Faculty of Science and Technology
Department of Computer Science

User-Aware Conflict Resolution

Ragnar Helgaas

INF-3981 Master's Thesis in Computer Science - June 2023

Contents

List of Figures

0.1 Abstract

A large-scale system that prioritizes high availability over extensive synchronization must make a design trade-off and implement a weaker form of consistency. Conflict-free Replicated Data Types(CRDTs) can enable replicas in the system to communicate asynchronously and achieve strong eventual consistency. SynQLite aims to implement CRDTs on top of relational databases with its addition of Conflict-free Replicated Relation CRR. SynQLite enforces integrity constraints defined by an application, in a strictly automatic manner. However, enabling user-aware conflict resolution, where the user can make application specific altercation to conflict resolution, can remedy some of the limitations of the last-write-protocol used in SynQLite.

1 Introduction

This thesis introduces the proposed concept of user-aware conflict resolution, an extension to SynQLite's automatic conflict resolution that addresses some limitations.

The thesis discusses trade-offs between availability, partition tolerance, and consistency, as defined by the CAP theorem. It explores the challenges that occur when prioritizing availability and partition tolerance over strict consistency. CRDTs are introduced as a solution to achieve strong eventual consistency. The concept of CRDT is discussed, highlighting applications in distributed systems through the SynQLite system design.

Additionally, the thesis explores the enforcement of integrity constraints through conflict resolution mechanisms. It discusses the use of automatic conflict resolution and its limitations, including potential undesired outcomes. Finally, it introduces the proposed concept of user-aware conflict resolution, which allows users to participate in the resolution process and influence the outcome based on their preferences. Some limitation to this solutions and use cases are described.

2 Technical Background

2.1 The CAP Theorem

In computer science, the CAP theorem[2] is a popular theory that states that any distributed system can only ever fully achieve two of the following guarantees;

- **Consistency(C)**
- **Availability(A)**
- **Partition Tolerance(P)**

In an arbitrary system, high consistency ensures that when a user requests data from a site, the response will be accurate and up-to-date. However, if the system also guarantees partition tolerance, such as through distributed replicas of data, additional measures must be taken to maintain consistency across replicas. It becomes crucial that a user, whether accessing data from one site, denoted as S_i, or another replicated site, denoted as S_j, receives the same response. Synchronous communication between sites is a common solution to achieve this goal. Whenever an update occurs at S_i, S_j also needs to receive the same update before responding to any requests. However, if the system aims to be highly available, problems may arise. Prioritizing consistency and partition tolerance leads to longer response times due to synchronization. Therefore, it is essential for any distributed system to carefully consider trade-offs in its design.

2.2 Weaker levels of consistency

Eventual consistency

Stronger forms of consistency are often prioritized in systems that require strictly serialized operations, such as banking. Inconsistent ordering of operations may result in incorrect calculations, inaccurate account balances, or violations of business rules. Such variations can damage the reliability and trustworthiness of the system, potentially compromising the integrity of the system's design.

However, for systems where fast access is deemed crucial, a weaker form of consistency is required to be implemented. A system designed for an e-commerce web application, such as Amazon's Dynamo [1], is an example of the prioritization of fast user access to enable instant updates without significant synchronization delays. In Dynamo, servers are strategically replicated in close proximity to their respective users, ensuring minimal response times. This trade-off is made based on the fact that ensuring that the website is available for customers to make purchases is an important objective for Amazon. While data consistency is still important, the impact of a slow website or a website prone to synchronization delays outweighs the consequences of temporary inconsistencies. This approach prioritizes availability and partition tolerance, at the cost of adopting a weaker form of consistency. The weaker form of consistency used in Dynamo is known as eventual consistency, which guarantees that updates will eventually propagate to all replicas, in a lazy fashion, resulting in a convergence toward a consistent state. this effectively means that sites in a distributed system will eventually be consistent with each other if further updates were to halt.

Strong eventual consistency

Strong eventual consistency is another form of implementing weaker consistency in a distributed system. This alternative adds the guarantee that sites that have received the same updates, will be in the same state. This allows systems that require a high level of availability to still offer a significant level of consistency, although it does not guarantee that operations are performed in the same order as the sequential consistency mentioned earlier. This added focus on consistency makes it particularly useful for distributed database applications that offer a cloud-based solution. With added guarantees comes added complexity to a system that chooses this approach. A major challenge is how updates are communicated to replicas and as well as to be ensured maintain causal relationships.

Mixed levels of consistency

A more adaptive approach is to utilize both a strong and weak consistency model. For example, implementing support for altering consistency requirements based on what the overlaying user application needs. A successful implementation can Potentially yield performance gains in areas that allow for weaker consistency. This form of consistency is difficult to implement as it requires a set protocol for switching between consistency models to prevent the balancing between the two to result in a performance loss. "Making geo-replicated systems fast as possible, consistent when necessary"[3] proposes a "RedBlue" consistency, where the goal is to reduce the complexity of designing such a system by coloring operations that require stronger consistency, red, and coloring operation that allows for weaker consistency, blue.

Causal consistency

In contrast to the sequential consistency briefly mentioned in our banking example is causal consistency[4]. In a system that is causally consistent, operations are observed in a consistent order across replicas rather than that the same operations happens in the same order. This guarantee works well with weaker forms of consistency, as not every operation has to be executed in the same order, sequentially, across replicas.

2.3 Last-write-wins

In distributed systems operating across network partitions, conflicts can occur due to replicas performing conflicting operations concurrently. To maintain eventual consistency, it becomes necessary for the system to establish an agreement on which operations to retain and which to discard when the replicas eventually synchronize. In such scenarios, the last-write-wins (LWW) strategy is used as a reliable approach for conflict resolution, offering consistent and deterministic outcomes. The principle of the LWW strategy is relatively straightforward, the latest update is considered as the winner in any conflict resolution. To determine the "latest" update, timestamps or version numbers are commonly used. Timestamps provide a chronological order to updates, allowing replicas to establish the relative recency of each operation. Alternatively, version numbers can be assigned to updates, enabling replicas to identify the update with the highest version as the winner. Figure 2.1 shows two replicated sites that both performs an two concurrent operations on the value A. On synchronization, the most recent timestamp t_2 is kept while the operation with timestamp t_1 is discarded.

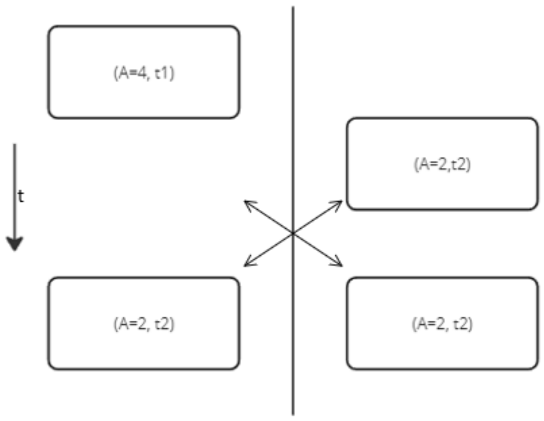

Figure 2.1: Last-write-wins conflict resolution

A limitation of the LWW protocol is potential loss of data. To address this, it is common to combine it with additional techniques such as conflict detection, where the system identifies conflicts and flags them for resolution by applications or users. Advanced conflict resolution strategies, such as merging conflicting updates or application-specific logic, can be used to handle conflicts.

2.4 CRDT

Conflict-free Replicated Data Types(CRDTs) [6] add metadata to replicated data in the form of tuples. The additional metadata can be seen as causal context that helps maintain causal consistency, often causal length or timestamps. This abstraction of data is designed to ease the implementation of strong eventual consistency. CRDTs inherently convey the principles of the last-write-wins strategy through their construction. The metadata reflects the ordering and causal relationships among updates, enabling replicas to determine the most recent update and resolve conflicts accordingly.

In the case of CRDT, causal history is acting as a mechanism for version control. It maintains a historical record of states and tracks the causal relationships between updates. Notably, lookup queries do not impact the causal history and are excluded from the count. Causal history represents a sequence of states, allowing replicas to maintain a comprehensive understanding of the history of the data. During the synchronization process, replicas merge their respective causal histories. This merging operation has the effect of merging the updates themselves, resulting in replicas reaching the same state. By tracking the causality of updates, replicas can independently perform necessary updates while monitoring the potential conflicts that may occur. In a large-scale system, where updates are eventually propagated to all replicas, the causal histories also converge and merge over time. As updates and their associated causal histories are spread across replicas, the convergence process ensures that all replicas eventually reach a consistent state.

This convergence of causal histories and the resulting convergence of data states enables the achievement of strong eventual consistency in CRDT-based systems. By tracking causality and merging updates during synchronization, CRDTs enable replicas to independently handle concurrent updates and eventually converge towards a consistent state. Later in Section 3.2, it is described how SynQLite[7] uses causal length as a form of causal history to keep track of the causality between insertion and deletion operations.

3 The SynQLite System Design

In pursuit of its objective to ensure strong eventual consistency while maintaining high availability and partition tolerance, SynQLite utilizes CRDTs. CRDTs play a crucial role in facilitating automatic conflict resolution during merging of updates across sites in a multi-synchronous environment where asynchronous communication is vital. SynQLite extends the conventional (key, attribute) relational database system by incorporating a CRR(conflict-free relations) layer implemented by CRDTs, introducing causal context to enhance conflict resolution capabilities.

While the utilization of CRDTs has been successfully demonstrated in small-scale systems by other solutions [5], the challenge lies in extending support to integrity constraints in a multi-synchronous system that employs relational databases. Ensuring data integrity and consistency across distributed replicas in a multi-synchronous context presents unique complexities that need to be addressed to fulfill the requirements of a robust and reliable system.

By adopting CRDTs within its architecture, SynQLite aims to overcome these challenges and provide comprehensive support for integrity constraints in a multi-synchronous setting. The integration of CRDTs with relational databases enables SynQLite to achieve strong eventual consistency while enforcing the integrity of the data across distributed sites, even in the face of concurrent updates and asynchronous communication.

3.1 Two layered design

SynQLite extends on SQLite and operates on a relational database that consists of two layers;

- **Application Relation(AR) layer**
- **Conflict-free Replicated Relation(CRR)**

The AR layer of SynQLite serves as an intermediary between the application and the underlying system, providing an API that closely resembles the well-established SQLite framework. Within this layer, applications have the ability to define their desired integrity constraints based on their specific requirements and preferences.

When performing a lookup or query operation on a specific entry, the entire process can be executed exclusively within the AR layer, specifically on the Application Relation denoted as R. As these operations do not impact the overall state of the site, they can be efficiently handled within the AR layer itself. The AR-layer requests operate on the familiar key, attribute relation structure represented as $R(K, A)$, allowing for seamless interaction with the stored data.

By shrouding the interaction between the application and the underlying system, the AR layer in SynQLite streamlines the integration process and offers a familiar interface through which applications can define integrity constraints. Such constraints are further discussed in 4.This layer also provides a high performing and isolated environment for executing lookup and query operations, with minimal impact on the overall system state.

CRR

While lookup operations are relatively straightforward, performing update operations introduces complexities related to consistency. In SynQLite, these challenges are addressed by using CRDTs (Conflict-Free Replicated Data Types) [6]. By extending the standard application relations, SynQLite incorporates conflict-free replicated relations (CRR) [7], enabled by CRDTs.

When an update request is received, SynQLite translates it into an operation on an augmented version of R in the CRR layer, denoted as $\tilde{R}$. This augmentation involves the addition of a timestamp τ and a causal length L" to the CRR layer. Both the relation instances of the AR layer, denoted as r_i, and the CRR layer, denoted as $\tilde{r}_i$, maintain a history of updates. Through the merging process, two sites can effectively combine their updates and eventually converge into a consistent state.

Consistency between two sites is achieved when their relation instances are observed as equivalent. This signifies that both sites have the same state after synchronizing. By incorporating the CRR layer, SynQLite ensures that updates from different sites are properly merged, leading to convergence and consistent states across replicas. The addition of the CRR layer remains transparent to the client application. The client only needs to be aware of how the AR layer handles read and update operations. This design simplifies the integration process and hides the underlying complexities of maintaining consistency in distributed systems from the clients.

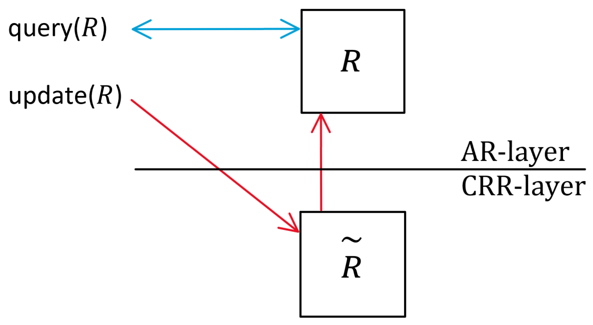

Figure 3.1: Flow of operations in two layer design

Figure 3.1 illustrates the two-layer design, where the AR layer accepts queries, while updates are redirected to the CRR layer before they can execute at the AR layer. This illustration is based on fig.1 from [7].

3.2 Updates

As mentioned in section3.1, an application sends update requests to the AR layer. Before carrying out the update on the local instance of Application Relation R, the update is first translated to an update on the CRR $\tilde{R}$. As an example, an update $u(r_i, K, A)$ on the existing entry received from the application. The update is first augmented, as discussed in the previous section, into $\tilde{u}(\tilde{r}_i, K, A, \tau, L)$ before being carried out. Where causal length differentiates between a row insertion and row deletion. Odd valued L means there has been an insertion, and even valued L means the previous operation was a row deletion. Update requests are handled differently if it's an insert, update or delete.

Insert

A local row insertion at site S_i is received as $insert(r_i, (K, A))$ and augmented at the CRR layer into $insert(\tilde{r_i}, (K, A))$. If the key K is a new entry, the following CRR operation is executed;

$insert(\tilde{r_i}(K, A, \tau_{currentTime}, L = 1))$

On the other hand, if K corresponds to an already existing entry, noted as $\tilde{r}(K)$, the following update is executed instead;

$update(\tilde{r_i}(K, A, \tau_{currentTime}, L(\tilde{r}(K)) + 1)$

Since there is issued an insert operation on an already existing entry, the entry must have been previously deleted with an even causal length $L \mod 2 == 0$. An increment of the casual length adds the insert operation to the causal history.

Delete

A local delete request is received as $delete(r, K)$ at the AR layer and executed at the CRR layer if $\tilde{r}(K)$;

$update(\tilde{r}(K, -, L(\tilde{r}(K)) + 1))$

Where "$-$" denotes the attribute A at $r(K, A)$ as deleted. Similarly to row insertion, the causal length is incremented by one to add the row deletion to the causal history. If $r(K)$ does not exist, the operation is aborted.

Update

An attribute update request is received as $update(r, K, A)$ and executed if $r(K)$ exists as such;

$update(\tilde{r}(K, A, \tau_{currentTime}, L(\tilde{r}(K)))$

If $r(K)$ does not correspond to an existing entry, the operation is aborted.

An attribute update only executes on an existing entry from a previous row insertion, therefore the causal length is not incremented as it does not affect whether the last operation was a row insertion or a row deletion.

3.3 Synchronization

The two-layered design of SynQLite assumes a distributed system where sites do not have shared memory. Updates that are executed locally at each site are kept in a local queue. State updates are inflationary, meaning that updates can only increase state values, in this case timestamp and causal length, such that a agreement can be formed on merge and support the notion of strong eventual consistency.

Synchronization at the individual replicas happens trough a pull request to all other sites in the system. The local queue of each site is received and the site can merge updates. At the merge, a site S_i considers incoming updates and how they will affect its local instance of $\tilde{r_i}$, by running it through the augmented CRR layer before updating the application layer.

If an incoming update from S_j has the state $\tilde{r}(K, A_j, \tau_j, 3)$ and S_i locally has $\tilde{r}(K, A_i, \tau_i, 5)$, the states are considered as an equivalent row insertion of A, assuming $A_j == A_i$.

Any updates that violate integrity constraints are rolled back with an undo operation. Integrity constraints and enforcement of these are described further in chapter 4.

In a system where the network is unreliable, as is the case for cloud-edge environments, strong eventual consistency is still supported. Sites store their local queue of updates on solid memory such that if the site goes offline or crashes, updates are not lost from the rest of the distributed system. Rather, update queues are propagated once the site is back online, allowing the site to conveniently execute updates locally while being blissfully unaware of its connection to other replicas in the system.

4 Integrity Constraints and Conflict Resolution

In a multi-synchronous system, sites are able to update without immediately broadcasting their changes to other sites. This allows for not only fast access to data, but also adds the ability to perform asynchronous updates. Asynchronous updates are an advantage for clients with unreliable connections, but this simultaneously adds complexity to keep the system consistent. To be able to guarantee strong consistency under such conditions, predefined rules need to be enforced to make sure every site will eventually converge into the same state without communication, as is the case for cloud-edge environments. Integrity constraints are rules that set the standard on how each site should resolve conflicts at merging updates with other sites.

Integrity constraints in a system can be defined by an application, according to what type of system the client aims to operate. Set integrity constraints can be "violated" locally without a problem, as integrity constraints are only enforced once a site merges with a different site's updates. This resolution is resolved using the causal context added through the previously discussed history of augmenting update request $r_i(K, A)$ into $\tilde{r}_i(K, A, \tau, L)$, where timestamp $t\tau$ and causal length L is added to the relation. Resolution happens at the CRR layer and offending updates are rolled back. When the incoming updates no longer consist of any operations that lead to integrity violations, the updates are translated back to the AR layer for a refresh of the local state.

SynQLite aims to support the automatic resolution and enforcement of three different types of integrity constraints;

- **Uniqueness constraints**: Constraints on a system having more than one unique entry after a merge, defined by application based on id or value.
- **Referential constraints**: Constraints on a system updating a referenced entry without updating its reference.
- **Numeric constraints**: Numeric boundaries to prevent a value or a counter to exceed a set limit.

4.1 Uniqueness Constraints

Uniqueness constraints set a standard on how a system allows for non-unique entries in a system. As an example on such a standard; lookup on $r(K)$ should only respond with a single entry. To solidify this as a consistent behavior throughout the system, a uniqueness constraint that only allows for unique keys needs to be set. In a system with an unreliable network, two sites may execute a local row insertion consisting of the same key at both sites without having to resolve a uniqueness violation. Another example of a uniqueness constraint could be that a user cannot have duplicate values, like home addresses, associated with its key.

Resolving uniqueness constraints

The resolution of a uniqueness constraint is handled by comparing the causal context among conflicting updates. The one update with the lowest timestamp will be chosen as the winner, the lowest timestamp meaning the timestamp that corresponds to the earliest time. The other violating operations are undone through an undo operation, described in section4.4. In the rare case of concurrent updates, $\tau_i == \tau_j$, it can be decided based on causal length, L, to prioritize an outcome where most updates are kept, and the least updates need to be undone

Figure 4.1 illustrate a situation where site S_i and S_j both perform a row insertion of an entry with key K, with K representing a non-unique entry. When the two sites merge updates, a uniqueness violation is detected. Following the set integrity constraints, both sites choose the update with the lowest timestamp, where $\tau_0 < \tau_1$. S_j performs an undo operation and accepts the incoming insert update from S_i, resulting in a consistent state between the two sites.

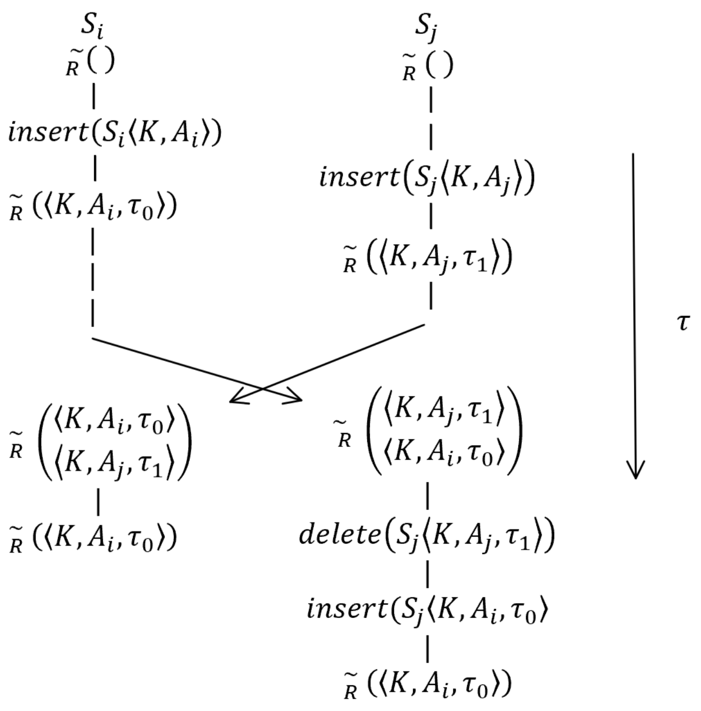
S_i
$\tilde{R}()$
$insert(S_i\langle K, A_i\rangle)$
$\tilde{R}(\langle K, A_i, \tau_0\rangle)$
S_j
$\tilde{R}()$
$insert(S_j\langle K, A_j\rangle)$
$\tilde{R}(\langle K, A_j, \tau_1\rangle)$
τ
$\tilde{R}\begin{pmatrix}\langle K, A_i, \tau_0\rangle \\ \langle K, A_j, \tau_1\rangle\end{pmatrix}$
$\tilde{R}(\langle K, A_i, \tau_0\rangle)$
$\tilde{R}\begin{pmatrix}\langle K, A_j, \tau_1\rangle \\ \langle K, A_i, \tau_0\rangle\end{pmatrix}$
$delete(S_j\langle K, A_j, \tau_1\rangle)$
$insert(S_j\langle K, A_i, \tau_0\rangle$
$\tilde{R}(\langle K, A_i, \tau_0\rangle)$

Figure 4.1: Uniqueness violation

4.2 Referential Constraints

Another type of constraint that is supported by SynQLite, is referential constraints. A referential constraint is set to prevent a referenced entry from being updated, without updating a reference to that entry.

Resolving Referential constraints

Resolving a referential integrity violation is done by performing an undo on the row insertion that added the invalidated reference. One could consider undoing the deletion of the referenced entry if the timestamp allows it.

Figure 4.2 illustrate two sites, S_i and S_j, with tables T_1 and T_2. S_i performs an insert of foreign key reference $r(K)$ at $T2$, while S_j performs a row deletion of K. When S_i merges update with S_j, the reference is referring to an entity that has been deleted. To resolve the conflict, the insertion of reference $R(K)$ is undone. This results in a consistent state between the two sites after the merge. The states are equivalent as the causal length at S_i is even, which from the perspective of the application, is the same as a removed entry.

$$\begin{tikzcd}
S_i & & S_j \\
\tilde{R}\begin{pmatrix} T_1\{\langle K, A, \tau_0, 1\rangle\} \\ T_2\{\,\} \end{pmatrix} \arrow[d, dash] & & \tilde{R}\begin{pmatrix} T_1\{\langle K, A, \tau_0, 1\rangle\} \\ T_2\{\,\} \end{pmatrix} \arrow[d, dash] \\
insert\big(T_2\langle r\langle K\rangle\rangle\big) \arrow[d, dash] & & delete(T_1\langle K\rangle) \arrow[d, dash] \\
\tilde{R}\begin{pmatrix} T_1\{\langle K, A, \tau_0, 1\rangle\} \\ T_2\{\langle r\langle K\rangle, \tau_1, 1\rangle\,\} \end{pmatrix} \arrow[dr] & & \tilde{R}\begin{pmatrix} T_1\{\langle K, -, \tau_2, 2\rangle\} \\ T_2\{\,\} \end{pmatrix} \arrow[dl] \\
& \tilde{R}\begin{pmatrix} T_1\{\langle K, A, \tau_0, 1\rangle\} & T_1\{\langle K, -, \tau_2, 2\rangle\} \\ T_2\{\langle r\langle K\rangle, \tau_1, 1\rangle\,\}\text{'} & T_2\{\,\} \end{pmatrix} \arrow[dl, dash] \arrow[dr, dash] & \\
delete(T_1\langle K\rangle) \arrow[d, dash] & & \tilde{R}\begin{pmatrix} T_1\{\langle K, -, \tau_2, 2\rangle\} \\ T_2\{\,\} \end{pmatrix} \\
delete\big(T_2\langle r\langle K\rangle\rangle\big) \arrow[d, dash] & & \\
\tilde{R}\begin{pmatrix} T_1\{\langle K, -, \tau_2, 2\rangle\} \\ T_2\{\langle r\langle K\rangle, \tau_3, 2\rangle\,\} \end{pmatrix} & &
\end{tikzcd}$$

Figure 4.2: Referential constraint violation

4.3 Numeric Constraints

Numeric constraints are set boundaries of numeric values. This boundary could be a lower or upper boundary.

A violation can occur when different sites concurrently increment a value or a counter through an update operation. Locally there is no sign of any violation, but on merge, the increment operations could collectively cause a violation of an application's set numeric constraint.

Resolving Numeric Violations

The process of resolving a numeric violation is similar to the process of resolving a uniqueness violation. Out of the conflicting updates, the update with the lowest timestamp is chosen as the winner. All other conflicting updates are undone. An important difference between resolving a numeric violation compared to a uniqueness violation is that sites are more likely to perform many value increments between each synchronization, for example, increment operations on counters. This adds the possibility of having a long history of updates that, in turn, have to be undone on a merge. This could be costly.

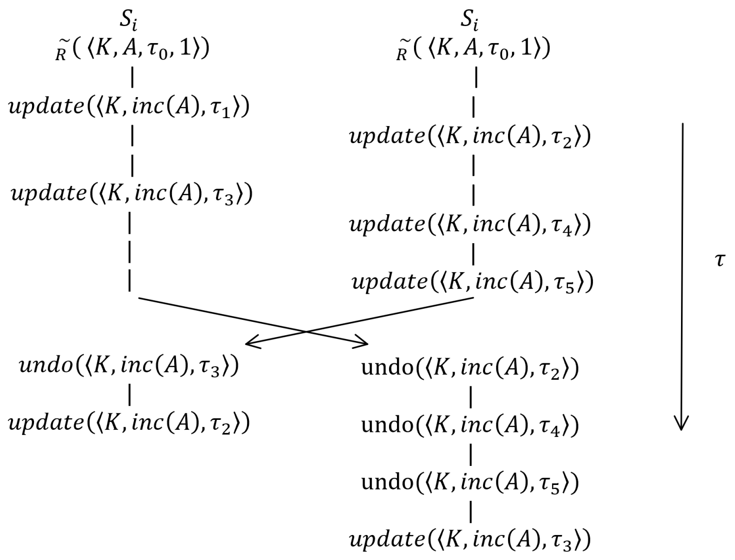
S_i
$\tilde{R}(\langle K, A, \tau_0, 1\rangle)$
$update(\langle K, inc(A), \tau_1\rangle)$
$update(\langle K, inc(A), \tau_3\rangle)$
S_i
$\tilde{R}(\langle K, A, \tau_0, 1\rangle)$
$update(\langle K, inc(A), \tau_2\rangle)$
$update(\langle K, inc(A), \tau_4\rangle)$
$update(\langle K, inc(A), \tau_5\rangle)$
τ
$undo(\langle K, inc(A), \tau_3\rangle)$
$update(\langle K, inc(A), \tau_2\rangle)$
undo$(\langle K, inc(A), \tau_2\rangle)$
undo$(\langle K, inc(A), \tau_4\rangle)$
undo$(\langle K, inc(A), \tau_5\rangle)$
$update(\langle K, inc(A), \tau_3\rangle)$

Figure 4.3: Numeric constraint violation

Figure 4.3 illustrates a situation where site S_i and S_j both have the inserted entry with key K and attribute A. The system has a numeric constraint for this attribute to not exceed three more increments. At each site, increments are performed to A. S_i performs two updates while S_j performs three. The merge of updates between S_i and S_j results in a numeric violation. Out of all updates, the ones that have the lowest timestamp are chosen as winners. The rest of the updates are undone until the numeric violation is no longer present. This results in a consistent state between S_i and S_j.

If the example system were to consist of additional sites, there could be a chain of many undo operations between merges. It is not easy to estimate how this will function in practice as there is not yet implemented support for the resolution of numeric violations.

4.4 Undo Operation

In the previous sections, the first part of resolving an integrity constraint violation was described, which is to pick a winner out of the offending updates. After picking a winner, the rest of the offending updates are effectively undone with an undo operation. How to perform an undo depends on which type of integrity constraint is enforced and what operation is to be undone. The goal of the undo operation is to return to a consistent state and not just revert the update.

Undo Uniqueness Violations

Update operations that violate uniqueness integrity constraints include row insertion and attribute updates. These operations are effectively undone by a row deletion or an attribute update. A row deletion resolves the issue illustrated in figure 4.1, where out of offending insert operations, the lowest timestamp wins. The other row insertion is undone by performing a row deletion on that entry. This effectively rolls back the incoming updates from site S_j to a state where it would be globally consistent, as S_j would not accept the request for a row insertion if it would have been aware that the insertion was already performed at site S_i.

Undo referential Violations

Undo of violations of referential constraints are mainly handled by a row deletion. As the example in section 4.2 illustrates, the row insertion of a conflicting reference is to be undone. Row insertion is undone by a row deletion. There could be situations where an undo of row deletion at S_i would be preferable, assuming a favored timestamp or causal length on the side of site S_j. However, the row deletion at S_i could be a result of a previous resolution of a uniqueness violation. This cascades into an endless cycle of undoing a row deletion that reinstates a uniqueness violation. This is not likely to happen too often and can be avoided by taking into account the history of integrity constraint enforcement. This workaround is more relevant to implement for the user-aware conflict resolution discussed in chapter**??**. The reason is that it adds unnecessary complexity to the automatic conflict resolution, which prioritizes supporting a system that eventually converges into a globally consistent state.

Undo numeric violations

Update operations that violate numeric constraints are either attribute updates or counter increments. Attribute updates are undone by setting the attribute to the older value, rolling back the effect of the update. Violating counter increments are undone by creating and performing an inverse of each update. This will in theory roll back the effect of counter increments. However, merges between multiple sites can create a difficult situation where the inverse of increment updates are also performed concurrently at different sites.

4.5 Overlapping integrity constraints

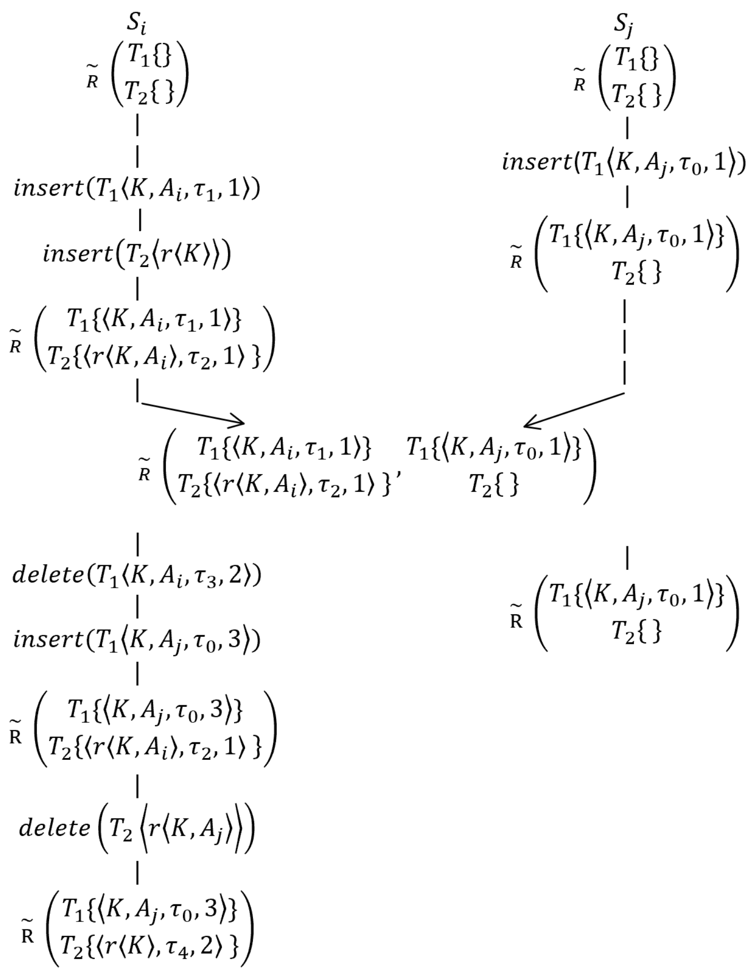
S_i
S_j
$\tilde{R}\begin{pmatrix}T_1\{\}\\T_2\{\}\end{pmatrix}$
$\tilde{R}\begin{pmatrix}T_1\{\}\\T_2\{\}\end{pmatrix}$
$insert(T_1\langle K, A_i, \tau_1, 1\rangle)$
$insert(T_1\langle K, A_j, \tau_0, 1\rangle)$
$insert(T_2\langle r\langle K\rangle\rangle)$
$\tilde{R}\begin{pmatrix}T_1\{\langle K, A_j, \tau_0, 1\rangle\}\\T_2\{\}\end{pmatrix}$
$\tilde{R}\begin{pmatrix}T_1\{\langle K, A_i, \tau_1, 1\rangle\}\\T_2\{\langle r\langle K, A_i\rangle, \tau_2, 1\rangle\}\end{pmatrix}$
$\tilde{R}\begin{pmatrix}T_1\{\langle K, A_i, \tau_1, 1\rangle\} & T_1\{\langle K, A_j, \tau_0, 1\rangle\}\\T_2\{\langle r\langle K, A_i\rangle, \tau_2, 1\rangle\}' & T_2\{\}\end{pmatrix}$
$delete(T_1\langle K, A_i, \tau_3, 2\rangle)$
$\tilde{R}\begin{pmatrix}T_1\{\langle K, A_j, \tau_0, 1\rangle\}\\T_2\{\}\end{pmatrix}$
$insert(T_1\langle K, A_j, \tau_0, 3\rangle)$
$\tilde{R}\begin{pmatrix}T_1\{\langle K, A_j, \tau_0, 3\rangle\}\\T_2\{\langle r\langle K, A_i\rangle, \tau_2, 1\rangle\}\end{pmatrix}$
$delete\left(T_2\left\langle r\langle K, A_j\rangle\right\rangle\right)$
$\tilde{R}\begin{pmatrix}T_1\{\langle K, A_j, \tau_0, 3\rangle\}\\T_2\{\langle r\langle K\rangle, \tau_4, 2\rangle\}\end{pmatrix}$

Figure 4.4: Overlapping integrity constraints

Consider site S_i, S_j with T_1,T_2 as illustrated by fig 4.4. S_i performs an insert of A_i with key K at T_1 and adds a foreign key reference at T_2. Additionally, S_j performs insert A_j with key k at T_1. locally there will be no violation of integrity. At merge it is discovered that the insertion of K in both sites violates uniqueness constraints set by the application. In this example, A_j is timestamped with a lower timestamp than A_i. Following the automatic constraint resolution, the insertion of A_i will subsequently be undone. As a consequence, $r(K)$ is reduced to a foreign key to a value that no longer exists. Therefore, it creates a situation where both a uniqueness constraint and referential that needs to be handled.

Different approaches are available and should be considered. Undo entry A_i and reference $r(K)$, undo A_j ,or, delete all. If the only thing considered is that the system enforces integrity constraint, option a is sufficient, as the system will eventually converge into a consistent state.

5 User-aware conflict resolution

So far, we have discussed the concept of automatic conflict resolution in achieving strong eventual consistency. However, it is important to acknowledge that this approach has limitations and can sometimes lead to undesired outcomes. As previously mentioned, the last-write-wins protocol, while effective in achieving strong eventual consistency, can potentially result in data loss. Certain applications may prioritize different outcomes that differ from the automatic resolution provided.

To address this issue, a user-aware conflict resolution is proposed, allowing users to manually adjust the priority of conflict resolution. By introducing this capability into SynQLite, users can integrate the system with applications that require alternative resolution outcomes to the default automatic approach. However, incorporating user-aware conflict resolution introduces complexities to the overall scheme. Notably, a history of constraint enforcement needs to be stored and consistently maintained across distributed sites. Additionally, support for queries on the history of conflict resolution becomes necessary.

5.1 Query History of Operations

In order to enable user-aware conflict resolution, previous conflicts needs to be queried and brought to reconsideration. SynQLite's use of CRDTs implements a stable history of operations across all replicas through the addition of the augmented history table in the CRR layer.

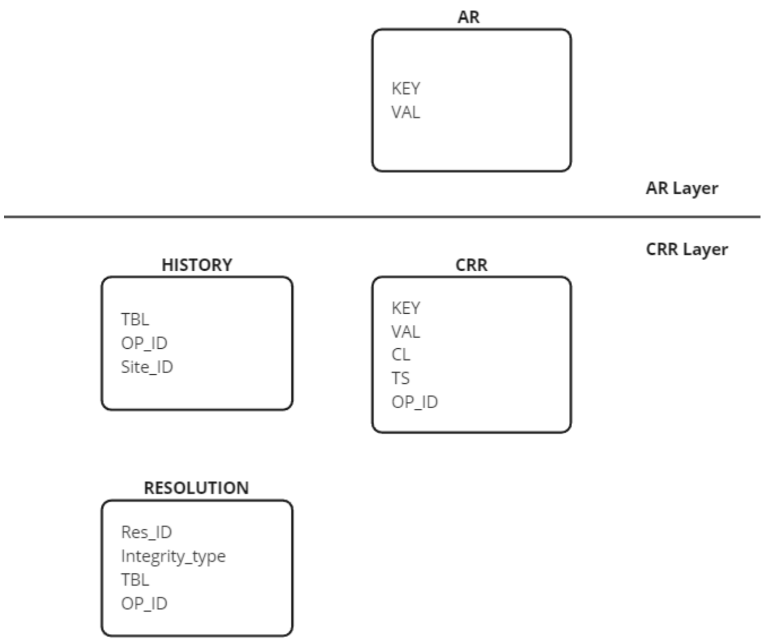

Figure 5.1: Relations between tables

To query for previous operations, the operation id(OP-ID) can be traced through to find if the site has previous any interaction with the operation. Figure 5.1 provides an overview of the relevant relations involved in conflict resolution, along with their corresponding keys. Expanding on the earlier discussed 3.1, in the AR(Application Relation)-layer resides the AR table with the common relation(Key, Value). In the CRR(Conflict-Free Replicated Relations) layer resides the CRR table which has additional metadata values. Any CRR relation involved in an integrity violation has a corresponding resolution relation inserted into the resolution(RES) table. A new relation inserted into the resolution table will include what type of integrity is being enforced and priority from conflict resolution.

There is a history table where a new relation is inserted for every operation executed on the current site. This includes resolution operations, like an undo operation. Inserted relations notably contain at which table the operation is executed(TBL), OP-ID and Site-ID.

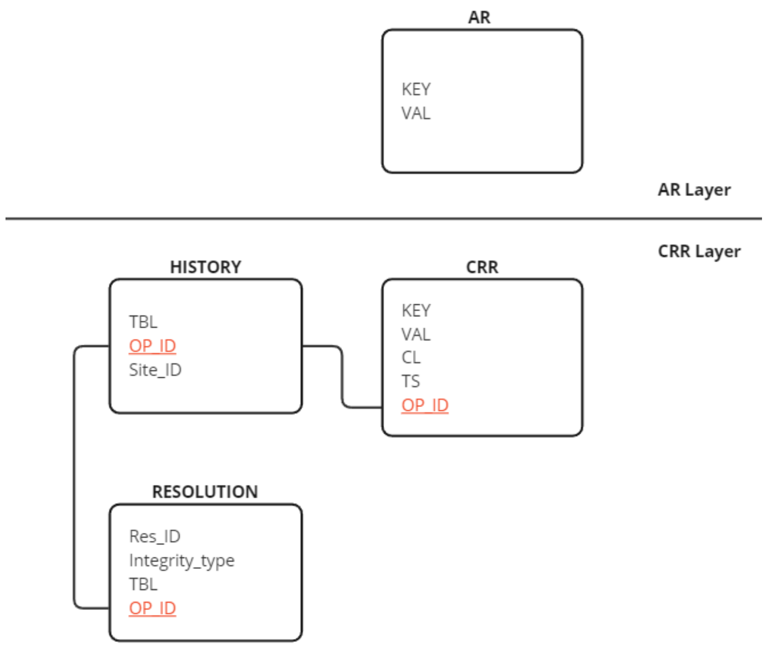

Figure 5.2: Querying OP-ID to retrieve resolution info

To enable this process, the resolution information must be included when sites synchronize. This ensures that the necessary resolution data is accessible and utilized during rollback operations. By incorporating the resolution data into the information, SynQLite enables retrieving and applying the correct historical changes to achieve the desired rollback.

5.2 Pick new winner in conflict resolution

Resolution list

When an integrity constraint violation is detected, the conflicting operations are evaluated to determine a winner. After calculating priority, considering timestamp and causal length, operations are ordered in a resolution list. The resolution list can contain different types of operation depending on the different types of violations;

- **Uniqueness violations**: Insertion or deletion
- **Referential violations**: Deletion of entry or insertion of reference

- **Numeric violations**: Increments or decrements

From the example displayed in 4.1, a conflict between two insert operations at two different sites triggers a uniqueness violation at merge. The resulting resolution list is processed by integrity constraint enforcement.

The OP-ID that has the highest priority will be considered the winner and the rest is considered losers. Under the automatic violation resolution, the winner is propagated to the AR layer whilst the rest of the list is iterated and undone4.4. Support for user-aware conflict resolution demands the possibility to choose another outcome than the one displayed by the automatic resolution. Considering the same resolution list, the OP-ID of the previous loser is instead defined as the winner in this instance. The value of priority is not changed, so that the manual change of the winner does not affect the overall convergence toward cross-site consistency.

Finally, the conflicts are resolved and inserted data is propagated back to the AR layer. Before a relation can be inserted into AR, the existence of previous operations on the relation needs to be considered. If the relation have previous history and exists, it has been previously deleted and is stored in CRR as $(\tilde{r}(K, -, L)$, where causal length $L \mod 2 = 0$. The causal length is then incremented by one with the updated value and is reinserted into AR. If the relation does not exist as an CRR, the relation is simply inserted.

5.3 Conflicts on rollback

The design of SynQLite's conflict resolution defines rules that are consistently enforced across replicas. An example is what to do when additional conflicts can occur as a result of an undo operation. In the situation mentioned in 4.5, violations of referential constraints are automatically resolved by undo on the operation that inserted the reference. This is to prevent the possibility of an endless cycle of uniqueness resolution. However, when a manual resolution is an option, the outcome of choosing to keep needs to be implemented. For example, if an application would rather keep the reference pair than delete it due to many updates since the last merge. In this situation, there is still a potential for an endless cycle. To prevent this as a possible outcome, additional measures need to be implemented to scout ahead and recognize the possibility of this happening. This can be done by iterating the list of conflicts before the actual resolution.

5.4 Resolution groups

User-aware conflict resolution introduces new challenges to the guarantee of eventual consistency. In a network with multiple sites, different resolution groups will be formed. A resolution group is the processed resolution list from a conflict resolution and is a result of synchronization between different sites in a multi-site environment. In a purely automatic implementation of conflict resolution, the sites will converge into a consistent state by following the same rules and merging resolution groups. A challenge arises in how sites follow the same rules in an environment where the user can alter how sites behave.

A solution to this is to add relevant information about resolutions in the data transmitted to requesting sites on synchronization. On merge, sites should also merge resolutions history in addition to the local queue. This enables more extensive considerations of previous enforcement of integrity constraints.

6 Discussion

6.1 Limitations

SynQLite is missing real-world use-cases to understand the frequency of integrity violations and conflict resolution, and the potential side effects and challenges associated with user-aware conflict resolution.

To implement robust integrity enforcement, it is important to evaluate its performance and effectiveness in real-world scenarios. By analyzing real-world use-cases, the frequency of integrity violations and the subsequent frequency of conflict resolution can be determined. This information is important in designing a suitable solution that balances complexity and performance. If integrity violations occur less frequently, more complex and resource-intensive solutions can be considered without significant performance degradation. On the other hand, if violations occur frequently, a lightweight solution that minimizes performance impact becomes an option.

User-aware conflict resolution, proposed as a means to address the potential data loss of automatic conflict resolution, introduces a new set of considerations. While the thesis has explored the benefits and limitations of user-aware conflict resolution, it is important to note that there may be additional side effects that have not been discovered or discussed. The thesis has focused on addressing the data loss issue by enabling manual divergence from the automatic conflict resolution. However, user-aware conflict resolution may introduce unintended consequences or conflicts that occur due to varying user preferences and priorities.

To prevent potential unwanted side effects, a comprehensive understanding of previous integrity constraint enforcement is necessary. Keeping track of historical data on constraint enforcement allows for better prevention of unwanted side effects, leading to the development of more robust conflict resolution mechanisms. However, this introduces new challenges such as data storage requirements, and the impact on system performance. Balancing the need for historical is an important consideration in the design of user-aware conflict resolution.

SynQLite's integrity enforcement and user-aware conflict resolution offer promising solutions, further exploration is needed to address the frequency of integrity violations and their impact on conflict resolution. Future research should focus on real-world use-cases, and defining strategies to minimize unwanted side effects while ensuring the integrity and performance of distributed systems.

6.2 Use-cases of User-aware Conflict Resolution

User-aware conflict as a concept works well with systems where the limitations of an strictly last-write-wins protocol creates issues. In these types of systems, alternative conflict resolution strategies, such as application-specific conflict resolution policies, may be more suitable to ensure data integrity and consistency. Systems that involve concurrent updates to the same data from multiple sources can have crucial faults when some data is overwritten.

For example in collaborative editing systems, such as document editing or collaborative design tools. Multiple collaborators edit the same text or data, and some changes from one collaborator can overwrite the work of another collaborator working on the same data. It is then desirable to be able to manually pick what data is overwritten. More relevant to relational databases, such as designed for SynQLite, is version control while developing a system, similar to git.

7 Conclusion

This thesis have explored the challenges and solutions associated with prioritizing availability and partition tolerance in distributed systems while enforcing integrity constraints. We have discussed SynQLite's use of Conflict-Free Replicated Data Types (CRDTs) as a means to achieve strong eventual consistency, even in the presence of network partitions and concurrent operations.

While automatic conflict resolution provides a baseline solution, we have identified its limitations, including potential undesired outcomes such as data loss. To address these limitations, the proposed concept of user-aware conflict resolution comes forward, enabling users to influence the resolution process based on their preferences and specific application requirements.

SynQLite's integrity enforcement and user-aware conflict resolution offer promising solutions as a concept. However, further exploration is needed to address the frequency of integrity violations and their impact on conflict resolution. Future research should focus on real-world use-cases, and defining strategies to minimize unwanted side effects while ensuring the integrity and performance of distributed systems.

Bibliography

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. volume 41, pages 205–220, 10 2007.

[2] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. *ACM SIGACT News*, 33, 11 2002.

[3] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. pages 265–278, 10 2012.

[4] Mawahib Musa and Jan Lindström. Causal consistent databases. *Open Journal of Databases*, 2, 01 2015.

[5] Nuno Preguiça, Joan Marquès, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. pages 395–403, 06 2009.

[6] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. volume 6976, pages 386–400, 07 2011.

[7] Weihai Yu and Claudia-Lavinia Ignat. Conflict-free replicated relations for multi-synchronous database management at edge. pages 113–121, 10 2020.