UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Multi-site multi-synchronous support for SQLite augmented for Local-First Software

Tabassum Khan

INF-3990 Master's thesis in Computer Science - May 2023

Abstract

At the one end of the spectrum, locally installed applications allow users to work offline and at their convenience but do not support collaboration among multiple users or devices. At the other end, cloud-based applications offer collaboration and data synchronization but require an uninterrupted internet connection. To address these limitations, Local-First software was introduced to merge the advantages of both approaches. This allows users to use applications offline and synchronize data with peers when they are back online. This approach offers an appealing solution for users requiring both offline capabilities and collaboration [1].

Conflict-free Replicated Data Types (CRDTs) have emerged as a promising technology for implementing Local-First software since they address the challenge of replica convergence and multi-synchronous data access [1]. Basically, they are abstract data types with well-defined interfaces that can be replicated across multiple sites and allow a replica to be modified without coordination with other replicas [2]. The replicas converge to the same state when they receive and apply the same set of updates. If CRDTs are integrated with relational databases (RDBs), they can facilitate multi-synchronous data access and extend RDBs with Local-First characteristics. This integration process is known as Conflict-free Replicated Relations (CRRs) [3].

SynQLite is an implementation that augments SQLite databases with CRR to make them Local-First software [4]. It aims to extend existing SQLite-based applications with multi-synchronous access, enabling offline usage and seamless collaboration across multiple users and devices. However, its existing implementation has a critical limitation in its ability to support multiple sites. It cannot manage states when multiple replicas or sites are frequently connected and disconnected, which is a vital requirement for real-world applications and presents a challenge that must be addressed for SynQLite to be more widely adopted.

The goal of this Master's thesis is to extend the existing version of SynQLite by adding multi-site multi-synchronous support. This was achieved by implementing new features and resolving existing bugs on top of the existing code-

base. Our implementation was rigorously evaluated through various tests and experiments, and the results demonstrate that we successfully achieved our goal.

Acknowledgements

I would like to take this opportunity to extend my heartfelt appreciation to my esteemed supervisor, Associate Professor Weihai Yu, for his instrumental role in acquainting me with and enabling me to engage with an exciting topic for my master's thesis. Additionally, his invaluable guidance and unwavering support throughout my research expedition have been pivotal to the successful completion of my thesis. His inspiring and motivational demeanor, coupled with his immense knowledge in the subject matter, have contributed significantly to my academic growth and development. Beyond his steadfast academic support, I am grateful for his empathy and assistance in navigating personal challenges that surfaced during the research process. I am immensely beholden to Professor Yu and acknowledge the immeasurable impact of his mentorship on both my academic and personal spheres.

I am deeply indebted to my family, whose constant inspiration and encouragement have been a driving force behind my academic achievements. Their earnest belief in my abilities to pursue and achieve my goals has been a source of motivation and strength throughout my academic journey. I would like to convey a special acknowledgment to my husband for his unflagging support and for making significant sacrifices to ensure that I can focus on my thesis.

Lastly, I express my profound gratitude to all the individuals who provided me with guidance and motivation during this research endeavor. Their invaluable support and encouragement have played a critical role in this success, and I am grateful for their contributions to my academic journey.

Contents

List of Figures

List of Code Snippets

1 Introduction

1.1 Context

The emergence of the Internet has caused a groundbreaking shift in the way people perform their tasks, shifting the paradigm from a centralized to a distributed system and from single-user to multi-user applications. Contemporary software applications now feature collaborative capabilities that enable multiple users to engage in real-time work across several devices. Although conventional cloud-based solutions provide such features, they are limited by the requirement for uninterrupted internet connectivity. This poses a challenge when users experience network outages or partitioning or are in offline areas. These connectivity issues can impede system usability and result in synchronization and data loss. In response to this challenge, the concept of Local-First software has been proposed, allowing users to use the application even when disconnected from the network or offline, as though the application were locally installed. When online, they are synchronized with each other. This approach facilitates seamless collaboration across multiple devices and users, much like online collaborative applications [1].

Whatever the scenario is, managing data across multiple devices within a distributed system presents a formidable challenge. The CAP theorem, a renowned theorem within the realm of distributed systems, outlines three desirable properties of a distributed data store or application: consistency (C), availability (A), and tolerance to network partition (P). Consistency entails ensuring a single, up-to-date copy of the data is available across all replicas of the data; availabil-

ity guarantees the ability to access the data for both read and update operations, while network partition tolerance allows the system to function despite network failures. The theorem postulates that in a distributed system where data are stored across numerous devices or sites, it is only feasible to guarantee any two of these three properties concurrently [5]. Therefore, the simultaneous achievement of all three properties within a distributed system is unattainable. Moreover, network failures are ubiquitous in distributed systems, compounding the challenge. Consequently, it is crucial to tolerate network partitioning in any distributed system. Given this reality, when dealing with distributed systems, one must strike a balance between consistency and availability.

Therefore, it is apparent that cloud services prioritize consistency over availability, whereas Local-First software emphasizes availability even at the cost of consistency. This is because Local-First software is designed to remain accessible at all times, even during network outages. However, compromising consistency in favor of availability does not negate the importance of maintaining consistency across replicas. Ultimately, all replicas must possess the same state to enable collaborative functionality among multiple users. Nevertheless, ensuring consistency in a distributed system is complex and remains a topic of ongoing research [6, 7].

The inventors of the concept of Local-First software have identified Conflict-free Replicated Data Types (CRDTs) as a potential cornerstone technology for implementing Local-First software [1]. CRDTs were developed to address the complex issue of maintaining consistency in a distributed system where multiple nodes or sites access data concurrently [2]. Essentially, CRDTs are abstract data types with well-defined interfaces that can be replicated across several nodes or sites. Each replica can be independently modified without the need for coordination with other replicas. Consistency or convergence is achieved when all replicas receive and apply the same updates, ensuring they are all in an identical state.

The process of integrating CRDTs with relational databases (RDBs) to enable multi-synchronous data access and extend RDBs with Local-First characteristics is commonly known as Conflict-free Replicated Relations (CRRs) [3]. The introduction of CRRs provides an avenue for augmenting existing RDB applications with multi-synchronous access, requiring minimal modifications [3].

SQLite is an open-source RDB engine that has been recognized as the most commonly used database engine worldwide [8]. It is considered an excellent option for Local-First software due to its ability to operate independently of network connectivity. In [4], the authors showcased their work in progress that augments SQLite databases with CRR to make them Local-First software. This implementation, dubbed as **SynQLite**, aims to extend existing SQLite-based ap-

plications with multi-synchronous access, facilitating offline usage and seamless collaboration across multiple users and devices and locations or sites.

The existing SynQLite implementation serves as a seamless service that can be accessed through simple command-line instructions and does not necessitate any modification to the application that utilizes the database. With the aid of SynQLite, a conventional SQLite database can be augmented with CRR support, enabling it to be replicated or cloned across multiple sites. This allows users to update the database and synchronize with their peers through SynQLite's PULL and PUSH commands, thereby facilitating multi-synchronous access and collaboration.

However, the existing implementation of SynQLite encounters a crucial limitation in its ability to support multiple sites, which is a crucial requirement for real-world applications. Specifically, the existing synchronization program can accurately converge the states of a database only when two sites or replicas of the database are present. However, when the number of sites surpasses two, the program fails to converge the database states to a unique state. After [4], an attempt was made in [9] to provide the multi-site multi-synchronous support to SynQLite. Unfortunately, these endeavors were also unsuccessful in achieving their intended goals.

1.2 Goals

The existing version of SynQLite suffers from adequate state management capabilities when multiple sites are frequently connected and disconnected within the system. Consequently, the existing implementation fails to guarantee that all sites reach a consistent state after performing the synchronization operation. This Master's thesis aims to address this limitation by providing multi-site and multi-synchronous support for SynQLite.

To achieve the primary goal of implementing multi-site and multi-synchronous support for SynQLite, we have identified the following sub-goals and requirements that must be met.

- Design and implement features on top of the existing SynQLite codebase, ensuring consistency across multiple sites after the synchronization operation.

- Regardless of the number of sites that have been cloned, it is essential that any synchronization operation performed by a particular site should receive all the updates from every other site present at that moment

except the partitioned sites.

- In Chapter 2, we will examine how CRDTs adhere to the principle of Strong Eventual Consistency (SEC), wherein a replica can converge with another site's updates without requiring any further coordination with the sending site. Hence, it is imperative that our implementation guarantees the convergence of sites without necessitating additional coordination.

- Evaluate the correctness and performance of our implementation.

1.3 Achievement

Following a thorough examination of the existing SynQLite codebase and the underlying principles of Local-First software, CRDT, and CRR, we identified a set of new features to be implemented and existing bugs to be resolved, as detailed in Chapter 6. These enhancements, if successfully implemented and rectified, would render SynQLite functional for multiple sites. We proceeded to implement and address all of the identified features and issues and subsequently conducted numerous tests and experiments, as outlined in Chapter 7. After confirming the accuracy and reliability of our implementation through these tests and experiments, we can confidently affirm that we have successfully developed a multi-site and multi-synchronous version of SynQLite.

The subsequent chapters of this thesis will provide a comprehensive discussion of the topics introduced in this opening chapter.

2

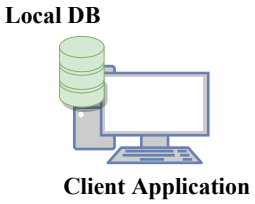
Technical Background

This chapter presents the foundational technologies that are required for a comprehensive understanding of the remainder of the thesis. Thus, it is advisable for the reader to delve into this chapter to understand these technologies before proceeding with the reading.

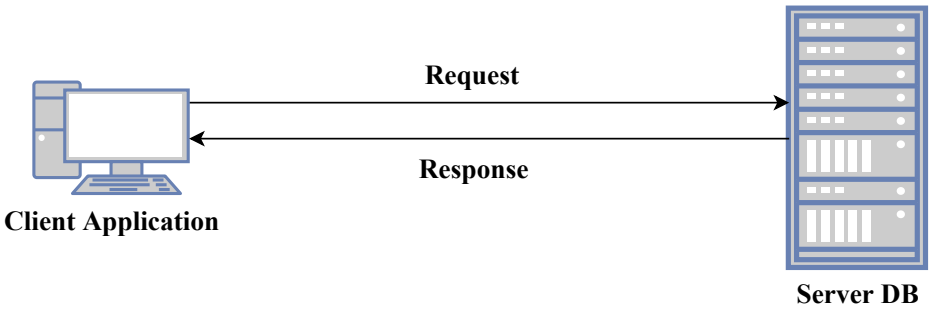
2.1 Local-First Software

As we delve into our thesis topic of "**Multi-site multi-synchronous support for SQLite augmented for Local-First Software**", it becomes apparent that the first technical concept to be addressed is "Local-First software". It is, therefore, essential to thoroughly explore this concept, including its definition, significance, and practical implementation.

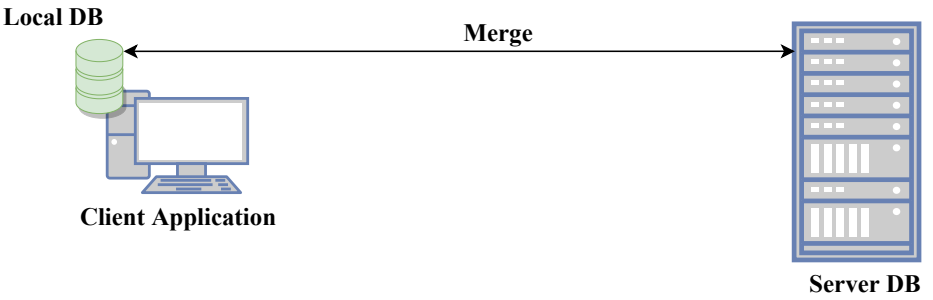
The "Local-First software" concept was first introduced in [1]. Local-First software is a set of principles that seeks to combine the strengths of collaboration and data ownership into software design. Collaboration can be referred to as the ability to cooperate or work together with multiple users and devices. At the same time, ownership means having control over the data, including the ability to access it offline. Cloud apps allow for collaboration across multiple users and devices but can limit data ownership by preventing users from working offline. On the other hand, traditional local apps provide users with authority over their data, but they do not facilitate collaboration.



(a) Traditional Local App



(b) Traditional Cloud App



(c) Local-First software

Figure 2.1: Comparison among the traditional local app, cloud app, and Local-First software

Local-First software aims to bridge the gap between cloud apps and traditional local apps. Cloud applications typically adopt a client-server architecture in which the data is kept on a server, and any interactions with the data are performed through the server. While the client side may cache some data, any changes to the data must be transmitted to the server to be applied. On the other hand, Local-First applications save the primary copy of data in the local filesystem of each device, allowing users to read and write data anytime, even offline. The data is then synchronized with other devices whenever there is an internet connection or when the user opts to synchronize. Figure 2.1 illustrates

the comparison among the traditional local app, cloud app, and Local-First software. It is crucial to note that the design of Local-First software prioritizes the accessibility or availability of data over ensuring constant consistency across all copies. This means that without constant monitoring, it is not possible to know the exact state of all copies at any given moment, but eventually, all copies will be in the same state.

According to [1], CRDTs have the possibility of being a fundamental technology in implementing Local-First software. Therefore, having a thorough understanding of CRDTs is crucial in the context of Local-First software. But before discussing CRDT, it is essential to understand the distinctions between Strong, Eventual, and Strong-Eventual Consistency. Understanding the concept of CRDT can be difficult if one is not aware of this knowledge. Therefore, we will cover the differences between these types of consistency before delving into CRDT.

2.2 Strong, Eventual, and Strong Eventual Consistency

In a distributed system, replication and consistency go hand in hand. When multiple copies of data or replicas are present, adhering to a consistency model is crucial to guarantee that all replicas are in an identical state or have the same view of the data at a particular moment in time. Strong Consistency (SC) means that every replica will show the same data, but this results in elevated latency and reduced availability since updates made to one replica need to be instantaneously transferred to all other replicas, which may cause potential delays in read-and-write requests while the update is taking place. Thus, in large distributed systems where there are significant network lags or frequent partitioning, Eventual Consistency is considered a more suitable option, especially in cloud computing or peer-to-peer systems [7]. Eventual Consistency (EC) permits short-term discrepancies in the data but guarantees that all replicas will ultimately have the same data state. While modifications to one replica may not be immediately reflected in all others, they will eventually be transmitted to all replicas. This consistency model offers greater availability and reduced latency but may result in conflicts when more than one replica updates the same data. Such conflicts are usually resolved using consensus algorithms or rollbacks [10]. Despite extensive research, conflict resolution techniques in replicated systems are still poorly established and remain a subject of ongoing research. Several algorithms that were previously thought to be correct have been shown to be flawed, even those backed by formal proofs of their correctness [6, 7].

On the other hand, [2] proposed a new theoretically-sound approach to Eventual Consistency referred to as Strong Eventual Consistency (SEC). Unlike traditional EC, SEC guarantees that once two replicas receive the same updates, regardless of their order, they will immediately converge. Any conflicting updates will be automatically merged without the need for consensus algorithms or rollbacks. In contrast, general EC only ensures that copies of data will eventually match after conflicting updates are resolved through consensus algorithms or rollbacks. However, to achieve SEC, a replica object or the replica's underlying data type or data structure must possess certain mathematical properties that ensure the absence of conflicts and, thus, convergence. That is where CRDT comes into play.

2.3 Conflict-free Replicated Data Type (CRDT)

The origin of the CRDT concept can be traced back to the authors who created the theory of SEC. According to their definition, CRDTs are data structures such as sets, lists, hash maps, graphs, or sequences that have been designed to adhere to certain mathematical properties like commutativity[1], idempotence[2], associativity[3], etc. to prevent conflicts and ensure convergence. This eliminates the need for conflict resolution through consensus algorithms and rollback, making CRDT replicas highly resilient and accessible, even in challenging situations such as failures, high network latency, faults, or network partitions. As a result, CRDTs adhere to the principles of SEC by guaranteeing CRDT replicas will eventually converge to a correct and identical state.

2.3.1 Categories of CRDTs

CRDTs can be classified into two primary categories: Operation-based and State-based. In Operation-based CRDTs, the convergence of replicas is achieved through the dissemination of local operations to all other replicas. Conversely, State-based CRDTs attain convergence by the exchange of local states among replicas. To illustrate, in the context of Operation-based CRDTs, when a replica updates through the execution of an operation, it notifies other replicas to carry out the same operation to guarantee the consistency of the data across

1. Commutativity refers to the property where the result of a series of operations is the same irrespective of the sequence of their execution, i.e. $x \circ y = y \circ x$
2. Idempotence refers to the property where repeating the same operation on a variable has no cumulative effect, and the value remains the same, i.e. $x \circ x = x$
3. Associative refers to the property of certain binary operations where changing the grouping of the elements within an expression will not alter the final outcome, i.e. $(x \circ y) \circ z = x \circ (y \circ z)$

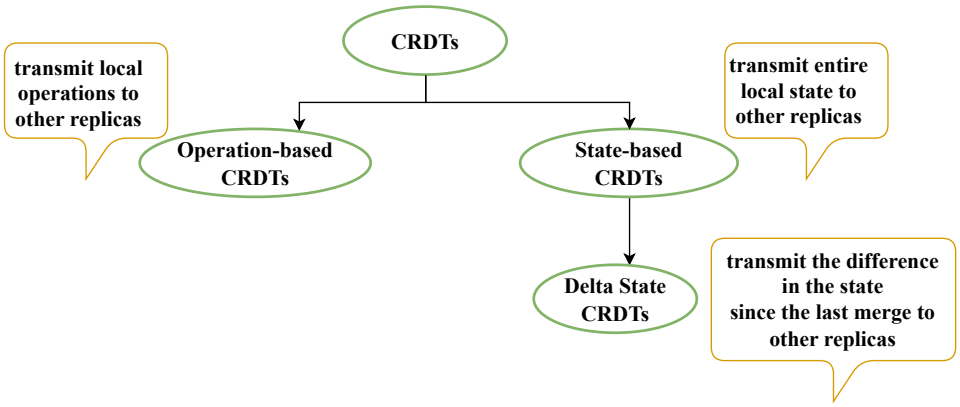


Figure 2.2: Categories of CRDTs

all replicas. In contrast, in the case of State-based CRDTs, the updating replica transmits its updated state to other replicas. The receiving replicas integrate the received state with their local state by executing a merge function.

In a distributed system utilizing a reliable causally-ordered broadcast communication protocol[4] , Operation-based CRDTs necessitate only the commutativity of operations. Conversely, State-based CRDTs require not only commutativity but also idempotence. This implies that while Operation-based CRDTs may encounter overhead from the requirement for reliable causally-ordered communication, State-based CRDTs are free from this concern due to the idempotent property, which ensures that repeated merging of states will result in no alteration. Additionally, there exist specific preconditions for the implementation of State-based CRDTs, including partial ordering[5] of all possible states from all replicas and the existence of a join between any two states. The merge function must join any pair of states to form a join-semilattice[6], and the internal states of State-based CRDTs must also exhibit monotonically non-decreasing behavior across updates. This is because the elimination of any information may render the merge function incapable of correctly merging a pair of states.

One significant disadvantage of State-based CRDTs is the communication overhead associated with the transmission of the entire state, which can become quite substantial in size. Delta-State CRDTs have been introduced as a solution

4. Delivers every message to every recipient exactly once and in an order consistent with happened-before relation [2].
5. A partial order on a set is a way of ordering its elements to say that some elements precede others, but allowing for the possibility that two elements may be incomparable without being the same [11].
6. A join-semilattice is a partially ordered set with finite colimits, or equivalently, a partially ordered set with finite coproducts [12].

to tackle this problem. Instead of sending the complete state to other replicas, Delta-State CRDTs only transmit the δ, or the subset of values that have been changed since the last merge, to other replicas. This not only reduces communication costs but also speeds up the merge process, as the merge function will have fewer elements to merge. However, this advantage is accompanied by the additional effort required to generate deltas. Figure 2.2 highlights the categories of CRDTs.

2.3.2 Examples of CRDTs

At this juncture, it is worth highlighting that we have adopted Delta-State CRDT in SynQLite. Particularly, we have leveraged the CL-Set and LWW Register CRDTs. On the other hand, these two CRDTs have evolved from more fundamental CRDTs, which we will be reviewed along with the two aforementioned CRDTs in this section.

Counter CRDT

A Counter CRDT represents a particular type of CRDT where the underlying data structure is a simple, integer-like counter that supports increment and decrement operations to update it, and queries return the value of the given counter. The counter can be replicated across multiple sites and updated in a manner that guarantees SEC among the replicas. The literature on Counter CRDTs encompasses a variety of designs and implementations, including the G-Counter, PN-Counter, Non-negative Counter, and others [13].

Register CRDT

A Register CRDT represents a specific type of CRDT where the underlying data structure is a memory cell that stores an opaque unit or object and supports two operations: to update the value of the register cell and to query to retrieve the value of the given register. Register CRDTs must resolve conflicts in updates to remain conflict-free. This is typically achieved by considering newer updates as correct and overwriting older ones. However, concurrent updates pose a problem that requires resolution. The solution is either the "Last-Writer-Wins Registers" approach or the "Multi-Value Register" approach, which determines the order of updates to resolve conflicts [13].

The Last-Writer-Wins Register (LWW-Register) determines the correct update by assigning a unique timestamp to each update. The latest update with the highest timestamp is considered to be correct. Timestamps are considered to

be unique, totally ordered, and consistent with causal order [14]. The update operation overrides the new value with the old one and generates a new timestamp. When a replica sends the state to other replicas, the merge operation selects the value with the maximal timestamp.

SET CRDT

A SET CRDT is a specific variant of the CRDT, which utilizes a set as its underlying data structure. Sets are defined by their capability to store unique values only and their support for adding and removing elements. Importantly, this data structure disallows any modifications to the values stored within it. However, the non-commutative nature of sequential add and remove operations may result in conflicts when updates are made to different replicas, thus raising the consistency issue among replicas. To address this challenge, various approximations of sequential sets have been developed and implemented as SET CRDTs, such as the Grow-Only Set (G-Set), Two-Phase Set (2P-Set), Last Write Wins Element Set (LWW-element-Set), and Causal-length set (CL-Set), with the aim of attaining SEC while avoiding conflicts [13].

Grow-Only Set (G-Set) The G-Set CRDT is designed to address consistency issues in merging sets with both addition and deletion operations. The idea is to avoid the removal operation altogether. It only supports add and query operations and therefore is not a fully general-purpose CRDT. A trivial check is carried out to confirm the existence of an element in a G-Set. To add new elements, a subset containing the element is created and merged into the G-Set via a union operation. By performing a union operation on the two sets, it is possible to merge two G-Sets together [13].

Two-Phase Set (2P-Set) Since the G-Set CRDT only allows adding elements and not removing them, its applicability can be restricted in some scenarios where removing elements is a crucial operation. The 2P-Set CRDT addresses this issue by allowing both the addition and deletion operations, but one limitation is that the deleted elements cannot be re-added. A 2P-set comprises two G-sets; one is for added elements, and the other is for removed elements. To be deemed as present in the set, an item must be included in the "add-set" and not present in the "remove-set" The process of adding elements involves adding them to the "add-set". In contrast, removing elements requires checking if they are present in the "add-set" and then adding them to the "remove-set". To merge two sets, a union operation is performed on both sets [13].

Last Write Wins Element Set (LWW-element-Set) The LWW-element-Set is

an improved version of the 2p-set that addresses its limitations by allowing the re-insertion of removed elements. Each element in the add and remove sets is recorded as a tuple with an associated timestamp, which is used to determine its existence. The procedures of adding, removing, and merging elements are analogous to the 2p-set. However, to determine the existence of an element, it is necessary to verify if it has been added to the "add-set" and if its timestamp is newer than any corresponding entry in the "remove-set". If a newer entry exists in the latter, the element is deemed non-existent. The limitation of this variant is that the precision of the system clock is an essential aspect for the set to operate correctly [15]. Another challenge is that elements are not physically deleted from the add and remove sets; rather, newer elements are simply added with timestamps. As a result, the sets may accumulate a significant number of "ghost elements" [13].

Observe Remove Set(OR-Set) The OR-Set was created to address the limitations of the LWW-element-Set. It is more space-efficient by eliminating outdated elements and eliminates the issue of concurrent updates in LWW-element-Set, which are dependent on the allocation of timestamps. OR-Set utilizes a unique identifier instead of timestamps and adds elements by creating a new pair with a new identifier and adding it to the "add-set." After that, all pairs with the same element in the "remove-set" are removed. When removing an element from the OR-Set, pairs that contain the element are added from the "add-set" to the "remove-set" and then removed from the "add-set". Additionally, merging two instances of OR-Sets involves extracting each "add-set" and eliminating the opposing "remove-set". Afterward, the two "add-set"s are combined, and any conflicting "remove-set"s are reconciled using a join operation that takes the union of the "remove-set"s. This guarantees that the merged set includes all elements added to either set and that conflicting removals are resolved to produce a consistent final state across all replicas [13].

Causal-length set (CL-Set) The CL-set represents a specialized variant of SET CRDT that offers a unified set containing a distinctive integer for each attribute referred to as the causal length. The causal length feature determines the state of each element within the set, eliminating the need for timestamps, unique identifiers, or multiple sets superfluous. Checking for the presence of an element entails examining its value within the set and the parity of its causal length - if odd, it exists, and if even, it does not. Adding an element requires checking its existence and causal length, after which the causal length is incremented by one if the element exists with an even causal length. The element is appended to the set with a causal length of 1 if it does not exist. When removing an element, the opposite procedure occurs. To merge two CL-Sets, all unique elements

within each replica are combined into a singular set, and the highest causal length is selected in cases where conflicts arise [3].

2.3.3 How CRDT can realize the Local-First software

Since CRDT are the data types that guarantee SEC, a CRDT replica may continue to receive and serve read and write requests in case the replica gets disconnected from other replicas through network failure or partition. The replica can always be made available for both read and write operations regardless of the network condition, thus allowing the ability to work offline, which is the data ownership Local-First property. On the other hand, any communicating subset of replicas gives the guarantee to converge, thus giving the multi-synchronous property of Local-First software.

2.4 Conflict-free Replicated Relation (CRR)

CRR is a principle that implies the integration of CRDTs with RDBs to extend RDBs with Local-First characteristics. With CRR, it is ensured that the replicas of RDBs will converge with each other when they apply the same updates, similar to how the replicas of CRDTs converge. The CRR approach was developed as RDBs are more complex and cannot guarantee SEC like CRDTs. Essentially, CRR involves utilizing various types of CRDTs with the relations of RDBs to extend the RDBs with Local-First characteristics.

3

SynQLite Overview

This chapter will delve into the details of **SynQLite**, a python-based implementation of **CRR**, which forms the basis of this thesis. **SynQLite** enhances the capabilities of an **SQLite** database as a Local-First software by enabling it to be replicated and synced across various locations or sites, allowing for individual offline usage and later synchronization. It functions as a service that can be easily accessed through simple command-line commands and does not require any modifications to the application utilizing the database.

3.1 Design and Implementation

SynQLite adds a CRR layer on top of the existing database instance, generally referred to as the *Application Relation* (AR) Layer. The CRR layer is nothing but the additional tables and triggers that provide the Local-First property to the database. Note that, SynQLite does not modify the schema of the AR layer tables and triggers. It just adds the CRR layer tables and triggers. Users interact directly with the AR layer and are not aware of the CRR layer. After augmenting the CRR layer through SynQLite, the database can be cloned or replicated to another site. Communication between sites for synchronization is done through the CRR layer using SynQLite commands. Figure 3.1 depicts the overview of the SynQLite.

It is worth mentioning that SynQLite was first introduced and implemented in

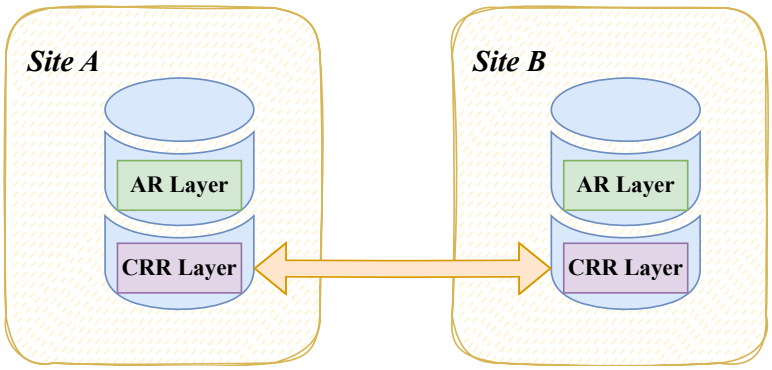


Figure 3.1: SynQLite Overview

[4] and later modified in [9]. To differentiate between these two implementations, the implementation from [4] will be referred to as the *stage 1* implementation, and the implementation from [9] as the *stage 2* implementation in this thesis.

3.1.1 CRR Layer

The CRR layer can be divided into two parts: CRR layer tables and CRR layer triggers. The following subsections will describe both of them.

CRR Layer Tables

For each table R in the AR layer, SynQLite will generate a corresponding $\tilde{R}$ table in the CRR layer. For example, if the AR layer has Department, Student, and Subject tables, the CRR layer will have crr_Department, crr_Student, and crr_Subject tables. If the schema of the AR layer table is R(a1, a2,...), the corresponding CRR layer table schema would be $\tilde{R}$(crr_id, cl, t_a1, t_a2,...a1, a2,...). For instance, if the Department table schema is R(DepartmentId, DepartmentName) the crr_Department table schema would be $\tilde{R}$(crr_id, cl, t_DepartmentId, t_DepartmentName, DepartmentId, DepartmentName). $\tilde{R}$(crr_id) is a universally unique identifier (UUID) for the particular row [16]. $\tilde{R}$(cl) is the causal length of the row and $\tilde{R}$(ti) is the timestamp for $\tilde{R}$(ai), meaning the last time when the attribute was updated. The tuple (crr_id, cl) symbolizes the **CLSet CRDT**, and tuple (t_ai, ai) symbolizes the **LWW register CRDT** which we discussed in Chapter 2. Please take a look at Figure 3.2 to get a clear visual understanding of the correlation between the R and $\tilde{R}$ tables.

In addition to the $\tilde{R}$ tables, SynQLite also generates the meta_site(S) table,

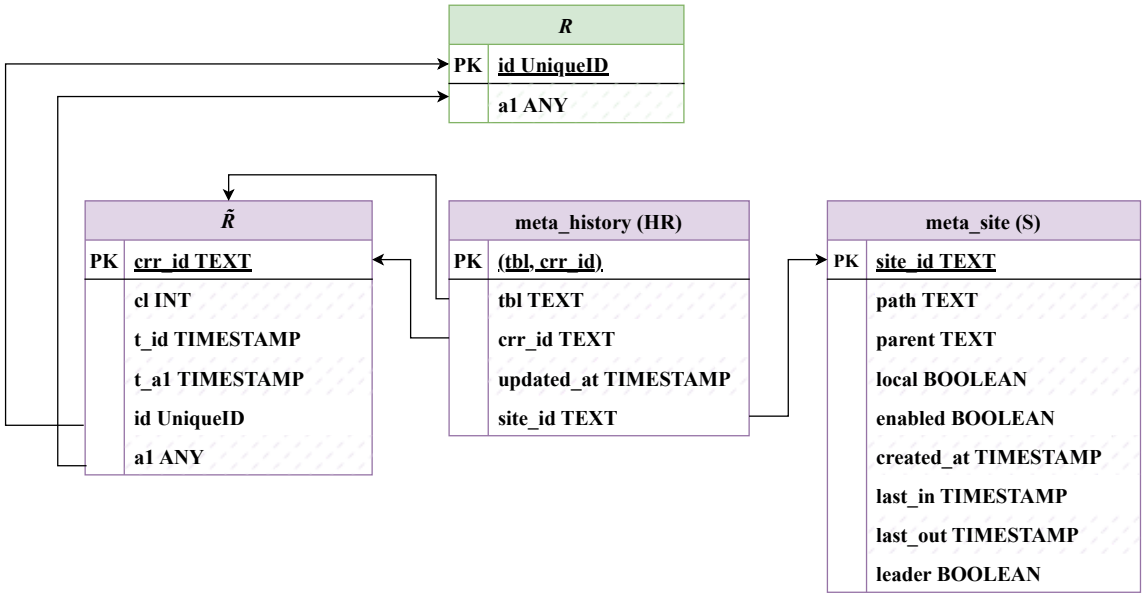


Figure 3.2: CRR Layer tables

which serves as a repository for the information pertaining to the sites that are associated with this database. The schema of the meta_site(S) table is depicted in Figure 3.2. This table comprises the site's unique identifier S(site_id), the location of the database S(path), an indicator for whether the site is the local one S(local), the timestamp for when the database was augmented with CRR support S(created_at), timestamps for when data is transferred between sites S(last_in) and S(last_out), and a flag indicating whether the site is a leader for continuous synchronization S(leader). Specifically, if the CRR support was added through the initialization operation, then the value of S(local) will be set to true. Conversely, if CRR support was acquired through the cloning process, which we will explain later, the value of S(local) will be false. Furthermore, the S(path) value may be either a local file path or an SSH path, contingent on the site's location with respect to the local site. The S(last_out) timestamp denotes the time when the local site transferred its state to the remote site, which can be accomplished either by the remote site pulling the state from the local site or by the local site pushing its state to the remote site, as we will elaborate on later. In contrast, S(last_in) designates the time when the remote site transmitted its state to the local site. Note that the S(leader) attribute was introduced in the *stage 2* implementation process. By setting S(leader) to true, a site role can be made as a leader for continuous synchronization, which we will discuss later in this chapter.

The meta_history(HR) table is another crucial table in the CRR layer that keeps

track of all the rows of all the $\tilde{R}$ tables concerning when a row was last updated and on which site. The HR(crr_id) and HR(tbl) are the corresponding $\tilde{R}$(crr_id) and the table name of $\tilde{R}$ table respectively, HR(updated_at) is the timestamp of the most recent update and HR(site_id) refers to the S(site_id) where the update took place. The tuple (tbl, crr_id) serves as the unique identifier for each row. By consolidating essential data from all rows of all the $\tilde{R}$ tables, the meta_history table enables SynQLite implementation to easily access and traverse the information of the $\tilde{R}$ tables. See Figure 3.2 for a better understanding of the table structure.

There are a few other important tables, such as meta_clock, meta_islocal, etc. To prevent clock divergence at each site meta_clock table is used, which maintains the local clock information at each site and updates it with the latest timestamp during synchronization with other sites. On the other hand, the meta_islocal is used to turn on and off the CRR layer triggers. With the help of all of these tables, SynQLite achieves the Local-First properties.

CRR Layer Triggers

SynQLite employs several triggers in the database to aid the operation of the CRR layer. Each table in the AR layer is associated with five triggers, namely insert, delete, update, hist_insert, and hist_update triggers. For instance, if the AR layer includes a Student table, SynQLite generates crr_Student_insert, crr_Student_delete, crr_Student_update triggers, hist_crr_Student_insert and hist_crr_Student_update triggers.

When a row is inserted into the R table, the corresponding insert trigger adds a row to the $\tilde{R}$ table, with one as the initial value of $\tilde{R}$(cl). When a row is deleted, the delete trigger increases the $\tilde{R}$(cl) value of the corresponding row. If a row is reinserted, the insert trigger is invoked again, which further increases the $\tilde{R}$(cl) value. Therefore, the cl value being odd indicates that the row is present in the database, and an even cl value denotes that the row has been deleted. If there is an update to R(ai), the update trigger updates $\tilde{R}$(ai) and $\tilde{R}$(t_ai) according to the LWW-register CRDT rules. These triggers thus establish a connection between the AR layer and the CRR layer.

On the other hand, When a new row is added to the $\tilde{R}$ table through the insert trigger, the hist_insert trigger creates a new row in the meta_history table. The HR(crr_id) value is set to $\tilde{R}$(crr_id), HR(tbl) is set to the name of the $\tilde{R}$ table, HR(updated_at) is set to the current timestamp, and HR(site_id) is set to the site id from meta_site. Furthermore, for each update, delete, or reinsertion in the $\tilde{R}$ table, the hist_update trigger is executed, updating the corresponding row in the meta_history table. These triggers thus connect the $\tilde{R}$ table and the

meta_history table in the CRR layer.

3.1.2 SynQLite Operations

SynQLite's operation can be segmented into three primary stages: initialization, cloning, and synchronization.

Initialization

Initialization is SynQLite's first function, which entails incorporating the CRR layer into an already existing database. This process involves not only adding the CRR layer tables but also populating them with data taken from the AR layer tables and the specific table logic. Moreover, SynQLite's initialization code includes the metadata of the site in the meta_site table.

Cloning

The subsequent function in SynQLite is the clone operation, which is relatively straightforward. The process starts by verifying if the database to be cloned has CRR support initialized. If it does, the entire database is copied as a file to the intended site via SSH File Transfer Protocol (SFTP) [17]. The implementation of SynQLite employs Paramiko, a Python implementation or library for SFTP protocol, to establish connections between the sites, transfer files, and execute commands on the remote site [18]. Upon successfully copying the entire database to the cloned site, the metadata of the cloned site is recorded in the meta_site table of both the original and cloned sites. To clarify the cloning process, we generally use a parent-child relationship between the original and cloned sites, with the original site being referred to as the parent site and the cloned site as the child site. As such, during the cloning process, the S(parent) of the cloned site is assigned with the S(site_id) of the original site, while the S(local) of the original site in the cloned site's meta_site table is set to false. Additionally, the logic for triggers on the cloned site is modified to ensure they function correctly for the cloned site. The modifications are necessary since the logic of triggers includes the local site id.

Synchronization

The synchronization process involves generating and transferring the deltas(δ) from one site to another site, which are followed by a merging operation on the recipient site that merges the incoming state with its local state. δ means the

unseen or updated rows that need to be transferred from one site to another site so that the recipient site converges with the sender site. With the help of the $\tilde{R}$ table, meta_history(HR) table, and S(last_in) and S(last_out) attribute of meta_site table, the δ is generated, which is referred to as the delta generation logic. The database's entire set of changes or the total δ of the database is composed of the changes to both the $\tilde{R}$ table and the meta_history table. The δ are stored in a temporary file and then transferred to the other site. Note that a site communicates with other sites through the S(path) of the meta_site table, which is configured during the initialization or cloning process.

Upon receiving the δ, the site initiates the merging process by adding the δ rows to the CRR layer and trickling them up to the AR layer for convergence. To prevent the triggers from affecting the CRR layer, the meta_islocal(up) is set to true, and the triggers only execute when meta_islocal(up) is false. During the merging process, the site first iterates through (δ)HR and adds it to HR. Next, it merges the (δ)$\tilde{R}$ row with its $\tilde{R}$ table while adhering to the CRR rules. If there's already a row in $\tilde{R}$ with the same crr_id, the merge operation compares the causal lengths and chooses the one with the highest one. The merge operation also compares the t_ai for each attribute and selects the attribute with the most recent timestamp. Finally, the updated row in $\tilde{R}$ is trickled up to the AR layer. Once the merging process is complete, the triggers are reactivated by setting meta_islocal(up) to false.

When a site wants to synchronize with other sites a site performs a pull or push operation. If a site wants to perform a PULL operation, it sends a command to other sites; other sites then generate the δ for this site using the delta generation logic of SynQLite. Afterward, they transfer the δ. The former site then merges the δ. On the other hand, PUSH is the opposite of PULL. If a site wants to perform PUSH, it generates the δ and then pushes the changes to other sites; other sites then merge the δ.

The SynQLite system offers discrete PULL and PUSH operations that can be initiated at any time as required. In the *stage 2* implementation, a continuous synchronization feature was added that uses a centralized network topology, where one site is designated as the leader. The leader site is identified by the S(leader) attribute in the meta_site(S) table and runs an API server that listens on a specific port. To communicate with the leader, the other sites, called member sites, establish a TCP connection via an SSH tunnel. This tunnel is the only means of communication between the members and the leader.

In discrete PULL and PUSH operations, a site communicates with all other known sites through its meta_site table, whereas in continuous synchronization, a member site communicates only with the leader site. During continuous synchronization, a member site sees only two sites in the distributed system:

itself and the leader site. After establishing a TCP connection with the leader, the member sites continuously push and pull from the leader. When a member site pushes or pulls, the leader locks its database and performs the necessary operations as described earlier. The difference is that during merging, the leader changes the information to make it look like the changes were made by the leader so that members are only aware of the leader site.

3.2 User Manual of SynQLite

From a user perspective, **SynQLite** is not a background process that needs to be up and running all the time for the user to access the database. Instead, it is a service that empowers users to augment their SQLite databases with CRR support, replicate the database to other locations, and communicate with replicas for synchronization. To use **SynQLite**, the user must have the SynQLite code on their machine and have **Python3** and other necessary libraries such as **paramiko** and **sshtunnel** installed. Users can then run various **SynQLite** commands as outlined below.

Command: INIT

Functionality Augments CRR support to an existing SQLite database instance.

Full Command *python3 -m synqlite.cli init <db_file>*

Command: CLONE

Functionality Copies a remote augmented database instance to a local location

Full Command *python3 -m synqlite.cli clone <source_db> <destination_directory>*

Command: PULL

Functionality Fetches and merges remotely applied updates to the local instance

Full Command *python3 -m synqlite.cli pull <db_file>*

Command: PUSH

Functionality Sends and merges locally applied updates to a remote instance

Full Command *python3 -m synqlite.cli push <db_file>*

Command: API

Functionality Starts a centralized leader. Other members can connect with it for continuous synchronization.

Full Command *python3 -m synqlite.cli api <db_file>*

Command: SYNC

Functionality Members connect to the centralized leader for synchronization.

Full Command *python3 -m synqlite.cli sync <db_file> <sync_time>*

4

Methodology

In the field of Computer Science, there are three main research methodologies that are commonly used to achieve various goals - these are theory, abstraction, and design. Firstly, the theory is based on mathematical principles and is used to create a logical and valid theory. Conversely, abstraction uses the experimental scientific method to investigate a concept or phenomenon and involves creating models of potential implementations. Lastly, design, rooted in engineering, is used to build systems or devices to solve problems [19].

More specifically, the design methodology is a structured approach that is made up of four distinct stages, including identifying the problem or need, outlining the specifications, designing and implementing the system, and finally, evaluating the system to confirm that it has met its intended objectives, as shown in Figure 4.1. It is a step-by-step process that begins with recognizing the problem and ends with verifying that the system was successful in solving it [19].

It is clear that the methodology used in this research falls under the category of design, as we are not creating a new theory or modeling potential implementations of a Local-First database, which has already been done in reference [3]. To clarify, the primary objective of this thesis is to expand the existing version of SynQLite to provide multi-site multi-synchronous support while maintaining the Local-First property of the system. We have designed and implemented additional features to meet these requirements, which have been documented in Chapter 6. The implementation was then tested through various experiments, the results of which are presented in Chapter 7.

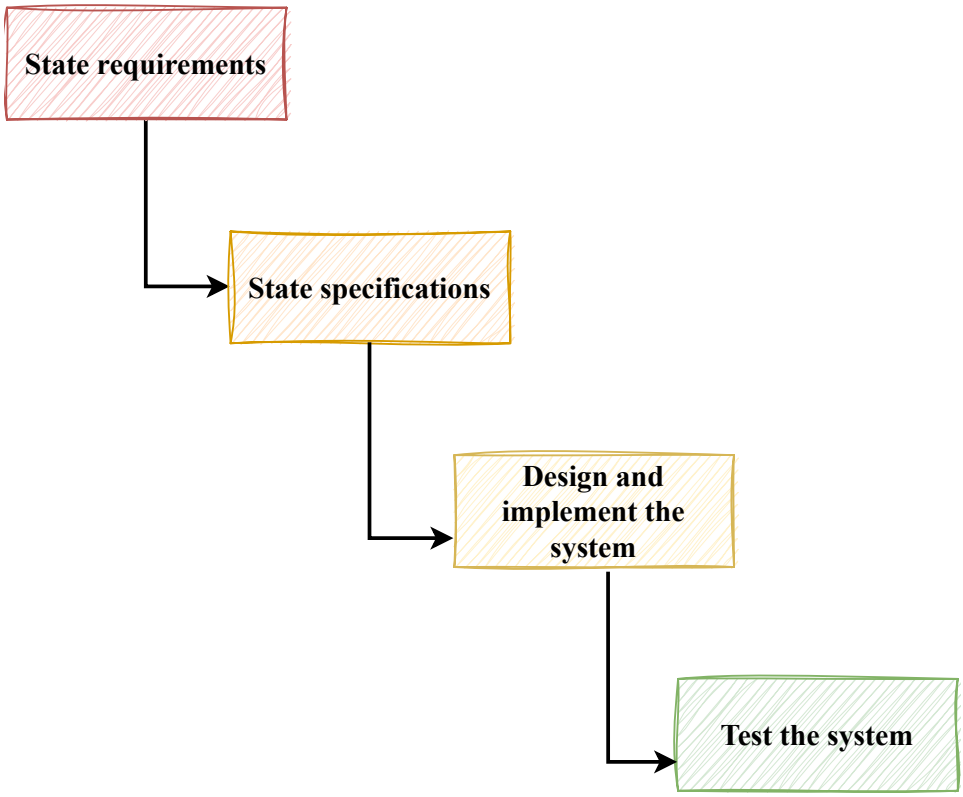


Figure 4.1: Research Methodology

5 Approach

This chapter will shed some light on the approach that has been taken to materialize the following ultimate goal of this thesis -

> Multi-site multi-synchronous support for SQLite augmented for Local-First Software.

In this thesis, we began by utilizing SynQLite [4] as our starting point. The initial implementation of SynQLite augments CRR with the SQLite database in order to add the Local-First property to an existing SQLite database. Local-First property means having the ability to work offline and then synchronize with peers or other replicas while online. However, since the implementation was in its early stage, it had limited support for managing the state when multiple sites were frequently connected and disconnected. Hence, the primary objective of this thesis was to extend the existing implementation of SynQLite by providing multi-site multi-synchronous support. In other words, to improve or update SynQLite to make it practical and ready for end-user usage.

5.1 Agile Software Development Model

We decided to follow the Agile software development model which is an incremental process approach that allows actualizing the system in small incremental steps [20]. In the development phase, the system is developed in *Sprints*,

which is an iterative cycle where a single unit of the system or a single feature is implemented, evaluated, and corrected. My supervisor and I used to have bi-weekly meetings throughout the entire thesis period. Since Agile is an iterative design and development model with short cycles or sprints that enable fast verification and corrections, we found it the most fitted approach for us to leverage. In addition to that, another essential property of the Agile approach is that it is an adaptive approach with probable emergent new risks that also allows us flexibility during the entire period.

I employed a substantial amount of time investigating the property of the Local-First software, CRDT, CRR, and the existing codebase, at the inception of our thesis. This immensely helped me to find out a bunch of potential features which could augment the SynQLite with multi-site multi-synchronous support. With those features in hand, I broke down our main requirements and specifications into several small requirements and specifications for each individual feature. After that, I started developing those features in the Agile software development model fashion. My supervisor and I decided on two weeks as our Sprint since we used to have bi-weekly meetings to track our progress.

At the beginning of each Sprint in our meeting, we would discuss and then pick up the most potential feature from our remaining feature list. After that, I would try to implement that feature. After the implementation, I would perform a technical evaluation test to see whether the implementation met the requirement or not. Since we were working on a distributed system, sometimes it became really difficult for us to test the feature because of the unpredictable nature of the distributed system. As a result, some feature development took more than one Sprint. During this period, I also used to find more potential features and bugs, and if I could find any, then I would discuss them with my supervisor in the next meeting. And after the discussion, we used to append them to our feature list. Figure 5.1 shows the approach we have taken during the entire thesis period. Needless to say, besides the fixed bi-weekly meeting, if I needed to discuss something with my supervisor in the middle of a Sprint, he would find some time to arrange a meeting. With his constant support, motivation, and guidance, and with my persistent dedication, we achieved our final goal.

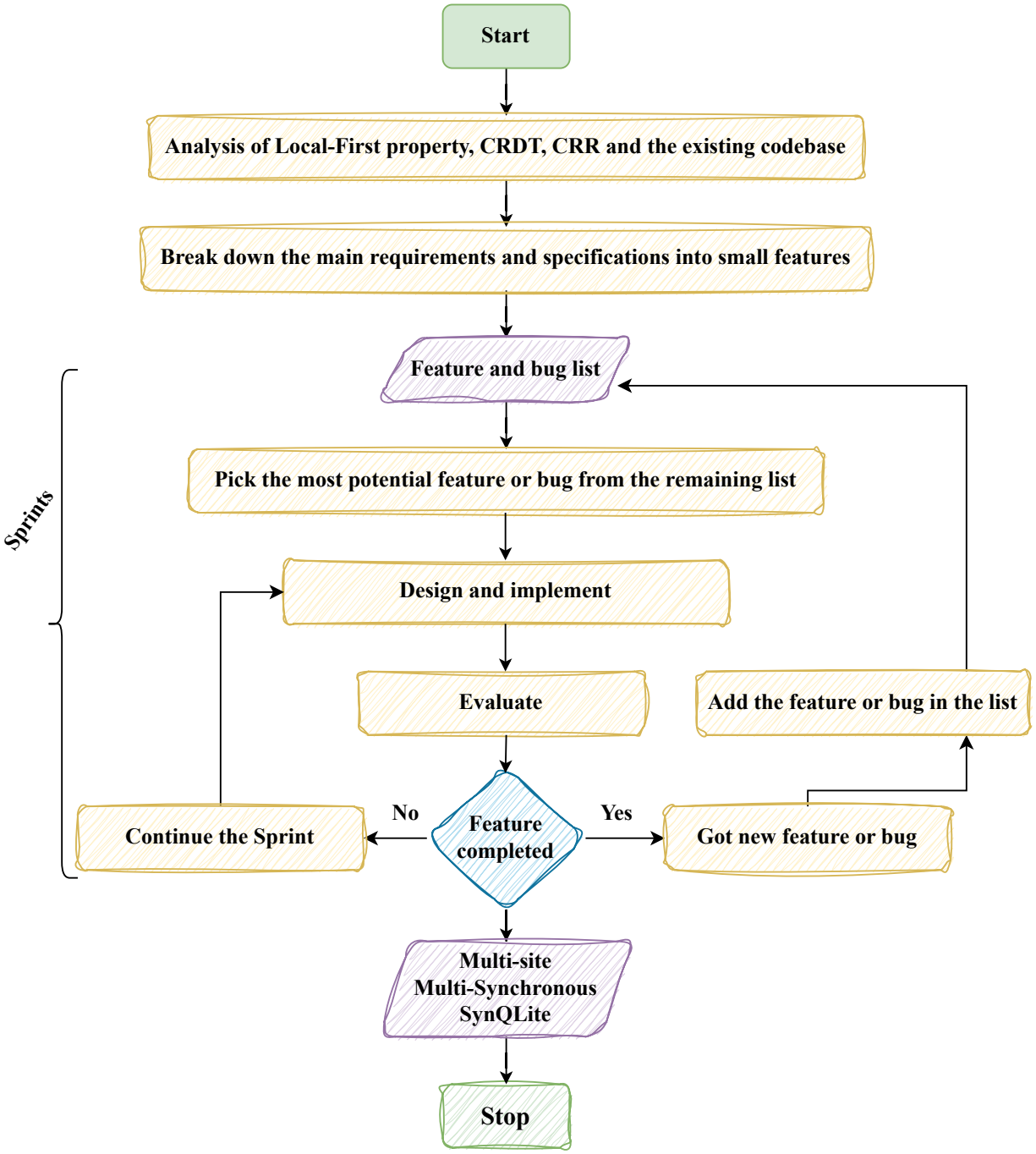


Figure 5.1: Approach overview

5.2 Technology Choices

5.2.1 Implementation

Since I have added additional features on top of SynQLite, I was obliged to use the same technologies it was implemented with, such as Python3, CRDT, CRR, etc. Additionally, I also have employed *Docker Container* to simulate a remote site, which allowed me to test the system's ability to work with multiple sites [21].

5.2.2 Evaluation

I have utilized *Pytest*, a Python testing framework, to write unit tests for a few of our features and bugs. After the implementation of a single feature, I would run all the test cases to test whether the new implementation has broken down the earlier implementations or not. This approach tremendously gave me a better control over the code. I also have utilized *Python* and *Shell Scripts* extensively to automate and design various test case scenarios for multiple sites so that I can verify whether we have been able to accomplish our goal to make the SynQLite multi-site multi-synchronous [22].

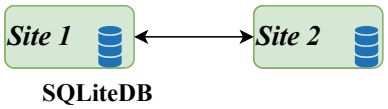
5.2.3 Development Environment

We took advantage of Git version control throughout the development process of our system [23]. I used to maintain separate separate Git branches for implementing a single feature. This allowed me to experiment and modify the existing codebase robustly without the fear of affecting previous work. Besides, I have done all the implementation on a Windows machine. However, I have leveraged Windows Subsystem for Linux [24] to evaluate the implementation in both Windows and Linux environments.

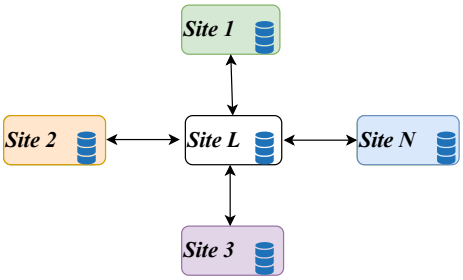
6

Design and Implementation

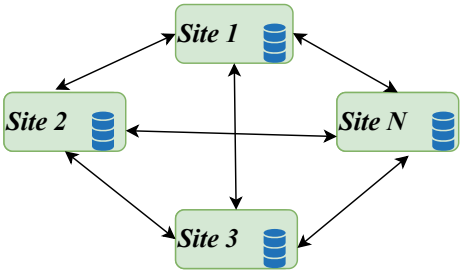
This chapter aims to showcase the features we have implemented on top of the existing SynQLite codebase to enable multi-site multi-synchronous support. In Chapter 3, we discussed the existing state of the SynQLite implementation, which adequately handles two-site scenarios, as shown in Figure 6.1a. In other words, when only two sites exist in the distributed system, they can work independently while offline and synchronize with each other when online to maintain a consistent state using SynQLite commands like PULL and PUSH. In Chapter 3, we also discussed an attempt that has been taken in [9] to address multi-site synchronization issue with a centralized leader-based approach, where a single site is elected as a leader in the distributed system and multiple sites connect with this single leader for synchronization. From the member site's perspective, there are only two sites, the leader and the member itself. Setting aside concerns about performance and potential single points of failure, it should be noted that the existing implementation of the centralized leader-based approach still falls short in terms of ensuring a consistent state when multiple sites connect and disconnect frequently, as depicted in Figure 6.1b. Our thesis focuses on ensuring correctness, meaning all sites should converge to a consistent state after synchronization, a fundamental property of Local-First software, as shown in Figure 6.1c. We want to emphasize that our approach is intended to handle multi-site scenarios, not just two, which we will demonstrate in the following sections.



(a) The existing implementation functions correctly only for a scenario involving two sites



(b) States of multiple sites diverged with Leader based existing Implementation



(c) The goal of this thesis is to make SynQLite work with multiple sites

Figure 6.1: Difference between existing SynQLite functionality and our goal for this thesis

6.1 Feature Lists

This section presents a list of features that have been identified through an in-depth analysis of the characteristics of Local-First software, CRDT, CRR, and the existing SynQLite codebase. These features aim to ensure state convergence in scenarios where multiple sites are frequently connected and disconnected to a great extent. The features will be outlined as follows:

- Partition Handling
- Adapting the delta-generation logic to support multiple sites
- Eventually, all sites should be aware of each other
- Fix the idea that the primary key is always auto-incremented

- Fix the idea that a table can not be empty during the clone operation
- Cross-platform Support

6.2 Implementation

This section aims to provide a comprehensive account of each feature mentioned in Section 6.1, detailing their implementation and functionality.

Feature: Partition Handling

Description Assuming that all nodes in a distributed system will always be up and running is not a practical approach, and it is not sustainable to rely on all nodes being constantly accessible. At the moment, if, for some reason, a site that is known to another site becomes unavailable, then the latter site will not converge with other available sites, which goes against the principle of the Local-First nature of the service. Figure 6.2 shows the loophole of the existing feature.

In Figure 6.2, we see that *Site B*, *Site E*, *Site F* have been cloned from *Site A*. On the other hand, *Site C* has been cloned from *Site B*. In addition, *Site G*, *Site H* have been cloned from *Site C*. With the existing implementation, if for some reason *Site B* becomes unavailable, then *Site A* or *Site C* will not be able to perform a PULL or PUSH operation. However, in order for the Local-First property to be upheld, it is important for both sites to continue functioning even in the event of any other site becoming unavailable. They should get updates from other available sites such as *Site E*, *Site F*, *Site G*, and *Site H*. We have solved this partition handling issue. Moreover, if we now delete *Site B* permanently, other sites will get updates from each other which were not possible with the previous implementation.

Implementation Properly handling exceptions in a program is crucial to prevent program execution from halting abruptly without any further progression. Unfortunately, in the existing implementation, exceptions were not adequately managed. Specifically, during synchronization, the program first iterates over its meta_site(S) table and attempts to establish an SSH connection with the sites listed in the table. This connection enables the site to exchange updates with other sites. However, an exception occurs if a site is unavailable when the program tries to connect with it, and the program terminates without attempting to connect with other avail-

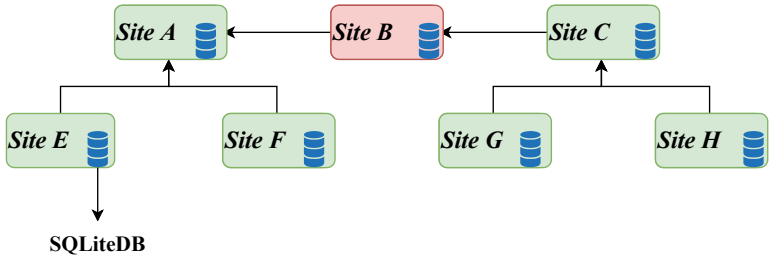


Figure 6.2: Partition Issue

If Site B becomes unavailable, Site A and Site C will not be able to converge with other available sites

able sites. We addressed this issue by enhancing exception handling in the existing implementation, effectively resolving the partition handling problem.

Feature: Adapting the delta-generation logic to support multiple sites

Description In Chapter 3, we learned that SynQLite's implementation utilizes Delta-State CRDT. A significant aspect of this approach involves the generation of deltas. However, in the previous implementation of SynQLite, the delta-generation logic was solely designed for two sites, so it only generates correct deltas when only two sites are involved in the system. Suppose we clone multiple sites from one another. In that case, the delta-generation logic does not work properly, causing the sites to not converge after synchronization operations as depicted in Figure 6.3.

In Figure 6.3, we see the states of *Department* table in three different sites. The snapshot of the states is shown after *Site B* and *Site C* have been cloned from *Site A* when the *Department* table had a single tuple (1, CSE). After that, *Site B* inserted tuple (2, ME) into the table, and *Site C* inserted tuple (2, EEE) into the table. Then we performed the synchronization operation at all the there sites. We clearly see that the table states remained diverged after the synchronization operation. While *Site A* received updates from both *Site B* and *Site C*, sites *B* and *C* did not receive updates from each other. This is due to faulty delta-generation logic, which we have solved in this thesis.

Implementation To solve this issue, we slightly changed the schema of the CRR layer. We found that the S(last_in) and S(last_out) attributes of the

Id	DepartmentName
1	CSE
2	ME
3	EEE

(a) Department Table at *Site A*

Id	DepartmentName
1	CSE
2	ME

(b) Department Table at *Site B*

Id	DepartmentName
1	CSE
2	EEE

(c) Department Table at *Site C*

Figure 6.3: Delta-generation logic issue

States of the sites are different after synchronization with one another due to the delta-generation logic issue

meta_site(S) table from Figure 3.2 are sufficient to generate the corresponding changes or deltas (δ) between only two sites. The previous delta-generation logic is shown in Code Snippet 6.1 and as an example visually explained in Figure 6.4. In the example, (3) if *Site A* generates δ for *Site B*, (4) the previous delta-generation logic gleans newly inserted, updated, or deleted rows from $\tilde{R}$ and meta_history(HR) table from *Site A* (5) with a later timestamp than S(last_out) for *Site B*, i.e., HR(updated_at) > S(last_out) where S(site_id) is *B*'s site id.

In this case, if, in the meantime, (1 & 2) *Site A* receives any updates from other available sites in which HR(updated_at) are less than *Site B*'s S(last_out), then (6) these rows will not be included in the δ. On the other hand, if, in the meantime, *Site B* got updates from other sites, *Site A* may also include the duplicate rows in the δ. The latter scenario is not a critical issue since, during the merge operation, *Site B* will handle the rows which are already in the CRR layer. However, (7, 8 & 9), because of the former issue, the state of the sites will diverge since *Site B* will not get all the unseen or modified rows.

Code Snippet 6.1: Previous Delta Generation Logic.

```
        def _select_delta_sql(site_id):
            sql_query = f"""
                SELECT *
                FROM ~R as crr
                inner join HR as hist
                on (crr.crr_id == hist.crr_id )
                WHERE hist.updated_at >
                ( SELECT last_out FROM S as site
                WHERE site.site_id = { site_id })
                AND hist.site_id != { site_id }"""
        # site_id is the id of the site that the deltas are for
```

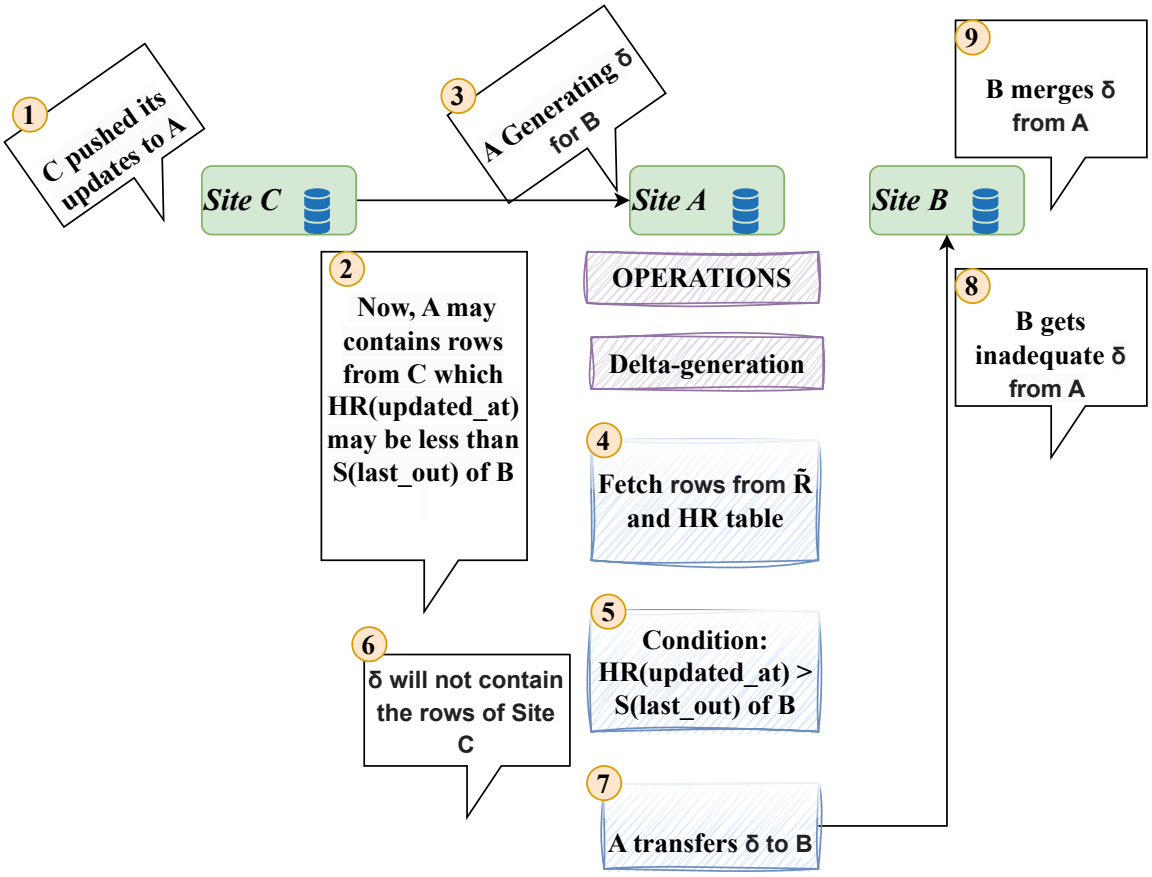


Figure 6.4: Explanation of previous Delta-generation logic

Therefore, the former delta-generation logic can be considered faulty for multiple sites. In fact, we have discarded S(last_in) and S(last_out) variables from the table *S* and added a completely new table named

meta_site_state(SS) in the CRR layer. In Figure 6.5, SS(site_at) indicates the S(site_id) where an update has occurred, SS(site) specifies the S(site_id) who is responsible for the update, and SS(last_update) is the timestamp when SS(site) last updated the SS(site_at).

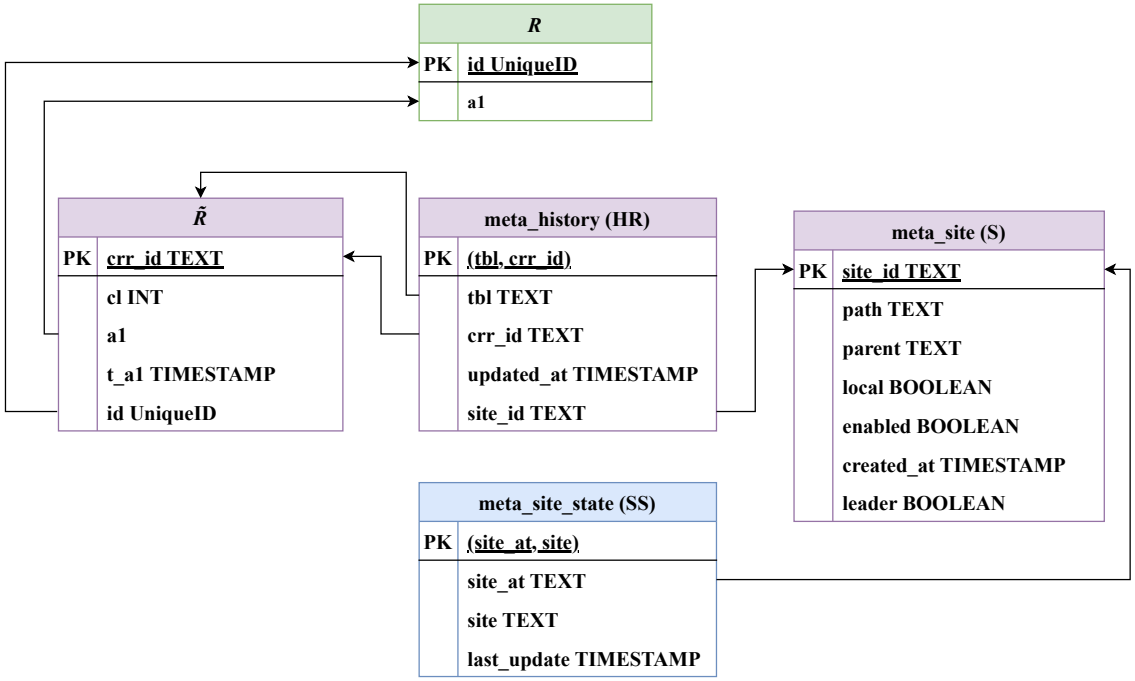


Figure 6.5: Updated CRR Layer tables

In fact, the *SS* table is a way of implementing the state vector in the distributed systems [25]. A state vector is actually a data structure that keeps track of the overall state of the entire system and stores information about the state of each node or process in the system. State vectors are generally used to detect and resolve conflicts that arise when several nodes update the same data at the same time since state vectors are causally ordered, meaning they retain the sequence of events inside the system. To maintain a consistent and up-to-date representation of the system's overall state, each node retains and updates its state information in the state vector and then propagates it to all other nodes. As shown in Figure 6.6 and described in the subsequent paras, we record all of the update information for all possible combinations of sites in the SS table using the format (site_at, site, last_update). As a result, this table serves as a representation of the global state of the entire system and aids in generating the correct δ when there are more than two sites in the distributed system.

site_at	site	last_update
A	A	NULL

(a) When initialized at *Site A*

site_at	site	last_update
A	A	1.5

(b) When Updated at *Site A* at 1.5

site_at	site	last_update
A	A	1.5
B	A	1.5
A	B	NULL
B	B	NULL

(c) When Cloned at *Site B*

site_at	site	last_update
A	A	1.5
B	A	1.5
A	B	NULL
B	B	NULL
C	A	1.5
C	B	NULL
A	C	NULL
C	C	NULL

(d) When Cloned at *Site C*

site_at	site	last_update
A	A	1.5
B	A	1.5
A	B	NULL
B	B	NULL
C	A	1.5
C	B	NULL
A	C	2.5
C	C	2.5

(e) When updated at *Site C* at 2.5 by *Site C* and then PULLED by *Site A*

site_at	site	last_update
A	A	1.5
B	A	1.5
A	B	NULL
B	B	NULL
C	A	1.5
C	B	NULL
A	C	2.5
C	C	2.5
B	C	2.5

(f) When PULLED By *Site B*

Figure 6.6: State information in meta_site_state (SS) table

How the *SS* table is populated During the initialization process, SynQLite inserts a single row into the *SS* table, where both the SS(site_at) and SS(site) are set to the site_id of the initialized site, and the SS(last_update) field is initially left as NULL and then updated when the site is updated as shown in Figure 6.6a and Figure 6.6b.

During the cloning process, the *SS* table is also updated along with the *S* table as mentioned in Chapter 3. SynQLite inserts all tuples of the form (cloned_site_id, site, last_update) into the cloned *SS* table, where the "site" value is extracted from all tuples of the form (original_site_id, site, last_update) that exist in the *SS* table during the cloning time. Additionally, it appends two more tuples to the cloned *SS* table, which are (original_site_id, cloned_site_id, NULL) and (cloned_site_id, cloned_site_id, NULL). In the original site's *SS* table, SynQLite also inserts the same tuples. The state information after the cloning process is depicted in Figure 6.6c and Figure 6.6d.

During the Synchronization operation, if *Site A* generates δ for *Site B*, at first SynQLite inserts all possible tuples of the form (*B*'s site id, site, last_update) in *Site A*'s *SS* table. The tuples are gleaned from existing rows in the table of the form (*A*'s site id, site, last_update). If (*B*'s site id, site, last_update) is not already in the table, then the last_update is set to NULL. Then SynQLite runs the adapted delta generation logic at *Site A*, which we are going to describe in the following para, aggregating all newly inserted, deleted, or updated rows from all the sites in the δ for *Site B*. Afterward, *SS* is again updated for the tuples of the form (*B*'s site id, site, last_update), this time the last_update is set to max(last_update of *A*, last_update of *B*) since after generating the δ, *Site B* now will have all the updates. Similarly, after merging the δ, at *Site B*, we update the *SS* table with the same tuples. The state information after the synchronization process is shown in Figure 6.6e and Figure 6.6f.

The adapted delta-generation logic is presented in the Code Snippet 6.2 and, as an example, has been explained visually in Figure 6.7. In this case, (2) if *Site A* generates δ for *Site B*, (4) at site *Site A*, we extracted rows from $\tilde{R}$ and HR table and then (5) join the rows with *SS* table using the condition that SS(site_at) matches *B*'s site id. Subsequently, (6) we filter the resulting rows using the condition: HR(updated_at) > SS(last_update). We also prune out the rows if the update has been solely done by *Site B*. Since (3) we update the *SS* table in such a way to include all possible tuples of the form (*B*'s site id, site, last_update) as discussed in the previous paras, (7) the δ will contain all the updated rows performed by any site, (9, 10, 11, & 12) which need to be sent to and merged at site *Site B* in order to ensure that it is consistent with all other

sites. Therefore, using the *SS* table as a state vector assures that all the rows will be included in the deltas, and thus the sites will converge to the identical state.

Code Snippet 6.2: New Delta Generation Logic.

```
        def _select_delta_sql(site_id):
            sql_query = f"""
                SELECT *
                FROM ~R as crr
                inner join HR as hist
                ON (crr.crr_id == hist.crr_id)
                INNER JOIN
                (SELECT site, last_update
                FROM SS
                WHERE site_at = {site_id})
                AS state
                WHERE state.site != {site_id} AND
                        hist.updated_at > state.last_update
                ORDER BY hist.updated_at"""
        # site_id is the id of the site that the deltas are for
```

Feature: Eventually, all sites should be aware of each other

Description As previously mentioned, the existing approach is effective when only two sites exist. This thesis aims to create multiple copies of databases in a distributed manner and ensure that all replicas have the same information when updates are exchanged between them. However, the existing system only allows for synchronization between sites that have a direct parent-child relationship. This limitation prevents the convergence of sites if an intermediate site stops updating its state.

Consider an example, as shown in Figure 6.8, we see that *Site B* has been cloned from *Site A* and *Site C* has been cloned from *Site B*. Now if we update *Site C* and then, we do *Pull A*, then *Site A* will not get the updates from *Site C* since *Site A* is not the direct parent of *Site C*. But after that, if we do *Pull B*, then *Site B* will get the updates from *Site C* since *Site B* is the parent of *Site C*. Afterward, if we do *Pull A*, then *Site A* will finally get the updates and will converge since now *Site B* contains the updates of *Site C*. But if, for some reason, the user who uses *Site B* stops using *Site B* and never do *PULL B*, then the updates of *Site C* will never be propagated to *Site A*.

Therefore to ensure consistency across all sites in a distributed system, each site must have knowledge of the meta information of all other sites.

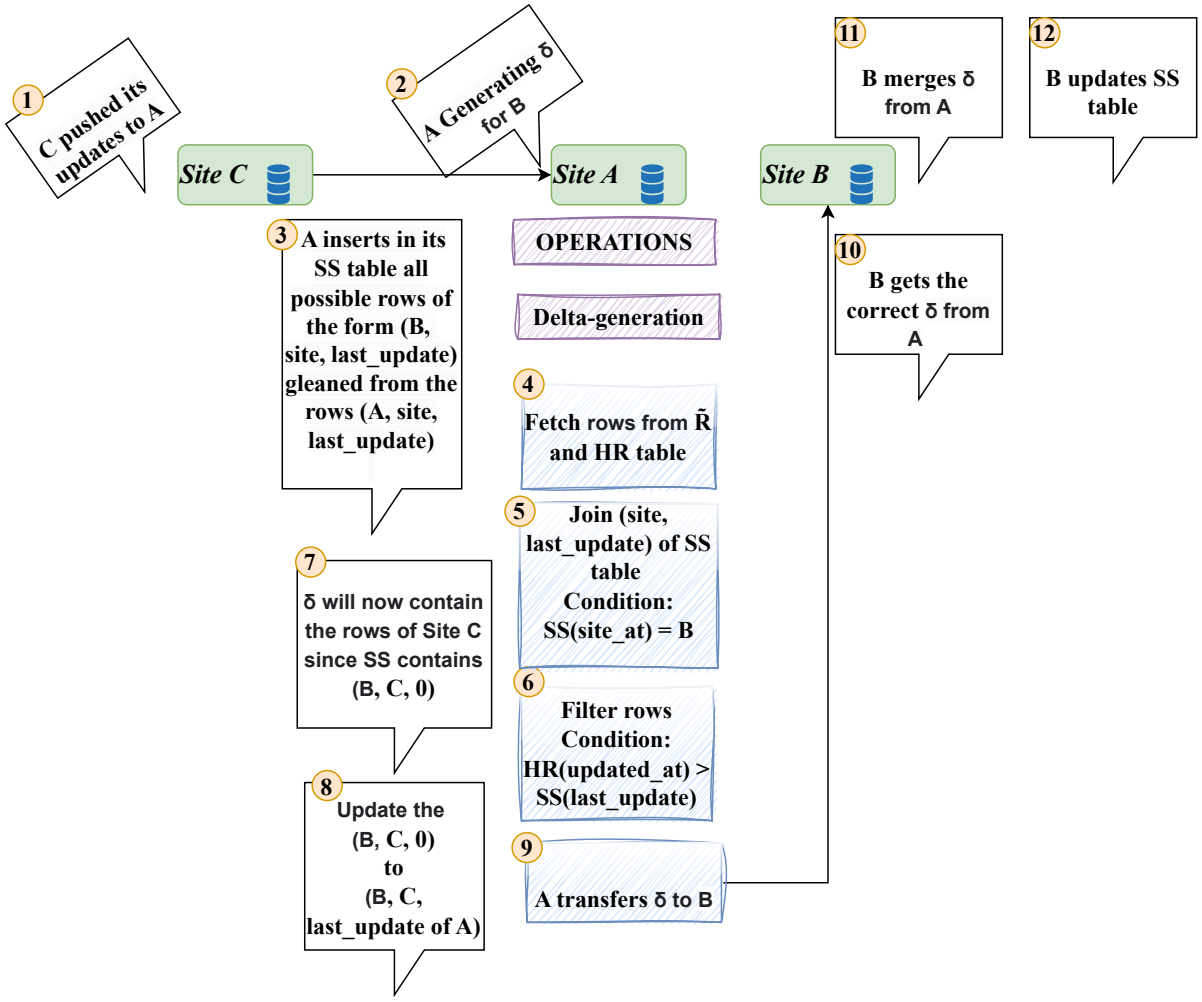


Figure 6.7: Explanation of adapted Delta-generation logic

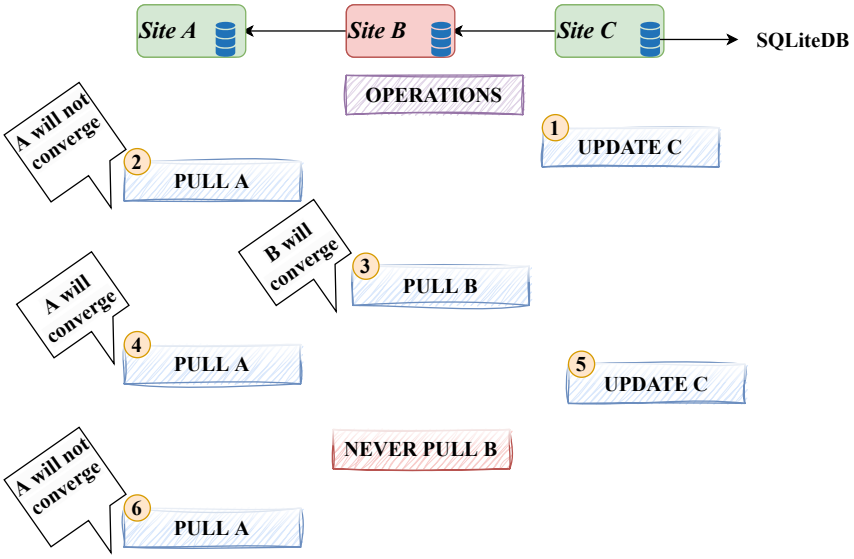


Figure 6.8: Parent site may never be converged with child's child site

Implementation We know that in the CRR layer of a database, the meta_site(S) table serves as the repository for all the pertinent information about all known sites. With the previous implementation of SynQLite, the *S* table is only updated when a site is cloned from another site. After the database of a site has been successfully copied to another site through the cloning process, the cloned site would inherit all the information from the parent site's *S* table, and then both the cloned and parent's *S* table would be updated with the newly created site's information. However, with this approach, the *S* table of other sites in the distributed system would not receive information about the newly created site. We have solved this issue with two approaches.

- Once the database is successfully copied to a new site, we will inform not only the parent site but also all the sites listed in the parent's *S* table about the creation of the new site. This way, all sites will be aware of the new site. However, if some sites are offline or there is a partition between certain pairs of sites during the cloning process, those sites will not receive the information about the new site. To address this potential problem, we have implemented a logic to ensure that all sites are informed about the creation of the new site during the synchronization process.

- During the process of synchronization in SynQLite, δ are prepared and transmitted to other sites to achieve convergence through PULL or PUSH operations. These δ are typically composed of updated or unseen rows from the $\tilde{R}$ and HR tables, limiting the convergence to only these tables. However, in our implementation, we have also generated δ of S table and combined them with the previous δ. After transmitting the δ to the other sites, it receives information about the new sites and proceeds to perform the corresponding PULL or PUSH operations for these new sites by iterating through them.

Note that we not only update the S table of all the sites present in the distributed system during the cloning or synchronization process as described above but also the SS table with the logic described in the implementation part of the previous feature. Through this, we guarantee state convergence for multiple sites.

Feature: Fix the idea that the primary key is always auto-incremented

Description We know that whenever any modifications are made to relations in the AR layer table, identified as R, the CRR layer triggers will automatically make corresponding changes in the $\tilde{R}$ tables of the CRR layer. The previous implementation assumed that the R tables would always have auto-incremented primary keys. After inserting a relation into the R table, the insert trigger will check if the corresponding row is already present in the $\tilde{R}$ table by looking at the primary key. If it is present, the trigger will simply increase the $\tilde{R}$(cl) or causal length and no other changes will be made.

In a scenario where the user does not define the primary key in the R table as an auto-incremented primary key; after a row is deleted and a new row is inserted, the new row will have the same primary key as the deleted row in the R table, as illustrated in Figure 6.9a. In this case, since the primary key already exists in the $\tilde{R}$ table, the insert trigger will not insert the new relation into the $\tilde{R}$ table. Instead, it will only update the $\tilde{R}$(cl) and not update any other column values as shown in Figure 6.9b and Figure 6.9c in the $\tilde{R}$ table.

In the scenario outlined above, the R table on the local site may have the most recent information. However, because the $\tilde{R}$ table is not updated correctly, if the local site pushes its updates to other sites or if other sites perform a pull operation, they will not receive the most recent information. This is because, when updates are sent to other remote sites, the δ

pk	a1	a2
1	DS	INF-3200
2	PP	INF-3201
2	ADS	INF-3203

(a) R table

crr_id	cl	t_a1	t_a2	a1	a2
-	1	-	-	DS	INF-3200
-	2	-	-	PP	INF-3201

(b) $\tilde{R}$ table after the first deletion

crr_id	cl	t_a1	t_a2	a1	a2
-	1	-	-	DS	INF-3200
-	3	-	-	PP	INF-3201

(c) $\tilde{R}$ table after the following insertion

Figure 6.9: Fix Auto-incremented Primary key issue

rows in the $\tilde{R}$ table are sent and merged with $\tilde{R}$ table of the remote site, and then the relations trickle up to the R table of the remote site. In this case, the new row in the R table of the remote site will have outdated information since the $\tilde{R}$ table does not contain the new information; it just has an increased $\tilde{R}$(cl). This will cause the states of the sites to diverge from each other.

Implementation The easiest solution to this problem would be to impose a restriction on SynQLite users that their tables must have an auto-incremented primary key. However, this would limit the functionality and usability of our SynQLite service. Therefore, to avoid limiting the functionality and usability of SynQLite, we have modified the insert trigger logic from Code Snippet 6.3 to Code Snippet 6.4. Note that a simplified version has been presented to simplify the trigger logic for easy understanding.

The previous logic has been explained in the description section. Under the new logic, when an insert trigger is invoked, it checks whether the primary key of the R table already exists in the $\tilde{R}$ table. If it does, the trigger uses the earlier crr_id and causal length instead of generating new ones. Note that we have employed the SQL COALESCE function for this, which returns the first non-null value. To guarantee the insertion of a new row into the $\tilde{R}$ table, a "REPLACE INTO" command is utilized. This command deletes the previous row with the same primary key, which is the crr_id in this case, and inserts the new row with updated information.

On the other hand, using the INSERT command would result in failure to insert a new row with an already existing primary or unique key.

Code Snippet 6.3: Previous Insert Trigger.

```
def insert_trigger():
    sql_query = f"""
        INSERT INTO ~R
        (crr_id, cl, id, t_id, ...)
        VALUES
        (new_crr_id, 1, new.id, timestamp, ...)
        WHERE NOT EXISTS
        (SELECT * FROM ~R
        WHERE new.id = ~R.id)"""
```

Code Snippet 6.4: New Insert Trigger.

```
def insert_trigger():
    sql_query = f"""
        REPLACE INTO ~R
        (crr_id, cl, id, t_id, ...)
        VALUES
        (COALESCE(
        (SELECT crr_id FROM ~R
        WHERE new.id = ~R.id),
        new_crr_id),
        COALESCE(
        (SELECT cl FROM ~R
        WHERE new.id = ~R.id)
        , 1),
       new.id, timestamp, ...);"""
```

Feature: Fix the idea that a table can not be empty during the clone operation

Description There is another issue in the earlier implementation of SynQLite. If a table of a database is empty while it is being cloned, then after the clone operation, the state of the table will not converge with one another during the synchronization operation when multiple sites exist.

Consider Figure 6.10, where *Site B* has been cloned from *Site A* while the *Department* table was empty. Then *Site C* has been cloned from *Site B*. Afterward, we inserted a row in *Department* table both at *Site A* and *Site C*. Therefore, both the local site will have the row with the primary key one, as shown in the figure (1 & 2). Following that, if we perform the PULL operation on *Site B*, we will see that *Site B* will not converge with

Site A or *Site C*. Let's take a close look at the corresponding $\tilde{R}$ table of *Department* table at *Site B* (3). We will notice that it has got the update from both the sites, but due to the same primary key issue, the update will not trickle up to the *R* table or the *Department* table. If *Site A* or *Site C* also perform PULL operation or PUSH operation, they will also not converge with one another.

Just like the problem with the auto-incremented primary key, the simplest solution is to restrict SynQLite users from inserting at least one row before making a copy of the database on a remote site with CLONE operation. However, it would negatively impact the usability of the SynQLite service. Therefore we have solved the issue in this thesis.

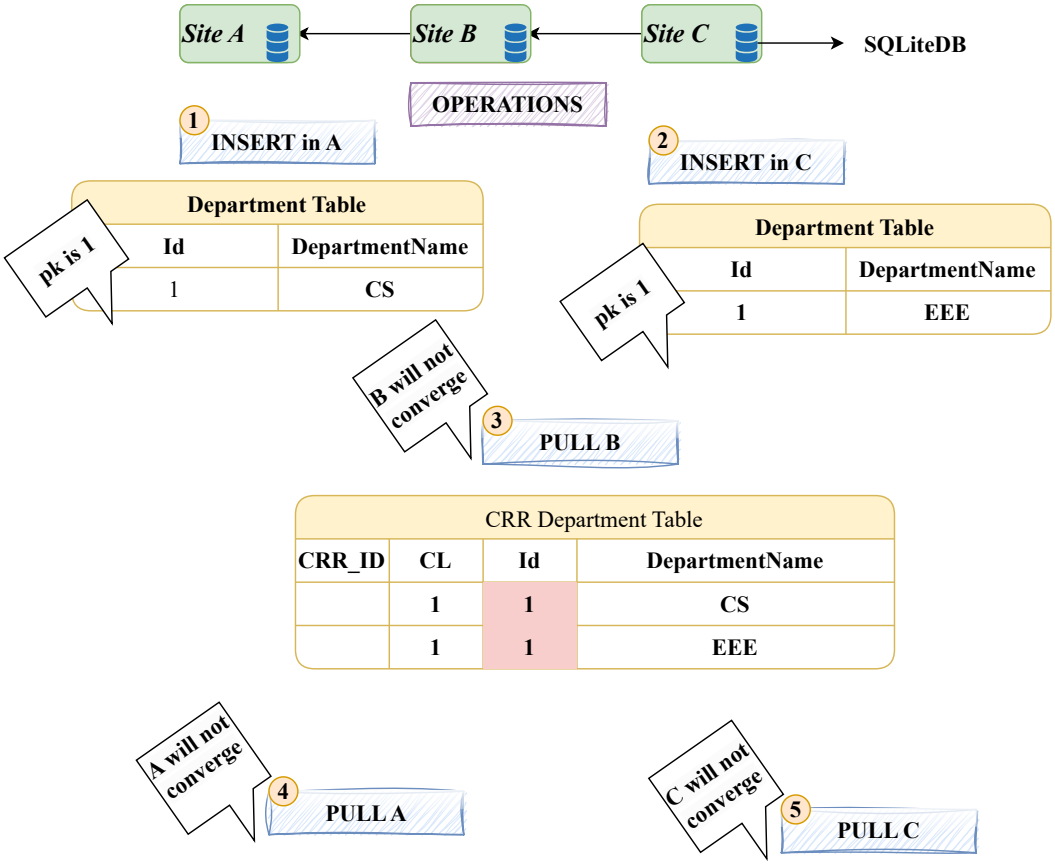


Figure 6.10: Empty table clone Issue

The tables must not be empty during the clone operation, as doing so sites states will diverge

Implementation After investigating the cause of the issue, we found that in the merge function, there was an assumption that the $\tilde{R}$ table in the merging site would not be null. The function fetches the primary key information from the existing rows from $\tilde{R}$ and takes the necessary steps to merge based on the information. If the $\tilde{R}$ table is initially empty, it will not get the necessary information and thus will fail to merge the rows. We removed that assumption from the implementation.

Feature: Cross-platform Support

Description Since we are developing a system or service for a distributed and diverse environment, we want to give users the flexibility to use any platform on their site. Previously, SynQLite was only compatible with Linux, but we have added support for Windows so that users on both operating systems can use the service simultaneously. When multiple sites are connected, regardless of the environment at each site, they will all reach a consistent state after performing PULL or PUSH operations with other sites. Figure 6.11 highlights the feature.

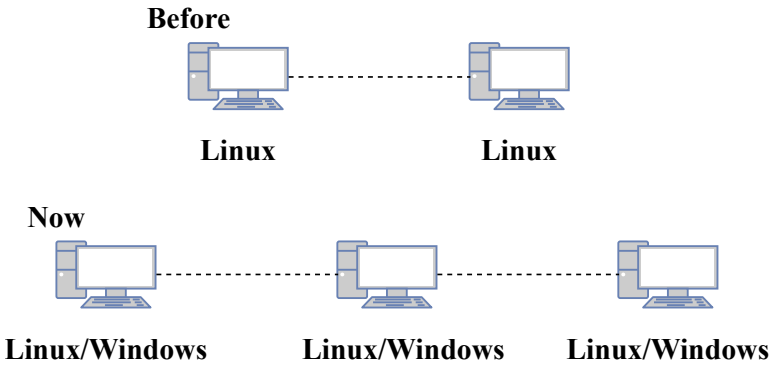


Figure 6.11: Cross-Platform support

Implementation During the cloning or synchronization process of a remote database file to a local system, SynQLite requires a temporary directory to store the file temporarily. However, the existing implementation of SynQLite assumes a Linux directory structure when creating the temporary directory to store the file. This can cause issues when attempting to clone a remote database onto a Windows machine, as the directory structure is completely different and may cause the temporary file creation to fail, resulting in SynQLite not functioning properly. Previously, the temporary files were stored in the "/root/tmp" directory. However, with the

updated implementation, the system environment is first checked, and the temporary path is then calculated based on this environment. We know from Section 3.2 that the user specifies the local path where they want to clone the database using the SynQLite CLONE command, and then SynQLite creates a temporary directory based on this path to store the file, ensuring that the cloning process is successful regardless of the system environment.

To sum up, implementing multi-site multi-synchronous support on top of SynQLite required us to address the key features and bugs mentioned earlier in this chapter. However, we didn't stop there and also made additional modifications to enhance the performance and functionality of the system. One noteworthy modification we made was adding a feature that allows temporary files to be wiped out after they have served their purpose. This optimization helps to free up storage space and contributes to an overall enhancement of the system's performance. Aside from this, we also refactored the codebase to enhance its readability and ease of maintenance. Overall, our efforts have helped to make SynQLite a more capable and efficient system for multi-site multi-synchronous support. These modifications have improved the performance and functionality of the system, making it a more reliable tool for managing data across distributed sites.

7

Experiments

This chapter outlines the experiments we conducted to justify our implementation of SynQLite. Our primary focus was on ensuring the accuracy and correctness of SynQLite, rather than optimizing its performance. The previous version of SynQLite was unable to function effectively in a distributed system with multiple sites. To adhere to the Local-First property, SynQLite should allow offline work and ensure that the states of the different sites converge when online. SynQLite is a service that provides the Local-First property to SQLite databases, so working offline is not a significant concern. However, the previous implementation of SynQLite failed to handle synchronization issues, which limited its effectiveness to only two sites. This is not practical in real-life scenarios where a service may be used by numerous users. Therefore, we implemented new features and addressed existing bugs to ensure that SynQLite functions appropriately for multiple sites. We have categorized our experiments into two types: the first type validates the accuracy and correctness of SynQLite, which is our primary goal, and the second type evaluates the overall performance of SynQLite after implementing our features.

7.1 Correctness Validation

In order to ensure the accuracy and correctness of our system, we developed several test cases. The purpose of these test cases was to demonstrate that when multiple sites are present in a distributed system, all sites converge to

an identical state following the synchronization operation. The test cases were written in both Bash and Python scripts. With the help of these test cases, we automatically simulate sites, create databases on a site using a database schema shown in Code Snippet 7.1, augment the CRR support to the database, clone the database to several other sites, randomly insert, update, and delete rows in database tables and then perform the synchronization operation. We ran these scripts on both the previous implementation and our updated implementation and then observed the state of the databases by outputting the states to a text file. To do this, We leveraged SQL **SELECT** operation to output the entire table state of a site and then write the output in a text file. We then compared the states to determine if they were identical. In this section, We will highlight some of our test cases that validate the correctness of our implementation.

Code Snippet 7.1: Database Schema for Test Cases

```
1  CREATE TABLE [Departments] (
2          [DepartmentId] INTEGER  NOT NULL PRIMARY KEY,
3          [DepartmentName] NVARCHAR(50)  NOT NULL
4  );
5  CREATE TABLE [Students] (
6          [StudentId] INTEGER  PRIMARY KEY NOT NULL,
7          [StudentName] NVARCHAR(50) NOT NULL,
8          [DepartmentId] INTEGER  NULL,
9          [DateOfBirth] DATE  NULL
10 );
```

7.1.1 Test Case 1

The design of Test Case 1 is shown in Code Snippet 7.2. Following the design, we developed a script to execute the test case. A simple output obtained from executing the test case on both the previous and updated implementations is illustrated in Figure 7.1. After Step 6, we notice that the Department table's states are not consistent across all sites when the test case is executed on the previous implementation. Regardless of the number of PULL operations performed and the number of times the script is executed from Step 4 to Step 6, the table's state did not converge after the PULL operation with the previous implementation. Upon close inspection, we observed that Site A was receiving the newly inserted rows from all other sites, while Site B, Site C, and Site D were receiving the newly inserted rows of Site A but not from each other. In contrast, our updated implementation produced an output that showed all the sites having all the newly inserted rows from each other, confirming the correctness of our implementation. Therefore, this test case successfully validates the accuracy and correctness of our updated implementation.

Code Snippet 7.2: Design of Test Case 1

```
1 Step 1: Create and initialize a database at Site A
2 Step 2: Insert rows in the Department table at Site A
3 Step 3: Consecutively Clone Site B, Site C, and Site D from Site A
4 Step 4: Insert a row in the Department table at all the sites
5 Step 5: Perform PULL operation at all the sites
6 Step 6: Output the state of the Department table of all the sites
7 Step 7: Perform Step 4 to Step 6 several times
```

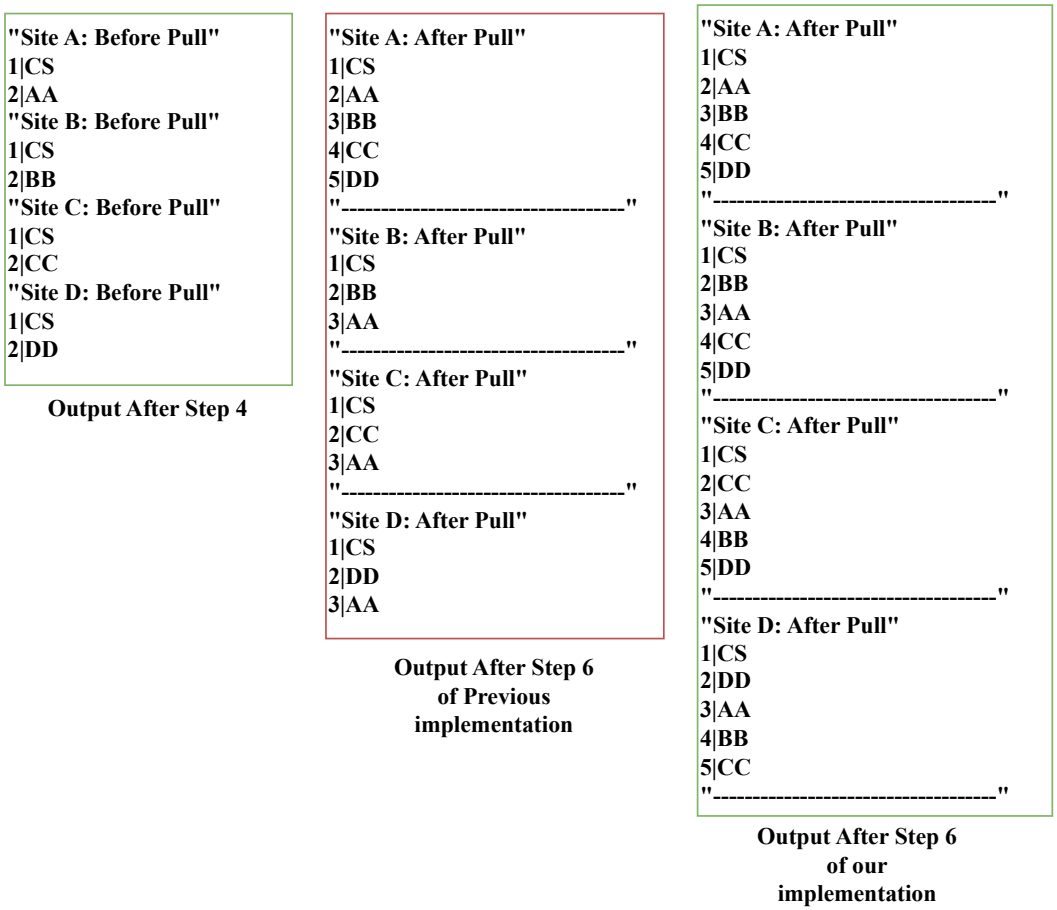


Figure 7.1: Output of Test Case 1

7.1.2 Test Case 2

Test Case 1 involved the examination of four different sites, three of which were created through the process of cloning from a single site. Test Case 2 was designed to present a more complex scenario, utilizing a total of seven sites as

illustrated in Figure 7.2. The process began by initializing a database at Site A and cloning Site B from it, followed by cloning Site C from Site B. Sites D and E were then cloned from Site A, while Sites F and G were cloned from Site B. Subsequently, rows were inserted in all sites, and a PULL operation was performed at each site. The state of all the sites was then observed, as depicted in Figure 7.3. The previous implementation resulted in states of the sites that were inconsistent with one another, while our updated implementation provided consistent states across all sites. Thus, the accuracy and correctness of our updated implementation were confirmed by Test Case 2.

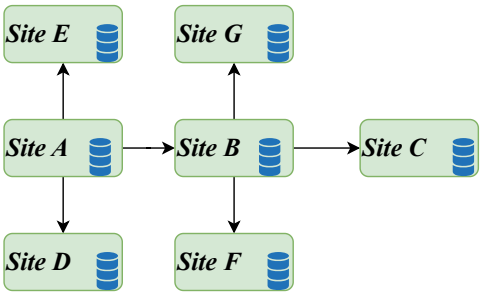


Figure 7.2: Sites Arrangement of Test Case 2

7.1.3 Test Case 3

Once we had tested our implementation with SQL insert operations and various site configurations, we proceeded to evaluate it with both insert and update operations using the same site arrangements. The design for Test Case 3 is presented in Code Snippet 7.3. The output of our test case successfully demonstrated that our implementation ensures the convergence of site states after synchronization operations.

Code Snippet 7.3: Design of Test Case 3

```
1 Step 1: Create and initialize a database
2 Step 2: Clone databases from each other
3 Step 3: Insert rows in a particular table at all the sites
4 Step 4: Perform PULL operation at all the sites
5 Step 5: Output the state of the table of all the sites
6 Step 6: UPDATE rows in that particular table at all the sites
7 Step 7: Perform PULL operation at all the sites
8 Step 8: Output the state of the table of all the sites
9 Step 9: Perform Step 3 to Step 8 several times
```

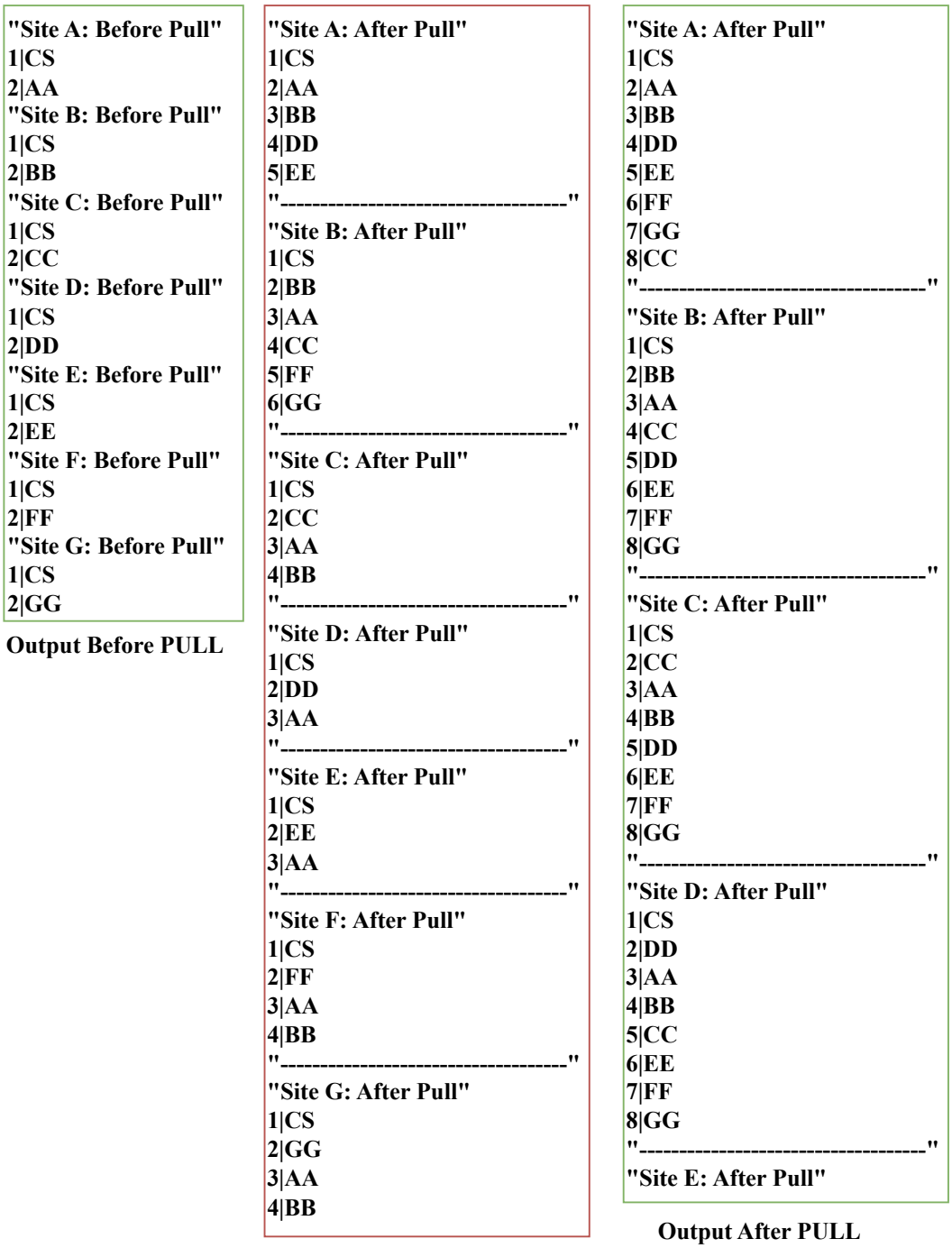


Figure 7.3: Output of Test Case 2

7.1.4 Test Case 4

Our implementation underwent comprehensive testing with all SQL operations, including the delete operation, as we evaluated the behavior of our system. For this test case, following the site arrangement used in previous test cases, we inserted data into the table at each site, performed a PULL operation, and examined the state of the database. We then deleted rows from each database and repeated the PULL operation to observe the states of the sites. However, we discovered that our previous implementation, which successfully converged the databases to a unique state for Test Case 1, Test Case 2, and Test Case 3, did not work when deleting rows and attempting to synchronize the database. An extensive investigation led us to identify the root cause of the issue, which was the assumption of a fixed auto-incremented primary key described in Chapter 6. We addressed the issue and tested our solution on Test Case 4, where we found that the sites converged after the synchronization operation. We retested the previous test cases and confirmed that our implementation effectively converged the states of the sites, even with multiple sites involved.

7.1.5 Test Case 5

During Test Case 5, we executed SQL insert, update, and delete operations across various sites and found that our implementation was able to converge the sites regardless of the operation type. However, we encountered an issue where simultaneous update and deletion of a row from different sites caused a divergence in the site states. We identified and resolved the root cause of the problem, and after rerunning the test case, we were able to achieve successful site convergence.

To conclude, as we conducted our research on distributed multi-site multi-synchronous databases, we made an effort to consider every possible corner case. However, it is conceivable that some corner cases were overlooked. Despite this, we can confidently report that our implementation achieved complete accuracy and correctness for both general and specific corner cases discussed in this thesis.

7.2 Performance Evaluation

Our primary objective was to extend the existing SynQLite implementation by providing multi-site multi-synchronous support, ensuring that the states of all sites would converge to a unified state after synchronization. To validate the correctness of our implementation, we conducted several test cases, as outlined

in the preceding section. However, during the implementation process, we not only focused on ensuring that the features worked but also on evaluating their efficiency. To this end, we carried out a series of performance tests on our implementation. In this section, we will discuss the performance tests we performed on our implementation. The specification of the machine on which all the performance tests were executed is:

Processor Processor 12th Gen Intel(R) Core(TM) i7-1255U, 1700 Mhz, 10 Core(s), 12 Logical Processor(s)

RAM 16 GB

Disk SAMSUNG MZVLQ512HBLU-00B 512 GB SSD

OS Microsoft Windows 11 Home

7.2.1 Experiment Setup

We set up our experiment as depicted in Figure 7.4. (1) At first, we utilized a cloning process to set up our experiment, where the first site was initialized, and the subsequent sites were cloned from the preceding site in a sequence, i.e., the second site was cloned from the first, the third from the second, and so on. We set the initial size of the database to zero. (2) Following site creation, we inserted an identical number of rows to each site. (3) After that, we sequentially performed a pull operation at each site. (4) We then timed how long it took for each site to synchronize. We averaged the synchronization time and incorporated the standard deviation as an error bar on the plotted results. This allowed us to assess the variability in our data and better understand the consistency and reliability of our experimental findings.

It is also important to note that we measured both the Wall time[1] and the CPU time[2] when evaluating execution or elapsed time [26]. This allowed us to distinguish the amount of time the CPU spent executing our program versus the total time taken to complete the program.

1. When discussing computer processing time, Wall time is used to refer to the actual time taken to complete a task. This includes the total duration of three primary elements: CPU time, I/O time, and communication channel delay
2. The CPU time refers to the length of time the CPU is actively engaged in executing a program. It quantifies the duration during which the CPU executes program instructions.

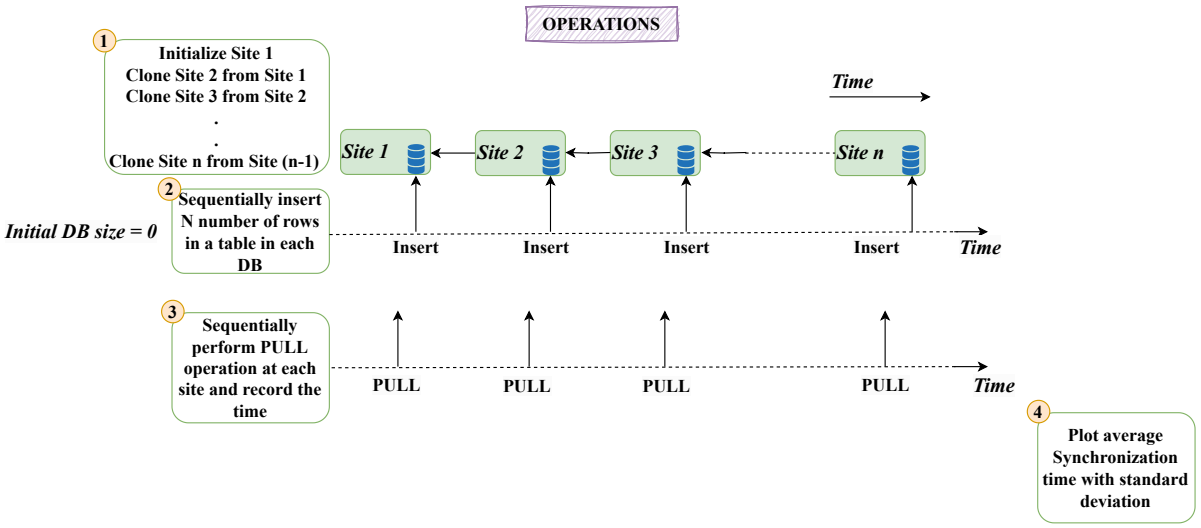


Figure 7.4: Experiment Setup

7.2.2 Number of sites vs Synchronization Time

Our primary focus in expanding SynQLite to function with multiple sites involved conducting an experiment to measure the time it takes for a site to synchronize with other sites and to determine how much this duration varies depending on the number of sites involved in the distributed system.

The result of this experiment is presented in Figure 7.5. We investigated the synchronization process with different numbers of sites, ranging from 10 to 100, with increments of 10. Following the site creation as described in Figure 7.4, we inserted a single row to each site for this particular test. We limited the row count to one to ensure that the number of sites and the synchronization time were not affected by other factors. However, then we performed the PULL operation at each site and recorded the time it took to synchronize with other sites. Our findings indicate that the synchronization time increased steadily and predictably for site numbers between 10 to 60, with each site taking less than 5 seconds to synchronize. However, beyond 70 sites, the synchronization time increased significantly for each site and became less predictable, taking much longer to complete. Specifically, when the system consisted of 100 sites, synchronization varied between 15 to 25 seconds for each site in the system. Overall, our experiment demonstrates that synchronizing with a large number of sites can be challenging and can result in unpredictable and time-consuming synchronization times.

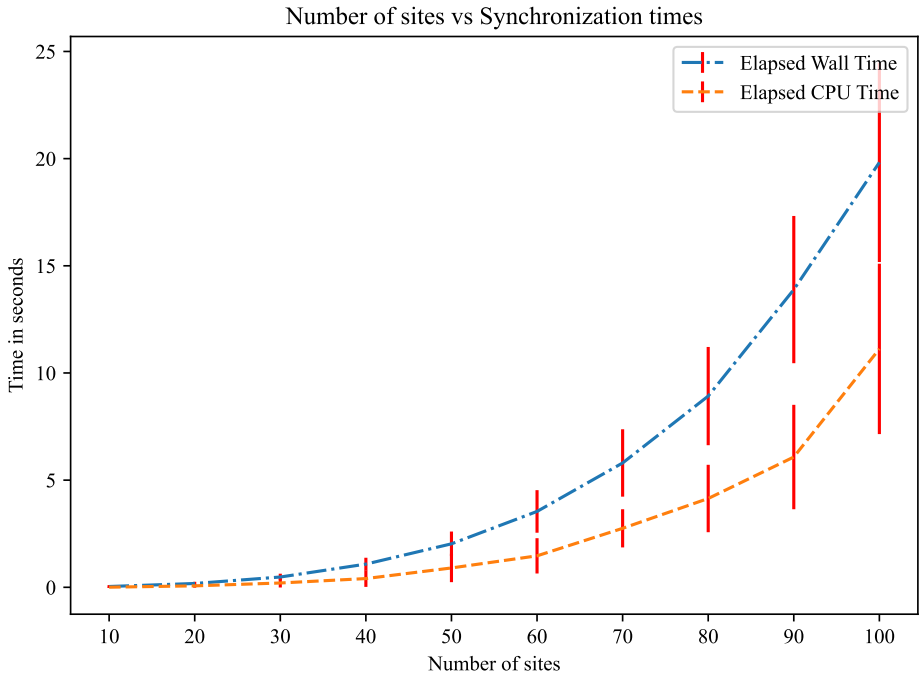


Figure 7.5: Plot of Number of sites vs Synchronization times

7.2.3 Very frequent synchronization vs long intervals of synchronization

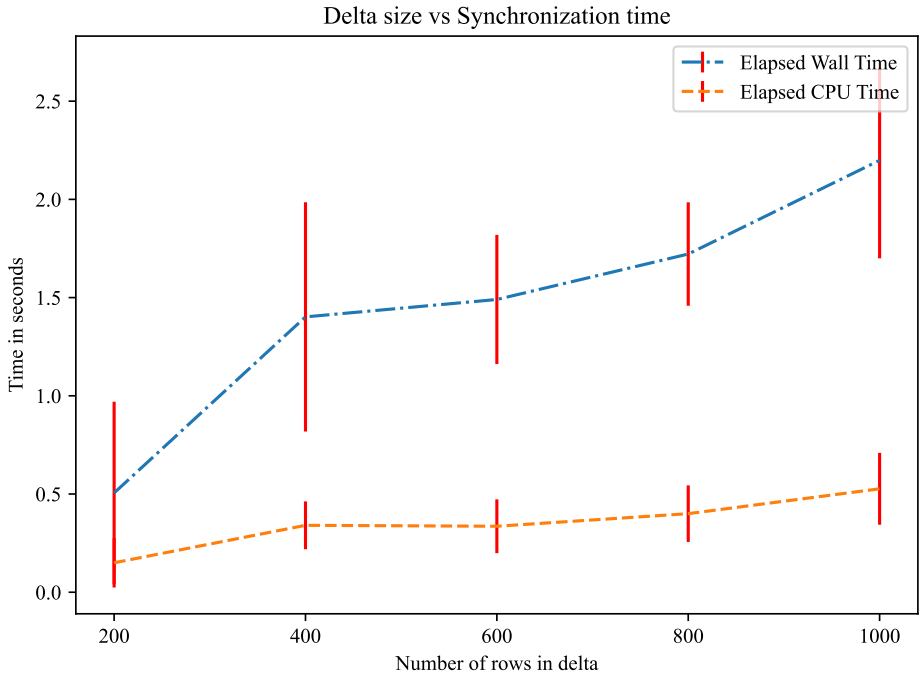
If synchronization operations are performed frequently, there will be a higher number of synchronization operations but a lower number of updated rows in the delta during each operation. Conversely, if synchronization operations are performed after a long interval, there will be a lower number of synchronization operations, but the delta will contain more updated rows. The frequency of synchronization operations and the number of delta rows being synchronized must have a great impact on the time required for a site to synchronize with other sites. To investigate this issue, we conducted an experiment where we inserted different numbers of rows into a database and performed synchronization operations on each site in the system, and recorded how long it takes for a site to synchronize with other sites. Figure 7.6 depicts the result of this experiment.

We used different numbers of sites in this experiment to see the effect of the experiment and the effect of the number of sites on this experiment. We inserted 200, 400, 600, 800, and 1000 rows in each site and performed the PULL operation at each site, recording the time it took to synchronize. Our observation indicates that the time taken to synchronize increases nearly linearly with the number of rows that exist in the delta regardless of the number of sites presented in the system. We found that when there are five sites in the system as portrayed in Figure 7.6b, then for approximately 1000 rows in the delta at each site, the synchronization time was around 10 to 14 seconds (Wall time), whereas, for 200 rows, it only took about 1 to 2 seconds. On the other hand, We see that when only two sites are present in the system, as shown in Figure 7.6a, then the synchronization is around 1.5 to 2.5 seconds for 1000 rows, and for 200 rows, it is notably insignificant.

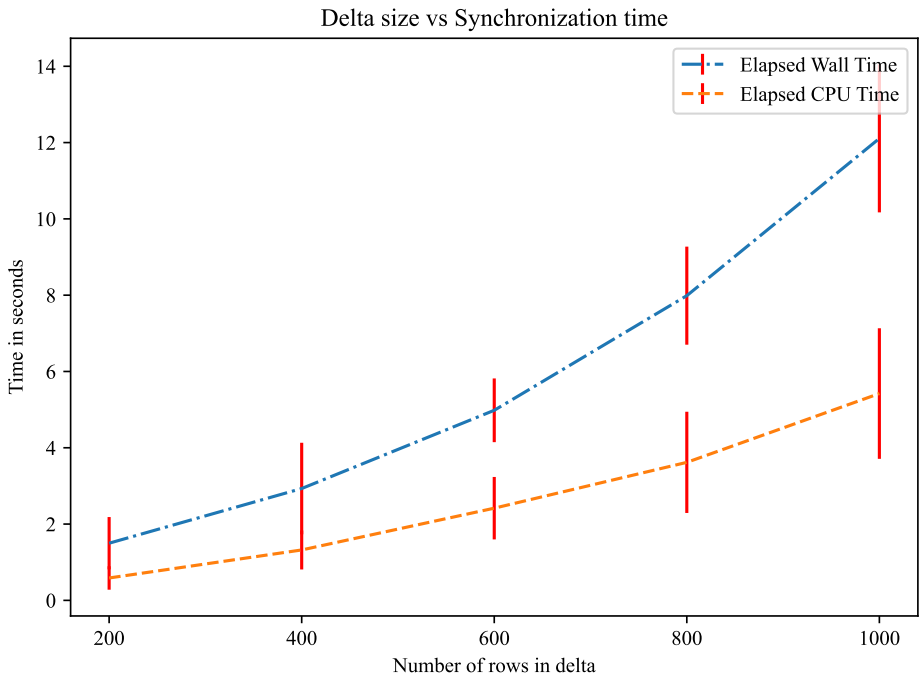
7.2.4 Delta Generation time vs Merging time

Synchronization time can be divided into two parts. One is the time to generate the delta at one site, and the other is to merge the delta at the other site. We also experimented to see which sub-part of the synchronization operation consumes the most time in the process. It will help future endeavors to focus particularly on this part to optimize its performance.

The graph shown in Figure 7.7a clearly indicates that the merging process is responsible for consuming most of the synchronization time. In this experiment, we have leveraged five sites where Site 2 generates the delta for Site 1, and then Site 1 merges the delta; similarly, Site 3 generates and Site 2 merges the deltas, and this process continues. We see that as the number of rows in

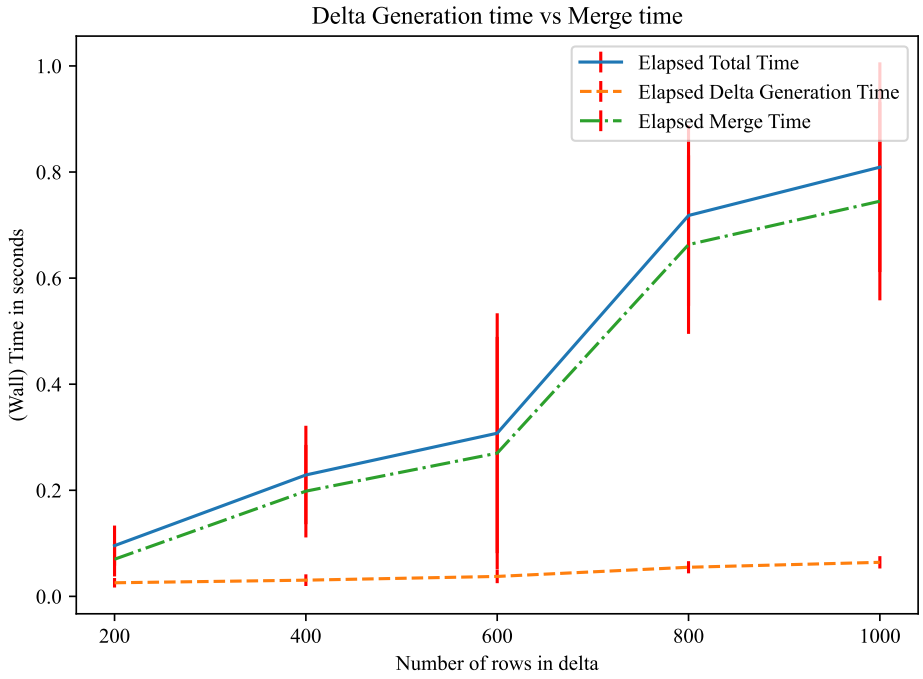


(a) Two sites are present in the system

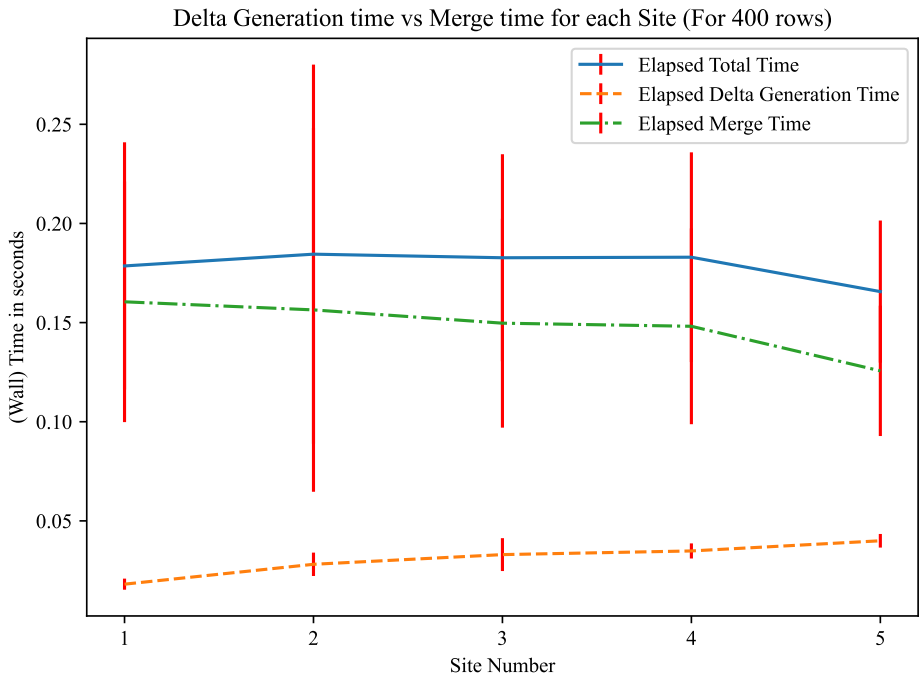


(b) Five sites are present in the system

Figure 7.6: Plot of Delta size vs Synchronization Time



(a) Plot of Delta Generation time vs Merge time



(b) Plot of Delta Generation time vs Merge time for each site (For 400 rows)

Figure 7.7: Plot of Delta Generation time vs Merge time

the delta increases, the merging time also increases significantly. In contrast, the time taken to generate the delta remains so insignificant, and though it is increasing, the rate of increase is remarkably slow. Consequently, it is evident that optimizing the merging process should be our primary focus in order to reduce the overall synchronization time.

We have also fixed the number of rows in deltas (400 rows), and then, for each site, we have plotted the delta generation and merge time along with the total time in Figure 7.7b. We see that the plot is aligned with the plot Figure 7.7a. Merging time is much larger than the delta generation time, and for 400 rows, each site takes approximately 0.20 seconds.

7.3 Resource Utilization

During our observation of SynQLite operations, we monitored the system's resource utilization, including CPU, memory, and disk, using the Task Manager. Our observation showed that the CPU utilization varied between 1.0 to 29.0% for most of the operations. In contrast, the memory utilization ranged from 100 MB to 2000 MB. In comparison, disk usage is not a significant concern.

To conclude, the performance experiment carried out as part of this thesis underscores the need to focus on enhancing the performance of SynQLite. While our main focus was on ensuring accuracy and correctness, we recognized the importance of performance but were limited in our ability to address this issue. Therefore, we recommend that future researchers allocate sufficient time and resources to improve the performance of SynQLite. By doing so, they can optimize its performance and make it more efficient for a broader range of applications.

8 Discussion

At this juncture, we will discuss how we have carried out our entire thesis work from the very beginning to the end. We will shed some light on the discussion of how well we have achieved our primary goal. Moreover, we will also discuss the potential improvements that can be further made on top of our implementation of SynQLite.

8.1 Overcoming Challenges of this thesis

We began our work by taking the already implemented version of SynQLite as our base and then tried to extend it to handle scenarios when multiple sites frequently connect and disconnect with the system. When we started, there were already thousands of lines of code, and we all know that it is really a difficult task to debug someone else code and find out the existing loophole. It took a substantial amount of time to understand the existing logic and to find out the bug in the logic. Side by side, we had to learn the technology and their implication that are being used in SynQLite as well as the fundamental technology for Local-First software, i.e., CRDT, CRR, etc. While learning these new technologies was not particularly difficult, it was initially challenging to correlate them with the existing SynQLite implementation. However, we overcame this challenge by breaking down the problem into smaller parts and gradually building our understanding of the system.

Though in research, we can not plan everything in advance, and we also had difficult times when we predicted to solve features within a timeline, but it took far more than that timeline. Sometimes, it took more time to understand a concept and also to understand the existing bug. Still, in the end, we are glad that with perseverance and a willingness to learn new technologies, we overcome all the obstacles and have been able to achieve our goal to make the SynQLite multi-site multi-synchronous.

8.2 Unreported Technical Tasks

To completely solve our issue, we had to learn and leverage a few other tools and technologies other than the properties of Local-First software, CRDT, and CRR, such as Docker, WSL, etc. We used Docker image to containerize and simulate a remote computer to which we have SSH access so that we can clone to and from it and then synchronize with the remote site. It took almost an entire month from our entire thesis period to become familiarized with it and to successfully create the remote site with the Docker container and access it through the local system or another Docker container. Though it snatches away a significant amount of time from our entire thesis work, to be honest, when it comes to thesis writing, it does not carry that much significance so that we can include the work in the writing separately. Not only technology-related things, but there are a few feature-related issues also on which we spent a substantial amount of time, but we did not find a way to include the work separately in the thesis writing. For instance, when we were simultaneously updating and deleting the same row from two different sites and then tried to synchronize them with each other, they were diverging from each other. It took a lot of time to find out the bug, but we did not find a way to include it separately in the thesis writing.

8.3 Learnings

The research process has been a transformative and valuable experience, as it has provided me with extensive knowledge, practical skills, and valuable insights. Throughout the process, I have gained a deeper understanding of new theories, technologies, and tools and learned how to apply them in practical settings. For instance, concerning theories and technologies, I learned Local-First software, how cloud application work, how CRDT can realize Local-First software, state-vector, and many more concepts of Distributed Systems. Concerning tools, I have learned Python, Pytest, SQL, Docker, WSL, etc. The addition of the meta_site_state (SS) table was an application of state vector in a practical

setting. The SynQLite implementation itself is a practical application of CRDT, CRR, and Local-First software for SQLite databases.

Moreover, this journey has enabled me to develop my research abilities and provided a substantial opportunity for personal and professional growth. I am immensely grateful to my thesis supervisor for his exceptional guidance and support throughout the research process. He helped me to focus on the most critical aspects of the research and taught me how to break down complex tasks into manageable steps, which enabled me to remain focused and productive throughout the process. For instance, breaking down the main goal of providing multi-site multi-synchronous support to existing SynQLite implementation into small achievable features and bugs that would materialize the main goal and then prioritize the features and bugs.

Through my research, I learned how to approach problem-solving and analysis from multiple perspectives, gaining exposure to different methods and techniques for resolving issues. My supervisor's mentorship was invaluable in navigating the uncertainties and risks inherent in research and enabled me to develop essential skills such as top-down and bottom-up approaches to problem-solving.

Going forward, I am confident that the knowledge, skills, and confidence I have developed through this experience will enable me to make meaningful contributions to my field and beyond.

8.4 Future Work

During the initial investigation of the SynQLite codebase and throughout the entire research journey, we identified a number of potential features and bugs that we could not incorporate into this thesis due to time and scope constraints. These features are listed in this section for the benefit of future efforts to work on SynQLite, providing insight into its current limitations and presenting possibilities for enhancement.

8.4.1 Handle the re-clone issue

It is crucial to understand that the complete SQLite database is saved in one file, making it possible to delete a cloned database unintentionally. If this occurs and an attempt is made to re-clone the mistakenly deleted database using SynQLite, an exception will be thrown. As a result, the database will be unable to transmit updates to and receive updates from other sites during the synchro-

nization process. This can cause a divergence in the database's state, which is a significant concern that must be resolved in future versions of SynQLite.

8.4.2 Add support to update the database schema

It is important to note that the current implementation of the SynQLite only permits the addition, deletion, and modification of rows within tables. However, it is common for users to introduce new tables, remove existing ones, or update table structures that are not currently supported. If a user performs any of these actions on the database, SynQLite will not update the corresponding CRR layer, resulting in an unstable database that cannot be further cloned and will not synchronize properly with other sites during the synchronization process. Therefore, it is essential for future versions of SynQLite to handle these events to ensure database stability and proper synchronization.

8.4.3 SSH path is not set properly in the meta_site table

This is an extremely critical issue as SynQLite does not accurately set the path of cloned databases in the meta_site table of other sites. After cloning a database from a remote site to a local site, SynQLite sets the original remote site's path properly in the cloned site's meta_site table since, during cloning, we specify the path from which we want to clone. After the database has been successfully cloned, SynQLite sets the cloned database path in the meta_site table of all other sites presented in the distributed system along with the original remote site. However, SynQLite mistakenly sets the local path of the cloned database to other sites' meta_site table instead of the required ssh path in the format "user@host:port:local_path". This results in communication issues between sites as they don't have the remote ssh path of each other. Since, during our implementation, we used to simulate the remotes in just a single PC, which means we used to perform all the operations in the same PC with different folders or paths, it worked properly. But this issue should be solved so that SynQLite actually works with remote sites. A potential solution may involve utilizing Python's socket library to calculate the username and host, then determining the path based on that. Future researchers should investigate ways to resolve this issue.

8.4.4 The initialization operation is not atomic

If an exception occurs during initialization, the CRR support will not be correctly augmented to the database. This makes it impossible to clone and synchronize the database to other sites, and there is no way to re-initialize it. As

a result, the database will become permanently unusable. To prevent this, it is crucial to roll back the initialization operation and display an error message explaining why the initialization failed when any exception occurs during the initialization process.

8.4.5 Handle the assumption of Synchronization issue

During the synchronization process, one site sends a delta to another site and then updates its own state. However, it is possible that the recipient site may not receive the delta due to various reasons such as a dropped update or network partition. In such cases, the first site assumes that the second site has received the update, while in reality, it has not. This issue can have serious consequences, as the first site may not resend the update during the next synchronization, assuming that the second site has already received it. Therefore, this issue requires careful consideration and should be addressed by future researchers.

8.4.6 Export and Merge delta separately

In order to provide users with greater flexibility, SynQLite should offer the ability to export a database's deltas as a separate file using a command. This delta file can be shared via email, USB, or other methods. Furthermore, SynQLite should have the functionality to merge this delta file with an existing database as a separate operation, providing users with greater control over their database management.

8.4.7 Synchronize with a particular site

Currently, synchronization operations, namely PULL and PUSH, cannot be executed for a specific site. Instead, when a PULL or PUSH operation is initiated on a site, it retrieves updates from or sends updates to all other sites within the system. It is recommended that a separate option be implemented in the future to allow for PULL from ALL or PUSH to ALL, as well as the ability to PULL from or PUSH to a specific site.

9

Conclusion

Our principal research challenge focuses on the insufficient level of support provided by the existing SynQLite implementation for state management in situations where multiple sites are connected and disconnected frequently to the system. More specifically, the existing implementation fails to guarantee consistency among replicas when there exist more than two replicas or sites of a database, thereby limiting its practical applicability in collaborative scenarios that require multi-site and multi-synchronous access.

The limitation poses a significant challenge as it impedes the scalability and effectiveness of the SynQLite system. Therefore, the primary goal of this research was to address this limitation of the existing SynQLite implementation by fine-tuning the existing implementation so that it provides multi-site multi-synchronous access to SQLite databases while ensuring consistency among replicas.

For this endeavor, we conducted an extensive study of the existing codebase and the properties of Local-First software, CRDTs, CRR, and other relevant factors. Our analysis exposed several flaws in the existing code base, which were causing problems when we attempted to add more than two sites to the system. Furthermore, we also identified specific features that, if integrated into the existing version of SynQLite, would enable it to function for multiple sites. After compiling a comprehensive list of these features and bugs, we commenced the development process.

A significant accomplishment of this thesis was the implementation of an adaptive delta-generation logic feature. The previous logic was only designed for systems with two sites, causing incorrect delta generation during synchronization for systems with more than two sites. This led to incorrect delta state merging, resulting in failure to converge. Also, existing issues like partitioning were not handled, leading SynQLite not to uphold the Local-First properties. We handle the partitioning issue. Additionally, making all sites eventually aware of each other, eliminating the fixed idea of the auto-incremented primary key, etc., mentioned in Chapter 6 also helped us to extend SynQLite with multi-site multi-synchronous support.

We designed various test cases using both bash and python scripts to experiment with our implementation to verify whether it converges the site states to a unified state when multiple sites are presented in the system as mentioned in Chapter 7. All tests and experiments were successful in demonstrating the convergence of site states, which proves the multi-site multi-synchronous capabilities of SynQLite. We believe that we have considered almost all the corner cases that can arise in a distributed system during our experiment. Still, it is true that in our limited human mind, some of the farthest corner cases can be missed. Though our main concern was not on enhancing the system's performance but instead on making it correct, we also did some performance tests. These tests show that future researchers should consider the performance enhancement of SynQLite.

Finally, We can delightedly announce that we have successfully accomplished the goal that we had set up during the inception of our thesis. Our implementation guarantees that even if multiple sites frequently connect and disconnect to the system, SynQLite will still ensure that they converge to the same state after performing the synchronization operation. We look forward to future research on SynQLite, building upon the resilience and robustness of our thesis implementation, and focusing on enhancing its performance.

Bibliography

[1] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 154–178, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.

[3] Weihai Yu and Claudia-Lavinia Ignat. Conflict-free replicated relations for multi-synchronous database management at edge. In *2020 IEEE International Conference on Smart Data Services (SMDS)*, pages 113–121. IEEE, 2020.

[4] Iver Toft Tomter and Weihai Yu. Augmenting sqlite for local-first software. In *European Conference on Advances in Databases and Information Systems*, pages 247–257. Springer, 2021.

[5] Armando Fox and Eric A Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178. IEEE, 1999.

[6] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. 1(OOPSLA), oct 2017.

[7] Werner Vogels. Eventually consistent: Building reliable distributed systems at a worldwide scale demands trade-offs? between consistency and availability. *Queue*, 6(6):14–19, 2008.

[8] Sqlite home page, 2023. URL `https://sqlite.org/index.html`.

[9] Gustav Heide Iversen. Continious synchronization of conflict-free replicated relations. Master's thesis, UiT Norges arktiske universitet, 2022.

[10] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5):172–182, 1995.

[11] partial order in nlab, 2023. URL `https://ncatlab.org/nlab/show/partial+order#definitions`.

[12] semilattice in nlab, 2023. URL `https://ncatlab.org/nlab/show/semilattice`.

[13] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[15] Hicham Marouani and Michel R Dagenais. Internal clock drift estimation in computer clusters. *Journal of Computer Systems, Networks, and Communications*, 2008, 2008.

[16] Universally unique identifier - wikipedia, 2023. URL `https://en.wikipedia.org/wiki/Universally_unique_identifier`.

[17] Ssh file transfer protocol - wikipedia, 2023. URL `https://en.wikipedia.org/wiki/SSH_File_Transfer_Protocol`.

[18] Welcome to paramiko! — paramiko documentation, 2023. URL `https://www.paramiko.org/`.

[19] Peter Denning, Douglas Comer, David Gries, Michael Mulder, Allen Tucker, Joe Turner, and Paul Young. Computing as a discipline. *Computer*, 22: 63–70, 03 1989. doi: 10.1109/2.19833.

[20] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

[21] Docker: Accelerated, containerized application development, 2022. URL `https://www.docker.com/`.

[22] Shell script - wikipedia, 2023. URL https://en.wikipedia.org/wiki/Shell_script.

[23] Git, 2023. URL https://git-scm.com/.

[24] Windows subsystem for linux - wikipedia, 2023. URL https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux.

[25] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[26] Vishal. Python get execution time of a program [5 ways] – pynative, 2022. URL https://pynative.com/python-get-execution-time-of-program/.