.A := CL.A \cup \{(e, cl+1)\} \setminus \{(e, cl)\}$
    else if $\nexists cl : (e, cl) \in CL.A$
        $CL.A := CL.A \cup \{(e, 1)\}$
CL.remove (element e)
    if has(e)
        $CL.A := CL.A \cup \{(e, cl+1)\} \setminus \{(e, cl)\}$
CL.merge (S)
    $CL.A := \{(e, cl) | (\exists S.A \wedge \nexists CL.A) \vee (\exists CL.A \wedge \nexists S.A) \vee$
      $((\exists cl' : (e, cl) \in S.A), (\exists cl'' : (e, cl) \in CL.A) | max(cl', cl''))\}$
```

Each element in a CL-set consists of a pair of an element and causal length. Causal length is a representation of whether the element exists or not. Whenever causal length is odd, the element exists. A deleted element has even causal length.

Updates like adding or deleting elements are now as simple as incrementing the causal length. There has to be an extra clause to ensure that causal length is even or non-existent before adding and odd before deleting, as not to accidentally re-adding or deleting the element by incrementing CL when not intended. If an element is non-existent on insertion a new one is generated.

A merge operation of two CL-sets adds both elements existing in one set but not the other, whether the CL is odd or even. However, if an element exists in both sets, the one with the greatest CL is the winner. A CL-set assures that if it deletes a value, all other replicas it merges with delete the value unless any other instance has both deleted and re-added it, which would result in a greater CL than just a deletion.

Overall, CL has the advantages of limiting disk and memory usage without the need for timestamps, all with a simple single-set architecture. The simplicity and efficiency made it the ideal candidate for a set to be employed for CRR [5]. Others would require more extensive metadata and multiple abstracted sets or relations, thereby more complexity and potentially more disk usage.

2.3.7 Last Write Wins Register: LWW-register

It is easy to see how rows in a relational database share similar characteristics to sets. Similarities allow for set-CRDTs to be applied to rows and determine their state, either existing or not, across a distributed system. However, there is also the issue of the data within the rows. Data within rows has established types and quantity and is not similar to sets. Inter-row data does usually not have two possible states, existing or non-existing. Instead, the data within rows have as many states as the number of different values that the underlying datatype supports. The data is altered whenever the row attribute is updated.

[3] defines a register as a datatype that can be queried or updated. Where updates change its underlying value and queries return the value of the given register. Registers have similar properties to the data in the columns of relations. A state-based register CRDT applied to relations allows multiple instances of a row to be merged into one.

Listing 2.6: LWW-Register

```
register L (val x, timestamp t)

L.query : val r
    $r = L.x$
L.update (val w)
    $x, t := w, now()$
L.merge (R)
    if $R.t \leq S.t$
        $L.v, L.t := R.v, R.t$
    else
        $R.v, R.t := L.v, L.t$
```

As seen in **Listing 2.6**, whenever a value is updated, a new timestamp is generated. Queries return the value. Merging two registers updates the value and timestamp of the register with the lowest timestamp to match that of the highest timestamp register.

LWW-registers allow any value existing across a distributed network to reach a unified state once merged fully with the entire network. Applying this to data within rows in a relational database allows for conflicts on which shared value within a shared row is the most up-to-date one to resolve. This is done in CRR [5].

2.4 Delta State CRDTs

Delta-State CRDTs are not a type of CRDTs, but a principle applicable to state-based set CRDTs [29]. To minimize the bandwidth necessary for merges, it limits message size by generating a subset of the set containing only the values changed since the last merge. Limiting scope by merging fewer rows means that the merge is potentially faster. However, this is offset by additional complexity in terms of generating the state deltas [30].

Delta mutations are the transformative functions applied to a set or other data structure that generate the deltas. CRR [5] does this progressively as it adds new rows. Thereby minimizing the time needed to generate deltas.

2.5 Conflict-Free Replicated Relations: CRR

First, for the rest of the thesis, an important distinction to make is the difference between CRR as a principle and CRR[5] as a previous database software implementation. CRR defines the principles of applying CRDTs to database relations. CRR [5] is a database application that uses CRR.

CRR [5] applies CRDTs to database relations. As the principal inspiration and groundwork for this project, it applies local-first principles to databases through the use of CL-set. The original work [5] uses an Object Relation Mapper (ORM) for the application, thereby not being generally applicable to any application or system.

CRR [5] uses a two-layer architecture, the Application Relation(AR) layer and the Conflict-Free Replicated Relation (CRR) layer. The AR layer is just normal relations in a database, as they appear in any other database application. The CRR layer reflects the AR layer with CL-set applied to each row and LWW-register applied to each attribute. Each row in the table becomes like an element in a CL-set with a Causal Length defining its state. The AR layer acts as a cache of the CRR layer.

Unlike sets, relations are not static elements, but each row functions as a changeable element, as data within the rows can be altered. To handle this, CRR adds timestamps of the latest updates to each element in the row, as done in an LWW-register, and a Universally Unique Identifier (UUID) [31] for each row. For example, a row of *(x, y, z)* in the AR layer is represented as *(UUID, x, tx, y, ty, z, tz, cl)* in the CRR layer. By adding LWW-register functionality, the state of rows is no longer limited to existing and non-existing; the state can also change by updating attributes. For merging, the database applies the attribute with the greatest timestamp to the state of each merging replica on matching rows.

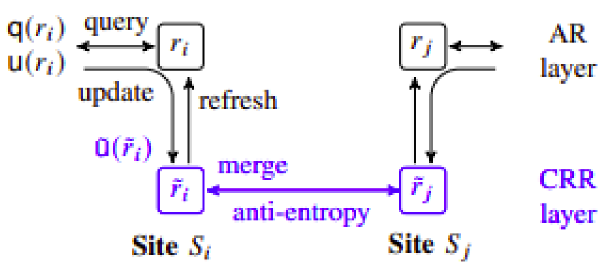

Figure 2.3: Relationship of CRRs AR and CRR layer [5]

CRR [5] defines functions to alter the database through an ORM which in turn executes SQL statements. The primary methods are insertions, deletions, updates, and queries to retrieve data.

Deletions and insertions of rows in the CRR layer function as deletions and additions of CL-set elements. That leaves the AR layer, which must reflect the state of the CRR layer in the form of a typical database table. When deleting a row, its CL, like in CL-set, increments to an even value, then a normal row-deletion is done of the corresponding row in the AR layer relation.

CRR [5] handles updates a little differently. Updates to the CRR layer generate a timestamp for each to-be-updated attribute. The program propagates updates to the AR layer, where the values are updated to match the corresponding values in the CRR layer.

Because of the two-layer architecture, the AR layer can handle queries, as seen in **figure 2.3**. At the same time, insertions, deletions, updates and merges are done on the CRR layer and propagated to the AR layer. Selecting rows are as simple as querying the AR layer.

The application performs database transactions locally. It adds affected rows to a queue forming a delta state for merging two CRR databases. The application handles merges as a CL-set, with the highest CL being the winning row applied to both merging databases. However, each row attribute is adjusted to be that of the attribute with the latest corresponding timestamp. Finally, both merging database instances propagate the changes from their now matching CRR layers to their AR layers for consistency across both layers.

CRR [5] employs hybrid-logic-physical [32] clocks to counteract timer drift between replicas. Whenever two replicas merge, they move the internal clock of the slowest one forward to match the clock of the faster one. This method does not ensure absolute ordering of operations like a vector clock [33] because that would have to rely on constant communication. However, it is still more synchronized than the internal computer clocks in their own right, resulting from communication counteracting desynchronization between replicas.

Applying CRR [5] to the earlier example of a Local-First client-server in **figure 2.1** gives insight into the internals of such a system, as seen in **figure 2.4**.

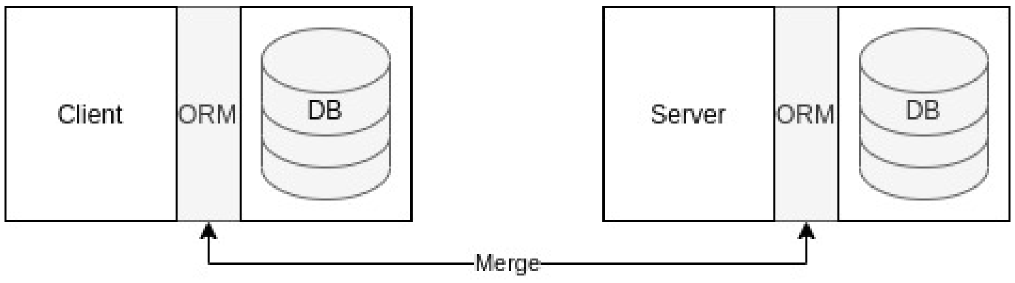

Figure 2.4: How CRR [5] uses ORM to access database and communicate

/3 Design and Implementation

3.1 Design Challenges: Generality with Pure-SQL Design

Before diving into the intricacies of the design and implementation, it is essential to reiterate the functionality that separates this project from previous work. We revisit generality and how it appears in this project's scope: generality is the ability to add functionality to something without introducing any new limiting factors to applications—for example, applying CRR to a database system without interfering with the user application. Aiming to make a general CRDT-based database manager, one has to make a method that, when applied to any SQL database, augments it to a CRDT-based database.

This project aims to achieve it in the following way: as far as possible, use nothing except SQL. New challenges become apparent, as SQL is not a complete programming language but instead a query language designed only to manage data in a database. Achieving generality is now a question of whether one can bend SQL's limited functionality to mimic what CRR [5] performs through the ORM Ecto [6] powered by Elixir, a complete programming language with extensive functionality [34]. This project aims to replicate the functionality of CRR [5] without the use of an ORM.

3.2 Supporting Technology: SQLite

SQLite is the clear candidate for the local-first databases, because it is a lightweight SQL manager designed to run locally. Other SQL systems like MySQL [35] and PostgreSQL [36] might have more functionality, like the ability to design functions [37] and more advanced locks for managing multiple DB instances. However, SQLite's barebones design means less bloat in terms of features and allows for our database to be an embedded one [21]. SQLite comes as a default software in any Android or iOS smartphone and is available on millions of devices. According to the SQLite homepage, they are the "Most Widely Deployed and Used Database Engine. " [38].

Other databases are more focused on being run as a singular server with client-server architecture, meaning they are more suited for a hierarchical distributed network than a peer-to-peer one [39]. While not having the most functionality, the advantages leave SQLite the ideal candidate for a general CRDT-based database manager. We chose a candidate by observing its advantages and disadvantages; it now becomes possible to design around its flaws and play to its strengths.

Advantages	Disadvantages
No Client-Server	No Functions
Lightweight and Embedded	No Variables
Database Files	Low Time Precision
General Applicability	No communication protocols

3.2.1 SQL and SQLite Functionality

Triggers [40] are what makes implementing a CRDT-DB in SQLite possible. Triggers allow one to define a set of SQL statements that execute any time a user performs a specified transaction on a specified table. The types of triggers supported by SQLite are deletion, insertion, and update triggers. A programmer can further specify triggers to trigger their set of instructions before or after a triggering transaction.

Tables are the main building blocks of SQLite, as even without any augmentation, it relies heavily on meta-tables[41, chapter 4]. As such, it is clear that mimicking SQLite's use of meta-tables can be used to overcome challenges. SQLite's meta-tables are a preexisting testament that this works reliably and efficiently, as it would not have been employed by SQLite if this was not the case.

SQLite ties each database of multiple relations to a single file, making it clean and easy for an overlying application to use SQL as it is as contained as possible. In addition, it is possible to access other database files through a database connection, meaning as long as the databases are on the same disk or appear to be so, a database connection can access multiple databases.

SQLite has no ability to define functions using SQL [41, chapter 9] or variables, meaning implementations rely only on tables and operations performed on them. There is, however, the possibility to apply user/application-defined functions, where one can add C-functions to SQLite [41, chapter 9]. User-defined functions allow for functionality such as encryption and regular expressions. The difference between what we refer to as functions and user-defined functions is the scope. Functions would mean something defined in SQLite that executes SQL statements, not defined in an extension that executes C-code.

User-defined functions open up whether it is applicable to make a SQL database into a CRDT-DB. While it allows for overriding SQLite's functions, this does not apply to the big three: UPDATE, INSERT, DELETE, and many other SQLite functions. This limitation is due to these functions being composites of many functions, where the application-defined functions only allow for functions limiting C-functions callable with the syntax *"uFunc("tbl", "id", 2)"* as opposed to something like and *"UPDATE tbl WHERE id = 2"*. There is the possibility to use functions like wrappers, where one could call SQL statements within the functions to achieve CRDT functionality. However, they will not be usable by existing SQLite applications and instead be more akin to using an ORM. Existing applications rely on the underlying SQL syntax remaining unchanged. Another downside is that this would require running each database connection with an additional executable containing the extension with the code. No other dependencies are needed when just using SQL to augment the database.

SQLite's time precision is only allegedly slightly better than a millisecond [41, chapter 8], meaning that with concurrent updates across multiple devices, there is a possibility of different events having identical timestamps. Minorly but in addition, there are issues when multiple local updates occur within the same millisecond time frame. The issue is that it is impossible to tell the order of these local operations.

Communication support within SQLite databases across multiple devices is non-existent, meaning they have to be left to the user application to handle. Merging two databases requires network communication unless the databases are on the same device.

3.3 Supporting CRR with SQLite

Firstly it is essential to identify what makes a database a database of conflict-free replicated relations, and therefore the following points are reiterated:

- Each row in a relation must store metadata to determine its state.
- The program must store an accurate timestamp with each attribute in each row.
- Each operation to the database is atomic across the CRR and AR layer to ensure that no data is left partially updated. For example, when deleting a row in the AR layer without incrementing CL in CRR layer.
- The relation must be syncable with other replications of the relation.

For the first two points, this project mimics the two-layer approach of the original CRR work [5]. Augmenting a normal relation to a CRR is, therefore, the process of managing the data across two layers.

3.3.1 Two-Layer Architecture

Tables can easily store the CRR metadata. Each table in a to-be augmented relation will need to be represented as two tables: one existing as-is, being the AR layer table, and another one with added columns for the metadata. For an AR layer of N columns, the CRR layer has *2N+1* columns. Each attribute *A* needs a timestamp *t(A)* attribute, and the row needs a Causal Length attribute to determine its state.

CRR [5] creates wrapper functions for updating databases, where each transaction is composed of two, an update for the AR layer and an update for the CRR layer. Updating both layers is possible even with SQLite limited function support, as mentioned in **section 3.2.1**. However, since the overlying applications have to conform to the new wrapper functions instead of SQL, user-defined functions do not support this project's goal of generality applicability.

Similarly, one might say that upholding CRR is the user application's responsibility. With this approach, the user application must propagate any update to the AR layer onto the CRR layer itself. Ideally, this would instead happen automatically to achieve this project's goal of general applicability.

We achieve general applicability through SQLite triggers instead of wrapper functions. We use triggers to propagate updates, insertions, and deletions from the AR layer to the CRR layer. The overlying application can remain unchanged, only needing to interact with the AR layer as it usually interacts with an underlying database. This approach remains generally applicable as inter-layer communication happens automatically and without extension to basic SQLite functionality.

3.3.2 Merging Tables/Relations

Merging two relations as done in CRR [5] would require the triggers to store each remotely edited row in a queue. This queue is a delta that can compare itself to the local data during a merge. Using tables as queues is not optimal. While SQLite stores commonly accessed tables in memory through caches [41, chapter 8], a growing queue table would eventually lead to a potential slowdown. Worth noting is that SQLite extensions support custom-made data types that are viewable as tables. Custom data types would allow for a queue-type structure. Still, custom data types do not have access to some essential SQL features that might be useful for the merging process and require even further extensions.

While CRR uses a continually generated queue delta, an alternative would be a delta generated on command. As each row has a timestamp, a consideration is generating a new delta table from an existing table based on this. Generating deltas would limit message size across networks and potentially speed up merges, as deltas are smaller than their parent tables. The disadvantage is the additional step of generating the delta. The advantage is not continuously updating a queue with every transaction. Whether this is worthwhile might depend on the overlying application's usage pattern of the database. Considering the number of merges versus updates might be the deciding factor.

It is also worth considering initially avoiding the use of CRR-deltas. If the program uses tables as the underlying data structure for deltas, the merge function can remain the same. Merging two deltas and applying them to the relations yield the same result as merging two relations [29]. Therefore, we initially focus on merging non-delta tables, as we can use this later to merge deltas.

3.3.3 Atomic Transactions and Integrity Constraints

Guaranteeing ACID transactions in the previous CRR design [5] is done by creating a savepoint before updates and rolling back to this point whenever there is an error with the update. Savepoints ensure that either both layers of the relation are updated, or neither is. By default, triggers happen within an atomic state. A trigger where an error occurs writes nothing to disk. However, this leaves the issue of undoing the corresponding update on the AR layer relation. This update already being committed to the database makes it impossible to roll it back to a savepoint. However, within the trigger, the old values of elements are accessible as "old.element". Having access to the un-updated values means adding functionality to triggers that check whether an error will happen and reapply the old values to the database.

An even simpler solution is making each trigger happen after the update. The order of updates will differ from the previous CRR[5]. However, it will ensure the option of simply canceling the update in the case of unwanted effects. With this, we achieve atomic transactions.

Integrity constraints are limitations defined along with the table schema on elements. Chief among them in terms of relevance is reference constraints. Enforcing reference constraints means a foreign key always has to correspond to an existing row or be null. Whenever a row referenced by another relation no longer exists, there is an integrity constraint violation. One can set SQLite either to enforce violations, meaning any operation resulting in an integrity constraint is canceled, or ignore violations.

The problems arise when merging multiple tables, as one table at site A might have deleted the row that another table at site B references. The challenge is to ensure that tables do not refer to *dead rows* while maintaining consistency. Selectively not applying some changes, where conflicts can arise when merging, would lead to inconsistency between the two tables after a merge. As it stands, there are two paths, proactive and reactive.

Proactively fighting integrity constraints would mean putting extra logic into a merge operation. After merging the CRR layer, all rows must be checked for potential constraint violations. If rows refer to a row with even causal length, this will cause a constraint when propagated to the AR layer. In this case, we must undo the transactions that cause the violations.

Reactively checking for constraints can be done by checking one of SQLite's meta-tables: the constraint table. Located in this table are all rows breaking the reference constraint. There is the potential to use this info to undo any integrity constraint resulting from a merge.

3.3.4 Communication Between Replicas

Finally, there is the issue of communication and network maintenance. A concession point will have to be network communication since basic SQLite supports no such features. Lack of communication protocols means that while the internal and local parts of CRR functionality can be achieved with only SQLite, accessing the database files on other devices is impossible.

The overlying application or user can make communication between nodes possible. Users being responsible for communication means that while the local version contains all it needs to merge with another replica, it cannot automatically merge across networks by itself. This responsibility falls on the overlying user application, as long as it can make it appear before a merge that as if that a foreign database file is a local one, SQL can handle the rest. The database can store addresses, paths, and other data needed for communication in tables. With this, accessing another database file might be as easy as using Secure Shell (SSH) [8] to make it appear as if local. Alternatively, database file deltas can be sent over the network and temporarily stored locally for merges.

While basic SQLite does not support communication, there is still the possibility of extensions with user-defined functions. Arising from this, one can employ communication between nodes, potentially allowing for *merge* to be a callable function(one might argue this still leaves merging to the application) or automatically happen periodically. The downside of this approach is that it forces users to use SQLite extensions, which has not been truly necessary thus far.

3.4 Design Overview

With the groundwork built from the design principles proposed in CRR [5] and further built upon to suit SQLs steep limitations, this project presents the following design outline.

3.4.1 Inter-layer Data Propagation

The two-layer architecture is continued for this design as well. We define a base database or relations B, and to augment it, we generate a CRR layer table to each corresponding B, leaving B as-is to be the AR layer. The CRR-table is an SQL table with a similar schema to the underlying table, adding a timestamp for each attribute and a CL for each row.

With the two layers clearly defined, the next step is to reiterate data between them. We exchange data between layers using SQL triggers. There are three different operations on the database, insert, update and delete. Three SQL scripts define triggers for each AR layer relation: after insert, after delete, and after update. These triggers will have to update the CRR layer with the information given in the updating query. We must generate three triggers, update, deletion, and insertion, tailored to the specific schema for each table in the AR layer database.

Each table in the database will propagate any insertion, deletion, or update to the CRR layer, which is responsible for applying CRDT functionalities to the database. Triggers propagating data entail that since the AR layer is the layer with the tables all operations reference, users will not need to worry about CRR. We ensure that the CRR layer is all handled by the triggers.

3.4.2 SQL Merge

An essential component to the overall design is to generate a merge operation between two replicated databases. A merge operation is a complex SQL script that merges all updates from one database into another while upholding the rules of CRR: the highest CL between the two rows determines whether the row exists or not, and the latest timestamp for each attribute determines each value. Once the merge operation has applied the changes to the CRR layer, it also passes them on to the AR layer.

The merge function is also the most logical place to resolve integrity constraints, as this is where they appear. The most logical place to do this is before the data is propagated to the AR layer, so there is no need to re-apply changes to the AR layer after resolving integrity constraints.

On an inter-nodal scale, an essential takeaway from merge is that it in some ways mimics git [42], as one merges into or onto a database, one gets all their changes, but not necessarily the other way around. Merges are similar to a 'git pull', and two instances will not be consistent between merges and updates unless both do a mutual pull operation. This design choice is to avoid multiple database connections writing to a single database file.

3.4.3 Vision and Application

The overall vision for the application as it will exist in this project is as follows: A two-layer CRR approach to the database, where the intricacies of CRDTs will need to be of no concern to the user. The database will function exactly like an ordinary SQLite database, and the user needs no changes in how they would usually use it. Further, the user can merge with other instances through a console command, all without halting the operations of the database. The

SQLite shell or any application can keep running during a merge.

We continue on our earlier example of a local-first client-server research database from **section 1.5**. The client will be able to perform each transaction locally and have a perfectly available database of research data for their needs. While being far from civilization, the client can still use the database and update it with his findings. Once in a place with an internet connection, the user can pull data from the server, which might also trigger the server to do the same. All the researcher's data is now available for analysis by other clients using the server.

In addition to users like the researcher, embedded edge sensor devices use the database. These edge devices periodically automatically merge when they need up-to-date data, but when not, they use their local store and limit bandwidth usage and computing power needed to merge constantly. Billions of edge devices come with SQLite support [38], meaning employing CRR on them was as simple as copying an instance of the CRR database to their local storage. A single SQLite database file contains everything needed for conflict-free replicated relations: tables, triggers, timestamp generation, and merge possibilities.

3.5 Implementation Details

This section mainly focuses on the intricacies of implementing the prototype database system of conflict-free replicated relations. We implement the triggers and functionality to merge multiple database instances to achieve a consistent state and explore the implementation additions needed to make this work. The three types of triggers are as follows: after updates, after deletes, and after insertions.

3.5.1 Triggers: After Updates, After Deletions, and After Insertions

To map rows between layers, we employ Universally Unique Identifiers [31] for each row. For the current implementation, both AR layer rows and CRR layer rows use UUIDs as primary keys.

When inserting, there are two cases: the user inserts a previously deleted row, or the row is a brand new row. For brand new rows, the trigger has to create a row in the CRR-relation, with all the data that will initialize the corresponding AR-row also added to matching columns in the CRR layer. Given an insertion of the row:

```
INSERT INTO table(x,y,z) VALUES(1,2,3);
```

the insertion trigger must also affect the x, y, and z of the CRR layer relation. Also, a timestamp must be generated for each of the newly created attributes and added to the CRR layer relation. In new rows, the trigger initializes causal length to 1. The now() function is a placeholder as we explain timestamps in **section 3.5.2**:

```
CREATE TRIGGER after_insert AFTER INSERT ON table
BEGIN
    INSERT INTO crr_tbl(x,crr_x,y,crr_y,z,crr_z,crr_cl)
        VALUES(new.x,now(),new.y,now(),new.z,now(),1);
END;
```

A repeating challenge is a lack of if and else statements in SQLite. To combat the limited logic, we must make two mutually exclusive SQL statements within the trigger. With this comes a guarantee that one can happen and the other can not, and by adjusting the parameters in the WHERE-clause of a query, we can mimic logical operations like if and else.

For an insertion trigger, we want just one of the following two to happen: it updates the CRR layer row if it has previously existed, or it inserts a CRR layer row if it has not. In practice, we do this to separate that either a new row is inserted or an old row is reinserted. The trigger has an update function for old rows where it updates the row of the given UUID, but if the UUID does not exist, there is no update. On the other hand, a WHERE statement ensures that a new row does not exist in the CRR layer. If no row exists with the given id, the trigger inserts a brand new row into the CRR layer. These two cases are mutually exclusive, as one can happen and not the other. The mutual exclusion ensures that no row is duplicated in the CRR layer when an insertion trigger happens after an insertion to a table. Achieving mutual exclusion culminates in WHERE-clauses looking like this, where UUID is from the existing CRR layer and new.UUID is from the insertion that caused the trigger:

```
/* Update existing row in CRR layer */
... WHERE UUID = NEW.uuid;
/* Insert new row to CRR layer */
... WHERE NOT EXISTS
    (SELECT uuid FROM crr_tbl WHERE uuid = NEW.uuid);
```

Deletions are the simplest of the three types of triggers, as all that has to happen is for the CL value for the row in the CRR layer to be incremented when a user deletes the corresponding AR layer row. Incrementing CL is done as a single simple update statement that triggers on deletion.

Updates trigger whenever a user application updates a value in an AR layer relation. When updates happen, corresponding values in the CRR layer are changed, and the triggers set new timestamps. While SQLite does not have if or else statements, CASE statements can be applied to single value changes. To check what timestamp attributes should be updated, an update statement looks for cases where the new value is different from the old. The trigger generates a new timestamp for all values where the values differ.

On the topic of timestamps, to combat an SQLite issue where timestamps only have millisecond precision, there have to be some changes to ensure that two concurrent updates do not get the same timestamp.

3.5.2 Timestamps

To generate accurate timestamps, we implement a time-table containing an id, a millisecond UNIX-like time, and a six-digit decimal number. The time functions use id to ensure that the time-table only has one row. The millisecond time is an SQLite timestamp obtained through SQLite's basic time functions. Furthermore, the six-digit decimal number is a specially randomized number used to emulate a greater time precision. Whenever a timestamp is updated or inserted, the user of the timestamp fetches a number consisting of the millisecond time value combined with the nanosecond emulation. Nanosecond emulation is done in the following way: $t(n) = (ms * 1000000) + ns$, where $t(n)$ is the emulated current time, ms is the millisecond value, and ns is the six-digit decimal of emulated nanoseconds.

The method for generating timestamps means that the program must prime the time-table with an update before any update involving timestamps. At the start of each trigger using timestamps, the time method generates a new ms value from the current time, and then this triggers the generation of a new ns value. In the case where the old ms value is equal to the new ms', meaning a new millisecond has not passed since the last update, the system generates a new ns' as follows:

when $ms = ms'$ then $ns' = ns + rand(1000000 - ns)$ where $rand(n)$ is a random number in the range 0 to n.

And given a new ms value: ms':

when $ms \neq ms'$ then $ns' = rand(1000000)$

The nanosecond time methods result in an emulation of a more precise time precision than millisecond time. Having the last six digits be a semi-randomized value ensures that the chance of two timestamps being the same across nodes is similar to that of a nanosecond time precision timestamp. The logic of adding upon the last timestamp in the case where a millisecond has not passed ensures that operations within a local database remain ordered, as would be the case when using actual nanosecond time.

3.5.3 The Merge Script

The process of merging databases is two-fold. Firstly, the database merges its CRR layer with another database's CRR layer, and then it updates its AR layer with data from the newly merged CRR layer.

Storing timestamps in a table allows for an opportunity to implement a hybrid-logical clock[32]. This clock prototype compares the timestamps of the latest updates on a merge between nodes and adds an offset if a time-drift is detected. The hybrid-logical clock stores the offset value in an additional field in the time-table.

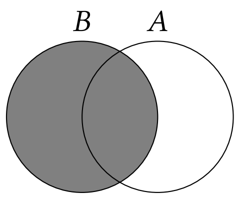

Figure 3.1: Left outer join of local table A and foreign table B

The approach to merging CRR layers is by updating the local table with values from the left joining the locally stored table and the foreign table, as seen in **figure 3.2**. The foreign table is set as the left table in this join, while the local table is the right table. They are joined on their UUID values, meaning all rows that exist in both tables are now easily comparable. If the database has multiple CRRs, then a complete merge does this for all CRRs. For each element in the table that will be the final table, there are two possibilities:

The first possibility is that the value only exists in the left of the two merging relations. A singular instance of a value happens when a row or value only exists in the left table. Using a left join, the case will never be that the value only exists in the right table, meaning a foreign table has created the row since

```sql
INSERT OR REPLACE INTO db1.crr_tbl(
    uuid,
    val,
    crr_val,    /*Timestamp of val */
    crr_cl      /*Causal Length of row*/
    )
    SELECT
      t2.uuid
      END,
      CASE
          WHEN t1.val IS NULL THEN t2.val
          WHEN t2.val IS NULL THEN t1.val
          WHEN t1.crr_val > t2.crr_val THEN t1.val
          ELSE t2.val
      END,
      CASE
          WHEN t1.crr_val IS NULL THEN t2.crr_val
          WHEN t2.crr_val IS NULL THEN t1.crr_val
          WHEN t1.crr_val > t2.crr_val THEN t1.crr_val
          ELSE t2.crr_val
      END,
      CASE
          WHEN t1.crr_cl IS NULL THEN t2.crr_cl
          WHEN t2.crr_cl IS NULL THEN t1.crr_cl
          WHEN t1.crr_cl > t2.crr_cl THEN t1.crr_cl
          ELSE t2.crr_cl
      END
    FROM db2.crr_tbl t2 LEFT JOIN db1.crr_tbl t1 USING(uuid);
```

Figure 3.2: SQL snippet of merging CRR layer of two DB instances

the last merge, and it, therefore, does not yet exist in the local replica, as shown in **figure 3.1**. When this happens, the merge method chooses the left value by default.

The second possibility is that the value exists in both tables. When this is the case, the value with the greatest corresponding timestamp is selected, as seen in **figure 3.2 lines 13-14**. The timestamp attributes are also updated, as seen in **figure 3.2 line 16-21**. The merge script selects the highest existing causal length for each row, just like in a CL-set CRDT.

With the two tables merged in the CRR layer, the next is to propagate the changes to the AR layer. The merge method propagates data to the AR layer in two steps: For all rows with an odd *Causal Length*, the penultimate step of the merge script uses SQL "INSERT OR REPLACE INTO" to add the new rows to the AR layer. Metadata like timestamps and CL is stripped away for each row, and the new values are applied to the AR layer. The reasoning for not using update is that this catches cases where the row did not exist before the merge. Finally, for all rows with even *Causal Length*, the SQL "delete" method removes all rows from the AR layer that should no longer exist post-merge.

The problem with updating the AR layer is that the AR layer relations have triggers attached to them. Both previous steps would lead to triggers that would cause inconsistency right after a merge. The solution to this, one of the most significant implementation challenges, is implementing the option to disable triggers using SQLite.

3.5.4 Disabling Triggers

The system needs to disable all triggers temporarily for the duration of the merge. On first thought, one could assume that since triggers happen in an ATOMIC state by default, wrapping the merge in an atomic transaction using SQLite's "BEGIN and COMMIT" would cause the triggers not to trigger. Relying on SQLite's atomic functionality does not work, possibly because triggers use the SQLite "SAVEPOINT and RELEASE" [41, chapter 4] transactions that allow for nesting.

Instead, this project implements a "lock" table. The locks are not mutexes but temporarily lock an AR layer table from performing triggers that propagate the data to the CRR layer. A row in the lock table has two attributes, the name of the table that should be locked and a boolean value deciding whether the lock is on or off.

The next challenge is making the lock block triggers from happening, which there is no way to do. There is no IF-clause in SQLite, so one can not simply state that if there is a lock, return. However, each update within each trigger for each table has a WHERE-clause. Simply changing this clause to check if the lock is on all updates to all rows would fail if the lock is active:

```
/* Update trigger propagating data */
...
WHERE uuid = OLD.uuid
    AND
    (SELECT trigger_lock FROM lock_tbl WHERE tbl = "crr_tbl") = 0;
```

The triggers still trigger with this method, but they ignores all changes as long as the lock is engaged. Wrapping merge operations in locks on the afflicted tables, as seen in **figure 3.3**, we guarantee the following: Locks ensure that the changes to the CRR layer can be propagated to the AR layer without a trigger inadvertently changing the CRR layer again.

3.5.5 Communication

At last, a separate program is built as a prototype to handle communication. The program uses SSH to access foreign databases as local tables. We externally define a function to merge the database, which copies rows from that database's CRR layer into a separate database file. What rows to copy are based on rows' timestamps, and the function copies any row with a timestamp greater than the time of the last merge to the delta database file. This new file acts as a delta and is then transferred to the local disk, and then the merge function merges the delta into the database.

We built the merge function to support merging delta relations and entire relations. Each node stores replications available to merge within a database table, along with the SSH path. To do a git-like pull of data into one's local replica, the user can call merge in the system shell. Using the merge function does not halt normal SQLite operations. Any applications using the database and the SQLite shell itself can continue functioning.

3.5.6 Integrity Constraints Portotype

CRR [5] resolves reference integrity constraints in two ways depending on what operation causes the violation:

1. For updates, the update is reverted to the previous value.

2. For insertion and deletions, the transaction is undone by incrementing the affect row's CL by one. Effectively removing an inserted row or reinserting a deleted row.

We create a prototype that mimics this functionality. To undo a deletion, one needs the old instance of the value; we define new relations to temporarily store values in CRR layer relations before merging. This prototype acts as a proof-of-work without fully realized functionality, see **section 6.7.1**.

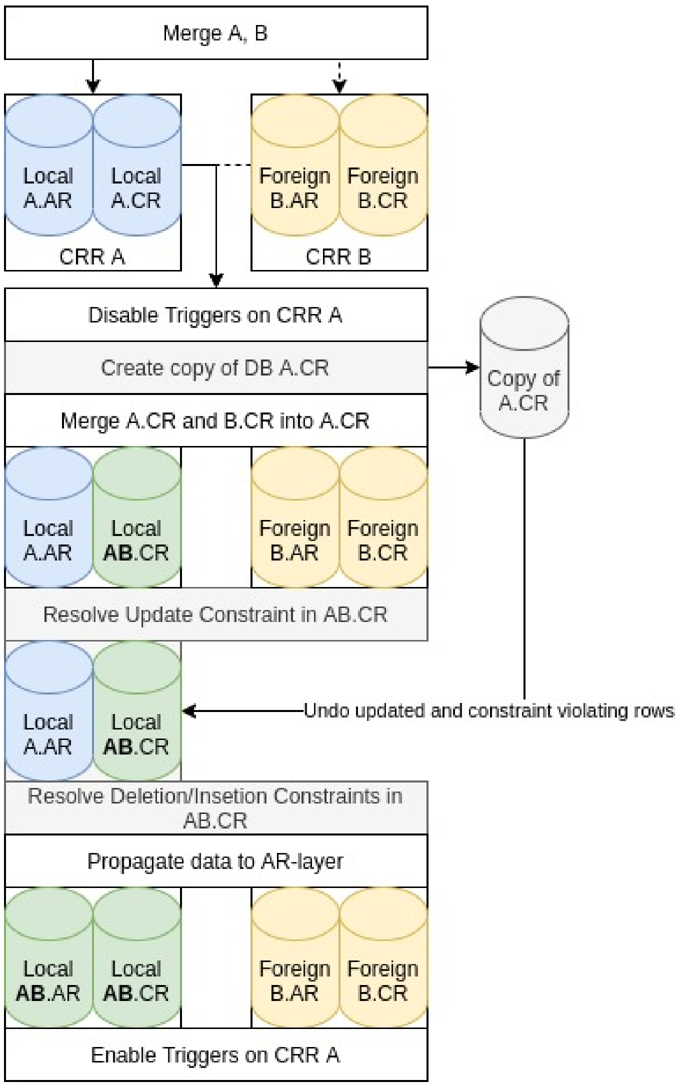

Figure 3.3: Merge pipeline, with resolution of reference constraint violations in grey

We proactively look for integrity constraints after merging the CRR layer but before propagating the data to the AR layer. This way, constraint violations are caught before they happen. If a foreign operation defined through a WHERE-clause causes a row to reference a row with even CL, we undo the transaction. We separate updates and deletions by comparing the pre-merge attributes in a WHERE-clause. If the CRR layer merge changed the attribute, it is an update, if not an insertion or deletion.

Undoing transactions is as simple as updating the CRR layer attribute to its old value or incrementing the CL. We perform the update undo first to catch the edge cases where an updated row still references a dead row in another relation after undoing an update. A deletion undo is performed if there is still a constraint on the row after an update undo.

3.6 SQLite CRR as a Local-First DB

To best encapsulate the advantages of this design and implementation, we return to the example of a research application using a local-first database from **section 1.5 and 3.4.3**. The researchers now wish to employ the application en-mass through mobile devices to gather data on a grander scale.

With SQLite-based CRR, replicating the database is done simply by copying the database from one device to another. All triggers and tables needed to make the two-layer architecture of a local-first database of CRRs work are stored right alongside the data within the SQLite database file. The overlying application can use any ORM or SQL methods to alter the data as it needs, without the concern of how the device propagates this data to other devices and the intricacies of avoiding consistency errors.

The standard complexity issue of gathering consistent data through such an application is avoided by easily being able to merge local data. Multiple devices can work asynchronously. Suppose the researcher wishes to move away from the centralized server structure and into a peer-to-peer model. In that case, researchers can quickly do this by merging with other devices and propagating data this way instead of merging with a central server.

To augment a database for such an application, we start with an ordinary SQLite database with several schemas or pre-existing tables containing the data. The augmentation process then generates a CRR layer: matching tables with timestamps for each row and a CL column for each table to be augmented. The application is now a two-layer database. Then we generate the three trigger schemas based on each of the AR layer tables and create them to apply them to the database. The augmentation program also initializes a time-table and a lock-table. With this all wrapped in the database file, all that remains to do is give each user application a copy of the database file. With this, the database is now a local-first database of CRRs.

4 Goals and Contributions

4.1 SYNQLite

In parallel to working on this project, Weihai Yu and I developed a prototype software. See **appendix A** for the paper describing the project and software. This prototype, named *SYNQlite*, was developed using the same methods as described in this project with the goal of automatic CRR support for any existing SQLite database. This paper, which is based on ideas from this thesis, has been accepted for presentation by the "International Workshop on Advanced Data Systems Management, Engineering, and Analytics (MegaData)" [43] in the ADBIS2021 conference.

It is therefore essential to distinguish what part of this project is my work and what I borrow from SYNQLite:

My work was the part of the application that applies CRR principles to SQLite. These parts include the triggers and how they are used to propagate data between layers. It also includes the merge function, the nanosecond emulated timestamps, and locking the triggers.

My collaborator focused on the communication and the automatic application of the SQLite CRR principles I discovered to an existing database and handling the communication between nodes.

4.2 Contributions

This project makes the following contributions:

- A prototype two-layer CRR architecture that functions using only SQL:
  - Methods to interact between the two-layer, making it possible for the user to use the database as a standard SQL database while triggers handle the CRR layer.
  - Timestamp generation with greater than millisecond time precision using only SQL.
  - A method to merge the database as an SQL script that one can reproduce for most schemas.
  - Implementing a prototype hybrid-logical clock using only SQL.
- This prototype contributes to the SYNQlite project. SYNQlite takes the augmentation used on this project's example database and applies them to any database.
- Models to measure consistency and availability in local-first systems used in experiments. To be described in **chapter 5**.

4.3 Priorities

When looking for what to prioritize, we reiterate the goal of the project: To create a general database manager of Conflict-Free Replicated Relations. Further, we define the sub-goals of the database needing to be general, meaning that it is applicable and usable by most systems without the need for external dependencies. At last, the database being a database of Conflict-Free replicated relations, which is a database with strong eventual consistency and constant availability through the use of a local-first approach. The question regarding testing and evaluation is then two-fold: which properties are feasible to test, and which should we prioritize to highlight the goal of this thesis?

Testing the generality of the database would mean proving that the functionality of the database is not limited to one case. Meaning it is as applicable as any other database and usable by all systems. For the second case, a simple proof by contradiction is offered: There are systems for which the database does not work. We further limit systems to systems with SQLite support. SQLite is already one of the most widely available, lightweight, and popular database managers. Systems that have no SQLite support probably have no support for relational databases. This project does not aim to create an even more lightweight database to be slightly more applicable.

As testing whether the database is usable by millions of different devices is outside of this project's scope, we instead focus on generality as an aspect of the database, keeping the functionality of SQLite, as this would mean the database is as general as an SQLite database. Therefore, tests are focused on breaking the functionality of the database through both chaotic test queries and systematic ones aimed at potential flaws, all while seeing if the database functions like an SQLite database.

For the second part of the goal, the database being comprised of Conflict-Free Replicated Relations, we reiterate the main advantages of CRDTs: Constant availability and Strong eventual consistency. These two attributes are thereby the clear candidates to prioritize when testing the feasibility and advantage of CRRs.

Finally, while performance based on throughput has not been a focus during the development of this project, it is a counterpoint to the advantages of CRRs. Therefore, to see the degree of tradeoff one does when applying CRR to a system, we still carry out experiments pertaining to the performance of the database management system. These experiments will allow for the comparison to other systems, as well as seeing what the penalty or overhead of having a database of CRRs is.

4.4 Expectations and Limitations

4.4.1 Generality

For generality, the experiments would revolve around the correctness of the system when applying different kinds of updates. We expect it to function as intended for most cases, as any update is applied in the AR layer just as it would be in a typical database. The only instance in which the underlying CRR changes the data as seen by the user is during a merge. Therefore, it is vital that the triggers that propagate data to the CRR and update things like the Causal Length function properly.

There are no metrics applicable to generality except applying it to many different systems and observing whether they still function. Testing if the system truly works on millions of different devices is out-of-scope of the current implementation. Instead, we perform extensive correctness tests.

- Insertion of already existing rows: Should not cause the CL to be incremented in the CRR layer, as the trigger will not fire when the triggering operation is not applied.
- Deletion of deleted rows: Should not increment the CL in the CRR layer since the trigger will not fire when the triggering operation is not applied.
- Updates of deleted rows: Should not update the deleted row as the trigger will not fire when the triggering operation is not applied.
- Reference integrity constraint after merge: The prototype implemented with some functionality acts as a proof-of-concept.
- Duplicate timestamp across nodes: Is statistically unlikely with enhanced time precision from emulating nanosecond time.

With this in mind, the expectation is that an overlying system will be able to perform almost all functionality that it would have been able to on a regular SQLite database, with the added benefits of merging with other databases.

4.4.2 Availability

Availability is the main strength of a local-first approach; as the data is stored locally, it is always available. Researchers commonly test availability as the number of requests that yield responses through a gradually deteriorating network [44]. In such a test environment, we expect that every request receives a response, as the CRR-DB stores data locally.

4.4.3 Consistency

Strong eventual consistency means that the system must guarantee that it can reach a consistent state through a merge. Testing eventual consistency would mean ensuring that two or more databases all have synchronized data after merging them to reach a unified state. As the approach revolves around the proven concept of the CL-set CRDT and using LWW-registers when it comes to row internals, the expectation is that as the underlying principle dictates, the state of the replicas should be consistent after a merge. It is important to note that since the application uses one-way merges instead of updating both databases whenever they merge, we only expect a consistent state when both replicas merge with each other.

4.4.4 Transaction Performance

In terms of database performance, we can use an ordinary SQLite database as a baseline and predict the overhead of adding the CRR layer through time complexity. In a single-layer architecture, the effect of an operation is that a single relation is updated. In a two-layer approach, each operation updates two relations, one in the AR layer and one in the CRR layer. An insertion, deletion, or update SQL query requires two operations instead of one. Therefore, an easy assumption is that augmenting a relation to CRR causes some level of drop in throughput.

In SQLite, a B+-tree is used to store data. Looking at a B+-tree of size n, the time-complexity of searching, deleting, or inserting data is given as $O(logtn)$. To further compare the potential throughput of a normal SQLite database and a CRR-augmented SQLite database, we look at how the time complexity changes with the addition of the CRR layer. For example, with an SQLite database relation of n tree nodes and a CRR SQLite database relation of m1 tree nodes in the AR layer and m2 tree nodes in the CRR layer, we assert the following:

- If the same insertion and deletions have been applied to both databases, then: $m_1 = n$, as operations are performed on the AR layer just as they would be in a normal SQLite database.

- Each row in m_2 is larger than a row in m_1 or n, as it contains additional metadata. Additionally, m_2 contains representations of deleted rows. Therefore, $m_2 > m_1 = n > 0$.

- Ignoring the time needed to generate timestamp or access the lock-table, as they contain very few elements, we compare updating, inserting, or deleting from a CRR database versus an SQLite database as followed: $O(logtm_1) + O(logtm_2) > O(logtn)$

- When querying a CRR SQLite database, only the AR layer is used, therefore when doing lookups, the following is clear: $O(logtm_1) = O(logtn)$

We observe that for insertions, deletions, and updates, the operations take more than twice the time compared to a standard SQLite database. However, for retrieving data through queries, the same throughput is expected. The expectations in terms of performance are thereby this: We expect no loss in throughput for selections, but we expect insertions, deletions, and updates to take a longer time than in a regular local SQLite database.

5 Experiments

This chapter details experiments we run on the system. Having narrowed focus in the priorities **section 4.2**, we know what to measure; the question now is how to measure it. This chapter focuses on stating the setup and observations of various experiments, the evaluation and analysis of these observations are in the following chapter.

We perform local experiments, with python3 [45] objects [46] used to simulate a node in a distributed system. Local testing draws testing focus away from communication aspects, which were SYNQlite's main contribution, and towards the underlying CRDT Relational DBMS, which is this project's main contribution.

The hybrid-logical clock prototype was not used for the tests, as it rarely does anything when simulating nodes locally. Emulated nanosecond timestamps and the rest of the system are just as described in **chapter 3**. The database used is a single relation database with the following AR layer schema: UUID, first name, last name, phone number and email.

5.1 System Hardware Specifications

Tests were performed using the following underlying hardware:

- **CPU:** Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
- **Disk:** SAMSUNG MZYLN256, 256GB SSD
- **Memory:**
  - Size: 8192 MB
  - Type: Type: DDR3
  - Type Detail: Synchronous
  - Speed: 1600 MT/s
  - Manufacturer: SK Hynix

5.2 CAP Theorem: Measuring Consistency and Availability

The CAP-theorem is the trifold relationship of Consistency, Availability, and tolerance to Partitioning and states that only two out of three are achievable in a distributed system. As the CAP-theorem is a widespread principle across distributed systems [10], it is also an obvious choice for comparison between them. Problems arise when comparing the CAP-theorem across different systems, however, as Consistency, Availability, and Partitioning are not measurable metrics but underlying attributes of a system. It is therefore important to clearly state how the experiments test the different attributes, as not to create unfair comparisons.

Consistency can be further subdivided into two categories, client-centric consistency, and data-centric -consistency [9]. Data consistency is that all instances of a data value are the same across the replicas. In the case of a SQL database, data consistency will occur after all nodes in the system have mutually merged. The client consistency model relies on the client and not the system's total state; if the client's value is not stale, there is client consistency. For example, the system might only be data-inconsistent, but if the most up-to-date value is within the client's relations, there is client-side consistency for that instance of data.

It is clear to see that the CRR approach is neither entirely a data-focused or client-focused consistency approach. This is because it can only guarantee that the data returned is correct after a merge where all nodes are responsive. Since this is not constantly the case across the system of replicas, there is no constant data consistency nor constant client consistency as being so would require inter-node communication on each request. However, the system will eventually reach a consistent state, where each instance of a data value is the same across the system. A way to measure data consistency can therefore be by measuring how long this process takes as done in [47]. On the other hand, measuring client consistency can be done by emulating how a client interacts with the system. Since the client consistency is dependent on whether the value in the local store is stale or not, there are two underlying factors: the data consistency of the system and whether the up-to-date version of data is in the local store by pure chance.

Availability would be more complex in a system where updates are asynchronous. In SQLite, updates are all scheduled, and one cannot be executed before the previous one is written to disk. A module exists that allows for asynchronous updates; it states the following trade-off: “You lose Durability with asynchronous I/O, but you still retain the other parts of ACID: Atomic, Consistent, and Isolated. Many applications get along fine without the Durability.” [48]. As we employ a local-first approach and updates are synchronous, no update can cause another to not trigger because of the database’s ACID functionality. The success rate will be measured, but to measure a more interesting aspect of availability, we look at the latency in the system as a way to measure the degree of availability [10].

5.3 Client Consistency and Data Consistency

We measure consistency in the following ways: N different replicas will be employed locally, and the experiments perform randomized transactions on randomized replicas and rows. These transactions are: updates to a single attribute, deletions and re-insertions of deleted rows. We limit insertions to re-insertion of deleted rows to ensure that the number of rows in the CRR layer remains the same.

Periodically the average number of stale rows will be measured and given as a client consistency; given as the percentage:

$$p = avg(\frac{|r_1 \cap r_m|}{|r_1|}, \dots \frac{|r_n \cap r_m|}{|r_n|})$$

where r is a relation instance behaving as a mathematical set of rows. Moreover, $|r|$ is the number of rows in the relation r. Client consistency is the average of n relations consistency percentage gained by looking at deviation from what a fully merged relation r_m would be.

Data inconsistency will be measured by periodically merging all replicas into a separate test replica. Since this replica is one-way merged from all other replicas, it will contain the system's implied consistent state. Then each of the rows in the consistent test database is compared to each other row, and if any row does not match, that row is considered data-inconsistent. The data consistency is given as:

$$p = \frac{|r_m \setminus ((r_m \setminus r_1) \cup (r_m \setminus r_2) \ldots \cup (r_m \setminus r_n))|}{|r_m|}$$

where r_m is the correctness test replica with $|r_m|$ fully consistent rows and r_1 to r_n are the potentially inconsistent relations. $(r_m \setminus r_n)$ is a set of all rows in r_m with the ones from r_n removed. The resulting set is a set of all inconsistent rows in the given replica r_n. Subtracting a union of all inconsistent rows across the system from the consistent replica r_m will yield a set of all rows that are data-consistent, which we use to calculate the percentage of data-consistent rows.

5.3.1 Experiment: Consistency in non-merging system

We look at the consistency behavior of n replicated relations with s rows. i is the merge interval and will be relevant later. When i is equal to 0, there are no merges. We observe the behavior of a non-merging system.

As seen in **figure 5.1**, data consistency is close to equal across an increasing number of nodes. Client consistency is, however, dependent on the number of nodes. A correlation seems to form between the number of nodes and client consistency, where a greater number of nodes results in more rapid decay and a lower point of convergence.

From observations in **figure 5.2**, we see that by increasing the size of each relation, it becomes clear that the greater the relation, the more operations are needed for it to be inconsistent. The fact that client-side consistency is better than data consistency remains true.

Continuing, we will periodically mutually merge all replicas and observe how this changes client and data consistency.

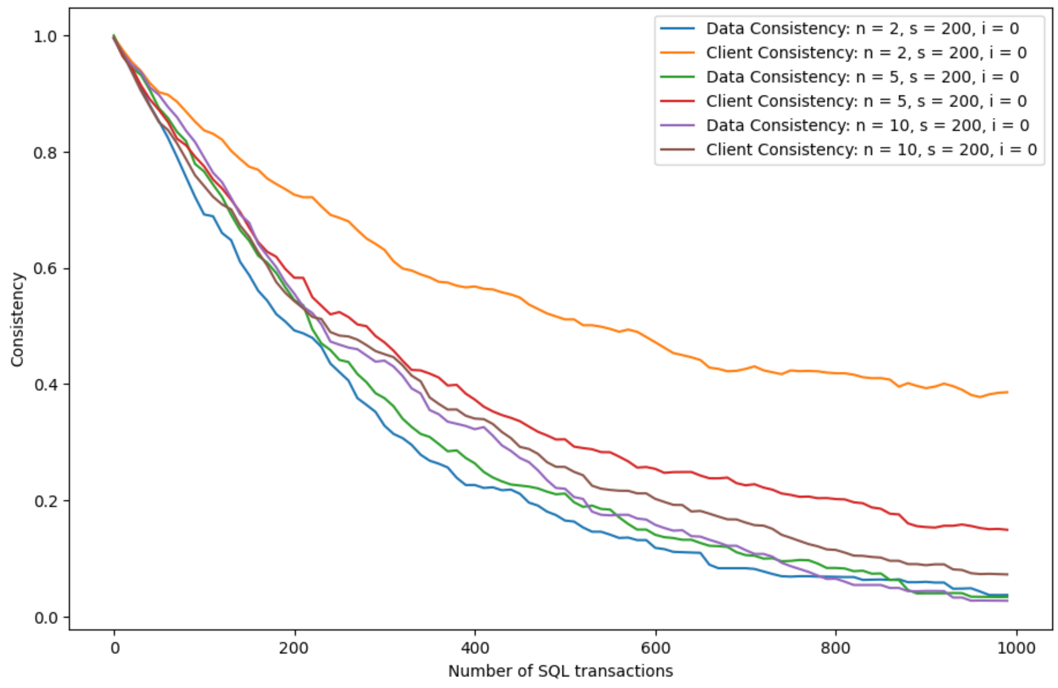

Figure 5.1: Client and data consistency in a non-merging system

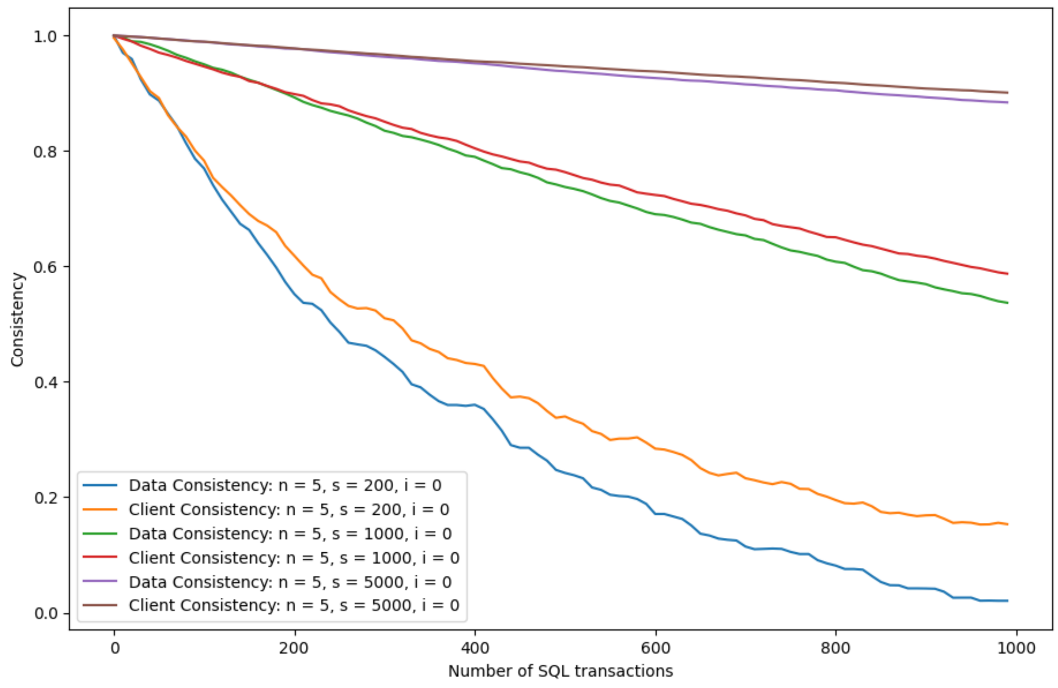

Figure 5.2: Effect of number of rows on client and data consistency

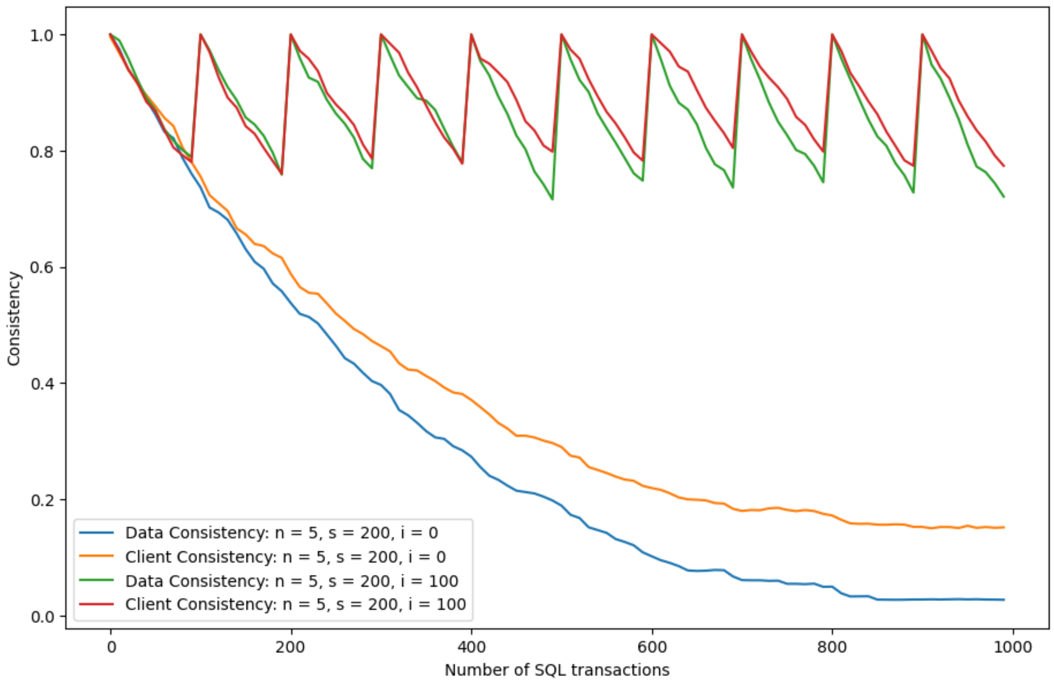

Figure 5.3: Periodical merges vs no merges

5.3.2 Experiments: Demonstration of eventual consistency through complete periodical merges

It is observed in **figure 5.3** that the non-merging and merging graphs follow a similar path until a merge happens. We observe merges as sudden spikes in consistency, where every complete merge restores the system to perfect client and data consistency.

On closer observation in **figure 5.4**, we see that the client consistency is still greater than data consistency while intermittently merging the system.

Looking at just data consistency and a best fit graph given as polyfit [49] in **figure 5.5** and comparing a merge interval of 50 to 100, one sees that the initial best fit consistency is 0.95 for 50 and 0.90 for 100. Twice the interval leads to twice the drop in consistency at these intervals. Another point of interest is that the average(best fit) data consistency drops off throughout the total runtime. The dips in consistency grow deeper as the total number of transactions increases.

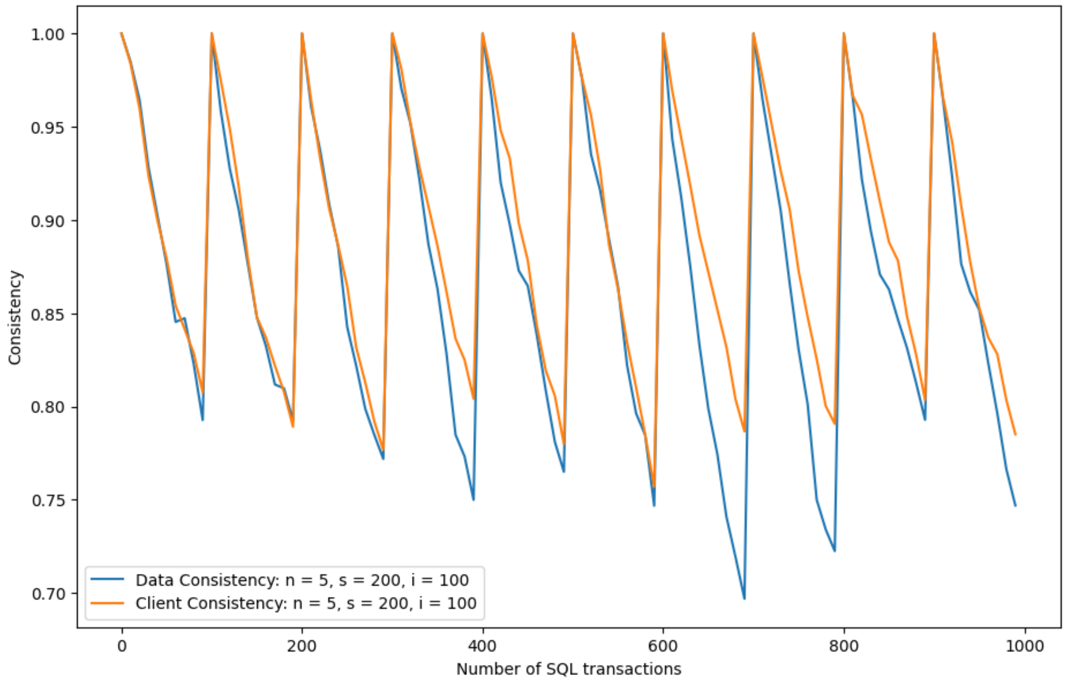

Figure 5.4: Periodical merges, 5.3 zoomed in

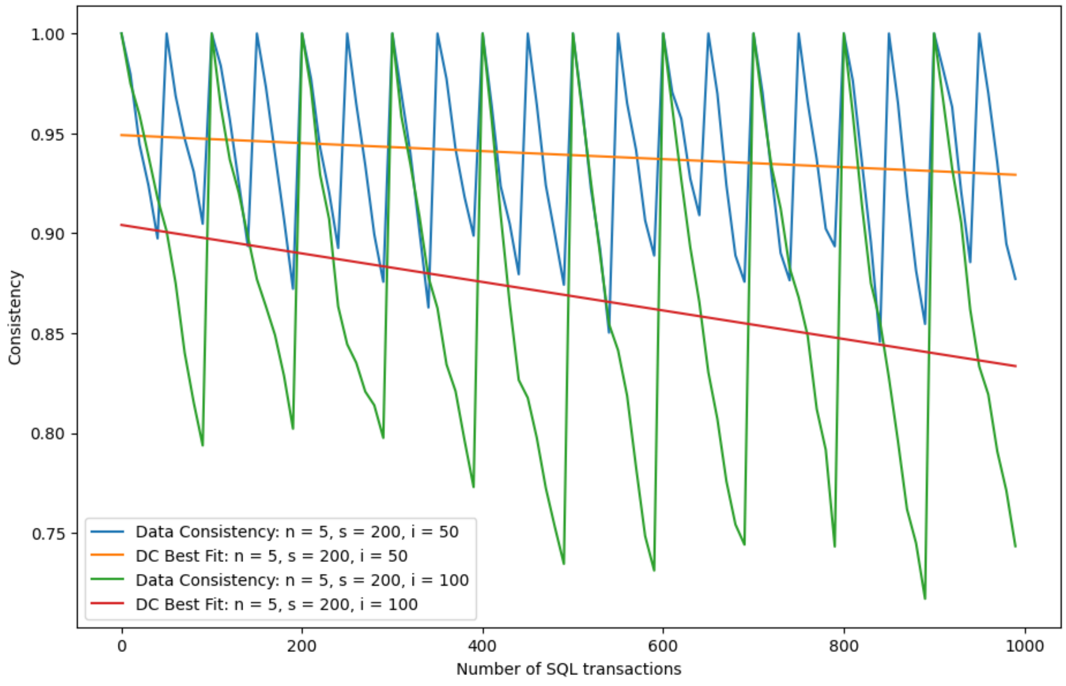

Figure 5.5: Data consistency when periodically merging with best fit

5.4 Consistency model 2: Partial Merge-Pulls

In a distributed system, one can not always rely on all nodes being up at a given time, and therefore relying on all nodes being up at all times is not an option. Moreover, an unstable network or connection is a repeated focal point of CRDTs and the approach in this project. Mimicking epidemic dissemination of information [50] and propagating information at random is an alternative to this. We observe what happens when nodes choose to do a "merge-pull" from others at the rate of every n operations performed by a given node. There are no periodical total merges of the entire system. Instead, the nodes merge with a single randomized node in the system.

5.4.1 Experiments: Eventual Consistency Without Fully Merging

As opposed to **figure 5.3**, **figure 5.6** shows no regularly repeating pattern. We also observe that the consistency seems relatively stable, as it operates within the 1.0 - 0.75 range. While being 1.0 initially, consistency never reaches this point again.

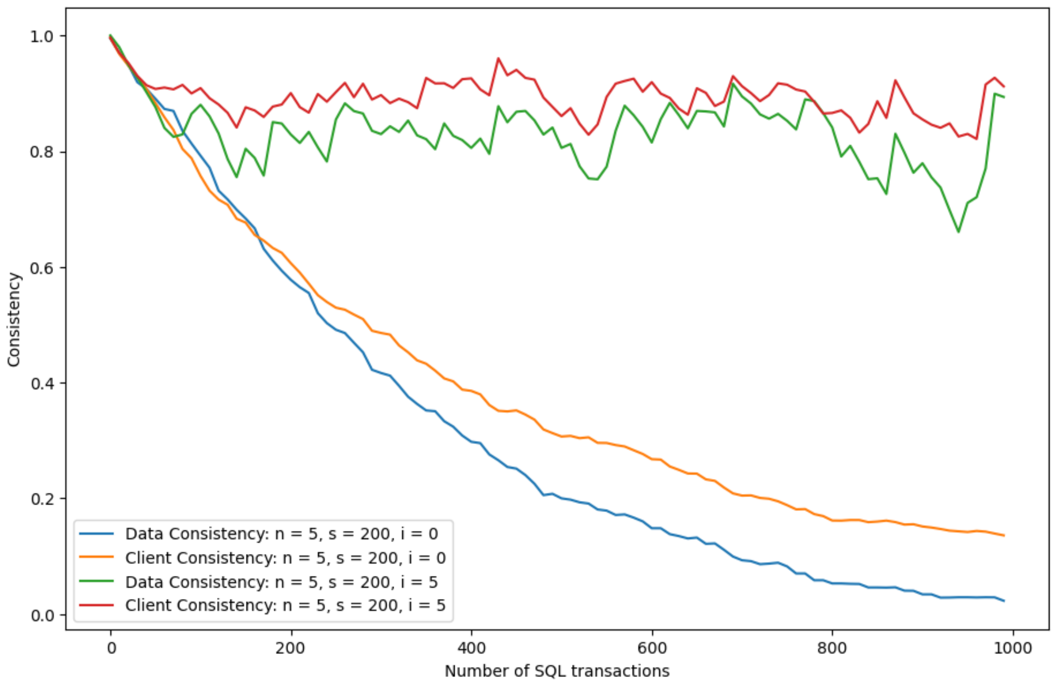

Figure 5.6: Data and client consistency with partial merge vs no merge

Observed in **figure 5.7**, the relationship between client and data consistency

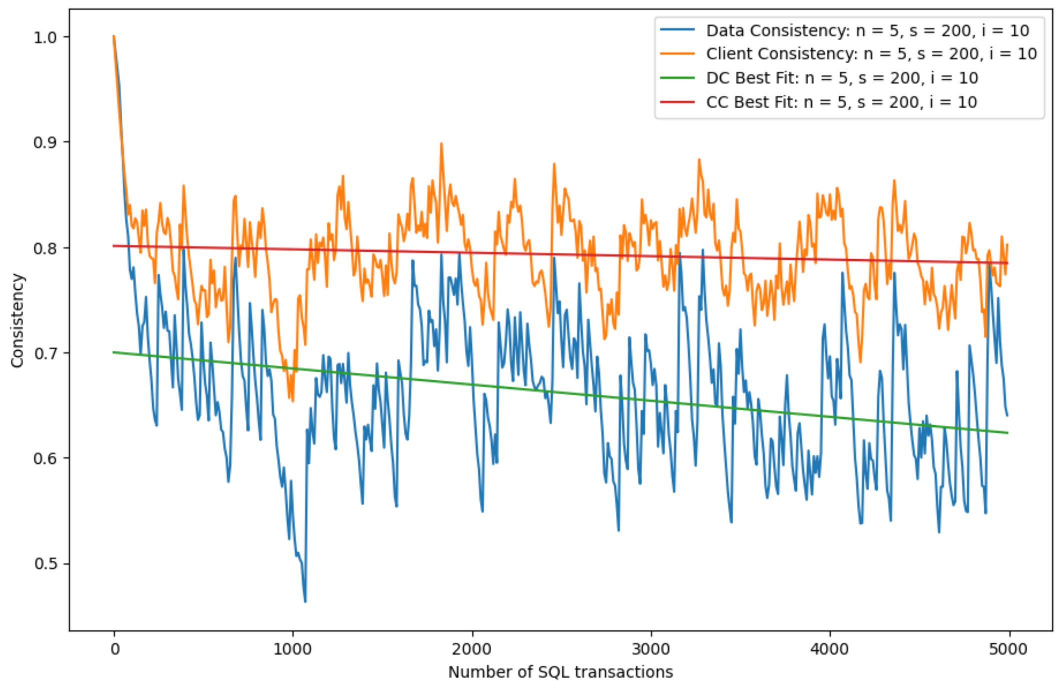

Figure 5.7: Client vs data consistency in a partially merging system

where client-side is greater (for n = 5) is also observed in this merge approach, just as in **figure 5.4**. Also true is that it seems that the data and client consistency generally follow the same high and lows throughout operations. We observe a big gap between best and worst consistency, where data consistency operates in a range of 0.475 - 0.8.

Figure 5.8 shows that as the number of nodes increase, the consistency of the system drops significantly. For two nodes, the consistency is always greater than 0.95 and seems to reach 1.0 very frequently. When increasing to 5 nodes, the average consistency decreases from 0.975 to 0.7, an average inconsistency increase of 1200 percent. Furthermore, the average consistency at n = 10 continuously drops from 0.4 to 0.2. However, this decrease is probably a best fit bias from starting at 1.0 consistency.

Finally, in **figure 5.9**, we look at how the frequency of random merges affects the data consistency. We observe that more frequent merges result in greater data consistency. There seems to be a non-linear relationship between average data consistency and the number of merges, as half the frequency of merges does not equal twice the inconsistency.

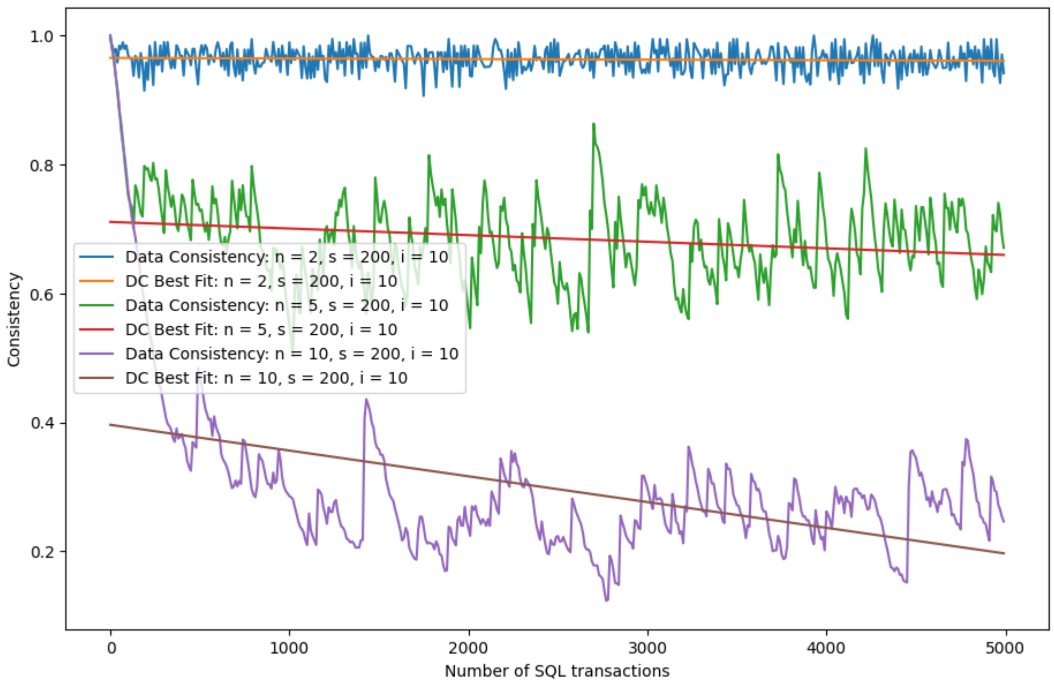

Figure 5.8: Data consistency based on number of nodes in a partially merging system

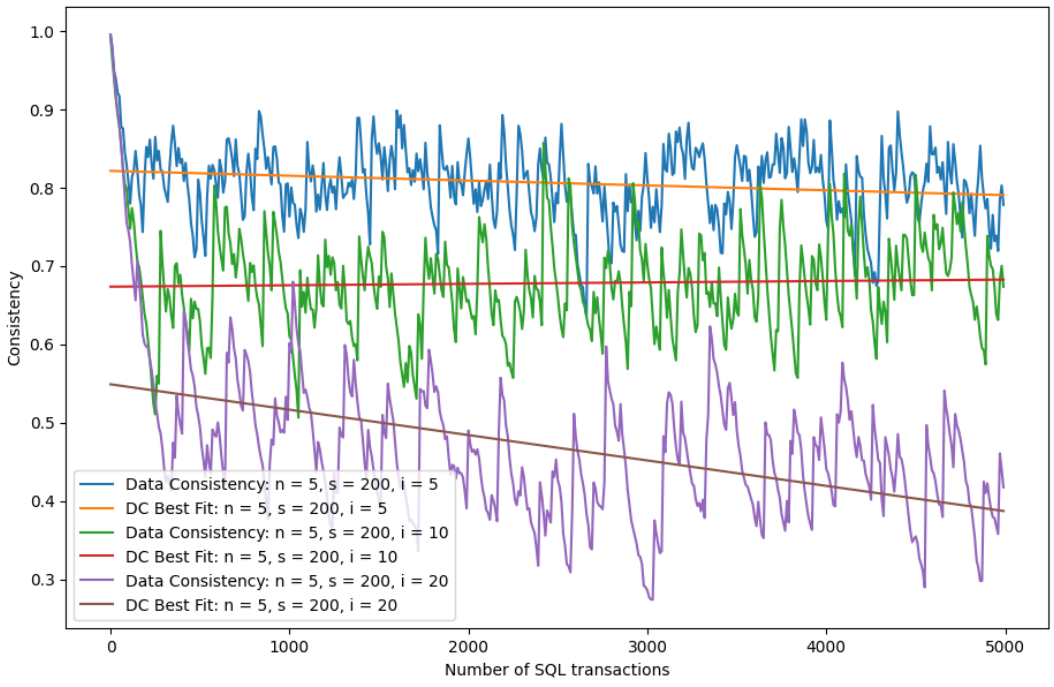

Figure 5.9: Frequency of random merges and data consistency

5.5 Time to Reach Eventual Consistency

Since consistency is dependent on merges, we look at the time needed to merge a system of a varying number of rows as a metric of how eventually consistent the system is [47]. We generate two replicas where some rows are different but where UUIDs are shared. The merge experiments use a complete merge that does not rely on deltas. We use a worst-case merge to avoid unfavorable comparisons, where all replicas share row UUIDs, but the rows are different. This method of merging compares each row to its counterpart from the other relation, and no rows get included just by being exclusive to a single replica. To minimize the effect of external factors and inconsistencies, each result given is an average of 10 runs.

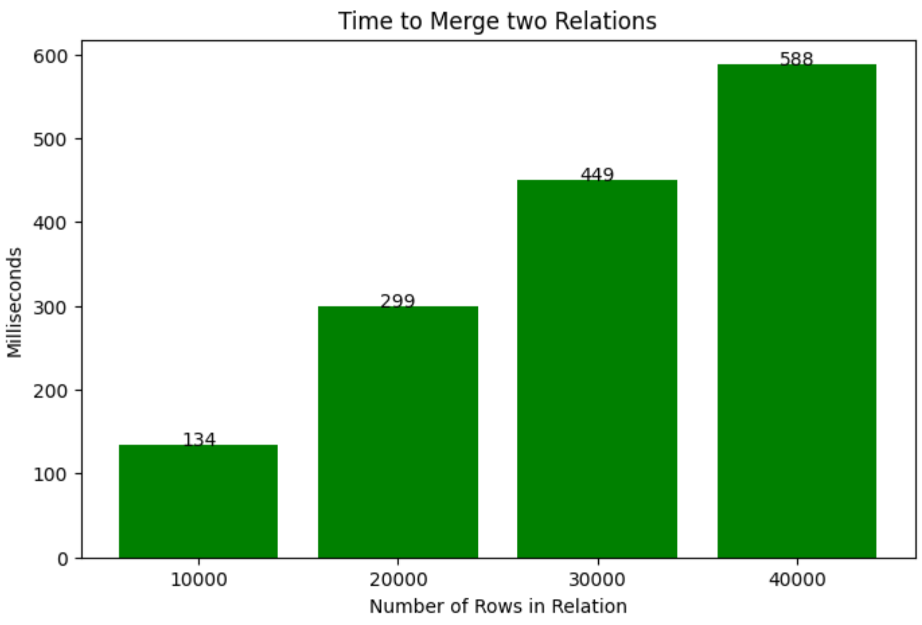

Figure 5.10: Merge performance based on number of rows

In **figure 5.10** there is a linear relationship between the increase in rows and the increase in time. Performance of merge, which also represents the time to eventual consistency, seems to scale linearly with the number of rows in the to-be merged relations or delta-relations.

5.6 Availability as Latency

We first distinguish the four different types of database transactions to be tested, select statements, insertions, deletions, and updates. We observe the time it takes to perform 10000 operations of each given type on a database with a varying number of rows. Deletions, selections, and updates use the row UUID as the index.

To perform the tests, we create two SQLite databases in python3[45], one with CRR applied and one without CRR. Then a number of rows given in the x-axis of **figure 5.11-5.14** were added to both databases. We generate 10 000 instructions to perform on the database and add them to a python list. We pre-generate instructions since integrating through the list of instructions has less overhead than generating instructions during execution. The results reflect the time in milliseconds to perform all 10000 instructions on a database.

We observe that for insertions in **figure 5.11**, deletions in **figure 5.12**, and updates in **figure 5.13**, there is a significant difference between the basic SQLite database and the database augmented to CRR. This difference ranges from 8 times the latency for updates, around 5 times for insertions, and between 3 and 4 times for deletion. For queries in **figure 5.14**, there is virtually no difference between the CRR-SQLite and the SQLite latency, where the latter is slower in some cases.

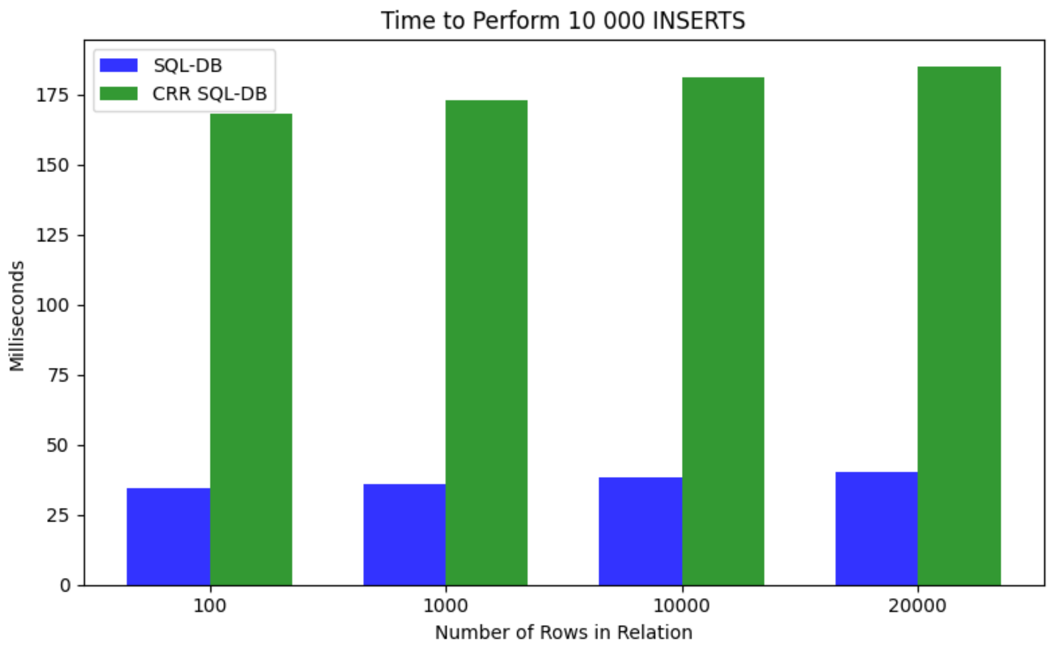

Figure 5.11: Insertion performance

Common to all CRR DB transactions is that they scale similarly to their SQLite counterparts. Deletions are the only transaction that shows any significant change with the number of rows in the relation changing.

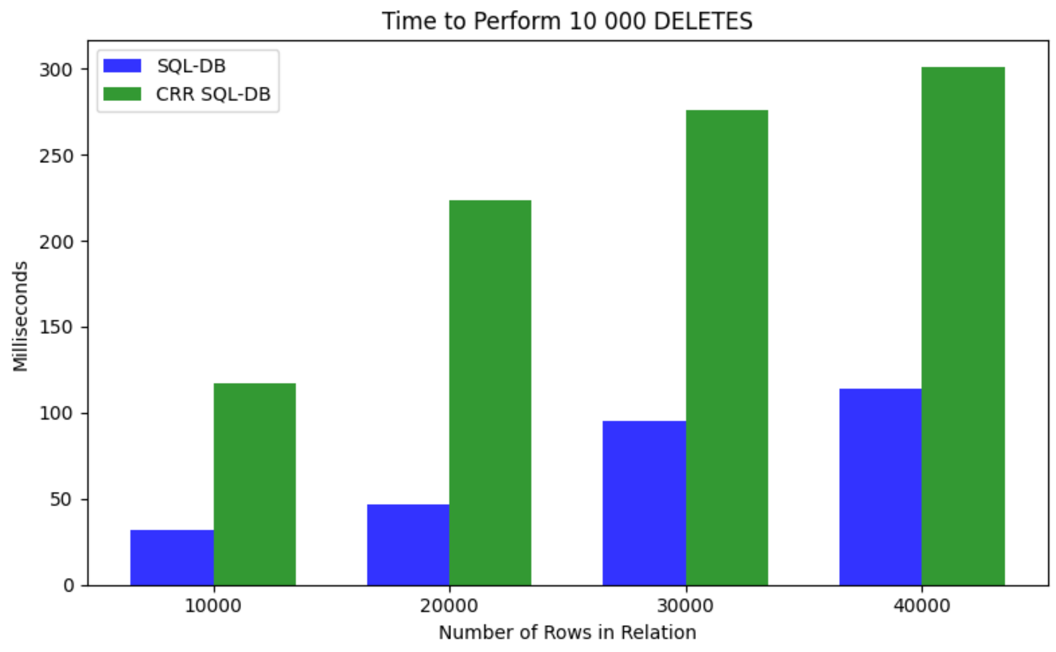

Figure 5.12: Deletion performance

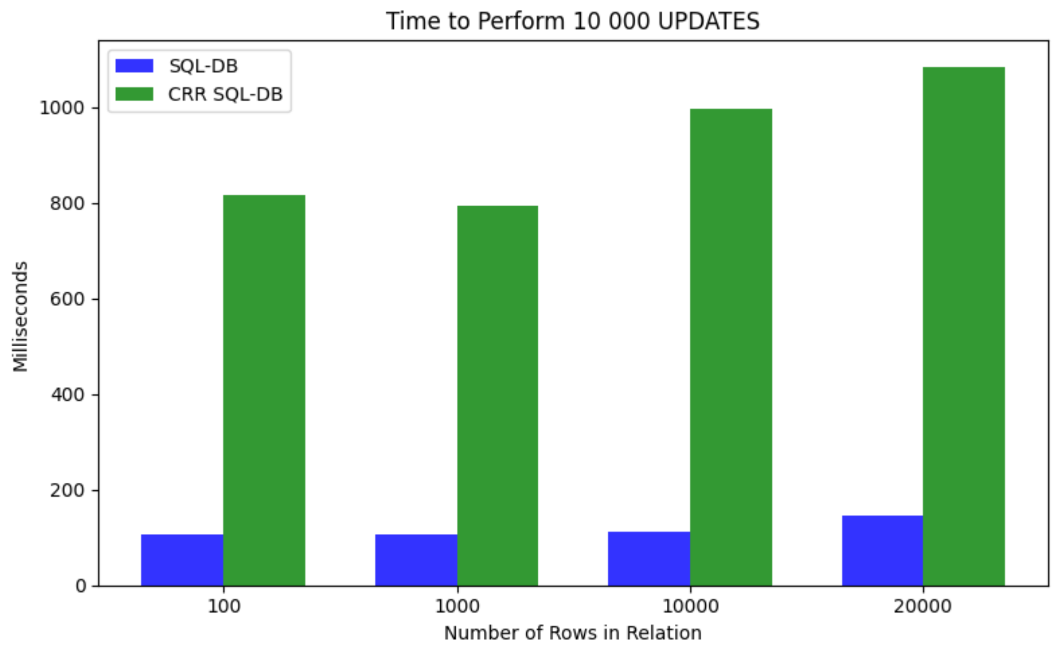

Figure 5.13: Update performance

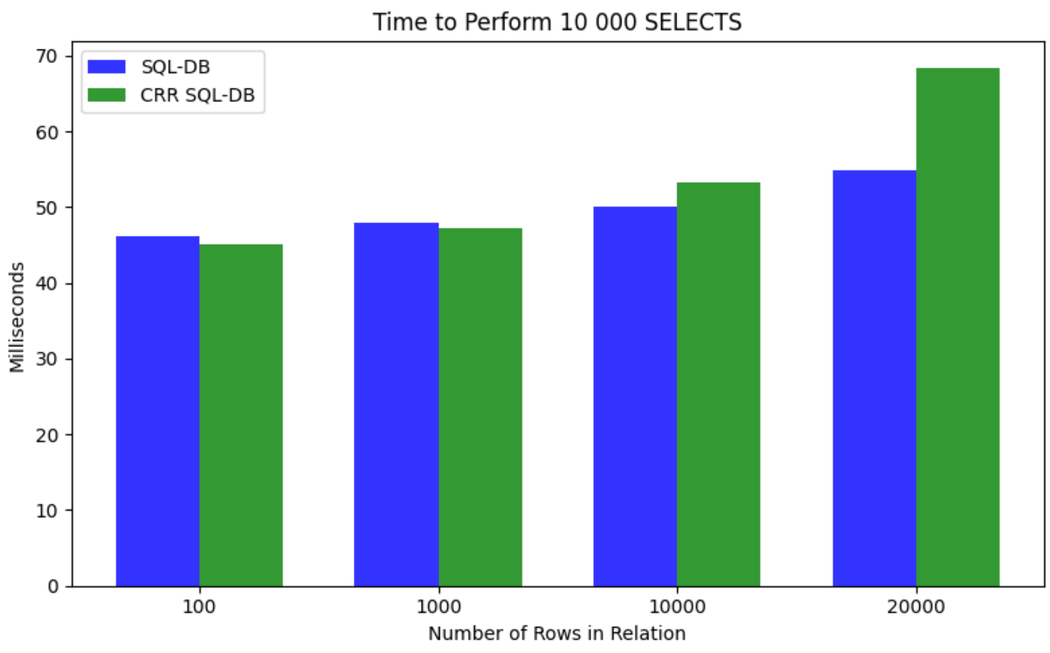

Figure 5.14: Select performance

6 Discussion and Evaluation

We start this chapter by reiterating the Main Issue and sub-requirements that lay the basis for this project:

- **Research Issue:** Is it possible to create a General Database Management System of Conflict-Free replicated relations?

Other Requirements:

- **CAP Properties:** The system should have strong eventual consistency from CRDTs, constant availability from having a locally replicated database, and be partitioned across multiple computers.

- **Generality:** The easiest way to ensure that a system is general is that it relies only on SQL and that there are augmentations that can be applied to any SQL system, new or preexisting, that augments it to a CRDT-DB without impeding the current functionality.

- **Performance:** While not a main goal, different performance metrics will be measured to make the current state of implementation comparable to other database management systems.

6.1 Evaluation of the Database Consistency

6.1.1 Client and Data Consistency

Experiments show how the system adheres to consistency with two models, data and client consistency. In a non-merging system, as seen in **figure 5.1**, we observe that the plots of data consistency remain equal across an expanding number of nodes. The correlation is accurate because if a user alters any row in the n replicas, that row is data inconsistent for all replicas. For example, given n nodes, if the nodes alter u of their s shared rows, those rows are inconsistent, and the consistency of the system is $\frac{s-u}{s}$. With the equation for data consistency being like this, the number of nodes n has no factor in the system's consistency.

For client consistency, we measure consistency from the average number of consistent rows in a node's local store. Given two nodes, it is clear that a consistent data instance must appear in one of the two nodes, as it must exist somewhere in the system. This observation would mean a worst-case client consistency of 0.5, or 1/n given n nodes, higher than the experiments show. The reasoning for this is because we measure worst-case client consistency. When a row is updated, the experiment counts the entire row as inconsistent, not just the updated value. The worst-case approach results in the possibility of a row being inconsistent in both replicas, and the system can have less than $\frac{1}{n}$ client consistency.

6.1.2 Eventual Consistency Through Full Merges

The expectation of eventual consistency is met. When fully merging the system, both models of consistency become a perfect 1.0. As seen in **figure 5.3**, a full merge operation allows the system to reset the drop inconsistency.

Merges do not need to happen periodically, and a user can merge as needed; in this case, a full merge will get the system to perfect consistency whenever the user wants it. This approach has the advantage of guaranteeing when the system is fully consistent. However, one might need a system that guarantees that consistency is at least a certain percentage. **Figure 5.5** highlights this possibility by showing the relationship between frequency of merges and consistency. We gauge that more frequent merges equal better overall consistency from the graph. As seen in **figure 5.4**, there is a slight deviation between data and client consistency with frequent merges. However, it is observed in **figure 5.1** that when the number of operations since the last merge increases, so does the gap between client and data consistency. With an increasing number of nodes, this gap gets smaller and smaller.

It is essential to factor in that a full merge is not without cost, as the system only supports a non-mutual merge; each node in the experiment merges with all other nodes to perform a full merge, requiring $n(n-1)$ total merges across all nodes. Therefore, it is increasingly expensive in terms of time and computing to fully merge a system as the number of nodes grows.

6.1.3 Eventual Consistency Through Partial Merges

As a counterpoint, we introduce a second method for merging, one where each node periodically merges with another singular node. This method counteracts some of the drawbacks of fully merging a system:

- A node does not need to tell other nodes to merge. Each node can merge independently instead of having to coordinate with the entire system.
- All nodes in the system do not have to be available.

With the second approach, the system is never entirely consistent, however, change to the database will eventually be spread to each other node, and there is, therefore, eventually consistency. **Figure 5.8** exemplifies these properties, where we see how the system reacts with an increasing number of nodes. For two nodes, the system needs only a single merge to propagate the data to each other node, so $n = 2$ is an outlier that offers high consistency for this model, akin to fully merging the system after every i operations. As the number of nodes increases, so does the number of merges needed to propagate the data across the system, and the chance of that row still being consistent when it is merged into every node decreases. Therefore, an increase in nodes results in lower consistency for both models.

While simply relying on randomized merges alone does not allow the system to guarantee that a fully consistent version of the database is stored locally, which is the case after fully merging the system, this approach can still guarantee that a system is at least a certain level of consistency. Even with the number of nodes reducing consistency, a system can counteract this by increasing the frequency of merges, as seen in **figure 5.9**. We theorize that through factoring size, number of nodes, and frequency of mergers, one can create a system with a specific average consistency.

A downside of this approach is its volatility. While a system can offer a specified average consistency, we still observe volatile outliers, where ranges can be as high as 40 percent data consistency, seen in **figure 5.9**. This might be partially due to a relatively low number of rows. Each operation has more impact with fewer rows, as seen in **figure 5.2**.

6.1.4 Consistency Method Comparison

To compare periodically merging the entire system to spreading updates through epidemic information dissemination [50], we define the following metric: updates per merge. For randomized merges, this is simply the value i, while for periodic full merges, it is $\frac{i}{(n-1)n}$. We see that for $n = 5$ and $i = 100$, the updates per merge are 5 in the full merge approach. When comparing the graph with these parameters, **figure 5.5**, to the corresponding partial merges one, **figure 5.9**, where $i = 5$ and $n = 5$ we observe the following:

- Fully merging results in a best-fit graph between 0.9 and 0.85
- Partially merging results in a best-fit graph 0.825

With these parameters, fully merging gives the best results in terms of data consistency per merge in the system. However, the partial merge has the advantage of being better equipped for a peer-to-peer approach as it does not require a fully available network. Both approaches have advantages, and they are not mutually exclusive; one can combine them as the system needs. Most importantly, they both highlight what eventual consistency is and that this project achieves it.

6.1.5 Time to Reach Eventual Consistency

We observe a near-linear growth in time spent merging when increasing the number of rows in a relation or relation-delta. Given two relations of sizes s_1 and s_2, the time complexity of $O(s_1 * log(s_2))$ is expected when joining on primary keys that are used to order a B+tree. Because for each row s_1 in the first relation, a lookup must be performed in the B+tree of the second relation. Each of the s_1 lookups has a time complexity of $O(log(s_2))$.

Another distinction to make when it comes to time to reach eventual consistency is that we can define the size of the delta as the number of updated rows since the last merge. As we know that the merge function seemingly scales linearly with the number of updates, we can further say that increasing the frequency of merges should not provide much additional overhead. Using **figure 5.5** as an example, increasing the frequency of merges increases the average consistency, but while the number of merges double, the number of rows that have been updated per relation are halved, which means a merge of i = 50 should only take half the time of an i = 100 merge. Defining scalable consistency to mean a system that achieves consistency without latency overhead, we now know the following to be true: a DBMS of CRRs provides scalable consistency when using a delta-CRDT model.

An argument against this is that smaller deltas equals a larger fraction of total time in the merge-function devoted to setup and communication. While the previous statement is true for delta sizes over 10000, it might not be the case when dealing with small databases and deltas.

Merging a system of CRRs does not scale only with the number of rows in a relation but also the number of nodes because the system must propagate the data more times. The merges done in the experiment were not the most efficient, as they required every node to merge with all other nodes. A more elegant approach would have one node merge with all others and then have all other nodes merge with that node, giving the number of merges as $(n-1)+n$. However, as the last merge can happen concurrently because all nodes have to read from the merged nodes delta to merge it into their own database, one can argue that it only takes the time of one merge, we then define the time of a complete merge as $t(n-1)+t(1)=t(n)$. This approach merges scales linearly with the number of nodes.

A more complicated way of fully merging the system is by having the nodes form a slightly different join semilattice to what we see in CRDT-theory [26] that considers that all merges are non-mutual merge-pulls. Assuming all nodes generate a delta δn, each node can update their local relation without altering their delta. Given nodes $n_1, n_2, \ldots .n_n$, each node n_x can merge with the delta δn_{x+1}, then δn_{x+2} and after n-1 steps, each node should have the same state. If we create a new delta with all the newly updated rows, i.e., $\delta(n_x, n_{x+1})$ after the first merge, then it only takes $log(n)$ merges to merge the system fully. The logarithmic scaling is because each consecutive merge will carry twice the information of the last, at the cost of more complex communication.

All the previous approaches aim for perfect data consistency. However, if one aims only to achieve perfect client consistency for a given node, one can merge a single node with all others, then that node has perfect client consistency. The number of merges needed is $n-1$.

6.2 Evaluation of Database Availability

6.2.1 Beyond Always Availability

What has always been the case as this is a local-first application is that every transaction yields a response, as it performs it on a local datastore. Short of a hardware error or implementation fault, there is no reason that a transaction performed should not have ACID properties. Throughout testing and experiments, with multiple hours and millions of transactions, there was never a case of the local store not being available.

Experiments are therefore done by testing availability beyond this, as proposed in [10]. Here a transaction's latency is also a factor in its availability, where more throughput means a more available system. It is also important to note that throughput performance was not a primary concern during this thesis implementation. The experiments do not reflect an ideally optimized CRDT-DBMS in SQLite but a proof-of-work application.

6.2.2 Performance of SQL Transactions

The expectations for insertions, deletions, and updates were that they would take more than twice the time of their base SQLite counterparts. The result shows in the experiments, where one can see the overhead caused by the triggers propagating data to the CRR layer and generating timestamps. Queries behave as expected, taking around the same time as their SQLite counterparts, which is because queries are only affected by the AR layer relation, which is identical to the base SQLite relation.

To understand the case of the low throughput, we look at deletions, which have the best throughput compared to their SQLite counterparts. Deletions differ by relying very little on other meta-data tables, as it does not use the time-table. Also, it has fewer SQL CASE-statements. The theory of what causes the reduction in throughput is then the following:

- The time-functionality and simulated nanosecond time precision have to write to disk and commit every transaction.

- SQLs CASE-statement is slow, and calling it multiple times results in reduced throughput.

The performance of transactions is not all negative, as they all scale as well as the basic SQLite transactions. There is little or no difference in almost all cases when increasing the number of rows. Deletions are the worst-scaling ones, but this is also the case for the SQLite deletions, and while not as good as the others, it is still better than linear scaling.

In summary, when viewing availability as latency of transactions, the system is less locally available than an unaugmented SQLite database, except when it comes to queries where the availability is the same. The availability scales with an increasing size of the local store.

6.3 Tolerance to Partitioning

While it has not been the main focus of the experiments, tolerance to network partitioning is worth bringing up as it is part of the CAP-theorem. A distributed system of CRR databases would rely little on communication between nodes. With the solid fundamentals of state-based CRDTs, the system needs no guarantee of messages between relations being successful for the system to work.

Many distributed systems rely on applying transactions at least once and no more than once, as with operation-based CRDTs from **section 2.3.1**. However, we designed this system using state-based CRDTs, so multiple merges between nodes have no adverse effects. If two nodes merge more than once, the system remains functioning, meaning dropped merge-messages can be redone at no risk.

6.4 Unexpected Behaviour in Experiments: Declining Data Consistency

An observation made in many experiments involving data consistency seems to be a slow decline when looking at the average best-fit graph. There are multiple potential reasons for this.

Firstly, when doing the partial merge method, all experiments were initialized to a perfect consistency. The initial consistency means that the best fit will be front-loaded and gives the illusion of overall consistency decline when in reality, the consistency drops to a certain point and then stabilizes. Initializing consistency to zero would probably yield the opposite result.

Secondly, there is the issue of full periodical merges. In **figure 5.5**, one sees that the graph gradually declines. To understand the reasoning for this, we look at the setup and how we calculate data consistency. Initially, each node has s rows in both their AR and CRR layer. To ensure that an increasing number of rows does not affect the consistency, as seen in **figure 5.2**, we cap the number of rows to s, and all insertions are only re-insertions of deleted rows.

Limiting the number of rows means that while initially there are s rows in the AR layer, as the experiments go on, there are most likely fewer, as some have been deleted and not yet re-inserted, and therefore only exist in the CRR layer. As a lesser number of rows equals a more significant inconsistency effect of each update, see **figure 5.2**, since the number of rows has slightly decreased since the initialization, so does the data consistency. The decreasing number of rows causes the effect of gradually decreasing data consistency when periodically fully merging.

6.5 Applying Results to Example

Continuing with the running example of two architectures from **section 3.6**, a peer-to-peer one and a server-client one, we look at how the results found in the experiments can predict issues and performance of such systems.

With two different merge methods, randomly merging or merging with all other available nodes. The one that applies most to a peer-to-peer system is the random-merge one. We know that one of the biggest killers of data consistency is the number of nodes in the system, as seen in **figure 5.8**. To guarantee a certain consistency, we calculate a predicted merge frequency to guarantee a consistency great enough for the application's need. When a node needs even greater client consistency, it can merge with all other known nodes. This way a overlying application can have a more consistent database instance when needed. As the total number of rows increases, the number of merges needed to provide the same consistency decreases. However, one might want a model that calculates not based on a percentage but on the number of inconsistent rows. In both cases, there might be a need for periodical recalibration of merge frequency.

For the more standard server-client architecture, the server periodically merges data from all clients, so its internal store is up-to-date, and the server has client consistency. The client nodes can merge with the server if they need the latest version of their replicated relations.

These examples highlight the versatility of a general database of conflict-free replicated relations, and this example serves as one of many potential use-cases. Circling back to the motivation, local-first approaches allow for the local store to be constantly available and the strong eventual consistency coming from the underlying CRDTs, and further highlighted in the experiments. These features sum up to a database that can function reliably in an unreliable environment, both physically and in terms of network reliability.

6.6 Advantages, Disadvantages, and Potential Improvements

The following are the advantages and disadvantages of the system as it exists in its current form.

Advantages:

- Strong eventual consistency.
- Constant availability in terms of ACID transactions.
- Selects are just as fast as in a normal SQLite database.
- Applications only rely on SQLite to uphold CRR functionality.

Disadvantages:

- There is a performance penalty on updates, deletions.
- Merging two nodes scales linearly with the number of rows that they have updated.
- Merging multiple nodes needs complex communication to achieve logarithmic scaling.
- Extra disk usage because of the two-layer database.

All of the disadvantages are just apparent in the current system. Some of the disadvantages may not be hard limitations stemming from the design. The performance of the DBMS was not a central concern in the current implementation, and following implementations can have a greater focus on this. There is plenty of data concerning database optimization [51][52], and much of this has possible applications here. Ideally, one could force commonly accessed relations to be stored in memory to limit the time needed to access them and optimize the queries within the triggers to ensure that the tables are only accessed once.

The advantages of the database are, as promised, constant availability and strong eventual consistency. While there is a performance overhead, it is essential to remember what an application gains at that cost. Comparing a database of CRRs to a standard SQLite database might be considered an unfair comparison. Unaugmented SQLites operations are only local, while CRR augmented

SQLite propagates transactions to any other node in a system. Therefore, this comparison is akin to comparing an update performed with a consensus algorithm to that of updating a local store.

When comparing this system to consensus algorithms [11][12], we see a clear advantage. Consensus algorithms generally do not scale beyond 5 nodes. In a distributed system based on this project's implementation, the number of nodes plays no factor in the time needed to apply a transaction, only the time needed to merge all nodes.

6.7 Future Work

6.7.1 Integrety Constraints

A vital part of relational databases is a user's ability to define constraints on particular data instances. We explore integrity constraints in this project in the prototype implementation of a reference integrity constraint handler.

The current iteration of constraint handling is more of a proof-of-work, where one can see the possibility of resolving constraints within SQLite. It lacks some functionality, especially when it comes to ordering constraint repairs across more than two nodes. The current system only separates local and foreign transactions and undo the foreign ones. A more elegant solution would undo the last update, insertion, or deletion, not based on which node did the transaction but based on which transaction has the latest timestamp.

6.7.2 Counter-CRDT as Row Attribute

Currently, the inconsistency of intra-row attributes is resolved purely by using timestamps as in an LWW-register [3]. However, as supported by [5], there is also the possibility of the data within rows being defined as other CRDTs. For example, instead of an integer, a value in a pure-SQL CRR-DB that symbolizes the number of a ware in stock can be depicted with a PN-counter CRDT [3]. As multiple instances of counter-CRDTs can merge by design, it absolves that attributes need for a timestamp.

Implementing this using SQL is a challenge. The expectation is that CRDT-values have to be defined in the database augmentation process. For counters, every AR layer counter-integer needs a plus and minus value in the CRR layer that makes the AR layer value when resolved. In addition, merges on counter values must be specified to take the max of both the values in the CRR layer and subtract the minus from the plus value to apply in the AR layer.

Updates are complicated, as SQLite does not have an increment function. It does support a syntax to increment as "*UPDATE Products SET Price = Price + 50*"*;* however, this is just an update based on a current value. This would mean all updates on a counter value must be interpreted as increments or decrements. Changing a 10 to 25 would mean increasing the plus counter in the CRR layer by 15 since that is how much the overlying value increments.

6.7.3 Future master-project: User-applications and merge-push

This implementation and project have described a prototype. We have made some assumptions that we have not tested in practice. An advanced simulation of a user application that uses the SQLite DB of CRRs would allow for more extensive testing and observation than what this project has done.

There are plans for a future master thesis following up on this thesis's findings, technology, and research. While it is currently not clear what this project will entail, there is a strong possibility that this project will apply the technology of this project to a further realized distributed system. An extension to the functionality of this project that a future thesis can explore is the ability to not only merge-pull from another node, but also merge-push into another node, or even perform mutual merges.

6.7.4 CRR in the Greater Database Landscape

While CRR aims to be generally applicable and usable in many different devices and architectures, it is important to explore where it stands when compared to modern database management system. We expect that it is not suited for big data, such as Apaches Spark SQL [53], as big data favors multiple connections simultaneously accessing data. SQLite handles multiple concurrent connections to a single database poorly.

Application developers typically have to make a choice between embedded databases, like SQLite [4], eXtreme DB [54], Realm DB [55], and Berkley embedded DB [56] or client-server databases like Redis [57], Cassandra [58], MySQL [35] and PostresSQL [36]. A DB of CRRs offers a bit of both, by being an embedded database that can also be part of a network. Future exploration can see how the embedded features compare to other embedded databases, and how the networking compare to other server-databases.

6.7.5 SQLite-Extensions Revisited

As stated in the design section, one can create and call user-defined functions in SQLite. While these functions offer little when it comes to helping to implement CRR, as having them serve as wrapper functions would mean using a non-SQL syntax to query the database, there are other options hidden within the possibilities of SQLite extensions: The ability to change how the database interacts with the underlying OS. This OS interface is commonly referred to as "VFS" [41, chapter 7].

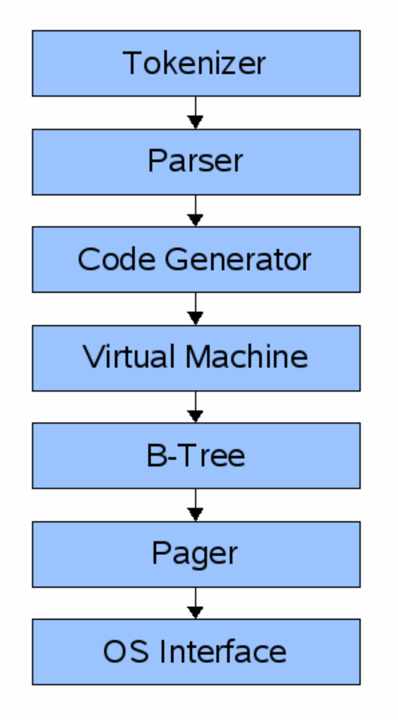

Figure 6.1: SQLites Internal Orginization[59]

Figure 6.1 shows a high-level overview of SQLite's internal organization. As seen, the final step is the OS interface, whose job is to write data to the SQLite .db file. VFS-extensions allow a user to change how SQLite stores data. An example of this is ZIPVFS [59], which compresses the data before writing it to disk and then uncompresses it as it is being read and passed up the structure.

A similar process might be possible to achieve CRR, as the program writes data to disk, it pads it with the necessary CRR meta-data. Moreover, as it is read, this CRR meta-data determines the data state. When inserting new data, the data is padded with a CL and timestamp, and when the data is deleted, the CL is incremented. When the data is updated, new timestamps are generated and stored with the data. Instead of queries being performed on an AR layer, the meta-data will just be stripped away as it is read and passed up the stack, with the CL determining the row's state as it is read.

It is important to note that this is not a realized design, and there might be unexpected factors at play that make it not realizable. There is also the challenge of merging relations when they are stored like this, and the feasibility of this approach relies on a brand-new merge method working directly on the data from disk.

This approach loses the advantage of not relying on extensions or anything that is not basic SQLite. However, all SQLite query syntax remains the same. It possibly gains advantages in terms of performance. The advantage is due to the current implementation doing multiple passes through the entirety of SQLite internal organization stack per transaction, while the VFS approach only needs one.

7 Conclusion

The database management system being general and accessible were some of the primary focuses of this thesis. Generality would entail that a user or application can use it without any changes to how it would use a standard database. A user or user application should be able to interact as usually with the database, while the DBMS silently and automatically augments the transactions to CRR transactions.

This thesis aims to achieve generality by relying only on SQL. SQL has many limitations that we overcame to achieve this, but also many unforeseen advantages that helped us overcome its limitations. Using SQLite, we have implemented methods to propagate data through the layers using triggers, new time functionality, and a way to merge the system is available as an SQL script. The only limit to what we cannot do in SQL is communication between other nodes, which is a concession this design makes.

When viewing generality as being able to use the database as one would any SQL database while also achieving what a database needs for Conflict-Free Replicated Relations, it is clear that the database is general. Any user could take an existing database and apply the principles used in this database to create a local-first database that functions without a connection to an overlying network while automatically having all the data needed to merge with another replica.

To further highlight the advantages of a general local-first database approach, we had a closer inspection of the CAP theorem and its application to local-first software. CRDTs provide the underlying methods to build CRRs, so from the very start, its ability to achieve consistency is unquestionable. CRR acts as a fusion of CL-set and LWW-register and can resolve any consistency conflict between replicas. In many ways, the consistency experiments only highlighted this and provided little-to-no new information. However, the expected confirmation offered by the experiments acts as testament to CRR's versatile and strong consistency capabilities.

The experiments also highlighted the versatility of an approach like this. The system is not only generally applicable to any overlying application but also many overlying system architectures. We saw how consistency could function in peer-to-peer and server-client networks. One can apply many different merging methods to fit the needs of the system. Moreover, no matter the system complexity, any information will eventually make its way to all nodes. From the standpoint of network architecture, this project's lack of inter-nodal communication could be considered an upside. Systems can define for themselves how nodes communicate, and this project's contribution wraps it in an easy-to-use package.

This project achieved a Database of Conflict-Free Replicated relations through only using SQL. This approach's replicability makes it generally accessible to any application and device able to use SQLite. We achieved it through a prototype that highlights the features and functionality. These features and functionalities are already employed in a more complete DBMS, SYNQlite, and are applicable to many more applications and systems because of their generality.

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*, 2019, pp. 154–178 (cit. on pp. i, 2, 8).

[2] A. Fox and E. A. Brewer, "Harvest, yield, and scalable tolerant systems," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, IEEE, 1999, pp. 174–178 (cit. on pp. i, 1).

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," 2011 (cit. on pp. i, 2, 10, 16, 68).

[4] *SQLite Home Page*. [Online]. Available: `https://www.sqlite.org/index.html`, (accessed: 04.05.2020) (cit. on pp. i, 3, 69).

[5] 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, 2020 IEEE World Congress on Services*, 2020 (cit. on pp. i, 2, 4, 10, 16–19, 21, 24–27, 35, 68).

[6] *Ecto, relational mapper for elixir*. [Online]. Available: `https://hexdocs.pm/ecto/Ecto.html`, (accessed: 04.05.2020) (cit. on pp. i, 21).

[7] W. Yu and S. Rostad, "A low-cost set CRDT based on causal lengths," in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–6 (cit. on pp. i, 10).

[8] T. Ylonen, C. Lonvick, *et al.*, *The secure shell (SSH) protocol architecture*, 2006 (cit. on pp. i, 27).

[9] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-hall, 2007 (cit. on pp. i, 46).

[10] E. Brewer, "CAP twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012 (cit. on pp. i, 46, 47, 64).

[11] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001 (cit. on pp. i, 9, 68).

[12] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319 (cit. on pp. i, 9, 68).

[13] V. B. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, “Verifying strong eventual consistency in distributed systems,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017 (cit. on p. 2).

[14] M. Kleppmann and A. R. Beresford, “A conflict-free replicated JSON datatype,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, 2017 (cit. on p. 2).

[15] P. Grosch, R. Krafft, M. Wölki, and A. Bieniusa, “AutoCouch: a JSON CRDT framework,” in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–7 (cit. on p. 2).

[16] M. Van Alstyne, E. Brynjolfsson, and S. Madnick, “Why not one big database? principles for data ownership,” *Decision Support Systems*, vol. 15, no. 4, pp. 267–284, 1995 (cit. on p. 2).

[17] A. M. Al-Khouri *et al.*, “Data ownership: Who owns “my data”,” *International Journal of Management & Information Technology*, vol. 2, no. 1, pp. 1–8, 2012 (cit. on p. 2).

[18] J. R. Groff, P. N. Weinberg, and A. J. Oppel, *SQL: the complete reference*. McGraw-Hill/Osborne, 2002, vol. 2, pp. 25–30 (cit. on p. 3).

[19] B. W. Kernighan and D. M. Ritchie, *The C programming language*. 2006 (cit. on p. 3).

[20] E. A. Lee, “Embedded software,” in *Advances in computers*, vol. 56, Elsevier, 2002, pp. 55–95 (cit. on p. 3).

[21] L. Junyan, X. Shiguo, and L. Yijie, “Application research of embedded database sqlite,” in *2009 International Forum on Information Technology and Applications*, IEEE, vol. 2, 2009, pp. 539–543 (cit. on pp. 3, 22).

[22] T. Colburn, “Methodology of computer science,” *The Blackwell guide to the philosophy of computing and information*, pp. 318–326, 2004 (cit. on p. 4).

[23] E. F. Codd, *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990, ch. 1, pp. 1–3 (cit. on p. 8).

[24] T. P.-W. Chris Wanstrath P. J. Hyett and S. Chacon, *GitHub*. [Online]. Available: `https://github.com/`, (accessed: 04.11.2020) (cit. on p. 8).

[25] C. Baquero, P. S. Almeida, and A. Shoker, “Making operation-based CRDTs operation-based,” in *IFIP International Conference on Distributed Applications and Interoperable Systems*, Springer, 2014, pp. 126–140 (cit. on p. 10).

[26] V. K. Garg, *Introduction to lattice theory with computer science applications*. Wiley Online Library, 2015 (cit. on pp. 10, 63).

[27] H. Marouani and M. R. Dagenais, “Internal clock drift estimation in computer clusters,” *Journal of Computer Systems, Networks, and Communications*, vol. 2008, 2008 (cit. on p. 13).

[28] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte, “An optimized conflict-free replicated set,” *arXiv preprint arXiv:1210.3368*, 2012 (cit. on p. 14).

[29] P. S. Almeida, A. Shoker, and C. Baquero, “Delta state replicated data types,” *Journal of Parallel and Distributed Computing*, vol. 111, pp. 162–173, 2018 (cit. on pp. 17, 25).

[30] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão, “Efficient synchronization of state-based CRDTs,” in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, IEEE, 2019, pp. 148–159 (cit. on p. 17).

[31] P. Leach, M. Mealling, and R. Salz, “A universally unique identifier (uuid) urn namespace,” 2005 (cit. on pp. 18, 29).

[32] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, “Logical physical clocks,” in *International Conference on Principles of Distributed Systems*, Springer, 2014, pp. 17–32 (cit. on pp. 19, 32).

[33] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196 (cit. on p. 19).

[34] *The elixir programming language*. [Online]. Available: `https://elixir-lang.org/`, (accessed: 04.05.2020) (cit. on p. 21).

[35] *MySQL Homepage*. [Online]. Available: `https://www.mysql.com/`, (accessed: 04.05.2020) (cit. on pp. 22, 69).

[36] *PostresSQL Homepage*. [Online]. Available: `https://www.postgresql.org/`, (accessed: 04.05.2020) (cit. on pp. 22, 69).

[37] *MySQL: CREATE PROCEDURE and CREATE FUNCTION*. [Online]. Available: `https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html`, (accessed: 04.05.2020) (cit. on p. 22).

[38] *SQLite: Most deployed*. [Online]. Available: `https://www.sqlite.org/mostdeployed.html`, (accessed: 04.05.2020) (cit. on pp. 22, 29).

[39] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, “A survey and comparison of relational and non-relational database,” *International Journal of Engineering Research & Technology*, vol. 1, no. 6, pp. 1–5, 2012 (cit. on p. 22).

[40] R. Cochrane, H. Pirahesh, and N. Mattos, “Integrating triggers and declarative constraints in SQL database systems,” in *VLDB*, Citeseer, vol. 96, 1996, pp. 3–6 (cit. on p. 22).

[41] J. Kreibich, *Using SQLite*. " O’Reilly Media, Inc.", 2010 (cit. on pp. 22, 23, 25, 34, 70).

[42] J. Hamano and L. Torvalds. (). “Git –local-branching-on-the-cheap,” [Online]. Available: `https://git-scm.com/`. (accessed: 03.11.2020) (cit. on p. 28).

[43] *International Workshop on Advanced Data Systems Management, Engineering, and Analytics (MegaData)*. [Online]. Available: `https://adbis2021.cs.ut.ee/megadata/`, (accessed: 28.05.2020) (cit. on p. 39).

[44] S. Singh, *An introduction to availability testing*. [Online]. Available: `https://www.qualitestgroup.com/white-papers/introduction-availability-testing/`, (accessed: 04.05.2020) (cit. on p. 42).
[45] *Python programming language*. [Online]. Available: `https://www.python.org/`, (accessed: 04.05.2020) (cit. on pp. 45, 56).
[46] B. J. Cox, "Object-oriented programming: An evolutionary approach," 1986 (cit. on p. 45).
[47] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior," in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, 2011, pp. 1–6 (cit. on pp. 47, 55).
[48] *SQLite AsyncVFS*. [Online]. Available: `https://sqlite.org/asyncvfs.html`, (accessed: 04.05.2020) (cit. on p. 47).
[49] L. Nan and P. Wonka, "Polyfit: Polygonal surface reconstruction from point clouds," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2353–2361 (cit. on p. 50).
[50] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "Epidemic information dissemination in distributed systems," *Computer*, vol. 37, no. 5, pp. 60–67, 2004 (cit. on pp. 52, 62).
[51] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi, "Learning state representations for query optimization with deep reinforcement learning," in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, 2018, pp. 1–4 (cit. on p. 67).
[52] 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, pp. 643–668, 2018 (cit. on p. 67).
[53] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394 (cit. on p. 69).
[54] *eXtremeDB*. [Online]. Available: `https://www.mcobject.com/extremedbfamily`, (accessed: 04.05.2020) (cit. on p. 69).
[55] *RealmDB Homepage*. [Online]. Available: `https://realm.io/`, (accessed: 04.05.2020) (cit. on p. 69).
[56] M. I. Seltzer, "Berkeley db: A retrospective.," *IEEE Data Eng. Bull.*, vol. 30, no. 3, pp. 21–28, 2007 (cit. on p. 69).
[57] *RedisDB Homepage*. [Online]. Available: `https://realm.io/`, (accessed: 04.05.2020) (cit. on p. 69).
[58] A. Lakshman and P. Malik, "Cassandra: Structured storage system on a p2p network," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, 2009, pp. 5–5 (cit. on p. 69).

[59] *SQLite ZipVFS*. [Online]. Available: `https://www.sqlite.org/zipvfs/doc/trunk/www/howitworks.wiki`, (accessed: 04.05.2020) (cit. on p. 70).

A
Appendix: SYNQlite paper

Augmenting SQLite for Local-First Software

Iver Toft Tomter and Weihai Yu

UIT - The Arctic University of Norway, Tromsø, Norway
`weihai.yu@uit.no`

Abstract. Local-first software aims at both the ability to work offline on local data and the ability to collaborate across multiple devices. CRDTs (conflict-free replicated data types) are abstractions for offline and collaborative work that guarantees strong eventual consistency. RDB (relational database) is a mature and successful computer industry for management of data, and SQLite is an ideal RDB candidate for offline work on locally stored data. CRRs (conflict-free relations) apply CRDTs to RDB data. This paper presents our work in progress that augments SQLite databases with CRR for local-first software. No modification or extra software is needed for existing SQLite applications to continue working with the augmented databases.

Keywords: Data replication, eventual consistency, conflict-free replicated data types, conflict-free replicated relations, edge computing

1 Introduction

Local-first software suggests a set of principles for software that enables both collaboration and ownership for users. Local-first ideals include the ability to work offline and collaborate across multiple devices [7].

SQLite is an open source RDB (relational database) engine. It is an ideal candidate for local-first software, because its operation does not rely on network connectivity. Citing its homepage[1]: "SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day."

One of the main challenges of supporting local-first software is the general limitation of a networked system, as stated in the CAP theorem [3, 5]: it is impossible to simultaneously ensure all three desirable properties, namely (C) consistency equivalent to a single up-to-date copy of data, (A) availability of the data for update and (P) tolerance to network partition.

CRDTs (conflict-free replicated data types) [10] emerged to address the CAP challenges. With CRDT, a site updates its local replica without coordination with other sites. The states of replicas converge when they have applied the same set of updates (referred to as *strong eventual consistency* in [10]).

CRRs (conflict-free replicated relations) apply CRDTs to RDBs [12]. In [12], we reported a CRR prototype that was built on top of an ORM (object-relation mapping)

[1] `https://sqlite.org`

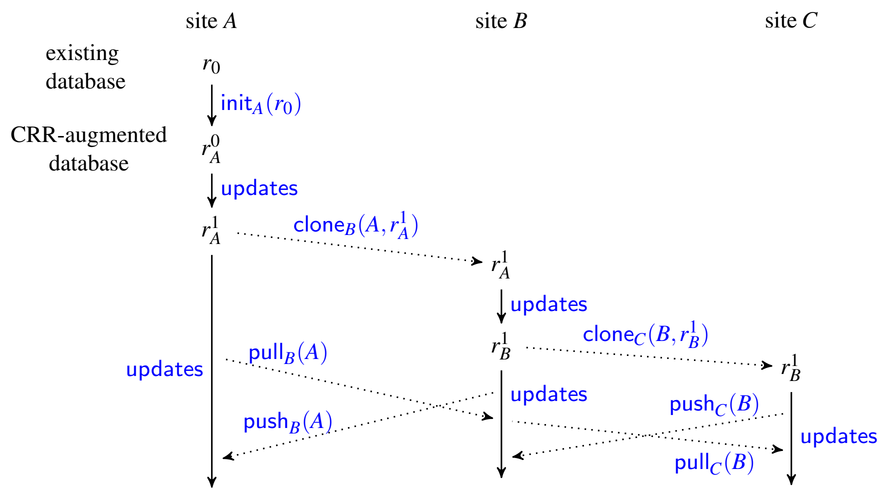

Fig. 1. A scenario of asynchronous database updates

called Ecto[2]. Applications are therefore limited to those using the particular ORM. Unfortunately, Ecto does not support SQLite in the latest versions.

In this paper, we report our work-in-progress implementation that augments SQLite databases with CRR support. With a single command, we augment an existing SQLite database with CRR. All applications using the database, including the `sqlite3` shell[3], continue to work without any modification. We can then clone the CRR-augmented database to different sites. We can query and update the database instances at different sites independently. We can synchronize the instances when the sites are connected. We are not locked in, though. We can easily drop the CRR-augmentation on any of the database instances without losing any of the original database features.

Fig. 1 shows a scenario of using our software. Initially we have a SQLite database instance r_0 at site A. We run $\mathsf{init}(r_0)$ to augment r_0 to r_A^0 with CRR support. We then apply some updates that lead to r_A^1. Now at site B we run $\mathsf{clone}(r_A^1)$ to get a clone of the database instance. Independently, we make updates on the local instances at sites A and B. Later, we make yet another clone from site B to site C. From now on, we make local updates at all three sites and occasionally push our local updates to remote sites and pull remote updates to local instances.

2 Requirements

A primary requirement for local-first software is that a site should be able to independently perform queries and updates on the local database instances.

[2] `https://github.com/elixir-ecto/ecto`
[3] `https://sqlite.org/cli.html`

When two sites are connected, one site should be able to merge the updates performed at the other site without coordination. In particular, the site should be able to resolve conflicts without collecting votes from other sites.

The instances at different sites should be eventually consistent, or convergent [11]. That is, when they have applied the same set of updates, they should have the same state.

Database integrity constraints should be enforced. In particular, a merge of concurrent updates may cause the violation of an integrity constraint, though none of the updates violated any constraint locally at the sites. When this happens, one of the offending updates should be undone. It is important that the sites independently undo the same offending update.

Finally, existing applications should continue to work without any modification. In particular, performing queries and updates on local instances should not depend on the augmentation or any additional third-party software.

3 Technical background

In this section, we review the necessary background information about CRDT and CRR.

3.1 CRDT

A CRDT is a data abstraction specifically designed for data replicated at different sites. A site queries and updates its local replica without coordination with other sites. The data is always available for update, but the data states at different sites may diverge. From time to time, the sites send their updates asynchronously to other sites with an anti-entropy protocol. To apply the updates made at the other sites, a site merges the received updates with its local replica. A CRDT has the property that when all sites have applied the same set of updates, the replicas converge.

There are two families of CRDT approaches, namely operation-based and state-based [10]. Our work is based on state-based CRDTs, where a message for updates consists of the data state of a replica in its entirety. A site applies the updates by merging its local state with the state in the received message. The possible states of a state-based CRDT must form a join-semilattice [4], which implies convergence. Briefly, the states form a *join-semilattice* if they are partially ordered with $\sqsubseteq$ and a join $\sqcup$ of any two states (that gives the least upper bound of the two states) always exists. State updates must be inflationary. That is, the new state supersedes the old one in $\sqsubseteq$. The merge of two states is the result of a join.

Fig. 2 (left) shows GSet, a state-based CRDT for grow-only sets [10], where E is a set of possible elements, $\sqsubseteq \overset{\text{def}}{=} \subseteq$, $\sqcup \overset{\text{def}}{=} \cup$, insert is a mutator (update operation) and in is a query. Obviously, an update through $\mathsf{insert}(s,e)$ is an inflation, because $s \subseteq \{e\} \cup s$. Fig. 2 (right) shows the Hasse diagram of the states in a GSet. A Hasse diagram shows only the "direct links" between states.

Using state-based CRDTs, as originally presented [10], is costly in practice, because states in their entirety are sent as messages. Delta-state CRDTs address this issue

$$
\begin{aligned}
\mathsf{GSet}(E) &\overset{\text{def}}{=} \mathscr{P}(E)\\
\mathsf{insert}(s,e) &\overset{\text{def}}{=} \{e\} \cup s\\
\mathsf{insert}^{\delta}(s,e) &\overset{\text{def}}{=} \begin{cases} \{e\} & \text{if } e \notin s\\ \{\} & \text{otherwise}\end{cases}\\
s \sqcup s' &\overset{\text{def}}{=} s \cup s'\\
\mathsf{in}(s,e) &\overset{\text{def}}{=} e \in s
\end{aligned}
$$

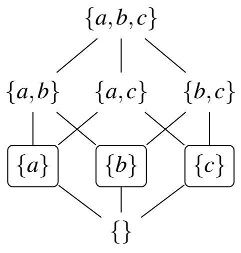

Fig. 2. GSet CRDT and Hasse diagram of states

by only sending join-irreducible states [1, 2]. Basically, *join-irreducible* states are elementary states: every state in the join-semilattice can be represented as a join of some join-irreducible state(s). In Fig. 2, insert^{δ} is a delta-mutator that returns join-irreducible states which are singleton sets (boxed in the Hasse diagram).

Since a relation instance is a set of tuples, the basic building block of CRR is a general-purpose set CRDT ("general-purpose" in the sense that it allows both insertion and deletion of elements), or more specifically, a delta-state set CRDT.

We use CLSet (causal-length set, [12, 13]), a general-purpose set CRDT, where each element is associated with a *causal length*. Intuitively, insertion and deletion are inverse operations of one another. They always occur in turn. When an element is first inserted into a set, its causal length is 1. When the element is deleted, its causal length becomes 2. Thereby the causal length of an element increments on each update that reverses the effect of a previous one.

$$
\begin{aligned}
\mathsf{CLSet}(E) &\overset{\text{def}}{=} E \hookrightarrow \mathbb{N}\\
\mathsf{insert}(s,e) &\overset{\text{def}}{=} \begin{cases} s\{e \mapsto s(e)+1\} & \text{if } \mathsf{even}\big(s(e)\big)\\ s & \text{if } \mathsf{odd}\big(s(e)\big)\end{cases}\\
\mathsf{insert}^{\delta}(s,e) &\overset{\text{def}}{=} \begin{cases} \{e \mapsto s(e)+1\} & \text{if } \mathsf{even}\big(s(e)\big)\\ \{\} & \text{if } \mathsf{odd}\big(s(e)\big)\end{cases}\\
\mathsf{delete}(s,e) &\overset{\text{def}}{=} \begin{cases} s & \text{if } \mathsf{even}\big(s(e)\big)\\ s\{e \mapsto s(e)+1\} & \text{if } \mathsf{odd}\big(s(e)\big)\end{cases}\\
\mathsf{delete}^{\delta}(s,e) &\overset{\text{def}}{=} \begin{cases} \{\} & \text{if } \mathsf{even}\big(s(e)\big)\\ \{e \mapsto s(e)+1\} & \text{if } \mathsf{odd}\big(s(e)\big)\end{cases}\\
(s \sqcup s')(e) &\overset{\text{def}}{=} \mathsf{max}\big(s(e), s'(e)\big)\\
\mathsf{in}(s,e) &\overset{\text{def}}{=} \mathsf{odd}\big(s(e)\big)
\end{aligned}
$$

Fig. 3. CLSet CRDT [12]

As shown in Fig. 3, the states of a CLSet are a partial function $s\colon E \hookrightarrow \mathbb{N}$, meaning that when e is not in the domain of s, $s(e) = 0$ (0 is the bottom element of $\mathbb{N}$, i.e. $\bot_{\mathbb{N}} = 0$). Using partial function conveniently simplifies the specification of insert, $\sqcup$ and in. Without explicit initialization, the causal length of any unknown element is 0. In the figure, insert^{δ} and delete^{δ} are the delta-counterparts of insert and delete respectively.

An element e is regarded to be in the set when its causal length is an odd number. A local insertion has effect only when the element is not in the set. Similarly, a local deletion has effect only when the element is actually in the set. A local effective insertion or deletion simply increments the causal length of the element by one. For every element e in s and/or s', the new causal length of e after merging s and s' is the maximum of the causal lengths of e in s and s'.

3.2 CRR

The RDB supporting CRR consists of two layers: an Application Relation (AR) layer and a Conflict-free Replicated Relation (CRR) layer (see Fig. 4). The AR layer presents the same RDB schema and API as a conventional RDB system. Application programs interact with the database at the AR layer. The CRR layer supports conflict-free replication of relations.

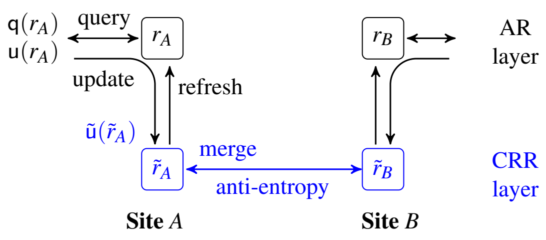

Fig. 4. A two-layer relational database system [12]

An AR-layer database schema R has an augmented CRR schema $\tilde{R}$. In Fig. 4, site A maintains both an instance r_A of R and an instance $\tilde{r}_A$ of $\tilde{R}$. A query q is performed on r_A without the involvement of $\tilde{r}_A$. An update u on r_A triggers an additional update $\tilde{\mathsf{u}}$ on $\tilde{r}_A$. The update $\tilde{\mathsf{u}}$ is later propagated to remote sites through an anti-entropy protocol. Merge with a remote update $\tilde{\mathsf{u}}(\tilde{r}_B)$ results in an update on $\tilde{r}_A$ as well as an update on r_A.

CRR has the property that when both sites A and B have applied the same set of updates, the relation instances at the two sites are equivalent, i.e. $r_A = r_B$ and $\tilde{r}_B = \tilde{r}_B$.

The two-layered system also maintains the integrity constraints defined at the AR layer. Any violation of integrity constraint is caught at the AR layer. A failed merge would cause some compensation updates.

We adopt several CRDTs for CRRs. Since a relation instance is a set of tuples or rows, we use the CLSet CRDT (Fig. 3) for relation instances. We use the LWW (last-write wins) register CRDT [6, 9] for individual attributes in tuples.

4 Work-in-progress implementation

We implement all functionality required for local updates inside the SQLite database, so no modification to existing applications or extra software is required for the applications to be able to continue working with the database. We implement the command-line features in Python (that in turn calls SQL statements).

4.1 Command-line operations

For command-line operations, we adopt `git`[4] operation names.

- `init` augments an existing SQLite database instance with CRR support.
- `clone` copies a remote augmented database instance to a local location.
- `pull` merges remotely applied updates to the local instance.
- `push` merges locally applied updates to a remote instance.
- `remote` queries and configures the settings of remote instances.

4.2 CRR-augmented database

For an AR-layer relation schema $R(A_1, A_2, \dots)$, we generate a new CRR-layer schema $\tilde{R}(K, L, T_1, T_2, \dots, A_2, A_2, \dots)$, ignoring all integrity constraints in R. K is the primary key of $\tilde{R}$. K values are globally unique. L is the causal-lengths (Fig. 3) of the tuples in $\tilde{R}$. T_i is the timestamp of the last update on attribute A_i. In other words, the (K, L) part represents the CLSet CRDT of tuples and the (A_i, T_i) parts represent the LWW register CRDT of the attributes.

In what follows, we write $t(K)$, $t(A_i)$ etc. for the K and A_i values of tuple t.

We use `randomblob(32)` of SQLite to generate K values. The chance that two tuples in the same relation have the same K value is extremely small.

For the AR-layer relations, we also generate triggers. We describe the triggers later in Section 4.3.

In addition to the augmentation of the AR-layer relations, we generate three more relations. A *Clock* relation implements a hybrid logical-physical clock [8] at a (virtual) nanosecond scale (Section 4.6). A *Site* relation maintains the information about the sites known at this instance. The information includes the hosts and paths of the remote instances, the last time this instance applied a push and a pull to the sites, etc. A *History* relation maintains a history of all the updates that have been applied.

4.3 Local updates

The `init` operation generates triggers on relation R. Every insertion, deletion and update on an instance r of R triggers the corresponding update on the instance $\tilde{r}$ of $\tilde{R}$.

When inserting a new tuple t into r, we insert a new tuple $\tilde{t}$ into $\tilde{r}$, with the initial $\tilde{t}(L) = 1$. When deleting t from r, we increment $\tilde{t}(L)$ with 1, so that the new $\tilde{t}(L)$ becomes an even number. When inserting the deleted t back to r, we increment $\tilde{t}(L)$

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

with 1, so that the new $\tilde{t}(L)$ turns back to an odd number. When updating $t(A_i)$ in r, we update $\tilde{t}(A_i)$ and $\tilde{t}(T_i)$ in $\tilde{r}$.

Since no integrity constraint is defined in $\tilde{R}$, a successful update on r will always lead to a successful update in $\tilde{r}$.

In addition to the triggers on R, the `init` operation also generates triggers on $\tilde{R}$. For every update on an instance $\tilde{r}$ of $\tilde{R}$, a trigger inserts a tuple in the *History* relation.

4.4 Merges

The pull of the concurrent updates from a remote site consists of the following steps: 1) generating the concurrent updates at the remote site; 2) transferring the generated updates to the local site (to be described in Section 4.5); 3) merging the received concurrent updates. A push is handled in a similar way.

As CRRs are based on delta-state CRDTs, the updates are join-irreducible states in a join-semilattice (Section 3.1). In our case, the updates are in fact the tuples in $\tilde{r}$ (Section 3.2). Using the *History* relation and the information of the last push and pull in relation *Site*, we can generate the updates since the last push or pull.

We generate the concurrent updates in a temporary database[5] and transfer it to the other site. This way, we avoid encoding individual tuples into an intermediate representation.

During the merge of received updates, we temporarily disable the generated triggers on AR-layer relation instances by setting a flag on the triggers.

An update on an relation instance $\tilde{r}'$ at a remote site is actually a tuple $\tilde{t}'$. If a tuple $\tilde{t}$ in the local instance $\tilde{r}$ exists such that $\tilde{t}(K) = \tilde{t}'(K)$, we update $\tilde{t}$ with $\tilde{t} \sqcup \tilde{t}'$ where the merge $\sqcup$ is the join operation of the join-semilattice (Section 3.1). Otherwise, we insert $\tilde{t}'$ into $\tilde{r}$. The merge $\tilde{t} \sqcup \tilde{t}'$ is defined as:

$$\tilde{t} \sqcup \tilde{t}' \overset{\text{def}}{=} \tilde{t}'', \text{ where } \tilde{t}''(L) = \mathsf{max}(\tilde{t}(L), \tilde{t}'(L)), \text{ and}$$

$$\tilde{t}''(A_i), \tilde{t}''(T_i) = \begin{cases} \tilde{t}'(A_i), \tilde{t}'(T_i) & \text{if } \tilde{t}'(T_i) > \tilde{t}(T_i) \\ \tilde{t}(A_i), \tilde{t}(T_i) & \text{otherwise} \end{cases}$$

After the update of $\tilde{r}$, we update r as the following. If $\tilde{t}(L)$ is an even number, we delete t (where $t(A_1) = \tilde{t}(A_1) \wedge t(A_2) = \tilde{t}(A_2) \wedge \ldots$) from r. Otherwise, we insert or update r with $\pi_{A_1, A_2, \ldots}(\tilde{t})$.

If the update on r violates an integrity constraint, we start an compensation update [12] (remaining to complete, see Section 4.7).

4.5 Network connections

At present, we support access to remote database instances in two possible cases: 1) the remote database instance is located on the same host as the local instance; or 2) the remote instance is located on a host where we have `ssh`[6] access.

[5] `https://sqlite.org/lang_attach.html`

[6] `https://www.ssh.com/`

When performing a clone, we specify a database instance stored on a remote host as `ssh://user@host#port:path/to/db`.

Since a SQLite database instance is stored as a file, we may (accidentally) copy or move the file to a different location. Every time we open an instance for push or pull, we verify the location information of the local instance stored in the *Site* relation and make modifications accordingly. We run the `remote` command-line operation to explicitly set or modify the location information of remote instances.

4.6 Timestamp values

The *Clock* relation, which the `init` operation creates, addresses two issues. The first issue is that the finest time resolution that SQLite provides is at a sub-millisecond level, so consecutive updates may have the same timestamp value. The second issue is that the physical clock (or "wall" clock) values are not sufficient to represent the happen-before relationship between updates, so a concurrent update may mistakenly win a competition when the physical clocks at different sites are skewed.

To address the first issue, the *Clock* relation has two attributes *Ms* and *Ns*, where *Ms* is the physical clock value in milliseconds and *Ns* is the offset within a millisecond at nanosecond scale. The *Clock* relation has only one tuple (τ_{ms}, τ_{ns}), which is the last clock (or timestamp) value that has been generated or merged. The comparison of two timestamp values is defined as $(\tau'_{ms}, \tau'_{ns}) > (\tau_{ms}, \tau_{ns})$ iff $\tau'_{ms} > \tau_{ms}$ or $\tau'_{ms} = \tau_{ms} \wedge \tau'_{ns} > \tau_{ns}$.

To generate a new local clock value, we first generate a new physical clock value τ'_{ms} (derived from the `julianday` function of SQLite) and a random number τ'_{ns} such that $0 \leq \tau'_{ns} < 10^6$. If $(\tau'_{ms}, \tau'_{ns}) > (\tau_{ms}, \tau_{ns})$, the new clock value is (τ'_{ms}, τ'_{ns}). Otherwise, we generate a new random number τ''_{ns} such that $\tau_{ns} < \tau''_{ns} < 10^6$ and the new clock value becomes (τ_{ms}, τ''_{ns}).

To address the second issue, we implement a hybrid logical-physical clock [8], which has the property that for two updates u_1 and u_2 with timestamp values τ_1 and τ_2, $\tau_1 < \tau_2$ iff u_1 happens before u_2 or u_1 and u_2 are concurrent (but never when u_2 happens before u_1).

At a merge, if a timestamp value (τ'_{ms}, τ'_{ns}) of the incoming tuple is greater than the (τ_{ms}, τ_{ns}) tuple in the *Clock* relation, we update the relation with (τ'_{ms}, τ'_{ns}).

4.7 Current implementation status

At the time of this writing, we have not finished all the features described in [12]. The remaining features include: enforcement of integrity constraints that are violated at the time of merge, and using a counter CRDT for lossless resolution of concurrent updates with additive update semantic. In addition to the merge in batch mode (push and pull), we are going to implement a continuous synchronization mode, like in [12], so that the sites can frequently exchange latest updates without explicit command-line push and pull. Since we have already implemented these features on top of an ORM [12], we expect it not to be as technically challenging as what we have implemented so far.

We currently focus on making a working prototype and have not put much effort on performance issues.

5 Related work

We limit the comparison to an alternative implementation reported in our earlier paper [12] and refer the interested reader to [12] for discussions on the other research work that are generally related to CRR.

Earlier, we implemented CRR on top of the Ecto ORM. One advantage of an implementation on top of an ORM is that it supports all RDBMSs (relational database management systems) that the ORM supports. It is even possible to synchronize between the instances running with different RDBMSs. There are some drawbacks, though. Only the applications using the ORM (and in the programming language of the ORM) can benefit from the CRR support. The supported RDBMSs are limited to those supported by the ORM. Unfortunately, the ORM of our choice, Ecto, does not support SQLite in the latest versions, and SQLite is an ideal candidate for local-first software (Section 1).

Implementing direct CRR support for SQLite addresses the above-mentioned drawbacks, at the cost of not benefiting from the advantages.

In [12], the implementation was mostly in the Elixir[7] programming language. Now, we try to keep the implementation as much in SQLite as possible. In particular, we aim at implementing all features related to local updates inside SQLite, so that existing applications continue to work without making any modification. We even restrict our implementation to the SQLite distribution that does not include any extension.

Since Elixir facilitates actor-based programs, data communication is built in. Little programming effort is needed for data communication. On the other hand, every database update is encoded and decoded between RDBMS and Elixir representations. This increases run-time overhead. Moreover, data security is not taken into account. Since we now transfers data through `ssh` connections, we do not have to worry about security and configuration issues.

There are some further differences in implementation details. In [12], a local update is first made in the CRR layer and then refreshed to the AR layer. Now the update first happens in the AR layer which then triggers updates in the CRR layer.

Currently, our implementation has not yet been as complete as the earlier implementation (Section 4.7).

6 Conclusion

We have presented a work-in-progress implementation of a software prototype that augments existing SQLite databases for local-first software. With a single command-line operation, we augment an existing database instance with CRR support. Existing applications using the existing database instance, without any modification, continue to work with the augmented instances. We can then maintain multiple instances of the same database at different devices and independently query and update the different instances. We can synchronize the updates at different instances when the devices are connected. The instances are eventually consistent. That is, they will have the same state when they have applied the same set of updates.

[7] `https://elixir-lang.org/`

References

1. ALMEIDA, P. S., SHOKER, A., AND BAQUERO, C. Delta state replicated data types. *J. Parallel Distrib. Comput. 111* (2018), 162–173.
2. ENES, V., ALMEIDA, P. S., BAQUERO, C., AND LEITÃO, J. Efficient Synchronization of State-based CRDTs. In *IEEE 35th International Conference on Data Engineering (ICDE)* (April 2019).
3. FOX, A., AND BREWER, E. A. Harvest, yield and scalable tolerant systems. In *The Seventh Workshop on Hot Topics in Operating Systems* (1999), pp. 174–178.
4. GARG, V. K. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.
5. GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (2002), 51–59.
6. JOHNSON, P., AND THOMAS, R. The maintamance of duplicated databases. *Internet Request for Comments RFC 677* (January 1976).
7. KLEPPMANN, M., WIGGINS, A., VAN HARDENBERG, P., AND MCGRANAGHAN, M. 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)* (2019), pp. 154–178.
8. KULKARNI, S. S., DEMIRBAS, M., MADAPPA, D., AVVA, B., AND LEONE, M. Logical physical clocks. In *Principles of Distributed Systems (OPODIS)* (2014), vol. 8878 of *LNCS*, Springer, pp. 17–32.
9. SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of convergent and commutative replicated data types. *Rapport de recherche 7506* (January 2011).
10. SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS 2011)* (2011), pp. 386–400.
11. VOGELS, W. Eventually consistent. *Comminications of the ACM 52*, 1 (2009), 40–44.
12. YU, W., AND IGNAT, C.-L. Conflict-free replicated relations for multi-synchronous database management at edge. In *IEEE International Conference on Smart Data Services (SMDS)* (2020), pp. 113–121.
13. YU, W., AND ROSTAD, S. A low-cost set CRDT based on causal lengths. In *Proceedings of the 7th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)* (2020), pp. 5:1–5:6.