UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Asynchronous replication of eventually consistent updatable views

Joachim Thomassen

INF-3971 Master’s thesis in Computer Science - May 2023

This thesis document was typeset using the *UiT Thesis LaTeX Template*.

“Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers. - usually, because the problem no longer fits on a single computer.”
–Mikito Takada

“Both optimists and pessimists contribute to our society. The optimist invents the airplane and the pessimist the parachute.”
–Gil Stern

“All models are wrong, but some are useful”
–George E. P. Box

Abstract

Users of software applications expect fast response times and high availability. This is despite several applications moving from local devices and into the cloud. A cloud-based application that could function locally will now be unavailable if a network partition occurs. A fundamental challenge in distributed systems is maintaining the right tradeoffs between strong consistency, high availability, and tolerance to network partitions. The impossibility of achieving all three properties is described in the CAP theorem. To guarantee the highest degree of responsiveness and availability, applications could be run entirely locally on a device without directly relying on cloud services.

Software that can be run locally without a direct dependency on cloud services are called local-first software. Being local-first means that consistency guarantees may need to be relaxed. Weaker consistency, such as eventual consistency, can be used instead of strong consistency. Implementing conflict-free replicated data types (CRDTs) is a provably correct way to achieve eventual consistency. These data types guarantee that the state of different replicas will converge towards a common state when a system becomes connected and quiescent.

The drawback of using CRDTs is that they are unbounded in their growth. This means they can quickly become too large to handle using less capable devices like smartphones, tablets, or other edge devices. To mitigate this, partial replication can be implemented to replicate only the data each device needs. This comes with the added benefit of limiting the information users obtain, thus possibly improving security and privacy.

The main contribution of this thesis is a new approach to partial replication. It is based on an existing asynchronously replicated relational database to support local-first software and guarantees eventual consistency. The new approach uses database views to define partial replicas. The database views are made updatable by drawing inspiration from the large body of research on updatable views. We differentiate ourselves from earlier work on non-distributed updatable views by guaranteeing that the views are eventually consistent.

The approach is evaluated to ensure it can be used for real scenarios. The approach has proved to be usable in the scenarios. The replication of database views has also been experimentally tested to ensure that our approach to partial replication is viable for less capable devices.

Acknowledgements

I want to thank my friends for providing some much-needed distractions during my years at university. Thank you for preserving my sanity.

I also want to thank my family for always being supportive of what I do.

A special thanks to my supervisor Weihai Yu. Without your enthusiasm, I might have chosen a much less exciting topic for my master's thesis. Thank you for providing a great challenge and a memorable end to my master's studies.

Contents

List of Figures

List of Listings

1 Introduction

There has been a large shift in the software industry toward deploying applications to the cloud. Client applications that could run locally have been moved to the cloud. Some have been enhanced with collaborative capabilities, such as Google Docs[18]. This move does not come without challenges, as users still expect fast response times and high availability out of the applications that they use.

Local-first software is an approach to software that prioritizes storing data on end-user devices without relying solely on cloud-based solutions [16]. This approach offers many benefits, such as improved privacy, security, faster performance, and greater control over data. It can enable applications to function even when faced with network partitions. This is where we face a fundamental challenge in designing distributed systems.

The CAP theorem states that a distributed system can't provide strong consistency, high availability, and partition tolerance simultaneously [8, 10]. By choosing local-first software, we should prioritize availability and partition tolerance. Consequently, to have an always-available system, we must give up on strong consistency and implement a weaker consistency model. Eventual consistency is one such weak form of consistency.

One way to achieve eventual consistency is through conflict-free replicated datatypes (CRDTs). These data structures allow multiple devices to store and modify data independently. It guarantees that the data structures will converge towards a common state when all devices have received all updates.

A challenge with using CRDTs is that they suffer from unbounded growth in terms of data size. Unbounded growth can become a problem for less capable devices. End-user devices may have limited storage, computation, and communication capabilities. Multiple devices may collaborate on a subset of data in large systems, not the entire dataset. The aggregate growth of information in the system may be more than some devices can handle.

Partial replication can be used to mitigate unbounded growth. Devices are likely only interested in smaller subsets of the entire dataset. Specifying relevant data subsets and replicating subsets to less capable devices might be valuable in building large-scale local-first systems. It is also important that the replicated subsets are updatable, keeping in line with the principles of local-first software.

There has been significant work on building an always-available, eventually consistent, relational database. Such a database can be used as local storage when building local-first applications. Conflict-free replicated relations (CRR) is the application of CRDTs to relational databases[29]. This thesis will focus on extending the work on CRR-augmented databases to allow partial replication. Partial replication can allow for more scalable local-first software solutions.

1.1 Overall goal

Previous work on an always-available, eventually consistent relational database has focused on local-first decentralized applications [26]. Each node in the decentralized system sends all its updates to other nodes in the system. This means that nodes will store all information when the system becomes quiescent.

This is probably not an issue for applications where nodes collaborate on a small working set, such as documents, message boards, or other real-time collaboration software. If, however, we extend the potential area of applicability to systems that generate large amounts of data from many nodes, it would be unreasonable to assume that any single node would be capable of storing the entire state of the system. The nodes in the system can be heterogeneous in computation, storage, and communication capabilities.

We focus on enabling users to replicate only an appropriate subset of the entire dataset. This subset would include all the data each node will be interested in analyzing and/or manipulating. The overarching goal of the thesis is described in the following statement:

> *The thesis aims to investigate the partial replication of CRR-augmented databases for large-scale, always-available, and eventually consistent systems.*

We present existing work on always-available, eventually consistent relational databases. We then explore how we can extend the existing solution to allow for efficient replication of data subsets. This exploration led us to investigate updatable views, an active research topic for decades. The requirements for updatable views and how this can be practically implemented will be detailed. We discuss the semantic implications of manipulating such views and how views can be put to practical use. We present an experiment showing whether or not partial replication is suitable for replicating data to less capable devices.

1.2 Scope, assumptions, and limitations

This thesis aims to present an approach that supports partial replication of data subsets. We do not consider any specific applications or scenarios in which our approach can be used. However, we try to reflect on the possibilities and limitations of our approach. This could inspire others to find practical use of the presented solution.

We do not consider underlying communication mechanisms as it has already been presented in previous work using SSH [26]. The actual synchronization protocols are retained, but there is no specific implementation of networked communication for synchronization.

There is a strong focus on how updates to data subsets are handled and how it relates to the existing approach for CRR-augmented databases. We perform a simple experiment that is aimed at demonstrating whether or not our approach is suitable. More complex experiments are omitted, as the behavior of our approach could vary widely depending on where it is applied.

1.3 Methodology

[6] identified three significant paradigms for approaches to computer science research. These paradigms are theory, abstraction, and design.

The **theoretical** paradigm comes from the world of mathematics and consists of four major steps. The first step is to characterize the object of study. This is done by creating definitions. Next, relationships between objects of study are hypothesized. This means that a theorem can be put forth. The hypothesized relationships are then examined to ensure that they do indeed exist. Finally, the results are interpreted.

The **abstraction (modeling)** paradigm is derived from the experimental scientific method. The experimental scientific method starts by first defining a hypothesis for some phenomena. Then a model is constructed, which can be used to make a prediction. The model is an abstraction of real-world phenomena. Then experiments are conducted, and data is collected. Finally, the results are analyzed to determine if the predictions made by the model were correct.

The **design** paradigm comes from the world of engineering. It is used to construct a system to solve a given problem. To achieve this, the requirements for the solution must be stated. These requirements lead to the creation of specifications.

In all cases, the subject performing work within these paradigms is expected to iterate the steps within each paradigm. As pointed out in [6], no single approach is more fundamental to computer science. The intertwined paradigms make it challenging to point to a singular approach.

Although these paradigms are intertwined, the approach used when working on this thesis resembles the design paradigm the most. The problem we face is that we would like to enable the construction of larger-scale, eventually consistent systems where central infrastructure is highly capable while end-user devices are not. We present requirements for the approach and use an iterative process until the requirements are met to a satisfactory degree.

In short, our approach should minimize the required storage, computation, and communication capabilities for end-user devices. Users should only need to store data they are interested in, data should be updatable, the updates should be correctly translated, and the system must be eventually consistent. We present an approach and implementation that meet these specifications and an evaluation that indicates how well it solves the problem.

1.4 Contributions

- An implementation of asynchronously replicated databases in Rust.

- A new approach for updatable database views that allows for asynchronous replication.
- A way to define what data should be partially replicated. Defining data subsets can be done using simple SQL definitions.
- A paper that has been submitted to ADBIS[2] and is currently under review. The paper has been put as an appendix at the end of the thesis.

1.5 Outline

The thesis is organized as follows:

Chapter 1 has just concluded. It presented the motivation for exploring partial replication in the context of local-first systems. It also briefly summarizes the general scientific approach, this thesis's scope, and main contributions.

Chapter 2 will present the necessary background knowledge to understand the approach and implementation. It also clarifies the terminology used to avoid confusing them with related terms.

Chapter 3 presents the approach to an asynchronously replicated relational database based on CRR. It then presents the approach for updatable views and how it is combined with the existing approach to CRR-augmented databases.

Chapter 4 covers implementation-specific details of how updates to data subsets are handled and how the synchronization procedure differs from the existing synchronization procedure for an asynchronously replicated database.

Chapter 5 will investigate the solution's applicability. This concerns how updates can be understood and whether they suit real scenarios. It will also present an experiment to evaluate whether partial replication is suitable for less capable devices.

Chapter 6 presents the work on which our approach to replicating views is based. It also presents earlier work on updatable views to which our approach is most similar. Finally, the main differences between our approach and other work are presented.

Chapter 7 highlights how well the current implementation satisfies the requirements. It also suggests future work that would improve upon the implementation that is to be presented.

Chapter 8 concludes the thesis with a summary of the motivation, the approach, experimental results, and our contributions.

2

Technical background

This chapter will present the technical knowledge needed to understand our approach. We start by providing a further explanation of local-first software. We explain what consistency means, as consistency has many interpretations depending on the context where the term is used. We then explain CRDTs and present several foundational CRDTs used in our approach. An optimization to state-based CRDTs, called delta-state CRDTs, must also be understood to understand our approach. We present the necessary knowledge from relational database theory. Finally, we clarify many terms to avoid confusion with similar terms.

2.1 Local-first software

Local-first software is a shift in thinking from applications that today use cloud services and applications for storing data in centralized infrastructure. Cloud services can create a belief that the correct data is the one that is stored in the centralized infrastructure. Devices that store copies of the data are merely intermediates, which are not necessarily proper replicas for the data they store.

Local-first applications are those in which data can be stored locally on end-user devices while having the data be true copies, which are no less important than what is stored in centralized infrastructure. This new way of thinking can bring benefits such as application responsiveness and availability improvements. As said in [16], conflict-free replicated data types (CRDTs) are general-purpose data types suitable for implementing local-first software.

Local-first is relevant within the context of this thesis as it is the primary motivation for creating an always-available database. The database is built using CRDTs. Partial replication is hypothesized as a possible solution to mitigate the performance overhead of ever-growing CRDTs and allow for more efficient implementations of local-first software.

2.2 Consistency

We explore the concept of consistency concerning replication in distributed systems. A data item may be present at multiple machines, and consistency can therefore be described as whether replicas are in the same state. Traditional approaches ensure replicas are in the same state by utilizing consensus protocols such as Paxos that require coordination[27, pp. 440-454]. Rather than focus on consistency as access guarantees for concurrent processes to shared objects or as consistency in which data must obey defined invariants, we will focus on whether or not the program outcomes are consistent with user expectations. This is similar to the definition used in [11].

Weak consistency models, such as eventual consistency [28], can guarantee that replicas will converge towards a shared state when the system comes to a quiescent state. Quiescence is achieved when there are no ongoing updates in the system. The replicas converge when all updates are propagated to all replicas. Achieving eventual consistency does not require coordination between nodes.

There is also a stronger form of consistency called strong eventual consistency. Strong eventual consistency guarantees that if two replicas have received the same set of updates, then the two replicas are in the same state [22]. The solution presented in this thesis guarantees strong eventual consistency.

2.3 CRDT

A conflict-free replicated data type is an abstract data type whose implementation guarantees convergence without coordination. The explanation of CRDTs is based on the work of [21]. There are two different approaches to building CRDTs. A CRDT can be either operation-based or state-based. Operation-based

CRDTs converge by sending update operations between replicas. For replicas to converge, the transmitted operations must commute. Any operations that do not commute must be delivered in a guaranteed order using some reliable communication channel.

State-based CRDTs synchronize by transmitting the state of the data type instead of operations performed on the type. When two instances of a data type synchronize, they do so using a merge operation. The state of a CRDT can be represented as a join-semilattice [9]. The merge operation must guarantee the least upper bound between the two semilattices to be joined. This implies that the merge operation is associative, commutative, and idempotent. The state information captured in the state can only ever increase monotonically. This is necessary to guarantee that the state converges towards the least upper bound of recent updates. This is defined in [21] as states being monotonic semilattices.

A general understanding of CRDTs is necessary to understand the existing database architecture and the contributions of this thesis. We give examples of CRDTs that lay the foundation for the existing database solution. We summarize relevant CRDTs below based on material from [21].

2.3.1 LWW-register

A Last Writer Wins register (LWW-register) is a register that contains a single value. The value in the register can be mutated at any replica. As the name suggests, the most recent write will be the current value in the register. To guarantee this behavior, the register holds the timestamp of the last mutation. Merging two replicas of the register involves checking for a greater timestamp and choosing the register value with the highest timestamp value. All timestamps must be unique and thus be sortable into a total order. It is also necessary for timestamps to be generated in strictly monotonically increasing order at a replica. Timestamps need not be monotonic between replicas, thus still preserving their coordination-freeness.

2.3.2 Grow-only set

There are many kinds of set CRDTs, and they differ only in their behavior for adding and removing elements. A grow-only set is a set in which the only allowed operation is to add an element. Its correctness is quite easy to prove. A grow-only set is a monotonic semilattice, as its updated state can only be a superset of its previous state. The merge operation is a regular set-union operation. A union of sets is associative, commutative, and idempotent and will, therefore, also produce the least-upper bound for the monotonic semilattices.

The grow-only set is presented here as it is used to compose new and more sophisticated CRDTs. A two-phase set (2P-set) is a CRDT composed of two grow-only sets. One set tracks insertions, and the second tracks removals. This means that elements can only be added and/or removed once. The main insight we can gain from this is that a composition of CRDTs can be used to create new CRDTs.

2.3.3 CL-set

We briefly describe the work on CL-sets as presented in [30]. The CL-set is a variation of the many kinds of sets that have been presented in CRDT theory. It uses causal lengths to determine whether or not an element is in the set. Each element is associated with a causal length, essentially a number that determines whether an element should be in the set. The associated element is in the set if the causal length is odd. On the other hand, an even number indicates that the element is not in the set. Intuitively, insertions and deletions can only happen in turn. An element that does not exist cannot be deleted, and an element cannot be added when it already exists in the set.

Listing 2.1: A simple implementation of CL-set in Python

```python
s = defaultdict(lambda: 0)

def in_set(s, e):
    if e in s and is_odd(s[e]):
        return true
    return false

def insert(s, e):
    if not in_set(s, e):
        s[e] += 1

def remove(s, e):
    if in_set(s, e):
        s[e] += 1

def merge(s1, s2):
    result = s1
    for e, cl in s2:
        result[e] = max(1, cl, s1[e])
    return result
```

Listing 2.1 shows a simple implementation of a CL-set where each element maps to some causal length. The variable *s* represents the state, and *e* denotes an element that can be added/removed from the set. Indexing into the state by element returns the causal length for that element. The keys of the map are the elements, and the values are the causal lengths of the element. Any element not represented in the set will have a causal length of 0 by default.

Inserting an element involves checking whether it is already in the set. If it is, adding once more will have no effect. If it is not in the set, the causal length is incremented. If the element does not exist in the map, it is created, and its causal length is incremented from 0 to 1.

Removing an element is done by checking whether the element is already in the set. If it is not, we do nothing. Note that the element does not need to be created if it does not already exist in the state. If it is in the set, it means that the causal length was odd, and incrementing it will remove it from the set.

Merging two CL-sets is a simple union of the two maps. The causal length of an element that appears in both sets is assigned the maximum causal length between the two states.

2.4 Delta-state CRDTs

One of the major critiques of using state-based CRDTs is that they need to transmit their state to some other nodes to synchronize. This can be a costly solution if the states of a CRDT are large. The ever-growing nature of CRDTs (monotonic semilattice) means synchronization will become costlier over time.

An optimization that can be employed is only to send state deltas. Delta-state optimization is presented in [3]. These delta-states are said to be join-irreducible states. Any state can be described as a set of join-irreducible states.

To synchronize, replicas only need to send join-irreducible states to other nodes. It is only necessary to send those join-irreducible states that may not exist at other replicas. This state can be much smaller than the entire state, thus reducing communication costs. It does induce an overhead of producing these delta-states.

Listing 2.2 shows merging a full state with a join-irreducible state. The join-irreducible states for CL-sets are the individual elements with their causal length. Merging with a join-irreducible state only takes in an argument *t* containing an element of the set and its causal length. It finds any corresponding element already present and takes the max causal length. Merging two full states can be done in terms of merging join-irreducible states. This is shown as a rewritten merge function in listing 2.2.

Listing 2.2: Relationship between a merge with a join-irreducible state and a merge with a full state.

```
def merge_irreducible(s, t):
    e, cl = t
    s[e] = max(1, cl, s[e])
    return s

def merge(s1, s2):
    result = s1
    for t in s2:
        merge_irreducible(t)
    return result
```

2.5 Relational databases

We focus on CRDTs applied to databases using a relational data model. The relational data model is a popular way of structuring information in databases. We summarize the main aspects of the relational data model based on the definitions given in [23, pp. 37-41].

A relational database contains a set of tables that contains a set of rows. Each table has columns defining the domain for each row's values. A column is also called an attribute. The rows within the table are logically related. For example, think of a table containing all residents in a country, where each resident maps to a single row. Each row has several attributes, including a person's address and social security number. Rows within a table can also reference rows in other tables, allowing the relational model to express relationships between data in separate tables.

It is also useful to understand schemas in relational databases. A database schema describes the design of the database. A relation schema defines attributes and their domains for a given named relation. A database schema consists of a set of relation schemas. A database instance is used to refer to the state of the database at a given point in time. A relation instance is used to describe the state of a relation.

2.5.1 Integrity constraints

Relational databases allow users to specify integrity constraints on their data model. These constraints are invariants that the database instance must always uphold [23, p. 145]. Reference constraints enforce an invariant that a referenced row must be present [23, p. 149]. Uniqueness enforces the invariant that no two tuples may contain the same values for a set of attributes [23, p. 147].

2.5.2 Materialized views

We will briefly discuss materialized views at various points throughout the thesis. A materialized view is a table that contains a precomputed set of results from a view [23, p. 778]. A view is simply a predefined query that has been named. Materialized views are used to speed up access to commonly used views. Materialized views will always contain redundant data extracted from the tables contributing to the view.

2.5.3 Functional dependencies

Functional dependency modeling is a technique from normalization theory for database systems. Functional dependencies define constraints between attributes [23, pp. 320-322]. For example, a social security number can be used to determine the address of a person. This means an address is functionally dependent on a social security number, as the number can uniquely identify a person with an address. We will briefly mention functional dependency as it was an integral part early in the project.

2.6 Terminology

It is necessary to clarify how different terms are used in this thesis. The first terms to clarify are the terms data subset and partial replication. A data subset is used to describe a logical subset of the data domain within a system. It is a logical partitioning of the data. Partial replication refers to database instances only replicating a specified data subset. This is in contrast to a regular replica, which replicates all data in the domain of a system.

The terms relation and table can be used interchangeably throughout the thesis. Both refer to a set of related records within a database. Rows and tuples may also be used interchangeably to refer to a record. The terms relation and tuple will be used when explaining the approach, while table and row are used when discussing implementation-specific details.

A CRR-augmented database is a qualification used for databases that have been modified to enable asynchronous replication and that guarantee eventual consistency. CRR is an acronym that comes from the work on conflict-free replicated relations [29].

3

Approach

This chapter presents an existing approach for an asynchronously replicated database based on conflict-free replicated relations [29]. We present the requirements for partial replication within the existing approach. This will lead us to investigate partial replication using database views and how database views can be updated. The required properties for the translation of updates are studied before defining actual translations. We will present our approach to updatable views using several examples.

3.1 Asynchronously replicated database

The high-level architecture of a CRR-augmented database is shown in figure 3.1. It shows that it is a two-layered architecture. The top layer is referred to as the AR layer. The AR layer is the application layer where applications can query and update data. The AR layer consists of all relations developers define in the application database schema. It is assumed that the database schema has been defined upfront.

The bottom layer is coined the CRR layer. The CRR layer is a superset of the information in the AR layer, as it contains metadata and logically deleted data that would not surface in the AR layer. Being a CRDT, we can never actually remove data.

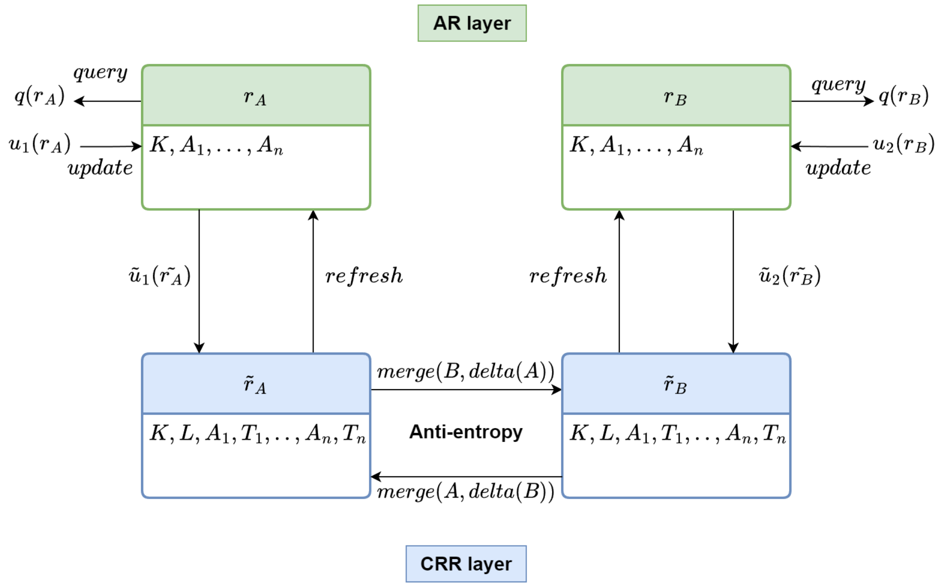


Figure 3.1: Two-layer architecture of CRR-augmented databases

The way the two layers interact is relatively straightforward. Any query in the AR layer can be satisfied without modifying the CRR layer. An update in the AR layer must trigger updates downstream in the CRR layer. This is to ensure that both layers are consistent with each other. An important point that should be made is that the AR layer is essentially a view of the underlying CRR layer. The difference is that metadata is stripped away, and logically removed rows are absent in the AR layer.

The two layers are each locally stored on each node, meaning that propagating updates from the AR layer to the CRR layer are immediate operations. Two nodes can synchronize by performing an anti-entropy procedure at the CRR layer. The synchronization is done by transmitting the state of one database instance to another and merging the state at the target database instance. This can be done using the entire database state or a set of join-irreducible states.

A relation is a set of tuples, as said earlier. We can model the relations in a relational database as CL-sets. The tuples in a relation are the same as the elements in a CL-set. Each tuple is therefore associated with a causal length. This means tuples can be inserted and removed. Synchronization can be performed according to the merging rules of CL-sets. This fails to consider key constraints on relation schemas.

Relations in relational databases often have a key. A superkey is a set of attributes uniquely identifying a tuple within a relation [23, p. 43]. If two nodes add a tuple with the same key but different non-key attributes, are these referring to the same row? We assume that the keys for a relation will always semantically refer to the same tuple. The causal length is therefore tied to the key, and merging would now not consider non-key attributes.

Eventual consistency will be violated if merging does not consider non-key attributes. Each non-key attribute can be defined as a Last-Write Wins register to guarantee eventual consistency. Each attribute gets an associated timestamp indicating the time it was last updated. Merging two relation instances with the same schema is as simple as merging equal tuples by key, taking the highest causal length between matching tuples, then merging all non-key attributes using the merge procedure of LWW-registers. If some tuples do not have an equivalent in the other database instance, they are added to the final merged result, just like a set union operation.

We are now equipped with the knowledge to explain figure 3.1 in more detail. Each relation schema $R(K, A_1, \ldots, A_n)$ at the application layer has a CRR-layer representation $\tilde{R}(K, L, A_1, \ldots, A_n, T_1, \ldots, T_n)$. The superkey for the relation is represented by K. The causal length is represented as L and, together with K, is logically a CL-set. A_i represents a non-key attribute, while T_i is its associated timestamp. The pairing of A_i and T_i makes up an LWW-register.

Now that the general layering has been explained, it is helpful to understand how a user may interact with the data. Updates can come in the form of inserting a new tuple or updating or deleting an existing tuple. A tuple may be inserted if it does not already exist in R. That means the tuple may exist in $\tilde{R}$ but with an even causal length. Insertion would then increment the causal length. If it does not already exist in $\tilde{R}$, the tuple is inserted with an initial causal length of 1. It is essential to note that inserting a tuple with the same primary key K but different non-key attributes will update the non-key attributes.

A tuple can be deleted if it exists in the AR layer. It gets deleted if its causal length is odd, meaning the tuple already exists. The key K identifies the deleted tuple, and deletion is done by incrementing its causal length. An even causal length means that it should not appear in the AR layer.

A tuple can only be updated when it is already present in the AR layer. That is, its causal length is odd. Updating non-key attributes is trivial, as it only needs to update the attribute and its timestamp. Updating any attribute that is also a key attribute is essentially the same as removing the old tuple and inserting a tuple with the new key.

The mechanism by which the AR-layer relations are refreshed will be discussed in chapter 4.

3.2 Synchronizing using delta-states

The existing approach allows for synchronization using delta-states. The join-irreducible states are the individual tuples of the relations within the database. It is only necessary to send all the tuples other nodes might not know. This is shown in figure 3.1 as the anti-entropy procedure. It involves calculating a delta at a node, transmitting it to a target node, and merging it. A delta contains the join-irreducible states to merge at the target node.

Each node has a vector clock[27, pp. 317-322] containing the last update received from any other node in the system. The vector clock contains other known node identifiers with the timestamp of the latest update received from that node. To generate deltas, nodes must keep a history of updates. This is necessary to ensure all updates from all sites are synchronized correctly. This information is kept in a history relation $\tilde{H}(R, K, T, N)$. The history relation tracks an update in relation R for every tuple identified by key K with an update time of T at node N.

The synchronization procedure is initiated by a node sending its vector clock to the node from which updates will be pulled. The node receiving the vector clock will inspect it and extract all tuples from its history relation that have an update time later than what is in the vector clock for all nodes. Since each update is associated with a node, it is easy to compare. If a node does not have an entry in the vector clock, all updates from that node will be transmitted.

After receiving the updates, the node will merge the tuples according to the anti-entropy procedure. It then updates the vector clock using the update history received from the other node. The update history for all tuples in the delta is always transmitted during a synchronization.

3.3 Integrity constraints

The existing approach to CRR-augmented databases can handle violations of integrity constraints. The integrity constraints that are handled are reference constraints and unique constraints. These are the most common integrity constraints in relational databases and should be sufficient for most situations. Integrity constraint violations can occur when two nodes concurrently make conflicting changes.

A reference integrity violation can occur if one node inserts a tuple referencing a tuple that was concurrently deleted at another node. The situation is resolved by reinserting the referenced tuple. A unique constraint can be violated when two sites concurrently insert a tuple with the same value that should be unique. The violation is resolved only by keeping the insertion that happened first.

Violations are resolved upon synchronization and are done by adding additional updates to solve the conflicts. The updates are deterministic in that concurrent resolution of the same conflict will always apply the same update. This means that strong eventual consistency is still preserved.

3.4 Requirements for partial-state replication

We now focus on supporting partial replication for CRR-augmented databases. Our primary focus is to provide valuable data subsets for nodes in a system. We reason that many nodes are only interested in a small subset of the information. We assume that the information in the system can grow quite large and that end-user devices cannot store all the information. It is necessary to use partial replication only to replicate data subsets that interest end-users.

Any partial-state replication solution should therefore fulfill the following requirements:

- **Data subsets are user-centric.** The solution should only replicate data of interest to nodes that may be less capable. This means that the data replicated to some set of nodes may differ substantially between the nodes. It should be easy to define a data subset for each node.

- **Clear update semantics.** The solution must allow updates, as read-only replication is considerably less useful. Being able to interact with data allows us to build useful collaborative applications. It also requires that updates are well understood with a clear semantic meaning. Changes to the replicated data should not have unintended side effects that users do not easily understand.

- **Updates have a well-defined translation.** Any update should correspond to some update that could have been applied to the source database. Updates cannot have ambiguous translations.

- **Guarantees strong eventual consistency.** The solution for replicating views must still guarantee strong eventual consistency. This requirement means that the new solution must fit into the existing approach, guaranteeing strong eventual consistency.

- **Minimize storage/computation/communication.** This requirement essentially captures the initial motivation for incorporating partial-state replication. It ensures that the solution for partial-state replication should work on less-capable devices.

3.5 Database views

Most relational database systems offer the possibility to define views. A database view is an abstraction over the database schema, and views may have some semantic meaning that is not easily interpretable just from looking at the database schema [23, p. 137]. It can also protect data by limiting the amount of information that users of a view can access.

For example, consider a database schema with tables for students, classes, and departments. A view can be defined to answer questions such as: "Which students are taking classes in the computer science department?" and "Who are the top K-performing students in the biology department?". This can be run as a single query on the full database schema. Still, it is often helpful to define views to clarify what is being queried and to make complex queries easy to construct by querying against a composition of views.

Defining views is a suitable technique for partial replication of databases, for their ability to convey semantics and limit the amount of information gained. Views are a mature technique and can be easily defined using a query language such as SQL (Structured Query Language [23, pp. 65-66]). This means that one should be able to define user-centric views.

Users of views are likely interested in modifying the information they receive. Views have traditionally only been applied as read-only abstractions, and having views be updatable has not been important. Despite this, updatable views have greatly interested researchers for many decades. Research on updatable database views ensures that semantics and translations of updates are well understood. We combine the mature research on updatable views and apply it in a setting where views are asynchronously replicated.

We expect that required storage, computation, and communication capabilities are minimized by limiting the amount of replicated information.

3.6 Translations of view updates

Supporting updatable views comes with many challenges. One particular challenge is that view updates must be well-defined. Any update applied at the view must have some translation that can be applied at the source state. We use the term *source* to refer to an instance replicating the entire database schema, while *view instances* only replicate a view. A translation of updates at a view to the source state is indicated with $T_\uparrow$. A translation of updates at the source state to the view is indicated with $T_\downarrow$.

$$\begin{tikzcd}
s \arrow[r, "T_\uparrow(s, u_v)", ""{name=T}] \arrow[d, "V"'] & s' \arrow[d, "V"] \\
v \arrow[r, "u_v", ""{name=B}] & v'
\arrow[dotted, from=B, to=T]
\end{tikzcd}$$

Figure 3.2: Well-defined translation of update u on view v applied to s. The figure is adapted from [15].

Figure 3.2 shows the process of extracting a view instance v from database instance s, and then performing an update u on the view. The extraction is done using a view definition V. The translated update, $T_\uparrow(s, u_v)$, is applied to the source state. Applying the translation at the source transitions it to a new state, which induces a new view state. The view state induced by the new source state should be the same as that resulting from applying the updates directly at the view. This can be concisely described by the following equivalence: $V(T_\uparrow(s, u_v)) = u(v)$.

Figure 3.3 shows how translations of view updates must behave within the context of the existing database architecture. The main difference to figure 3.2 is that translations are applied to join-irreducible states, meaning we can decouple translations from any specific target state.

$$\begin{tikzcd}
s_0 \arrow[r, "\Delta s'", ""{name=A}] \arrow[d, "V"'] & s_1 \arrow[r, "T_\uparrow(\Delta v')", ""{name=B}] & s_2 \arrow[r, "\Delta s'' = IC(s_2)", ""{name=C}] & s_3 \arrow[d, "V"] \\
v_0 \arrow[r, "\Delta v'", ""{name=D}] & v_1 \arrow[r, "T_\downarrow(\Delta s')", ""{name=E}] & v_2 \arrow[r, "T_\downarrow(\Delta s'')", ""{name=F}] & v_3
\arrow[dotted, from=A, to=E]
\arrow[dotted, from=D, to=B]
\arrow[dotted, from=C, to=F]
\end{tikzcd}$$

Figure 3.3: Well-defined translations for delta-states

A database instance with schema S has the initial state s_0. A view V is then defined on S such that a view extracted from state s_0 is the view state v_0. Updates are then applied independently at both the source and view states. The updates at the source are captured in $\Delta s'$ and make the source transition to state s_1. The updates at the view are captured in $\Delta v'$ and make the view transition to state v_1. The view state induced by s_1 is not the same as the view state v_1, as they have not synchronized yet.

Applying $T_\downarrow(\Delta s')$ to the view transitions the view into state v_2. Applying the translation $T_\uparrow(\Delta v')$ to the source makes it transition to intermediate state s_2. It is never observed since the update at the view violates an integrity constraint. The violation is resolved by applying an additional delta $IC(s_2)$ in the same manner as has already been presented in section 3.3. The translated resolution is then applied at the view upon the next synchronization.

The final states are s_3 and v_3. Both states were achieved by only transmitting translations of updates. The state s_3 must induce the final view state v_3 for view updates to be well-behaved.

The translations are delta-states, a set of join-irreducible states, and applying the delta is done using the same merge operation $\sqcup$ as is used in the existing database architecture. Incorporating view translations still guarantee strong eventual consistency since the merge operation is associative, commutative, and idempotent. The view extracted from the final state of the source instance will be the same as the final state at the view after applying all updates, which is captured in the following equation:

$$V(s_0 \sqcup \Delta s' \sqcup T_\uparrow(\Delta v') \sqcup \Delta s'') = V(s_0) \sqcup \Delta v' \sqcup T_\downarrow(\Delta s') \sqcup T_\downarrow(\Delta s'')$$

3.7 View updates are ambiguous

To support updatable views, we must ensure that updates have a well-defined translation and can be sensibly applied to the underlying database. Early research on updatable views has been concerned with the semantics of updatable views. Since views are abstractions which can be used to answer some query of interest, how would updates to a view affect the database in a way that makes sense in the context of the view?

Suppose we are interested in finding all students taking some class. What does it mean to delete a data item from such a view? Does it remove the student entirely, akin to expelling them? Or does it simply remove the relationship between the student and the class, such that the student can no longer be said to take the class? Or do we remove the class the student is a part of?

Earlier work has focused on the update semantics on views and how those updates are translated into changes on the underlying database. Updating a view is inherently ambiguous. This is because there is a many-to-one relationship between tuples in the contributing relations and the view.

An example of such ambiguity is illustrated in figure 3.4. The figure shows a view that is a natural join between relation instances r_1 and r_2. Two updates are then applied to the view. The updates are $-v\langle a2, b1, c1\rangle$ and $v\langle a1, b1 \nearrow b2, c1\rangle$. There are many possibilities for translating these updates.

For the removal, do we delete from r_1, r_2, or both? Deleting from r_2 will also remove the tuple $\langle a3, b1, c1\rangle$ from the view. There is also the problem of updating the join attribute. The originally referenced tuple may or may not have been deleted. Do we update the existing tuple or add a new one? What will happen if we update the existing attribute?

$$\begin{array}{c|cc} r_1 & \text{A} & \text{B} \\ \hline & \text{a1} & \text{b1} \\ & \text{a2} & \text{b1} \\ & \text{a3} & \text{b1} \end{array} \bowtie \begin{array}{c|cc} r_2 & \text{B} & \text{C} \\ \hline & \text{b1} & \text{c1} \\ & & \\ & & \end{array} \longrightarrow \begin{array}{c|ccc} v & \text{A} & \text{B} & \text{C} \\ \hline & \text{a1} & \text{b1} & \text{c1} \\ & \text{a2} & \text{b1} & \text{c1} \\ & \text{a3} & \text{b1} & \text{c1} \end{array}$$

$$\begin{array}{c|cc} r_1 & \text{A} & \text{B} \\ \hline & ? & ? \\ & ? & ? \end{array} \bowtie \begin{array}{c|cc} r_2 & \text{B} & \text{C} \\ \hline & ? & ? \\ & ? & ? \end{array} \longleftarrow \begin{array}{c|ccc} v & \text{A} & \text{B} & \text{C} \\ \hline & \text{a1} & \text{b2} & \text{c1} \\ & \text{a3} & \text{b1} & \text{c1} \end{array}$$

Figure 3.4: Update ambiguity on a join view

The update ambiguities lead us to formulate a principle for updates. The principle is that any update should be translated to a minimal change that achieves the desired effect. This is to restrict the effects of updates on one tuple affecting other tuples. Reasoning about such side effects may not be straightforward. Side effects may include the removal of one tuple leading to the removal of other tuples. This is likely not the intention of the updating user.

3.8 Disambiguating view updates

We look to previous work on updatable views to disambiguate the updates to a view. Previous work has used functional dependencies between relations to determine the mapping from a view to a source database [4]. The same approach is taken here. Most SQL relational databases have no way to define functional dependencies directly. Therefore, we rely on developers having specified superkeys for all relations and key relationships between relations. All relations will therefore have a functional dependency of the form $A \longrightarrow B$ where A is the set of attributes forming the superkey, and B is the set of non-key attributes determined by attributes in A.

A tuple in the view may have one or more source relations from which it originates. A view may join multiple tuples into a single tuple using a natural join on relation instances. It is helpful to note that tuples in views that only do selection and projection only have a single source tuple from which it originates. Knowing what source tuples contribute to a tuple in the view, we can specify precise update rules that satisfy the translation properties presented in section 3.6.

To ensure that it is known which tuples contribute to a given tuple, views cannot project away the superkey defined for each source relation. This restriction is necessary mostly for practical reasons. It means it is always possible to identify the source tuples that contribute to a tuple in a view. This makes it easy to define update rules that behave as expected.

It also fits the existing database architecture, which relies on superkeys for semantically identifying equivalent tuples. The join-irreducible states are individual tuples, which are identified by a key. We must know the causal length to determine whether a tuple exists and allow updates to tuples. The causal length is associated with the key, meaning no key attributes can be dropped. Any attribute appearing on the left-hand side of a functional dependency may not be projected away.

3.9 Translations by example

We now focus on specifying the translations of updates, which will uphold the previous translation properties and the restrictions laid out in the previous sections. The examples are borrowed from [4]. The database schema is slightly different, as we do not directly specify functional dependencies but only define relationships with keys. Three relations track what albums a store has and what tracks belong to what album. The database schema is illustrated in figure 3.5.

Figure 3.6 shows an example of a view. The source database contains the relations: R_a for albums, R_t for tracks, and R_{ta} for the relationships between tracks and albums. The view schema is defined as V, and a view is extracted using a sequence of queries on the source database instance.

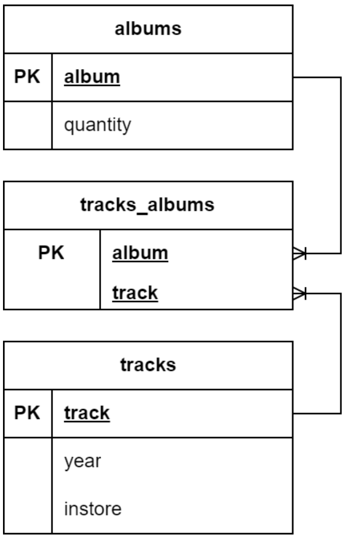
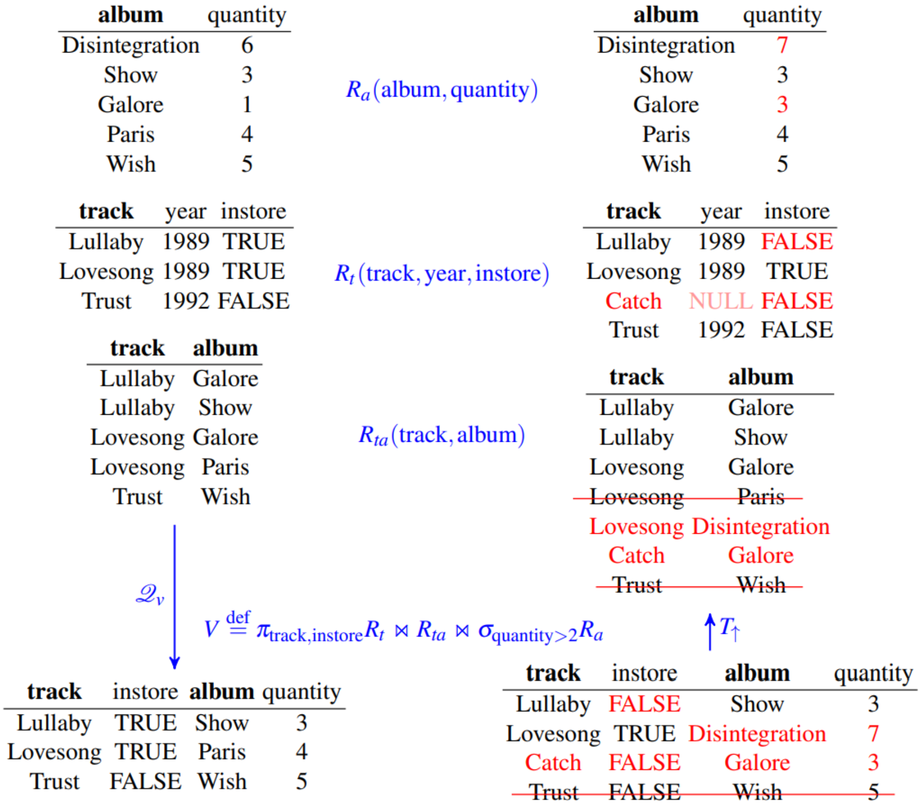


Figure 3.5: Database schema used for examples in our approach



album	quantity
Disintegration	6
Show	3
Galore	1
Paris	4
Wish	5

album	quantity
Disintegration	7
Show	3
Galore	3
Paris	4
Wish	5

track	year	instore
Lullaby	1989	TRUE
Lovesong	1989	TRUE
Trust	1992	FALSE

track	year	instore
Lullaby	1989	FALSE
Lovesong	1989	TRUE
Catch	NULL	FALSE
Trust	1992	FALSE

track	**album**
Lullaby	Galore
Lullaby	Show
Lovesong	Galore
Lovesong	Paris
Trust	Wish

track	**album**
Lullaby	Galore
Lullaby	Show
Lovesong	Galore
~~Lovesong~~	~~Paris~~
Lovesong	Disintegration
Catch	Galore
~~Trust~~	~~Wish~~

$$V \stackrel{\text{def}}{=} \pi_{\text{track,instore}} R_t \bowtie R_{ta} \bowtie \sigma_{\text{quantity}>2} R_a$$

track	instore	**album**	quantity
Lullaby	TRUE	Show	3
Lovesong	TRUE	Paris	4
Trust	FALSE	Wish	5

track	instore	**album**	quantity
Lullaby	FALSE	Show	3
Lovesong	TRUE	Disintegration	7
Catch	FALSE	Galore	3
~~Trust~~	~~FALSE~~	~~Wish~~	~~5~~

Figure 3.6: Example of an updatable view

A set of updates are then applied to the view and indicated using red in the figure. An update, u_i, is an update to a single tuple; each update in the figure is summarized below. $+v\langle\dots\rangle$ indicates an insertion, and $-v\langle\dots\rangle$ indicates a deletion from view v. The arrow symbolism, X $\nearrow$ Y, indicates that an attribute is updated from value X to value Y. The updates for figure 3.6 are shown below.

$$u_1 = v\langle \text{Lullaby, TRUE} \nearrow \text{FALSE, Show, 3}\rangle$$

$$u_2 = v\langle \text{Lovesong, TRUE, Paris} \nearrow \text{Disintegration, 4} \nearrow \text{7}\rangle$$

$$u_3 = +v\langle \text{Catch, FALSE, Galore, 3}\rangle$$

$$u_4 = -v\langle \text{Trust, FALSE, Wish, 5}\rangle$$

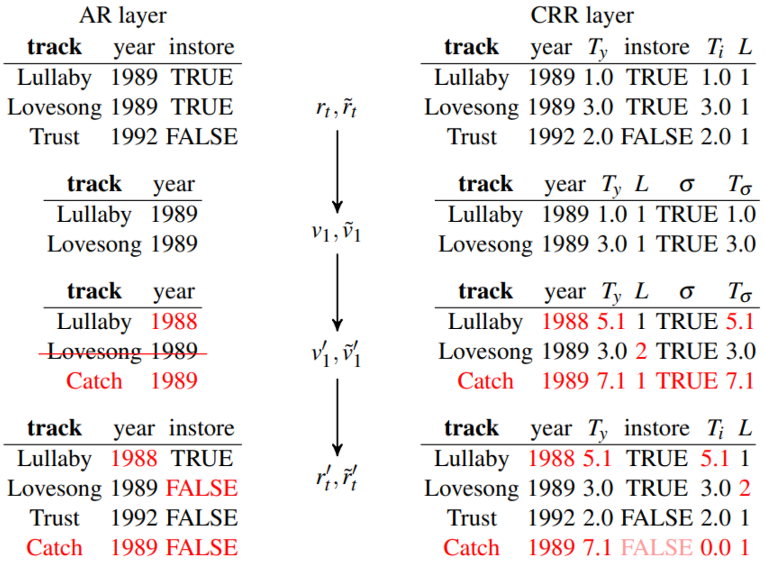
Below are the translations of the updates, shown as $T_{\uparrow}(u_i)$, that satisfy the properties in section 3.6. The following sections will explain how these translations are derived using simple examples.

$$\begin{aligned}
T_{\uparrow}(u_1) &= [r_t\langle \text{Lullaby, 1989, TRUE} \nearrow \text{FALSE}\rangle], \\
T_{\uparrow}(u_2) &= [r_a\langle \text{Disintegration, 6} \nearrow \text{7}\rangle, -r_{ta}\langle \text{Lovesong, Paris}\rangle, +r_{ta}\langle \text{Lovesong,} \\
&\qquad \text{Disintegration}\rangle], \\
T_{\uparrow}(u_3) &= [r_a\langle \text{Galore, 1} \nearrow \text{3}\rangle, +r_t\langle \text{Catch, NULL, FALSE}\rangle, +r_{ta}\langle \text{Catch, Galore}\rangle], \\
T_{\uparrow}(u_4) &= [\,-\,r_{ta}\langle \text{Trust, Wish}\rangle].
\end{aligned}$$

3.9.1 Translation of selection and projection

Views containing selections and projections are simple and are suitable for introducing translations of view updates. Views containing only selection and projection are less complex as each tuple in the view has a one-to-one relationship to its contributing tuples. We will study insertion, update, and deletion from these views.

Figure 3.7 illustrates how view updates are applied. The left-hand side shows the application layer, which is the logical view of the database. The right-hand side shows how the metadata is manipulated according to the previously presented rules for CRR-augmented databases. The relation r_t contains tracks identified by their name, release date, and whether they are in store. The view state is v_i, and v_i' is the view state after an update. r_t' is the state of r_t after the changes from v_i' have been applied. Timestamp attributes for LWW-registers are shown as T_x, where x are the first letters of the associated attribute. The L attribute is the causal length attribute in the CRR layer.



AR layer

track	year	instore
Lullaby	1989	TRUE
Lovesong	1989	TRUE
Trust	1992	FALSE

CRR layer

track	year	T_y	instore	T_i	L
Lullaby	1989	1.0	TRUE	1.0	1
Lovesong	1989	3.0	TRUE	3.0	1
Trust	1992	2.0	FALSE	2.0	1

track	year
Lullaby	1989
Lovesong	1989

track	year	T_y	L	σ	T_σ
Lullaby	1989	1.0	1	TRUE	1.0
Lovesong	1989	3.0	1	TRUE	3.0

track	year
Lullaby	1988
~~Lovesong~~	~~1989~~
Catch	1989

track	year	T_y	L	σ	T_σ
Lullaby	1988	5.1	1	TRUE	5.1
Lovesong	1989	3.0	2	TRUE	3.0
Catch	1989	7.1	1	TRUE	7.1

track	year	instore
Lullaby	1988	TRUE
Lovesong	1989	FALSE
Trust	1992	FALSE
Catch	1989	FALSE

track	year	T_y	instore	T_i	L
Lullaby	1988	5.1	TRUE	5.1	1
Lovesong	1989	3.0	TRUE	3.0	2
Trust	1992	2.0	FALSE	2.0	1
Catch	1989	7.1	FALSE	0.0	1

Figure 3.7: $V_1 = \pi_{\text{track,year}} \sigma_{\text{year}<1990} R_t$

It is worth pointing out that the CRR layer at the view has an additional LWW-register named σ. The purpose of the attribute is to track the evaluation of whether or not the tuple should be present in the view. It is dependent upon the selection criteria for the view. A tuple that fulfills the view predicate will have a $\sigma =$ TRUE. The attribute is evaluated upon every update. A tuple is in the view when its causal length is odd, and the tuple fulfills the view predicate. σ is an LWW-register as the evaluation depends on the state of attributes used in the selection criteria. The maximum timestamp among those attributes indicates when the selection was last evaluated.

Three updates have occurred at the view in figure 3.7. The updates that transitions v_1 to v'_1 are:

$$u_1 = v_1 \langle \text{Lullaby, 1989} \nearrow \text{1988} \rangle$$

$$u_2 = -v_1 \langle \text{Lovesong, 1989} \rangle$$

$$u_3 = +v_1 \langle \text{Catch, 1989} \rangle$$

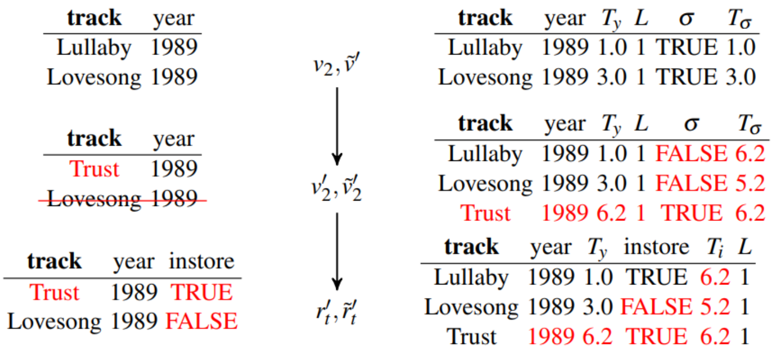
The update u_1 is a simple update to an attribute. The updated attribute is not a key and can therefore be updated directly in the CRR layer. The translation of u_1 is an update of the LWW-register for the attribute. Since the year attribute also is part of the selection condition, σ is reevaluated.

The second update, u_2, deletes a tuple from the view. Because of its one-to-one relationship to its source, we can increment its causal length (L) to remove the tuple at the view. It does not lead to an update of σ since no selection criteria were affected. Applying the update to the source state does the same update to the causal length.

The final update, u_3, inserts a tuple at the view. This tuple does not already exist, and it is therefore initialized with a causal length of 1, and σ will be true as it is in the view. Timestamps are initialized to the current time. The insertion of a tuple is a bit different from the previous updates. It is possible that an insertion does not have an equivalent tuple in the source relation. It is therefore missing a specified value for the in-store attribute, which has been projected away. Therefore, we require that any attribute projected away in a view has a default value. This default value can be NULL if no suitable alternative exists. Assigning a default value to dropped attributes is also done in [4].

3.9.2 Minimal changes to select-project views

Figure 3.8 shows a different set of well-defined translations on a view schema. The translations are based on the idea of minimal effect. Any update should produce minimal translations. The idea of minimal updates is presented in [15].



track	year
Lullaby	1989
Lovesong	1989

track	year	T_y	L	σ	T_σ
Lullaby	1989	1.0	1	TRUE	1.0
Lovesong	1989	3.0	1	TRUE	3.0

track	year
Trust	1989
~~Lovesong~~	~~1989~~

track	year	T_y	L	σ	T_σ
Lullaby	1989	1.0	1	FALSE	6.2
Lovesong	1989	3.0	1	FALSE	5.2
Trust	1989	6.2	1	TRUE	6.2

track	year	instore
Trust	1989	TRUE
Lovesong	1989	FALSE

track	year	T_y	instore	T_i	L
Lullaby	1989	1.0	TRUE	6.2	1
Lovesong	1989	3.0	FALSE	5.2	1
Trust	1989	6.2	TRUE	6.2	1

Figure 3.8: $V_2 = \pi_{\text{track,year}} \sigma_{\text{instore=TRUE}} R_t$

The view schema of figure 3.8 differs from the one in figure 3.7 as the attribute used in the selection is also projected away. This could be for security or privacy reasons. The actual selection condition cannot be sent to the view without leaking information. We instead rely on the previously mentioned σ attribute to indicate whether the tuple meets the selection criteria.

The updates shown in figure 3.8 are:

$$u_4 = -v_2\langle \text{Lovesong, 1989}\rangle$$

$$u_5 = v_2\langle \text{Lullaby} \nearrow \text{Trust, 1989}\rangle$$

Update u_4 removes the tuple from the view. There are several ways in which a tuple can be removed from a view. A contributing tuple can be removed, the attributes in the predicate can be set to non-selecting values, or a join condition can be changed to no longer hold. To preserve minimality, we change an attribute occurring in the predicate to a non-selecting value. Since the selecting attribute has been projected away, we only need to change the predicate evaluation attribute σ to false.

For update u_5, the key has been changed. Since the key has changed, it is not semantically the same row, and changing the key in any circumstance results in removing the old tuple and adding the new tuple. In this case, removing the tuple can be achieved by simply making the view condition σ false.

Although simply changing σ to false works at the view side, the update must still be applied to the source relation. Applying the updates will result in the in-store attribute being updated to a non-selecting value (false) for the removal. The insertion affects an existing tuple at the source, and its in-store value must be true.

In short, removing a tuple translates to an update that ensures it does not satisfy the predicate. An insertion must satisfy the predicate when the update is translated back.

We only consider these translations when the predicate has at least one boolean condition. The first boolean condition will be used for translations. This choice is arbitrary, as generalizing these translations to other predicate types, such as range predicates, may involve creating a framework for specifying custom removal policies.

3.9.3 Translation of updates in join views

The special thing about join views are that each tuple in the view is derived from multiple source tuples. Removing a tuple from a view can be done by removing one or more source tuples in a multi-way join.

Keeping in line with minimality, we remove tuples from at most one source relation contributing to a view. But which tuple to remove is not obvious.

For joins, a single source tuple may be repeated across multiple view tuples. Removing a repeated tuple will also remove all view tuples containing the repeated source tuple. We want a single insertion or deletion to only affect a single tuple.

The most practical way to enforce this is to restrict views to those in which the key of a source tuple is a superkey for the tuples within a view. This means that a graph of the references between source relations must form a rooted tree. We will refer to such a graph as a view-reference graph. The view-reference graph is similar to the view-dependency graph presented in [7].

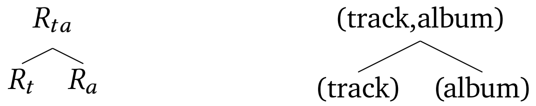


Figure 3.9: The left-hand tree shows the relationship between relations. The right-hand tree shows key relationships of the same relations.

The view-reference graph for the database schema in figure 3.5 is illustrated in figure 3.9. Tuples from the source relation at the tree's root will contain a superkey key into the view relation. In this case, the primary key (track, album) will uniquely identify a tuple within a view.

We also restrict join views to those in which the join condition is a key relationship between the relations. This is for the same reason as above. If this is not the case, then updating a single view tuple may have unintended side effects, such as adding or removing additional tuples not subject to an update.

Figure 3.10 shows a simple join between the R_a relation and the R_{ta} relation. There are only two relations where the R_{ta} relation contains a key to the R_a relation. The tree's root is the R_{ta} relation.

There is only a single update on the view:

$$u_6 = v_3\langle \text{Lovesong}, \text{Paris} \nearrow \text{Disintegration}, 4 \nearrow 7\rangle$$

The update affects the key for R_{ta} and R_a. Since R_{ta} is the root relation, an update to its key is translated into removing the old tuple and inserting the new tuple. The key for the albums relation has also changed, but since it is not the root source relation, we only ensure that the newly referenced tuple is present. No removal is done on R_a. Removal in the R_a relation might have affected other view tuples, which is undesirable.

The change is applied to the CRR-layer relations according to the update rules stated above, and no special handling is done apart from only deleting from

AR layer

track	**album**	quantity
Lullaby	Show	3
Lovesong	Paris	4
Trust	Wish	5

CRR layer

track	**album**	quantity	T_q	σ_t	T_{σ_t}	L_t	L_{ta}
Lullaby	Show	3	1.5	T	1.5	1	1
Lovesong	Paris	4	3.5	T	3.5	1	1
Trust	Wish	5	2.5	T	2.5	1	1

$v_3, \tilde{v}_3$

track	**album**	quantity
Lullaby	Show	3
Lovesong	Disintegration	7
Trust	Wish	5

track	**album**	quantity	T_q	σ_t	T_{σ_t}	L_t	L_{ta}
Lullaby	Show	3	1.5	T	1.5	1	1
Lovesong	Paris	4	3.5	T	3.5	1	2
Lovesong	Disintegration	7	7.7	T	7.7	1	1
Trust	Wish	5	2.5	T	2.5	1	1

$v'_3, \tilde{v}'_3$

album	quantity
Disintegration	7
Show	3
Galore	1
Paris	4
Wish	5

$r'_a, \tilde{r}'_a$

album	quantity	T_q	L
Disintegration	7	7.7	1
Show	3	1.5	1
Galore	1	4.5	1
Paris	4	3.5	1
Wish	5	2.5	1

track	**album**
Lullaby	Galore
Lullaby	Show
Lovesong	Galore
~~Lovesong~~	~~Paris~~
Lovesong	Disintegration
Trust	Wish

$r'_{ta}, \tilde{r}_{ta}$

track	**album**	L
Lullaby	Galore	1
Lullaby	Show	1
Lovesong	Galore	1
Lovesong	Paris	2
Lovesong	Disintegration	1
Trust	Wish	1

Figure 3.10: $V_3 = T_{ta} \bowtie \sigma_{\text{quantity}>2} R_a$

the root relation and ensuring the presence of tuples within all other relations. It is worth noting that updating an attribute referencing the key of another relation will only update the reference and ensure the existence of the newly referenced tuple. It will never delete previously referenced tuples.

4

Implementation

This chapter will present how the approach to partial replication and updatable views has been implemented. We briefly touch on our approach to software development during this project. We explain how the existing approach for an asynchronously replicated database is implemented. Finally, we present how updatable views are implemented and integrated with the existing approach.

4.1 Approach to software development

The previous solution for CRR was implemented in Python but was rewritten in Rust [20] at the start of this project. Rust was chosen due to its robust type system, which is helpful for properly modeling a domain. It makes it easier to catch mistakes when making numerous changes throughout the codebase. It also focuses on memory safety, preventing the most common memory management mistakes programmers make.

We use test-driven development to ensure the implementation behaves as expected. Since the Rust implementation is based on a previous implementation, it is trivial to implement test cases that cover the required core functionality. Some existing features have been omitted, such as network communication and flexible handling of integrity constraints. These features do not affect the implementation of partial replication.

Test cases for updatable views are based on the examples presented thus far. The implementation was first based on relational lenses but has been significantly simplified through many iterations. This means that both the implementation and the tests have evolved. The initial starting point for this project is discussed further in chapter 6.

4.2 CLI-tool

The implementation is based on the one presented in [26]. The implementation has been written in Rust as a CLI tool that provides several operations on a relational database. The tool currently only supports SQLite databases. The reason for choosing SQLite is that it is lightweight and suitable for use on small local devices[24]. The CLI tool can convert an existing database instance into one that supports asynchronous replication. This conversion process is called augmentation, referring to the fact that the conversion is non-destructive and can be reverted. The augmentation process creates the CRR layer.

The operations that are currently supported in the Rust implementation are:

- Initialize
- Pull
- Push
- Create view
- Pull view

The vigilant reader will notice the similarities to the commands in the popular Git version control system [19]. The similarity is no coincidence, as Git is a decentralized system where update conflicts are manually resolved. CRR-augmented databases can also be used in decentralized systems, but conflicts are resolved automatically. The current implementation uses a git-like synchronization model. This is illustrated in figure 4.1. We also borrow terminology such as a *remote* referring to a database at another location from the database instance considered to be *local*.

The initialize command augments an existing database instance. The augmentation process will create the underlying CRR layer and migrate existing data into the newly created CRR layer. After initialization, the database instance can

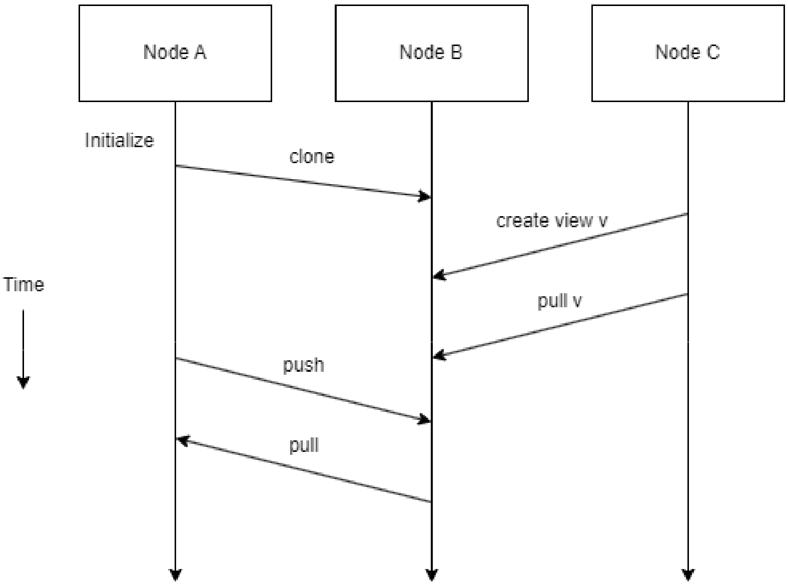


Figure 4.1: Synchronization example

function as a CRDT. The initialization procedure is idempotent. This means the database schema can be changed, and initialization can be run once more to ensure the entire database instance can be used as a CRDT.

The pull command is used to get updates from a remote database instance. The local database instance requests updates from some remote database instance. The remote node will then generate a delta state that is transmitted back to the local node. The local database instance merges this delta state into its state.

The push command is implemented by sending a delta of updates to a remote node. This impromptu sending of updates relies on previous knowledge of what updates the remote node knows of to calculate a delta state. An alternative implementation is a reverse pull, where the local node requests the remote to pull from itself.

Creating a view is done by contacting a remote node and registering a view that has been defined using SQL. This differs from database schemas, where we expect all tables to be defined upfront. The idea is to make views more dynamic. The only issue is that multiple nodes may register views with the same name. Views of the same name are currently required to have equivalent definitions.

The pull view command does almost exactly what the pull command does, but it only pulls updates for a view. This means that source database instances synchronize using the pull command. Instances that are partial replicas (view instances) synchronize using the pull view command.

4.3 Database schema

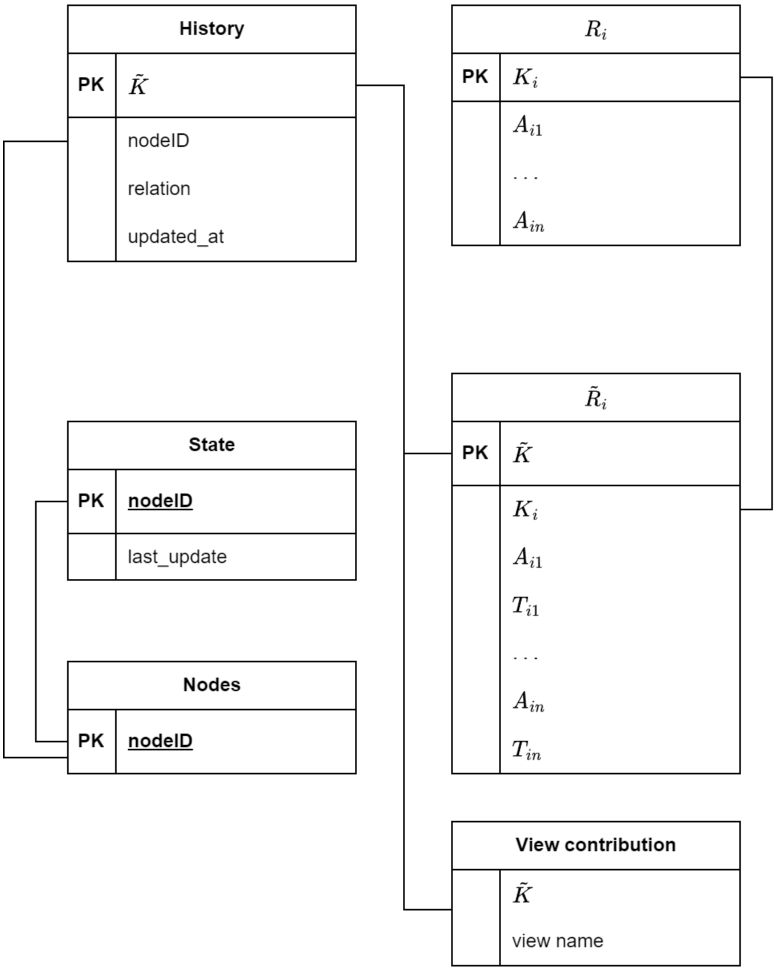


Figure 4.2: CRR-augmented database schema

The CRR layer contains several tables that store necessary metadata. The database schema for an augmented database instance is shown in figure 4.2. After augmentation, each table R_i will have a CRR-layer table $\tilde{R}_i$. The CRR layer table $\tilde{R}_i$ will be a superset of the AR layer, as it will contain the causal length and timestamps of all rows that have been present in the table.

The CRR layer needs to associate timestamps with each LWW-register. Timestamps are generated using time functionality built into the database management system. Generating timestamps is done through queries against a view. The clock view guarantees that the generated timestamps are unique and increase monotonically.

The history table tracks all updates done at what database instances. The updates are changes to individual tuples. Therefore, each table $\tilde{R}$ has a surrogate key $\tilde{K}$ that the history table uses to reference the tuples. Not having such a primary key would involve having separate history tables for all application-layer tables, which would likely be computationally inefficient. There may be multiple surrogate keys for a single row. This happens when a row identified by the same primary key is concurrently inserted at different sites. However, a single surrogate key may never exist for multiple distinct rows. The history table is used during delta generation.

We need to know what updates are already known at other sites to generate deltas. We, therefore, track the state of all other database instances using a table. The state table is a vector clock transmitted between instances to generate correct deltas. The table is used with the nodes table that tracks all other known database instances.

A view is created as a regular SQL view in a database instance. A view contribution table is used to track what rows have ever been replicated in a given view. It ensures that nodes replicating views synchronize correctly. It will be explained further when discussing the synchronization of views.

For database instances using regular replication, the entire database schema is replicated. There are also separate tables for AR layer tables. When replicating a view, only the CRR tables are replicated. The view is therefore calculated upon access using the replicated CRR tables. The replicated CRR tables are only the ones that contribute to a view, and only the contributing rows are replicated. This does mean that merging is considerably faster, but it might affect query performance.

4.4 Timestamps

Timestamps have been mentioned at various points throughout the thesis, but little attention has been paid to how timestamps are handled. Several aspects must be considered when dealing with timestamps. The first thing to realize is that it is impossible to have a globally synchronized clock in a distributed system [27, p. 3]. It is impossible to derive exact causal relationships between events. Computer clocks are also susceptible to clock drift which means they drift further away from the actual true time value.

It might seem that physical timestamps are unsuitable as they cannot guarantee ordering. We do not need to ensure exact physical times, only that timestamps for individual attributes have a total order. Recall that for LWW-registers, the timestamp component of the register must ensure a total order among all timestamps. Suppose an LWW-register is to be merged, where two concurrent updates receive the same timestamp. Which update is the last one is ambiguous and cannot be determined. The current system must have total ordering for timestamps since it relies on LWW-registers.

The second requirement for timestamps comes from delta generation. Delta generation works by transmitting all changes that might not be known between pairs of nodes. There is one problem: computers can do time travel. If a node performs a clock synchronization, its clock can be set back to an earlier time if the clock has drifted forward in time. If a node suddenly issues timestamps from an earlier time, the delta generation procedure would fall apart as we could not know all updates after some point has been received. The result is that timestamps generated at a node must be strictly monotonic.

The implementation solves both these requirements by combining two techniques. Strictly monotonic timestamp generation is implemented using something akin to a hybrid-logical clock [17]. A physical timestamp is generated and compared against a previously generated timestamp. A second logical component is updated if the new timestamp value is less than or equal to the previous value. The logical component can only increase. Clocks are compared lexicographically by first comparing the physical portion and then using the logical portion.

Total ordering is done by introducing randomness. Incrementing the previously mentioned logical component is done by using a random offset. This is used to emulate nanosecond scale timestamps. The entropy of the randomly generated number must be sufficiently high to ensure there won't practically be collisions.

An alternative would be to add a component to timestamps that references the node where it was generated. This could enable us to order timestamps based on the node doing the update. This could be more efficient if node IDs were small.

The current implementation does not consider clock skew as it is not essential to ensure the correctness of the system. We assume clock synchronization is handled by some other mechanism, such as using the NTP protocol [1].

4.5 Handling local updates with triggers

Any update to the AR layer must be translated into updates on the underlying CRR layer. The current implementation uses database triggers to propagate the updates. Database triggers are callbacks that invoke actions to be executed when an event is triggered [23, p. 206]. The actions are defined as SQL statements. Updates on tables trigger events that execute SQL statements as a side-effect of the update.

Every update to the AR table triggers an update to the underlying CRR table, which then triggers an update to the history table. The defined SQL statements extract and propagate the updated information to the CRR layer. The updates happen in a single local transaction such that update operations are atomic.

The triggers are created as a part of the augmentation process, and each AR table gets its own insertion, deletion, and update triggers for handling all possible update operations. Recall that the AR tables are only views on top of the underlying CRR tables. The AR tables are materialized views of the corresponding CRR layer table for fully-replicated database instances. This materialized view means that queries may be executed faster, but it uses a lot of storage capacity.

The current implementation of the replication of views uses a different approach. The application layer view is never materialized, and as a consequence, all updates only apply updates to the CRR table as well as the history table.

Updating views is also done using triggers that operate on the updated tuples in a view. How triggers are used when updating views differs from how triggers handle updates in regular replicas. The following subsections will therefore highlight the differences in how updates are handled between regular replication and view replication.

4.5.1 Insertion trigger

Inserting new rows into a normal table is relatively straightforward. The insertion is first applied to the AR layer table. If the tuple already existed, this would trigger an SQLite error. The insertion will trigger an update on the corresponding CRR table. The trigger will check whether a row with the same primary key exists in the CRR table. If it does, it increments its causal length and updates all updated non-key attributes. If not, a new row is inserted, and its causal length is set to 1.

The insertion triggers for views are quite similar, with a notable difference. View instances only replicate the source relations that contribute to the view. The view definition is therefore used to join tables and discard any removed tuples and tuples that no longer fulfill the predicate. Since the view is an actual SQL view, updates can be handled using "INSTEAD OF"-triggers [25] that operate on individual tuples in a view.

Each table contributing to a view needs triggers for handling updates. The insertion triggers inspect the added row and determine if the row contains a primary key for the contributing CRR tables associated with the trigger. The actual trigger logic is the same as for insertions on normal tables. Three insertion triggers will be created if a view has three contributing tables.

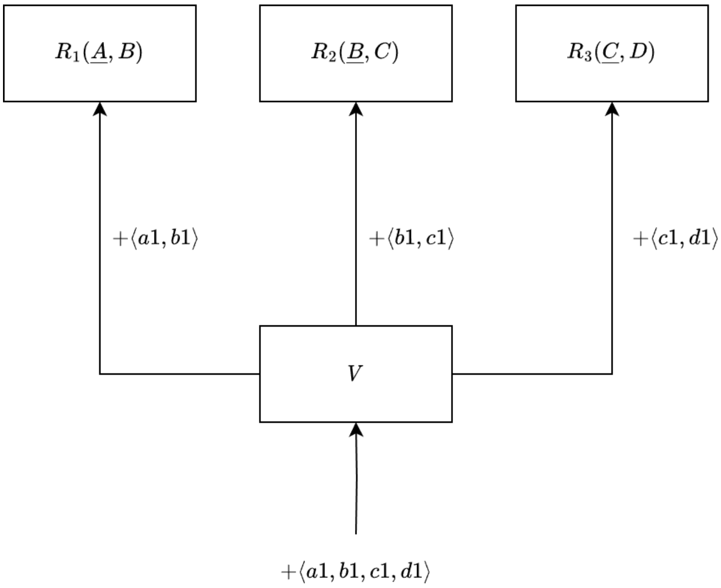


Figure 4.3: Insertion trigger

Figure 4.3 shows an example where the tuple $\langle a1, b1, c1, d1\rangle$ is inserted against a view V that is defined as a join of R_1, R_2, and R_3. The insertion of the original tuple will trigger insertions on all contributing tables. If a tuple already exists, the insertion is ignored.

4.5.2 Deletion trigger

Deletion of a row from a regular table is done by deleting it from the AR layer table. This triggers the deletion trigger, which finds the row in the CRR layer table and ensures its causal length is even.

Deleting a row from a view is quite different. Recall that there are multiple ways of removing a tuple from a view. The current implementation is opinionated because it only allows deletions from a single table. This means there will only ever be one deletion trigger for each view, which only deletes from a single contributing table. This contrasts insertions on a view, where each source table has an insertion trigger.

The relationships between the tables determine the relation to delete from. This relationship is captured in the previously mentioned view-reference graph. Kahn's algorithm[14] is used to topologically sort a graph where sets of keys are the nodes and edges are the reference relationships between the keys. The first element in the topological sort will be a table to which no other table has a reference (in-degree of 0). This is the root relation of the view-reference graph.

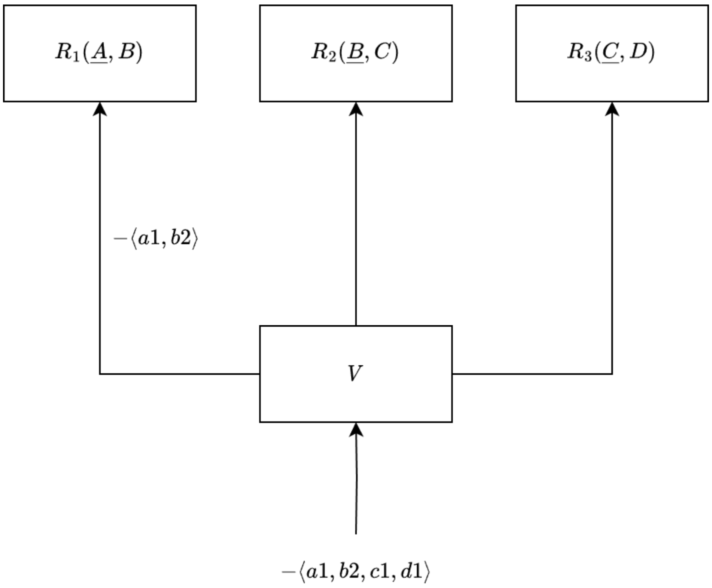


Figure 4.4: Delete trigger

Figure 4.4 shows the same view as in figure 4.3. The deletion of the tuple $\langle a1, b2, c1, d1\rangle$ leads to the removal of $\langle a1, b2\rangle$ from R_1. This is because R_1 is the root of the reference graph, as shown in figure 4.5.

4.5.3 Update trigger

Updates on regular AR tables trigger updates in the update trigger associated with the table. As previously mentioned, an update to non-key attributes applies the updates to the respective LWW-registers of the CRR layer table. If there is

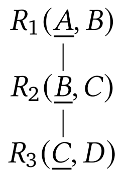


Figure 4.5: View-reference graph for figures 4.3, 4.4, 4.6, and 4.7

a change to the key attributes for a table, this is regarded as an insertion and a deletion. Recall that this is because different key attributes refer to different rows.

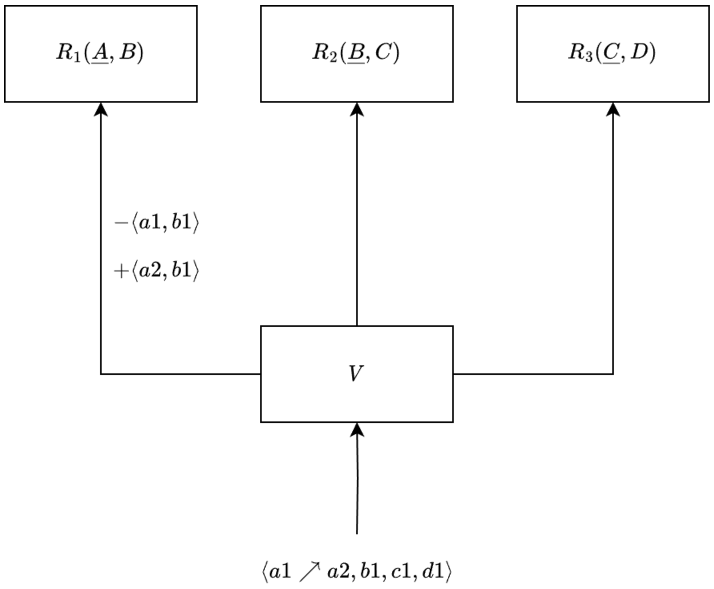

$R_1(\underline{A}, B)$
$R_2(\underline{B}, C)$
$R_3(\underline{C}, D)$
$-\langle a1, b1 \rangle$
$+\langle a2, b1 \rangle$
V
$\langle a1 \nearrow a2, b1, c1, d1 \rangle$

Figure 4.6: Update on the key of root table

Updates on views are handled similarly, but there is one important difference. Changing the key of a contributing relation is handled differently depending on whether the table is the root table in the view-reference graph. Updates to the key of the root relation are the same as a deletion and a subsequent insertion, the same as for the normal case. This is shown in figure 4.6, where the A attribute is updated.

However, if the updated attributes are part of a key used in a join condition, the referencing table will update its reference. The referenced table must therefore ensure that the newly referenced row exists by triggering an insertion, which does nothing if the row already exists. If an update were to delete and insert rows in a referenced table, this could remove additional tuples in the view. This is shown in figure 4.7, where the B attribute is updated.

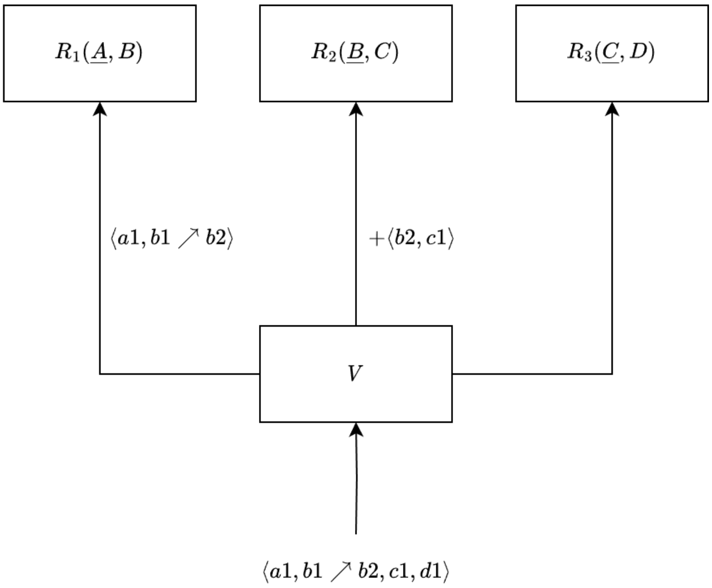

$R_1(\underline{A}, B)$
$R_2(\underline{B}, C)$
$R_3(\underline{C}, D)$
$\langle a1, b1 \nearrow b2 \rangle$
$+\langle b2, c1 \rangle$
V
$\langle a1, b1 \nearrow b2, c1, d1 \rangle$

Figure 4.7: Update on the key of non-root table

In summary, insertion will only insert one row into the view. Deletion of a row will only delete a single row. Updates not filtered by predicates will not have the side effect of adding or removing rows. Updates to non-key attributes may update multiple rows.

4.6 Synchronization

Instances using regular replication and instances only using view replication must be able to synchronize without issues. The approach to synchronization is fundamentally the same for both types of replication. There are some subtle differences in how synchronization is solved, however.

Synchronization is done in several steps. A node sends its vector clock to another node. The remote node then uses the vector clock to calculate a delta, which contains precisely the updates not present at the node when the vector clock was transmitted. The delta is a small SQLite database containing the updates identified when calculating a delta. The instance is shipped back to the local node, where all the updates are merged into the local state according to the merge procedure. The synchronization procedure is shown in figure 4.8, and the high-level process is the same for both regular and view replication.

There are subtle differences between the two approaches to replication. The

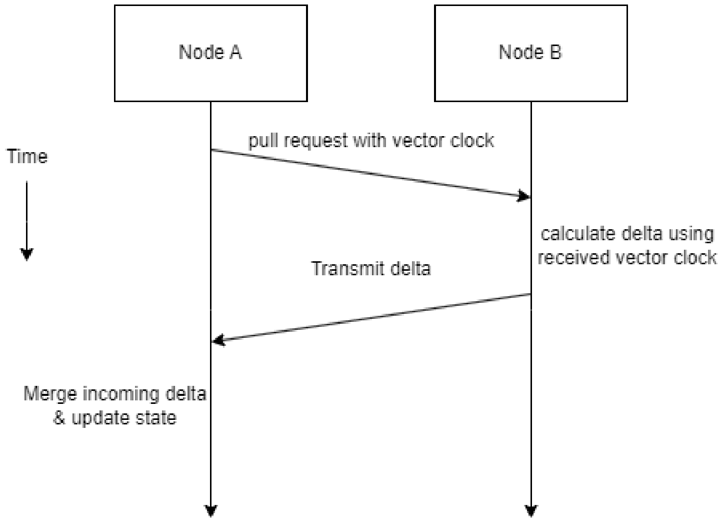


Figure 4.8: Synchronization procedure

first significant difference is how calculating delta-states is done. This is because delta generation should ideally only produce deltas with updates that affect the view in question. The other significant difference is how views deal with projections concerning security and privacy.

4.6.1 Calculating deltas

A pull-synchronization request will contain the vector clock of the initiating node, which tells the source node of the updates it already knows about. The delta state can then be generated by joining the CRR layer table against the history table and only selecting rows unknown to the other node based on the received vector clock. This is done for all AR layer tables in the database instance.

Calculating deltas for nodes only replicating a view is a bit different. In that case, we only want to send a delta containing all the rows in the view. A delta can then be calculated by doing the same query as regular replication and checking whether the row is currently a part of the view. There is one problem with this approach. After being replicated to a view, a row may be removed at a regular replica. Subsequent delta calculations will fail to send updates for the removed row. This will happen for all updates that remove rows from a view.

We must track which rows are or have ever been part of the view to fix it. This is where the view-contribution table comes into play. It tracks the rows that are, or have ever been, part of a view. The view contribution table can be merged between sites to ensure that removals can be propagated from any node to another. This method transmits the minimal information needed for proper synchronization.

A different approach to calculating deltas would be sending all the updated rows regardless. The view site can discard the information that it does not already store. This means we would need to transmit more information, but it is likely still efficient due to the already existing delta implementation. Recall that we only generate deltas for tables that are sources for the views regardless.

4.6.2 Push-based synchronization

We have only considered pull-based synchronization so far. This is because the implementation of push is not significantly different. An efficient push-based synchronization scheme could be supported by keeping the received vector clock from the other side and using it for subsequent pushes. This means that pushes could contain more rows than needed. Signaling another node to do a pull would be easier and might be more efficient. The vector clock at the pulling node should be more up-to-date. It will involve sending an additional message which increases latency. Which approach is better is likely dependent on usage.

4.6.3 Effect of projection on view definitions

A fundamental difference between regular and view replication is that attributes can be projected away. This can both reduce the storage required, but it may have the additional purpose of limiting what users of a view can learn. This may be for security and/or privacy reasons. We have already discussed that most translations on a view take place at the node replicating the view using triggers, but there are some exceptions. The exceptions are precisely those where we must deal with projections.

The most obvious difference is when there are insertions of new rows into a view instance that does not already exist at a source node. This means that a sensible value must be chosen for all the attributes that were projected away. This is done using a specified default value that must be defined on all tables contributing to a view. Assigning the default value is done when updates are sent to a source node and not at the view instance, as in most other cases. We also give the default value a timestamp of 0 (zero) to ensure that any direct update to the attribute takes precedence. Dropped attributes are always LWW-registers.

Another problem occurs when attributes that are projected away are also part of the predicate for the view. This means that selection information is discarded at the view instance. This is handled by introducing the σ attribute, which determines whether a row satisfies the predicate. Since attributes are projected away, the view definition used at the view instance must be changed. The predicate in the view definition must be replaced by a standardized predicate, which excludes information about the selection criteria. For example, it would not be fitting if a view instance knew that all rows in the view were of people from a particular region if it could infer this from seeing the predicate in the view definition.

Since we must transform views with selection criteria on attributes that are projected away, we must apply a transformation to the deltas generated for views. We must also apply the inverse transformation when deltas are received from views. All such transformations take place at source database instances and not at view instances.

4.7 Integrity constraints

Concurrent updates on relational databases do pose a particular challenge. A database schema may define several integrity constraints that constrain the set of possible states in which the database could be. By allowing concurrent updates, we can be in a situation where integrity constraints are violated.

Violations are detected upon synchronization by capturing errors from the SQLite database management system. The errors are inspected, and appropriate deltas are applied. A node may add a row referencing a row concurrently removed by someone else. This results in a foreign key violation. The row which is referenced is then brought back. The reasoning is that the referenced row must have existed at some point and that deleting it is more harmful than reinserting the referenced row. Deciding whether to delete or reinsert could be done based on the time ordering of the updates.

There are also unique constraints that concurrent insertions may violate. The way it is resolved is that the row which was inserted first will be kept while the other is removed. Otherwise, any later write would effectively overwrite any previous row, which might surprise whoever did the insertion first.

Regular replication enforces and handles violations of defined integrity constraints. Replication of views does not fully enforce the defined integrity constraints. There are a couple of reasons why this is the case. A view is an incomplete window into the entire database state. This means that it is not possible to enforce all integrity constraints.

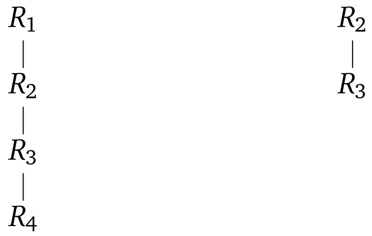


Figure 4.9: Left-hand side shows the database-reference graph. The right-hand side shows the view-reference graph.

Consider a database schema with four tables: R_1, R_2, R_3, and R_4. Figure 4.9 shows the database- and view-reference graphs. Suppose we define a view V where R_2 and R_3 are contributing tables. We then delete a row from R_2. There could be a reference from a row in R_1 to the deleted row in R_2, but this violation cannot be handled in a view instance because there is no way to know whether a reference exists. This means that resolution must occur at a node that knows about the state of table R_1.

There is a similar issue with references from R_3 to R_4. The state of R_4 is unknown at the view instance. It is impossible to know whether any rows in R_3 reference an existing row in R_4. Taking a step back, we realize there is likely little reason to have leaf tables in the view-reference graph that contain outgoing references. Such views should either include the referenced table or the reference should be projected away and given a suitable default value.

4.8 Additional considerations

There are additional considerations that must be considered when implementing updatable views. First, the view must respect the functional dependencies expressed through the key relationships. Recall certain contributing rows may be repeated in a view. An update to a non-key attribute of such a row must change all the repeated instances of that attribute. Changing a row currently affects only the contributing rows, making it easy to ensure this requirement is upheld. It does mean that certain updates to non-key attributes can affect multiple rows with a single update. This is desirable but perhaps confusing for application developers.

5

Evaluation

This chapter will present an evaluation of our approach and implementation. We present scenarios to assess the usability of updatable database views. It considers user-centric views and how update operations are applied to view definitions and the semantics of particular views. We then move on to an experiment that aims to demonstrate the effects of partial replication on storage, computation, and communication.

5.1 Applicability

The scenarios are borrowed from [7] as these are good practical examples. We discuss how our approach to updatable views can fulfill the requirements for each scenario. Figure 5.1 shows the application database schema for the scenarios.

5.1.1 Personnel management

The first use case is one in which personnel information is needed, and updates are used to manage personnel information. The identifier, age, address, designation, and skills of each employee information required for personnel management. In addition, the following updates must be allowed:

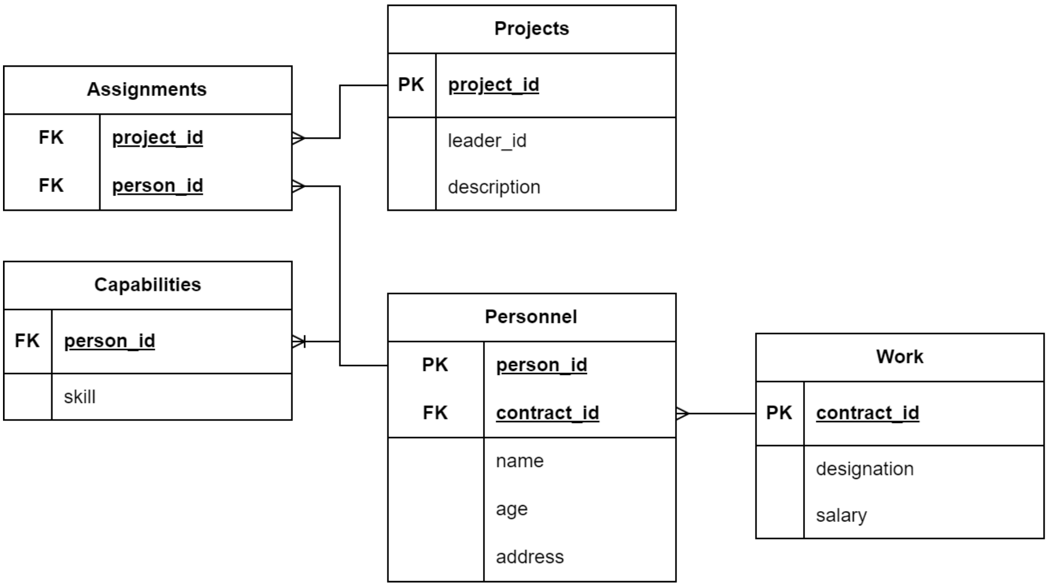


Figure 5.1: Application database schema used for evaluation

- Hire an employee
- Fire an employee
- Alter personal information about an employee
- Add a skill for an employee

On the first try, one might be tempted to create a view that captures all the information required to be in the view. This can be done using the SQL definition in listing 5.1.

Listing 5.1: First attempt at a personnel view

```
CREATE VIEW personnel AS
SELECT name, id, age, address, designation, skill
FROM employees
JOIN work
JOIN capabilities
```

This results in the view-reference graph shown in figure 5.2. It would be impossible to hire or fire an employee using the current definition of a view. This is because we cannot remove from the employee table using our approach to updating views. Even though the view is a rooted tree, deletions will only affect an employee's capabilities. This means we could add a new skill and change personal information, but we cannot hire or fire employees. The employee information will also be repeated for every skill, which is undesirable.

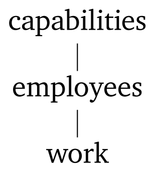


Figure 5.2: View-reference graph for the personnel view

The paper from which these examples are from requires that a single view must be able to fulfill these requirements. We take a more pragmatic approach. The views that extract required information must not necessarily also be updatable. Splitting a view into several views is possible, with each view exhibiting different update semantics. We can split the view into two views as shown in listing 5.2.

Listing 5.2: Personnel management using two views

```
CREATE VIEW personnel AS
SELECT name, id, age, address, designation
FROM employees
JOIN work

CREATE VIEW capabilities AS
SELECT id, skill
FROM capabilities
JOIN employees
```

employees
work
capabilities
employees

Figure 5.3: The reference graphs for the personnel and capabilities views

Figure 5.3 shows the new view-reference graphs. The query and updates that fulfill the requirements for personnel management using view replication are shown in listing 5.3. Recall that salary must be given a default value as it has been projected away. Creating two separate views is also not a problem, as it will use the same underlying CRR tables and will not waste storage capacity.

Listing 5.3: Updates to the views for personnel management

```
-- Hire an employee
```

```
INSERT INTO personnel(id, name, age, address, designation) VALUES (..., ...);

-- Fire an employee
DELETE FROM personnel WHERE id = ...;

-- Update information about an employee
UPDATE personnel SET address = "...", name = "..." WHERE id = "...";

-- Add a new skill to an employee
INSERT INTO capabilities(id, skill) VALUES (..., ...);

-- Combine the views to extract all necessary information
SELECT id, name, skill FROM personnel JOIN capabilities
```

5.1.2 Financial management

The second scenario involves a finance manager at a company who is interested in seeing the salary of all employees. There is only one requirement related to updates on the view: the finance manager should be able to update an employee's salary. The view definition and the update that fulfills the update requirement are shown in listing 5.4.

Listing 5.4: View for financial management

```
CREATE VIEW financial AS
SELECT id, name, salary
FROM employees
NATURAL JOIN work

UPDATE financial SET salary = ... WHERE id = ...;
```

The view definition does highlight a flaw with the current design. There is no way to restrict the update capabilities using views. The manager can hire and fire new employees by issuing insertions and deletions on the view.

5.1.3 Project management

The last scenario is one where a particular project manager is interested in querying information about the skills of the employees assigned to projects that the project manager is in charge of. Note that this is an example of a user-centric view since the view is for a particular user, a project leader.

The view should allow the following two updates to be performed:

- The project leader should be able to assign employees to a project, provided the project leader is responsible for it.
- Remove an employee from a project for which the project leader is responsible.

This problem is hard to solve with a single view, just as in the first case. It is hard to solve because it is challenging to reassign employees and provide information about every employee's skills on a project. If a view is created to retrieve all the required information, the view-reference graph would no longer be a rooted tree. It is not a rooted tree as neither the capabilities nor the assignment table has any inbound references, and both reference the employee relation. Updates would therefore be undefined according to our current approach.

The solution is, just as before, to split the view into two separate views where updates are well-defined. The new view definitions and the required update capabilities are shown in listing 5.5. The assignment view can be used to reassign employees using insertion, deletion, and updates, while the capabilities view is used to get the skills of every employee. This does not use more storage than necessary, as their common contributing tables will use the same underlying CRR tables. The views do give the project manager more update capabilities than needed. No authorization framework is implemented that can enforce that only certain users can access and update a view.

Listing 5.5: Project management views, as well as required updates

```sql
-- view of assignments
CREATE VIEW assignments AS
SELECT person_id, project_id
FROM assignments
JOIN employees
JOIN projects
WHERE leader_id = ...;

-- non-updatable view for information extraction
CREATE VIEW capabilities AS
SELECT person_id, skill
FROM capabilities
JOIN assignments
JOIN employees
JOIN projects
WHERE leader_id = ...;

-- Assign an employee using his id to a project
```

```
INSERT INTO assignments(person_id, project_id) VALUES (..., ...);


-- Remove assignment using a person_id
DELETE FROM assignments WHERE person_id = ...;
```

5.2 Experiments

An experiment has been conducted to evaluate the viability of partial replication using views. The experiment is designed to compare partial-state replication against full-state replication. The three aspects of interest are storage, computation, and communication. The experiment is similar to the one in [12]. There are two relations defined for the experiment. These are $R_1(K_1, A, B, K_2)$ and $R_2(K_2, C, D)$. K_1 and K_2 are the primary keys for R_1 and R_2 respectively. There is a foreign key constraint on R_1 referencing R_2 with the key K_2. K_1, B, K_2 are all integers, whereas A and C are 200-byte randomly generated strings. Before each run, R_1 is initialized with N values, while R_2 is initialized with $N/10$ rows.

K_1 is sequentially assigned each row within the range $[1..N]$. K_2 is a value in the range $[1..N/10]$ and is sequentially assigned in R_2. The K_2 reference in R_1 is evenly distributed such that ten total references from R_1 to every row in R_2. B is a value within the range $[1..100]$.

A view $V(K_1, A, B, K_2, D)$ is defined as $v = (\sigma_{B=3} r_1) \bowtie (\pi_{K_2,D} r_2)$. The key K_2 will be the superkey for the view when considering the view-dependency graph. Since B is in the $[1..100]$ range, selecting a single B value will mean the view will contain $N/100$ rows. In every run of the experiment, the D attribute of 10 rows is changed, meaning there is a constant number of updates regardless of the value chosen for N. We run the experiment multiple times with N in the range between $[20000..120000]$ rows.

The following hardware was used for the experiment:

CPU: Intel I7-8700K

RAM: 32 Gb of capacity with a clock speed of 2666 mHz

Storage: HDD with a capacity of 8 Tb and a rated speed of 6 Gb/s

5.2.1 Computation overhead

We measure the time to perform operations at a replica and compare regular replication to view replication. The most interesting operations are the delta-generation and merging procedures. We set up both a regular replica and a view replica that have been initialized to contain data as described in the previous section. We then apply ten updates to both the regular and view replicas. We then measure the time it takes to generate and merge incoming deltas.

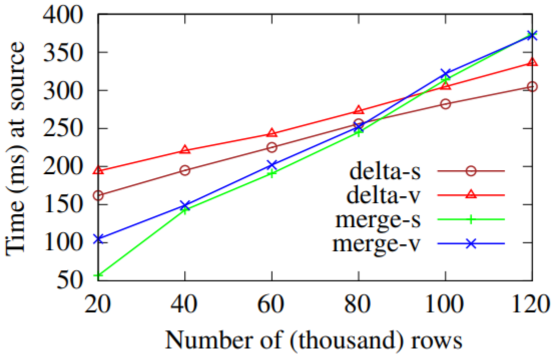


Figure 5.4: Time for delta-generation and merging at a source instance.

Figure 5.4 shows the time to generate deltas at a regular replica for another regular replica (delta-s) and a view (delta-v). The time taken to generate a delta for a source replica is slightly higher than for another view. This is because, in addition to identifying changed tuples, it must also ensure that the tuple is actually in the view instance. Checking if a row is present in a view is a constant cost that increases linearly with database size.

Merging incoming deltas at a source replica is also an operation that increases linearly with database size, and it is more expensive than delta-generation. This is shown with merge-s and merge-v. The higher growth rate is likely because the merging procedure is done in a high-level programming language. The delta generation is done entirely with SQL statements and can therefore be optimized by SQLite.

Figure 5.5 shows the time it takes to generate deltas and to merge at a view replica. The delta-generation and merging procedures are close to constant cost. The reason is that views are so small that the data size does not impact the time taken. The time would likely increase linearly when the view size increases, but this has not been verified.

What is not shown in the figures is that many changes in a source replica

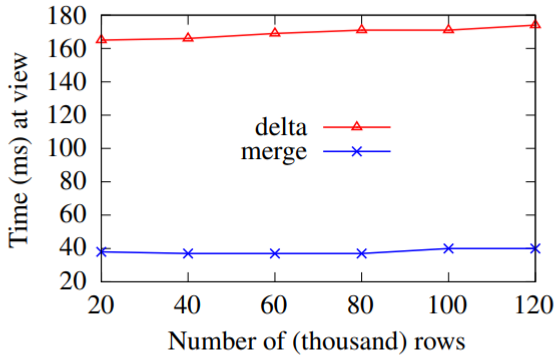


Figure 5.5: Time for delta-generation and merging at a view instance.

will not necessarily be replicated to views. Suppose there are 100 changes at a source replica, but only ten changes should be replicated to the view. The delta-generation and merge performance between source replicas would likely be much higher.

5.2.2 Storage overhead

We measured overhead for CRR-augmented databases for both regular and view replicas. The storage space overhead was close to three times the size of the data in the AR layer. The size of the view replica was about $1/100$ of the size of the regular replica. Recall that the view replicates $N/100$ the size of the view. The view is comparatively smaller as it does not store AR tables, and an additional attribute is projected away.

The amount of data within rows determines the exact storage space overhead. There is a constant amount of metadata per row in a table for CRR-augmented databases. This means that if the rows store more data, the storage overhead of CRR augmentation decreases. If the 200-byte strings were removed from the schemas in the experiment, the storage overhead would likely be more significant.

5.2.3 Communication overhead

The current implementation only runs experiments on a single machine. All experiments have been done locally, so it has been impossible to measure the communication overhead directly. It is possible to approximate the communication overhead by basing it on the amount of information that must be transmitted during synchronization. The information to transmit is a vector clock and a generated delta state. The transmission time of a particular message can then be calculated as $t = \frac{m}{b}$, where t is the transmission time, m is the message size, and b is the network bandwidth. For simplicity, we assume that the network bandwidth is fixed and there is no congestion. The network overhead will therefore grow linearly with message size, which will grow linearly with the size of a delta.

Another point to consider is the communication overhead in decentralized scenarios. Nodes can freely synchronize with their neighbors. This means that view replicas may synchronize with other view instances or regular replicas. By limiting the amount of information that a replica has, through view replication, we inhibit the ability to communicate information that may be of interest to other nodes. Consider a network partitioning scenario where nodes have a small set of neighbors. Nodes may need to communicate with more than one node if they replicate different but overlapping subsets of data. This may introduce higher overall communication costs in the system.

5.2.4 Time to consistency

Limited transitive information sharing will likely impact the time for any given node to reach a consistent state. If a node with information of interest is unavailable, there is less chance that other peers will have information of interest, as they may replicate views of different domains. This means that the node must wait until it comes into contact with another node that contains the information of interest. Limiting the amount of information a node has is excellent for security and privacy, but the impact on the dissemination of information may significantly reduce the usability of the approach.

5.2.5 Unfair comparison

It is essential to state that the view does not replicate the AR layer tables. This means that views do not need to ensure that the AR layer is synchronized with the CRR layer when merging. This general cost reduction may mean that comparisons between regular replication and view replication are unfair.

A few microbenchmarks that measure the insertion, update, and deletion performance on regular replicas compared to view replicas have been conducted. The time taken at view replicas is significantly lower than regular replicas since the AR layer does not need to be synchronized. The query performance is likely lower, but this has not been verified. The time to update the AR layer is a constant cost per row, and it is unlikely to affect the conclusions drawn from the experiment.

/6

Related work

This chapter will present related work that is closely tied to what is presented in this thesis. We will first explain the work that inspired our approach. We have strayed far from the initial starting point, and we therefore also present the work we ended up being closest to. Finally, we present how we differentiate our work from other related work.

6.1 The starting point

Database views were quickly identified as a mechanism that could support partial replication. An approach for updatable views based on relational lenses was presented in [4]. It formed the basis for the work presented in this thesis.

Their approach involved bidirectional lenses performing regular relational query operations in the get-direction, while the put-direction would apply an updated view state back onto a source state. The lenses are composable so that many views can be expressed. The lenses include selection, projection, and join lenses. Their focus was on well-behaved lenses. Well-behaved lenses ensure all updates are applied to a source that induces the same updated view state. They provide algorithms for the different lenses and typing rules that restrict what views could be expressed.

Our initial idea was to take their work on relational lenses and apply it in the context of CRR-augmented databases. Views would be expressed as SQL, and they would be parsed into relational lenses. Figure 6.1 shows the relational operations used to create a view. This would be interpreted as three instances of a selection lens, two instances of a join lens, and a single instance of a projection lens. The bottom-up evaluation would be the get-direction for the lenses. Traversing in a top-down fashion and propagating updates downward is equivalent to the put direction of the lenses.

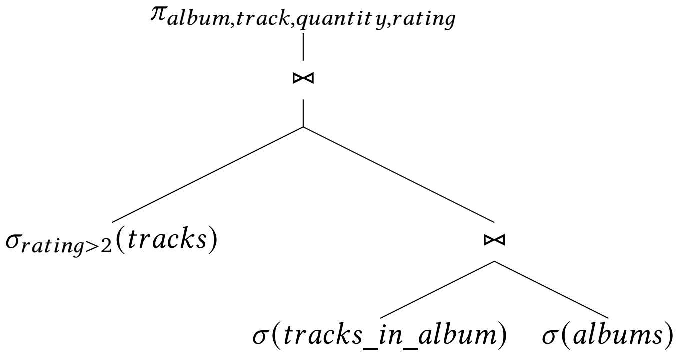

$\pi_{album,track,quantity,rating}$
$\bowtie$
$\sigma_{rating>2}(tracks)$
$\bowtie$
$\sigma(tracks\_in\_album)$
$\sigma(albums)$

Figure 6.1: An expression tree of relational operations for a view

Figure 6.2 shows their proposed algorithm for a selection lens. The get operation on relation M is simply a selection with a predicate P. The put operation involves doing a relational merge using the functional dependencies in the set F to determine the ordering of updates on attributes. They require that functional dependency form a tree. The tree of functional dependencies inspired our definition of the view-reference graph. Updates are propagated from the root of the functional dependency graph and down to the leaves. The relational merge may include updates that violate the rule that a lens should be well-behaved. The violating updates are captured in $N_{\#}$ and removed from the merged result.

$$\begin{aligned} get(M) &= \sigma_P(M) \\ put(M, N) &= M_0 - N_{\#} \\ \text{where } M_0 &= merge_F(\sigma_{\neg P}, N) \\ N_{\#} &= \sigma_P(M_0) - N \end{aligned}$$

Figure 6.2: Algorithm for a select lens

Applying this in the context of a CRR-augmented database meant making some adjustments. Each view tuple could only be produced from its contributing tuples. This was to ensure that metadata was preserved correctly at view instances. This would ensure that it could be locally updated while guaranteeing eventual consistency. In addition, the relational merge would have to be replaced by the CRDT merge operation to guarantee eventual consistency. The merge operation on two CRDT relations is a bijective function. Any tuple in a relation instance maps precisely to a row in another relation instance with the same schema. This means that M_0 would contain all rows of the view N, and consequently, $N_\#$ is always empty.

The change to the merging function also changes how the join-lens works. Figure 6.3 shows an algorithm for a join-lens. The join lens works by removing tuples from the left relation. It also removes violating updates using the temporary relation L. But since the CRDT merge is bijective, L will always be empty.

$$\begin{aligned}
get(M) &= M \bowtie N \\
put(M,N) &= (M', N') \\
\text{where } M_0 &= merge_F(M, \pi_U(O)) \\
N' &= merge_G(N, \pi_V(O)) \\
L &= (M_0 \bowtie N') - O \\
M' &= M_0 - \pi_U(L)
\end{aligned}$$

Figure 6.3: Algorithm for a join lens

The projection lens is shown in figure 6.4. It inserts the default value for the dropped attribute before doing a relational merge that overwrites the value if another attribute already determines it through a functional dependency. This is handled differently for CRRs, as we can set a timestamp of 0 to default values. A deliberate update to an attribute will always overwrite the default.

$$\begin{aligned}
get(M) &= \pi_{U-A}(M) \\
put(M,N) &= revise_{X \to A}(M', M) \\
\text{where } M' &= N \bowtie \{\{A = a\}\}
\end{aligned}$$

Figure 6.4: Algorithm for a project lens

After making the necessary changes, relational lenses were found to be unnecessary. The lenses became obsolete because of the significant difference between the merge operations since a direct CRDT merge of join-irreducible states could produce the expected results.

We decided to only delete a single tuple in the case of multi-way joins, which is different from the join-lens. We also restrict the views that can be updated, which are presented as typing rules in [4]. We have not included the typing rules from their work. This is because we take a more straightforward approach to what views are updatable.

6.2 Most similar work

Our final implementation is closer to the work of [15] than the work on relational lenses. They present several computational algorithms that translate updates unambiguously and apply them to a source database instance.

They propose algorithms for translating insertions, deletions, and replacements on views composed of selection, projection, and joins. We arrived at almost identical algorithms by starting with relational lenses and modifying the approach to suit our needs. The only difference is that our translations affect the CRR layer and must modify the underlying CRDT structures. The translations are the same when only considering the AR layer.

They also presented similar restrictions on the kinds of views that are updatable. The references between relations should form a rooted tree, and that key-attributes cannot be projected away. These are the exact requirements that were found to be suitable for our approach.

Most of the work in this thesis was done while being unaware of this early work. It is interesting how one arrives at an approach similar to one presented decades earlier.

Their work did inspire some changes to our existing approach. They present the idea of minimal updates. Updates should apply minimal changes to a source database to achieve the desired effect. For example, they argued that deletions from select views change the predicate to a non-selecting value if possible. This led to a change in our approach regarding deletions. We further restrict the idea of minimal deletion to selections where the predicate involves at least one boolean attribute.

6.3 Related work on updatable views

There has been a wide range of research on updatable views. This research has focused on translating updates back to one specific source instance. Our work guarantees that translations are applied in the same manner, independent of the state of the source instance. Our approach can therefore guarantee strong eventual consistency, which earlier approaches to updatable views could not.

Earlier approaches relied on applying translations to a specific source instance. Recall that our approach defines the translation of updates in terms of the join-irreducible states and is, therefore, independent of the target state where the delta is merged.

There has been a lot of research on incremental view maintenance. This primarily focuses on how tuples are derived and when tuples should be added and removed from the view. Maintaining materialized views is irrelevant, as we only use the concept of views. Views are recomputed upon access. The authors of [12] present an approach to incremental updates of views using relational lenses. Their work involves tracking changes and applying them to a source database instance. Our work relies on synchronization using join-irreducible states of a CRDT. It is an efficient approach for disseminating updates and therefore does not rely on earlier work on incremental maintenance on updatable views.

/7 Discussion

This chapter will discuss whether or not the presented approach satisfies the requirement for replicating data subsets. The possibilities and limitations of our approach are also highlighted. Several aspects of the current approach can be improved or investigated further. These aspects include security concerns, understandability, and the viability of the implementation.

7.1 Requirements

The requirements for partial replication were first presented in section 3.4. How well these requirements have been met is discussed in the following sections.

7.1.1 User-centric

The first requirement is that the replication of data subsets should be user-centric. In essence, this means that views need not be globally defined but rather customized for individual nodes or individual users. The current approach lets users define views of data using view definitions written using SQL. Users can thus express precisely what data is of interest.

This does mean that the database schema must be globally known and shared between all the nodes in the system. This is a small but necessary limitation. A solution would be to implement a way to handle schema changes dynamically.

7.1.2 Clear update semantics

As seen in the previous sections, view definitions affect the kinds of updates that can be applied. Different views provide different update capabilities. It is vital that it is easy to reason about what effects updates have when considering a view definition.

This is not currently as clear as one would hope. The current implementation allows users to change attributes constrained by a predicate to a non-selecting value. Doing such an update will remove the tuple from the view. It might even remove multiple tuples from the view at once. The usual semantics for an update is that it will neither add nor remove the updated tuple.

Another update that might not be easily understood is when deletions lead to the inversion of a boolean attribute used in a selection condition. Indeed, inverting the boolean will remove the tuple, but it differs from deletions from views that do not have such a predicate. This differing behavior means updates might not be well-understood.

Although the traditional updates on relational databases are insertion, deletion, and update of tuples, we must perhaps create more specialized operations that apply to views. An update that removes a tuple is a removal triggered by changing the attribute. We could create operations that convey the intention of the update while parameterizing how the operation is applied.

As an example, a deletion can either be done by deleting the tuple containing the superkey for the view, or it can be done by changing an attribute to a non-selecting value. Both cases intend to delete the tuple, but the parameters determine whether it should remove a tuple or update attributes to achieve removal.

Conversely, this means that an update will only ever update tuples in place, and removal is not a possible outcome. Deletion will always remove a tuple, while updates will never remove a row. If these invariants cannot be guaranteed, the operations should produce errors.

Insertion is a bit different. The current implementation is idempotent, ensuring all the tuples are present for the join condition to be true. It could fail if the insertion did not insert any new tuple. For example, it should not be possible to insert rows that would not appear within the view due to the inserted tuple not satisfying the selection predicate. The insertion should therefore produce an error if no insertion took place.

7.1.3 Well-defined translations

One of the requirements was that updates should have well-defined translations. By placing restrictions on what views are updatable, we have been able to define computational algorithms that unambiguously translate updates on views into updates that can be applied to source database instances. Our approach is pragmatic in the sense that we choose reasonable update algorithms. This does not mean the update policies are the only ones that can unambiguously translate view updates.

A drawback of the current implementation is that it does not enforce all the restrictions that have been described. It does not ensure that key attributes are present or that the view-dependency graph forms a rooted tree. These invariants should be evaluated upon creating the views and reevaluated if there are any changes to the database schema. It would be similar to enforcing the typing rules presented in [4].

Default values for attributes that are projected away should also satisfy any selection predicate defined on it. Otherwise, insertions on views will not be well-behaved since a view will have inserted a tuple that should never have been present. It will eventually be removed when the default value is applied, and the inserting node recomputes the view at a node that has applied the default value.

Another limitation of default values is that they are currently only defined in the global database schema. It would likely be helpful to let the definition of the view dictate what default values to set. This could allow an update to be correctly tagged with the user ID for the updating user in a user-centric view.

7.1.4 Guarantees strong eventual consistency

The current implementation does guarantee strong eventual consistency by utilizing CRDTs. Even though eventual consistency is guaranteed, it does not guarantee that data dissemination happens at any particular rate. This is determined by the views the nodes replicate and which peers they synchronize with. How replication of views affects the synchronization behavior concerning consistency should be investigated further.

It is important to stress that the current approach does not claim to be a general approach that can solve most problems for distributed systems. It is only meant for applications where only strong eventual consistency can be applied. The CALM theorem [11] describes what class of problems can be solved without coordination. Problems that do not require coordination are precisely those problems in which one can leverage eventual consistency.

The class of problems that are solvable without coordination are the problems that are logically monotonic. In essence, this means that the conclusions that can be made from an arbitrary state cannot be disproven by any updates that change the state. The design of CRDTs ensures that they are logically monotonic, which is a prerequisite for eventual consistency. Thus, developers are responsible for using replicated views to ensure that the problem being solved is logically monotonic.

7.1.5 Minimize required computer resources

The preliminary results shown in the experimental section show that replication of data subsets leads to a reduction in required hardware capabilities. Some additional aspects of replication can affect the performance of a system replicating data subsets.

Storage has been the primary focus when implementing the replication of data subsets. The reason is that limiting the amount of information will, in turn, limit the processing required for querying, updating, and transmitting updates. It does not mean limiting required storage is optimal; it depends on how the views are accessed.

A view is replicated by replicating subsets of all the tables contributing to the view. This means that the view is not a traditional materialized view. Materialized views speed up frequently used queries by precomputing the result. Our approach requires the view to be recomputed using the view definition upon each access. It likely has a higher computational cost than actually materializing the view. View replicas that replicate large views and do lots of queries will probably not be optimal in the current solution. If query performance is inadequate, techniques can be employed for materializing and maintaining views locally. They will not affect the implementation of the CRR layer as long as triggers propagate updates correctly and the materialized view is appropriately maintained.

It is assumed that views are small, so recomputing views should not be a huge issue. All joins within a view are on key attributes and, as such, will only contain as many rows as is in the root relation of the view-reference graph.

Since all joins are on keys, indexing ensures the joins are more efficient than a linear scan.

CRDTs are unbounded in their data size growth. Thus, any change to the replica's state can only add information to the system. Even though views replicate subsets of data, this only slows down data growth at each node replicating the view. Some view definitions can maintain constant data, but tuples are being added and removed at the same rate. Think of time-based views in which information from the last week is kept. These views should, in theory, limit the amount of data, but the constant growth may be infeasible over a larger period. In addition, the size of the deltas sent to views increases since all rows that have ever been in the view are sent to guarantee strong eventual consistency.

Even though there are a lot of impossibilities in distributed systems, reasonable assumptions can often lead to designs that are fit for use in practice. Removal of data could violate the guarantees for strong eventual consistency. It depends on how the system will be used. An approach to partially replicating CRDTs has already been presented in [5]. It presents two different system models, one pure peer-to-peer system and one in which there are authorities. Authorities are nodes that replicate the full state of a CRDT. They show that it is possible to grow and shrink the replicated subset of a state-based CRDT when the state of the partial replicas can be updated from or offloaded to authoritative replicas. This is similar to how version control systems are used. There is often a single authoritative copy for a project, and developers merge their local changes into the authoritative copy. In a pure peer-to-peer system, it is not possible to do such offloading.

7.2 Future work

Much work can be done to make the current approach more viable. Many practical issues should be resolved for the approach to be viable in production systems. The first is to build a better interface that makes it easy to use arbitrary communication protocols. Security concerns have also been omitted, although limiting information in the name of security and privacy has been a core motivation. The current supporting database technology has limitations that may warrant the creation of a custom database implementation.

7.2.1 Synchronization over the network

Previous implementations of CRR included network communication between nodes. One of the implementations used SSH to send commands and transmit delta-states between nodes. Another implementation used TCP/IP as the transport protocol while implementing a custom application-level protocol for synchronization between multiple nodes.

The current implementation does not include any implementation for synchronization over the network. The focus has instead been on how views can be implemented. Network synchronization is, therefore, out of scope. The CLI tool could instead be modified to allow the generation of deltas for view replicas and regular replicas. It can also allow the merging of incoming deltas. It would then be the application's responsibility to ensure that deltas are transmitted between nodes.

Therefore, future work could involve the creation of a better interface for different communication protocols to be used so that the actual CRDT implementation is not bound to any particular communication protocol. The only requirement for implementing a communication protocol is to use node identifiers generated by the augmented database and that the deltas are transmitted reliably.

7.2.2 Security concerns

The replication of views can be used to limit the amount of information that is replicated to nodes. This can improve security and privacy, as mentioned earlier. Several security concerns must be addressed before achieving a satisfactory degree of security. There is no limitation on who can create views and what can be created. This means that views can extract whatever information a user wants. There is no concept of identity for users. There is no authentication mechanism to prove the identity of users.

In addition to limiting access to information, there is the issue of limiting the updates a user can execute. This was shown as an issue when presenting use cases in section 5.1. Fine-grained control over what updates can be done on particular views should be investigated. Creating a view for a user should only allow that user to access the view. Restricting updates per user may also be necessary.

A complicating factor for incorporating authentication and authorization into the system is that existing frameworks are often centralized solutions. It should be possible for peers to verify that the data has been updated by a user authorized to update it. It must also be possible for nodes to receive synchronization requests to verify that the requesting node can receive the requested information. Having a centralized security solution will hurt system availability. It may also be helpful for users to restrict access to information they create and then delegate access. An approach to security in a decentralized system using CRDTs is presented in [13].

7.2.3 Beyond SQLite

The current implementation uses SQLite as a supporting database technology. Many issues have been identified during the development process. The primary drawback of using SQLite is that it is currently impossible to express complex integrity constraints. It is impossible to define constraints in SQLite that ignore tuples with even causal lengths. As said earlier, the AR layer is a view on top of the CRR layer where rows with odd causal lengths are present. It is currently required to have a materialized AR relation for SQLite to enforce integrity constraints autonomously. If the integrity constraints in SQLite could be made to ignore tuples of even causal length, the AR layer materialized view would become obsolete.

The primary benefit of the materialization of AR tables is that it is easier to define and enforce integrity constraints. Insertion of a tuple that already exists into an AR layer table will trigger an SQLite error. Using an actual view on top of the CRR layer makes the same error functionality harder to implement. The integrity constraints should only be enforced on those rows in the AR layer with an odd causal length. This kind of integrity constraint is not currently possible to express in SQLite.

Many tables contain metadata needed for synchronization, which is not hidden from the database users. The tables that are created during augmentation should be hidden from users. Metadata tables are only identified using a naming convention, which can break down if users name their tables interfering with the convention.

One possibility would be to implement a custom database solution where the internal tables for augmentation are hidden, and procedures for handling synchronization are built in. This would allow for the implementation of custom integrity constraints and additional CRDTs as custom attribute types (not only LWW-register), type-checking of view definitions, and enable the specification access control to views and tables for users.

8

Conclusion

The main contribution of this thesis is an approach to asynchronously replicated views that are updatable, and that guarantees eventual consistency. It combines classical research on updatable views with conflict-free replicated datatypes (CRDT) research. Our approach can apply updates to database instances independent of the state where updates are applied. Existing research on updatable views cannot independently apply updates at other database instances and guarantee that updates are resolved similarly at different instances. It also differentiates itself from existing research on CRDTs as the views can be defined using a query language, allowing for easily defining specialized partial replicas.

An implementation of conflict-free replicated relations (CRR) has been written in Rust. The implementation of CRR has been extended to allow the replication of views that can be defined using SQL syntax. The views can be updated, activating specific database triggers for each view. The triggers propagate updates from the view to the tables contributing to the view.

The development process involved creating test cases to ensure that view updates were well-defined. The implementation was first based on recent research on relational lenses. After implementing relational lenses, it was combined with the implementation of CRR-augmented databases. The properties of CRR-augmented databases lead to the simplification of the implementation. This meant relational lenses were no longer needed. The final solution is most similar to the work on updatable views in [15].

The implementation has also been evaluated to ensure it meets the requirements for partial replication. The requirements are summarized below.

- User-centric
- Clear update semantics
- Updates have well-defined translations
- Guarantees strong eventual consistency
- Minimizes the required storage, computation, and communication capabilities

The implementation has been evaluated to ensure that it fulfills the requirements. The evaluation included assessing the usability of the implementation of updatable views against several use cases. The assessment found that there were view definitions that would fulfill the requirements for all the use cases. The use cases were built upon what specific users may want from a system. It also included a use case tailored for individual users. The current implementation allows the definition of user-centric views.

The use cases also show that the views have clear enough update semantics. It is necessary to know the view definition to understand an update's effects on a particular view. Careful thought must be put into creating views to allow the required update capabilities. Naming a view is particularly important to convey what update capabilities the view has. The implementation also ensures well-defined translations of updates. All updates have an equivalent translation that can be applied to a regular replica storing the full state. This has been validated through rigorous testing using test cases.

The implementation guarantees strong eventual consistency, as it is built on the strong theoretical foundation of CRDTs. It uses the same join-irreducible states as a CRR-augmented database. It also uses mostly the same delta-generation and merge procedures. A few modifications have been made to delta-generation and merging. Delta generation is changed to only transmit tuples that are part of the view or have ever been a part of the view. Delta-generation and merging have also been modified to incorporate translations of projections and selection conditions. The design is largely the same, and our modifications will still guarantee strong eventual consistency.

An experiment was conducted to investigate whether or not introducing the replication of views reduced required storage, computation, and communication capabilities. The required storage was greatly reduced, as would be expected.

The synchronization procedures for delta generation and merging were efficient when using views. Views also are faster for updates as it only replicates data in the CRR layer, but they might be slower for queries on large views.

There is quite a bit of future work that can be done to improve the implementation. The implementation can be improved by allowing it to be used with multiple network communication protocols. Improved security and privacy were part of the motivation for implementing views. Still, much more work is needed to build a secure framework with proper authentication of users and authorization for access control. It might also be beneficial to move away from using SQLite as a supporting database technology in favor of a more custom solution allowing for a more flexible implementation.

In conclusion, this thesis describes a new approach to view-update problems in the context of local-first software. It shows that it can be implemented efficiently and that view updates have well-defined translations with clear semantics, and that the solution also guarantees strong eventual consistency.

Bibliography

[1] Network Time Protocol (NTP). RFC 958, September 1985.

[2] ADBIS. European conference on advances in databases and information systems. `http://adbis.eu/index.php`. Accessed on 2023-05-20.

[3] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018.

[4] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In Stijn Vansummeren, editor, *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 338–347. ACM, 2006.

[5] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free partially replicated data types. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 282–289, 2015.

[6] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, Paul R. Young, and Peter J. Denning. Computing as a discipline. *Commun. ACM*, 32(1):9–23, jan 1989.

[7] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

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

[9] Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility

of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[11] Joseph M. Hellerstein and Peter Alvaro. Keeping calm: When distributed consistency is easy, 2019.

[12] Rudi Horn, Roly Perera, and James Cheney. Incremental relational lenses. *Proc. ACM Program. Lang.*, 2(ICFP):74:1–74:30, 2018.

[13] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. Secure replication for client-centric data stores. In *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good*, DICG '22, page 31–36, New York, NY, USA, 2022. Association for Computing Machinery.

[14] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, nov 1962.

[15] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 25-27, 1985, Portland, Oregon, USA*, pages 154–163. ACM, 1985.

[16] 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)*, pages 154–178, 2019.

[17] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, pages 17–32, Cham, 2014. Springer International Publishing.

[18] Google LLC. Google docs. `https://www.google.com/docs/about/#overview`. Accessed on 2023-05-20.

[19] The Git Project. About. `https://git-scm.com/about/`. Accessed on 2023-05-20.

[20] Rust. Rust. `https://www.rust-lang.org/`. Accessed on 2023-05-20.

[21] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data

types. *Rapport de recherche*, 7506, January 2011.

[22] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400. Springer LNCS volume 6976, October 2011.

[23] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., USA, 7 edition, 2005.

[24] SQLite. About sqlite. `https://sqlite.org/about.html`. Accessed on 2023-05-20.

[25] SQLite. Create trigger. `https://www.sqlite.org/lang_createtrigger.html`. Accessed on 2023-05-20.

[26] Iver Toft Tomter and Weihai Yu. Augmenting SQLite for local-first software. In *New Trends in Database and Information Systems, (ADBIS2021 workshops*, volume 1450 of *Communications in Computer and Information Science*, pages 247–257. Springer, 2021.

[27] M. van Steen and A.S. Tanenbaum. *Distributed Systems*. distributed-systems.net, 3 edition, 2020.

[28] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, jan 2009.

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

[30] Weihai Yu and Sigbjørn Rostad. A low-cost set CRDT based on causal lengths. In *Proceedings of the 7th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*, pages 5:1–5:6, 2020.

A

Appendix: Paper submission to ADBIS

Eventually-Consistent Replicated Relations and Updatable Views

Joachim Thomassen and Weihai Yu

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

Abstract. Distributed systems have to live with weak consistency, such as eventual consistency, if high availability is the primary goal and network partitioning is unexceptional. Local-first applications are examples of such systems. There is currently work on local-first databases where the data are asynchronously replicated on multiple devices and the replicas can be locally updated even when the devices are offline. Sometimes, a user may want to maintain locally a copy of a view instead of the entire database. For the view to be fully useful, the user should be able to both query and update the local copy of the view. We present an approach to maintaining updatable views where both the source database and the views are asynchronously replicated. The approach is based on CRDTs (Conflict-free Replicated Data Types) and guarantees eventual consistency. We also present our preliminary implementation and performance results.

Keywords: Data replication, eventual consistency, updatable views, lenses, CRDT

1 Introduction

Local-first software suggests a set of principles for software that enables both collaboration and ownership for users. Local-first ideals include the ability to both work offline and collaborate across multiple devices [13].

There has been effort in supporting local-first database systems [18]. These systems replicate data at different sites and allow the sites to make immediate updates on local data. According to the CAP theorem [6, 8], it is impossible to simultaneously ensure all three desirable properties, namely (C) consistency equivalent to a single up-to-date copy of data, (A) availability of the data for update and (P) tolerance to network partitioning. We have to live with weak, or eventual, consistency [16], if we want immediate update on data that are offline.

CRDT [15], or Conflict-free Replicated Data Type, has been a popular approach to constructing eventually-consistent systems. With CRDT, a site updates its local replica without coordination with other sites. The states of replicas converge when they have applied the same set of updates. CRR [18], or Conflict-free Replicated Relation, is an application of CRDT to relational databases.

A user may want to maintain copies of views in her local devices, instead of the entire database. This may reduce both the amount of data stored on the device and the overhead of data communication and local data processing. This may also simplify user interaction as the user can focus on the data of interest. Furthermore, this may enhance the protection of data by preventing a user from accessing the data outside a view.

For a view to be fully useful, the user should be able to both query and update the local copies of the views. The updates must then be translated and applied back to the original source database.

Supporting updatable views has been an active research topic for decades [2,2–4,10, 12]. There is at present no existing work on supporting updatable views in a distributed setting where both the source database and the views are asynchronously replicated and local data can be updated even when the replicas are offline.

Our main contribution is an approach to supporting updatable views where both the source database and the views can be asynchronously replicated. The approach is an extension to CRR. It guarantees eventual consistency. That is, when the databases (or the views) have applied the same set of updates, their states converge. Furthermore, the approach preserves integrity constraints defined in the source database.

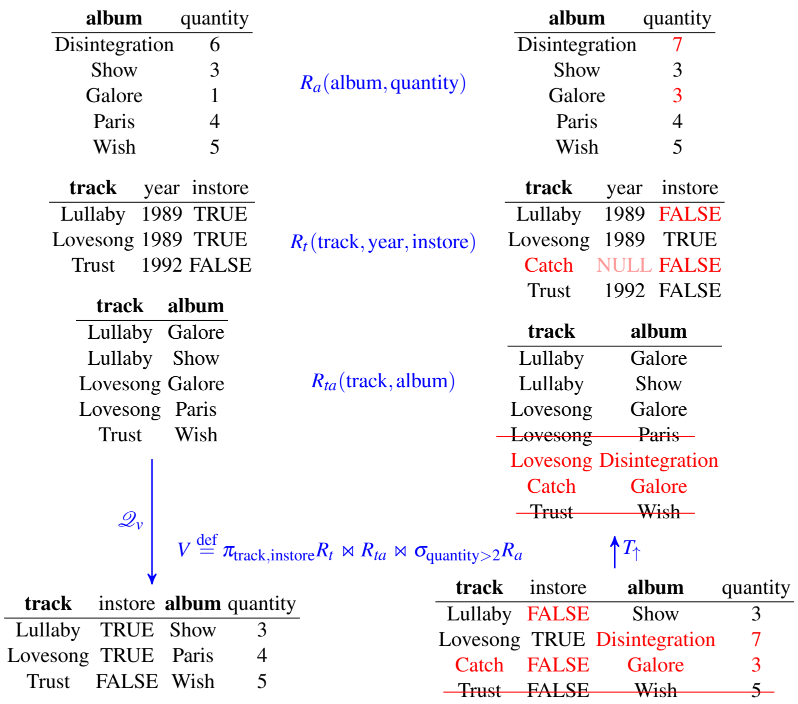
The paper is organized as the following. In Section 2, we first describe updatable views with an example. Then, we present the consistency issue when the database and the views can be asynchronously replicated. In Section 3, we briefly review the necessary background of CRDT and CRR. In Section 4, we give a high-level overview of our approach. In Section 5, we use examples to explain how we translate view updates to the source database. In Section 6, we present some early implementation and experiment results. In Section 7, we discuss related work. Finally, in Section 8, we conclude.

2 Updatable views and and eventual consistency

In this paper, a database query operation, ranging over q, can be a select σ, project π or join $\bowtie$. A database updating operation, ranging over u, can be a tuple insertion, deletion or update. We abuse the term *database* and *relation* for either their schema (ranging over S for source databases of views, R for base relations in a source and V for views) or instance (ranging over s, r and v), when it does not cause confusion.

A view V is defined with a sequence of query operations $\mathscr{Q}_v = [q_1, q_2, \ldots]$ over a source database s, i.e. $v = \mathscr{Q}_v(s)$. After an update u_v on view v, the new state of the view becomes $v' = u_v(v)$. A translation $T_\uparrow$ of u_v to source database s results in a sequence of updates in base relations $T_\uparrow(s, u_v) = [u_1, u_2, \ldots]$. According to [4], $T_\uparrow(s, u_v)$ *exactly performs* $u_v(v)$ iff $\mathscr{Q}_v(T_\uparrow(s, u_v)(s)) = u_v(\mathscr{Q}_v(s))$. $T_\uparrow(s, u_v)$ *exactly translates* $u_v(v)$ iff it exactly performs $u_v(v)$ and preserves integrity constraints defined in S. Traditional research on updatable views (e.g. [2, 4, 9, 12]) and later work (e.g. [3, 10]) focused on functional-dependency constraints. In this paper, we handle also integrity constraint that can be defined by the application, such as referential and uniqueness constraints.

Fig. 1 shows an example of an updatable view, adapted from [3, 10]. The source database $S = \{R_a, R_t, R_{ta}\}$. That is, there are three base relations in S, R_a for musical albums, R_t for musical tracks and R_{ta} for tracks in albums. The view is defined with



album	quantity
Disintegration	6
Show	3
Galore	1
Paris	4
Wish	5

$R_a(\text{album}, \text{quantity})$

album	quantity
Disintegration	7
Show	3
Galore	3
Paris	4
Wish	5

track	year	instore
Lullaby	1989	TRUE
Lovesong	1989	TRUE
Trust	1992	FALSE

$R_t(\text{track}, \text{year}, \text{instore})$

track	year	instore
Lullaby	1989	FALSE
Lovesong	1989	TRUE
Catch	NULL	FALSE
Trust	1992	FALSE

track	**album**
Lullaby	Galore
Lullaby	Show
Lovesong	Galore
Lovesong	Paris
Trust	Wish

$R_{ta}(\text{track}, \text{album})$

track	**album**
Lullaby	Galore
Lullaby	Show
Lovesong	Galore
~~Lovesong~~	~~Paris~~
Lovesong	Disintegration
Catch	Galore
~~Trust~~	~~Wish~~

$\mathcal{Q}_v$

$V \overset{\text{def}}{=} \pi_{\text{track,instore}} R_t \bowtie R_{ta} \bowtie \sigma_{\text{quantity}>2} R_a$

$T_{\uparrow}$

track	instore	**album**	quantity
Lullaby	TRUE	Show	3
Lovesong	TRUE	Paris	4
Trust	FALSE	Wish	5

track	instore	**album**	quantity
Lullaby	FALSE	Show	3
Lovesong	TRUE	Disintegration	7
Catch	FALSE	Galore	3
~~Trust~~	~~FALSE~~	~~Wish~~	~~5~~

Fig. 1. Example of an updatable view, adapted from [3]

$V \overset{\text{def}}{=} \pi_{\text{track,instore}} R_t \bowtie R_{ta} \bowtie \sigma_{\text{quantity}>2} R_a$. There are four updates in view v:

$$
\begin{aligned}
u_1(v) &= v\langle \text{Lullaby}, \text{TRUE} \nearrow \text{FALSE}, \text{Show}, 3\rangle,\\
u_2(v) &= v\langle \text{Lovesong}, \text{TRUE}, \text{Paris} \nearrow \text{Disintegration}, 4 \nearrow 7\rangle,\\
u_3(v) &= +v\langle \text{Catch}, \text{FALSE}, \text{Galore}, 3\rangle,\\
u_4(v) &= -v\langle \text{Trust}, \text{FALSE}, \text{Wish}, 5\rangle.
\end{aligned}
$$

Here, $+v\langle\dots\rangle$ and $-v\langle\dots\rangle$ are an insertion and a deletion of a tuple in v, and $v\langle\dots, a \nearrow a'\rangle$ is an update of an attribute of a tuple in v from value a to value a'.

Below is a possible translation of the view updates in s:

$$
\begin{aligned}
T_{\uparrow}(s,u_1) &= [r_t\langle \text{Lullaby}, 1989, \text{TRUE} \nearrow \text{FALSE}\rangle],\\
T_{\uparrow}(s,u_2) &= [r_a\langle \text{Disintegration}, 6 \nearrow 7\rangle, -r_{ta}\langle \text{Lovesong}, \text{Paris}\rangle, +r_{ta}\langle \text{Lovesong},\\
&\qquad \text{Disintegration}\rangle],\\
T_{\uparrow}(s,u_3) &= [r_a\langle \text{Galore}, 1 \nearrow 3\rangle, +r_t\langle \text{Catch}, \text{NULL}, \text{FALSE}\rangle, +r_{ta}\langle \text{Catch}, \text{Galore}\rangle],\\
T_{\uparrow}(s,u_4) &= [-r_{ta}\langle \text{Trust}, \text{Wish}\rangle].
\end{aligned}
$$

Notice that a translation of u_3 is only possible if the year-attribute of R_t either is given a default value or may be left unspecified with NULL.

Now, consider the situation where both the source database and the view are asynchronously replicated at different sites. In the existing work (e.g. [2–4, 9, 10, 12]), a translation is made on the current state of the source database. If one replica reaches state s_1 by deleting tuple $r_a\langle \text{Lullaby}, 1989, \text{TRUE}\rangle$, while another replica reaches state s_2 by updating the instore-attribute of the same tuple to FALSE. The translations of view update u_1 in states s_1 and s_2 are $T_\uparrow(s_1, u_1) = T_\uparrow(s_2, u_1) = [\,]$. Depending on the order in which the updates on the source database and the translations of the view update are applied, the instore-attribute of the tuple may end up with value TRUE or FALSE, or the tuple itself may be deleted.

Our work is based on delta-state CRDT [1,5] (Section 3.1), applied to replication of relational databases [18] (Section 3.2). The states of the source database and the views form a join-semilattice [7,15]. The updates in the database and the views are represented as join-irreducible states in the join-semilattice. A view update is translated into a set of join-irreducible states in the source database and the translation is independent of the state of the database. When the database (and equally view) replicas have applied the same set of updates (i.e. join-irreducible states), their states converge.

3 Technical background

In this section, we briefly review necessary technical background on CRDT and CRR.

3.1 CRDT

A CRDT [15] is a data abstraction specifically designed for data replicated at different sites. A site queries and updates its local replica without coordination with other sites. The data are always available for update, but the data states at different sites may diverge. From time to time, the sites send their updates asynchronously to other sites with an anti-entropy protocol. The sites also merge the received updates with their local data. A CRDT guarantees *strong eventual consistency* [15]: a site merges incoming remote updates without coordination with other sites; when all sites have applied the same set of updates, their states converge.

There are two families of CRDT approaches, namely operation-based and state-based [15]. In the primary state-based CRDT approach, a message for updates consists of the data state of a replica in its entirety. A site applies the received updates by merging its local state with the state in the received message. The possible states of a state-based CRDT must form a join-semilattice [7], which implies convergence. Briefly, the states form a *join-semilattice* if they are partially ordered with $\sqsubseteq$ and a join $\sqcup$[1] of any two states (that gives the least upper bound of the two states) always exists. State updates must be inflationary. That is, the new state supersedes the old one in $\sqsubseteq$. The merge of two states s_1 and s_2 is the result of $s_1 \sqcup s_2$.

Fig. 2 (left) shows GSet, a state-based CRDT for grow-only sets [15], where E is a set of possible elements, $\sqsubseteq \overset{\text{def}}{=} \subseteq$, $\sqcup \overset{\text{def}}{=} \cup$, insert is a mutator (update operation) and in?

[1] To avoid being confused with the join $\bowtie$ of relations, in the rest of the paper, we use the term *merge* for $\sqcup$.

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

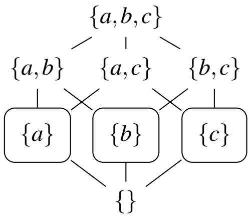


Fig. 2. GSet CRDT and Hasse diagram of states

is a query. Obviously, an update through $\mathsf{insert}(s,e)$ is inflationary, since $s \subseteq \{e\} \cup s$. Fig. 2 (right) shows the Hasse diagram of the states in a GSet. A Hasse diagram shows only the "direct links" between states.

Using state-based CRDTs, as originally presented [15], is costly in practice, because states in their entirety are sent as messages. Delta-state CRDTs address this issue by only sending join-irreducible states [1, 5]. Basically, *join-irreducible* states are elementary states: every state in the join-semilattice can be represented as a join of some join-irreducible state(s). In Fig. 2, insert^{δ} is a delta-mutator that returns join-irreducible states which are singleton sets (boxed in the Hasse diagram). We adopt delta-state CRDTs in our work.

Since a relation instance is a set of tuples, the basic building block of CRR is a general-purpose delta-state set CRDT ("general-purpose" in the sense that it allows both insertion and deletion of elements). We use CLSet (causal-length set, [18, 19]), a general-purpose set CRDT, where each element is associated with a *causal length*. Intuitively, insertion and deletion are inverse operations of one another. They always occur in turn. When an element is first inserted into a set, its causal length is 1. When the element is deleted, its causal length becomes 2. Thereby the causal length of an element increments on each update that reverses the effect of a previous one.

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

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

Fig. 3. CLSet CRDT [18]

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

3.2 CRR

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

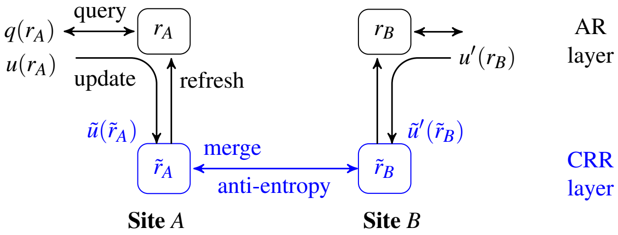


Fig. 4. A two-layer relational database system [18]

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

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

We adopt several CRDTs for CRRs. Since a relation instance is a set of tuples, we use the CLSet CRDT (Fig. 3) for relation instances. We use the LWW (last-write wins) register CRDT [11, 14] for individual attributes in tuples.

The join-irreducible states in a CRR relation $\tilde{r}$ are simply the tuples as the result of the insertions, deletions and updates. In the rest of the paper, we use the term *delta* for the tuple as the join-irreducible state of an operation. As we apply delta-state CRDTs, the tuples of the latest changes are sent to remote sites in the anti-entropy protocol.

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

When inserting a new tuple t into r, we insert a new tuple $\tilde{t}$ into $\tilde{r}$, with the initial $\tilde{t}(L) = 1$. When deleting t from r, we increment $\tilde{t}(L)$ with 1. Tuple t is in r, $t \in r$, if $\tilde{t}(L)$ is an odd number. That is,

$$\mathsf{in\_ar?}(\tilde{t}) \overset{\text{def}}{=} \mathsf{odd?}(\tilde{t}(L))$$

When updating $t(A_i)$ in r, we update $\tilde{t}(A_i)$ and $\tilde{t}(T_i)$ in $\tilde{r}$.

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

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

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

After the update of $\tilde{r}$, we update r as the following. If $\mathsf{in\_ar?}(\tilde{t})$ evaluates to false, we delete t (where $t(K) = \tilde{t}(K)$) from r. Otherwise, we insert or update r with $\pi_{K,A_1,A_2,\dots}(\tilde{t})$.

4 Approach Overview

We consider distributed database systems where data are replicated at multiple sites. For the purpose of, say, high availability, the sites may update the data without coordination with other sites. The system is said to be *eventually consistent*, or convergent, if, when all sites have applied the same set of updates, the sites have the same state. The system is said to be *strongly eventually consistent* [15], if the sites unilaterally resolve any possible conflict, i.e., without coordination with other sites. We focus on strongly eventually-consistent relational database systems.

We restrict on which views can be updated, similar to [3, 10, 12]. More specifically, a view can only project away non-primary-key attributes that are given default values or can remain unspecified with NULL when inserted without given values. Moreover, when joining two relations, the join attribute(s) must contain one of the primary keys.

For a source database S, we define a view V with $V = \mathcal{Q}_v(S)$. Suppose when the database state is initially s_0, the view state is $v_0 = \mathcal{Q}_v(s_0)$ (Fig. 5). Concurrently, the view applies updates with delta state $\Delta v'$ and the source database applies updates with delta state $\Delta s'$. The new states in the view and the database become $v_1 = v_0 \sqcup \Delta v'$ and $s_1 = s_0 \sqcup \Delta s'$ respectively. When the database receives $\Delta v'$, it applies the translated delta $T_{\uparrow}(\Delta v')$ to s_1 and the new state becomes $s_2 = s_1 \sqcup T_{\uparrow}(\Delta v')$. Similarly, when the view receives $\Delta s'$, it applies the translated delta $T_{\downarrow}(\Delta s')$ to v_1 and the new state become $v_2 = v_1 \sqcup T_{\downarrow}(\Delta s')$. One important property of the translations $T_{\downarrow}$ and $T_{\uparrow}$ is that they are independent of the target state in which the translation results are going to be applied.

Unlike traditional work on updatable views, we do not restrict to side-effect-free view updates. However, we do respect integrity constraints, including the ones defined by application programs, for instance, functional dependencies enforced with triggers.

$$\begin{tikzcd}
s_0 \arrow[r, "\Delta s'"] \arrow[d, "\mathscr{Q}_v"'] \arrow[dr, dotted] & s_1 \arrow[r, "T_{\uparrow}(\Delta v')"] & s_2 \arrow[rr, "\Delta s'' = IC(s_2)"] & {} \arrow[d, dotted] & s_3 \arrow[d, "\mathscr{Q}_v"] \\
v_0 \arrow[r, "\Delta v'"] \arrow[ur, dotted] & v_1 \arrow[r, "T_{\downarrow}(\Delta s')"] & v_2 \arrow[rr, "T_{\downarrow}(\Delta s'')"] & {} & v_3
\end{tikzcd}$$

Fig. 5. Delta-states in source database and view

In Fig. 5, if the state s_2 violates an integrity constraint, s_2 is never visible to the application. Instead, the view immediately applies some additional delta (as side effect of $T_{\uparrow}(\Delta v')$), $\Delta s'' = IC(s_2)$ for integrity-constraint preservation, and the new state s_3 does not violate any integrity constraint. Finally, the view applies the translation of $\Delta s''$. Our approach guarantees that the updates in Fig. 5 commute. That is,

$$\mathscr{Q}_v(s_0 \sqcup \Delta s' \sqcup T_{\uparrow}(\Delta v') \sqcup \Delta s'') = \mathscr{Q}_v(s_0) \sqcup \Delta v' \sqcup T_{\downarrow}(\Delta s') \sqcup T_{\downarrow}(\Delta s'')$$

Since the merge operation $\sqcup$ is commutative, when the different replicas of the source database (or the view) have applied the same set of delta states, their final states converge.

5 Translation of view-update delta states

The translation from source database to views, $T_{\downarrow}$, is traditionally know as incremental maintenance of materialized views. In this section, we focus on $T_{\uparrow}$, the translation of view-update delta states to the source database. We describe the translation through examples.

We start with select and project views. In Fig. 6, the base relation R_t (top left) is first augmented to a CRR-layer relation $\tilde{R}_t$ (top right). $\tilde{R}_t$ has an attribute L for the causal lengths of the tuples. In addition, every non-primary-key attribute is associated with a timestamp attribute, indicating the last time at which the attribute value was set.

A project view has the same causal-length and timestamp attributes as the base relation, unless the attribute is projected away. A select view has two more attributes σ and T_σ that tell the last time the select predicate was evaluated. Initially, all σ values are TRUE and the timestamp value T_σ of a tuple is the maximum of the timestamp values of the attributes that occur in the select predicate. For tuples in CRR-layer $\tilde{v}_1$ in Fig. 6(a), the T_σ values are set to the T_y values $\tilde{r}_t$. If later the year-attribute of a tuple is set to a value greater than or equal to 1990, the σ value becomes FALSE and the corresponding tuple disappears from the AR-layer view.

The delta state of an update is simply a tuple in a CRR-layer relation or view. For update $v_1\langle\text{Lullaby}, 1989 \nearrow 1988\rangle$ in Fig. 6(a), the delta state is $\tilde{v}_1'\langle\text{Lullaby}, 1988, 5.1, 1, \text{TRUE}, 5.1\rangle$. Here, $T_y = 5.1$ is the timestamp at which the new year-value is set. Since the year-attribute is used in the select predicate, T_σ is also set to 5.1.

For deletion $-v_1\langle\text{Lovesong}, 1989\rangle$, the L attribute of the delta state is incremented with 1. As it is an even number, the tuple is regarded as being deleted in the AR layer.

For insertion $+v_1\langle\text{Catch}, 1989\rangle$, the initial L value is 1 and all timestamps are set according to the current time. For all insertions in select views, the T_σ value must be TRUE.

AR layer

track	year	instore
Lullaby	1989	TRUE
Lovesong	1989	TRUE
Trust	1992	FALSE

CRR layer

track	year	T_y	instore	T_i	L
Lullaby	1989	1.0	TRUE	1.0	1
Lovesong	1989	3.0	TRUE	3.0	1
Trust	1992	2.0	FALSE	2.0	1

$r_t, \tilde{r}_t$ ↓

track	year
Lullaby	1989
Lovesong	1989

track	year	T_y	L	σ	T_σ
Lullaby	1989	1.0	1	TRUE	1.0
Lovesong	1989	3.0	1	TRUE	3.0

$v_1, \tilde{v}_1$ ↓

track	year
Lullaby	1988
~~Lovesong~~	~~1989~~
Catch	1989

track	year	T_y	L	σ	T_σ
Lullaby	1988	5.1	1	TRUE	5.1
Lovesong	1989	3.0	2	TRUE	3.0
Catch	1989	7.1	1	TRUE	7.1

$v'_1, \tilde{v}'_1$ ↓

$r'_t, \tilde{r}'_t$

track	year	instore
Lullaby	1988	TRUE
Lovesong	1989	FALSE
Trust	1992	FALSE
Catch	1989	FALSE

track	year	T_y	instore	T_i	L
Lullaby	1988	5.1	TRUE	5.1	1
Lovesong	1989	3.0	TRUE	3.0	2
Trust	1992	2.0	FALSE	2.0	1
Catch	1989	7.1	FALSE	0.0	1

(a) $V_1 = \pi_{\text{track,year}} \sigma_{\text{year}<1990} R_t$

track	year
Lullaby	1989
Lovesong	1989

track	year	T_y	L	σ	T_σ
Lullaby	1989	1.0	1	TRUE	1.0
Lovesong	1989	3.0	1	TRUE	3.0

$v_2, \tilde{v}'$ ↓

track	year
Trust	1989
~~Lovesong~~	~~1989~~

track	year	T_y	L	σ	T_σ
Lullaby	1989	1.0	1	FALSE	6.2
Lovesong	1989	3.0	1	FALSE	5.2
Trust	1989	6.2	1	TRUE	6.2

$v'_2, \tilde{v}'_2$ ↓

$r'_t, \tilde{r}'_t$

track	year	instore
Trust	1989	TRUE
Lovesong	1989	FALSE

track	year	T_y	instore	T_i	L
Lullaby	1989	1.0	TRUE	6.2	1
Lovesong	1989	3.0	FALSE	5.2	1
Trust	1989	6.2	TRUE	6.2	1

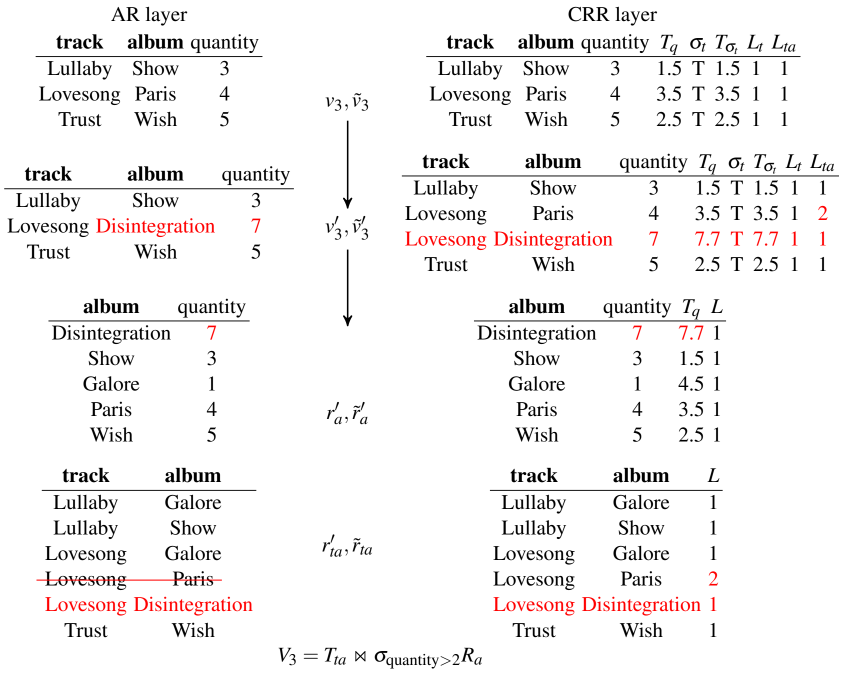
(b) $V_2 = \pi_{\text{track,year}} \sigma_{\text{instore=TRUE}} R_t$

Fig. 6. Updating select and project views

Recall that a project view is updatable only if it keeps the primary key of the base relation. Moreover, CRR-layer base and view relations keep all tuples regardless of whether they have been deleted or not. Therefore, for every tuple in a CRR-layer select-and-project view, there is exactly one tuple in the CRR-layer base relation.

Delta states of a view can be translated almost directly to the base relation. The only exception is for the attributes that are projected away. The instore-attribute of the Catch-tuple, which is missing in view V_1, is set to its default value (suppose it is FALSE). Its timestamp value T_i is set to 0.0, the smallest possible timestamp value. This means that a default value (or NULL) cannot override any value that is explicitly given.

Fig. 6(b) shows two additional cases. The first case shows that a deletion in some views can be handled differently. Here, we have an opportunity to achieve a least-effect translation of deletions in a view, when the select predicate includes a boolean attribute, such as the instore-attribute in $\sigma_{\text{instore=TRUE}}$. Now, for the deletion $-v_2\langle\text{Lovesong}, 1989\rangle$, instead of deleting the Lovesong-tuple in the base relation (i.e. by incrementing the L value), we set the σ value to FALSE. When translating to the base relation, we set the



AR layer

track	**album**	quantity
Lullaby	Show	3
Lovesong	Paris	4
Trust	Wish	5

$v_3, \tilde{v}_3$

CRR layer

track	**album**	quantity	T_q	σ_t	T_{σ_t}	L_t	L_{ta}
Lullaby	Show	3	1.5	T	1.5	1	1
Lovesong	Paris	4	3.5	T	3.5	1	1
Trust	Wish	5	2.5	T	2.5	1	1

track	**album**	quantity
Lullaby	Show	3
Lovesong	Disintegration	7
Trust	Wish	5

$v_3', \tilde{v}_3'$

track	**album**	quantity	T_q	σ_t	T_{σ_t}	L_t	L_{ta}
Lullaby	Show	3	1.5	T	1.5	1	1
Lovesong	Paris	4	3.5	T	3.5	1	2
Lovesong	Disintegration	7	7.7	T	7.7	1	1
Trust	Wish	5	2.5	T	2.5	1	1

album	quantity
Disintegration	7
Show	3
Galore	1
Paris	4
Wish	5

$r_a', \tilde{r}_a'$

album	quantity	T_q	L
Disintegration	7	7.7	1
Show	3	1.5	1
Galore	1	4.5	1
Paris	4	3.5	1
Wish	5	2.5	1

track	**album**
Lullaby	Galore
Lullaby	Show
Lovesong	Galore
~~Lovesong~~	~~Paris~~
Lovesong	Disintegration
Trust	Wish

$r_{ta}', \tilde{r}_{ta}$

track	**album**	L
Lullaby	Galore	1
Lullaby	Show	1
Lovesong	Galore	1
Lovesong	Paris	2
Lovesong	Disintegration	1
Trust	Wish	1

$V_3 = T_{ta} \bowtie \sigma_{\text{quantity}>2} R_a$

Fig. 7. Updating a join view

boolean value of the attribute as the negation in the select predicate. That is, $T_{\downarrow}(\tilde{v}_2\langle\text{Lovesong}, 1989, 3.0, 1, \text{FALSE}, 5.2\rangle) = [\tilde{r}_t\langle\text{Lovesong}, 1989, 3.0, \text{FALSE}, 5.2\rangle]$.

In this particular example, setting the Lovesong-track to be not-in-store is less destructive than deleting the track. When a select predicate uses multiple boolean attributes, we choose to update the truth value of the leftmost one in the view definition.

The next case that Fig. 6(b) shows actually applies generally to updates in both view and base relations. An update of (part of) a primary-key value is regarded as a deletion and an insertion. In the figure, the update $v_2\langle\text{Lullaby} \nearrow \text{Trust}, 1989\rangle$ is interpreted as $[-v_2\langle\text{Lullaby}, 1989\rangle, +v_2\langle\text{Trust}, 1989\rangle]$.

For a view of two-way join $R_1 \bowtie R_2$ to be updatable, we require, as in [12], that the join attributes contain a primary key of R_1 or R_2. We can make a graph from a view of a multi-way join. The nodes are the base relations. If the join attributes of $R_i \bowtie R_j$ contains the primary key of R_j, there is a link from R_i to R_j in the graph. Currently, we require, also as [12], that the view graph is a tree. The primary key of the view is the primary key of the root relation of the tree. Since the primary keys of the base relations are not projected away, for a tuple t_v in the view, we can find the tuples in the base relations that contribute to t_v via their primary-key values.

For view $V = R_1 \bowtie R_2$, the set attributes of the CRR layer $\tilde{V}$ is the union of the sets of attributes of the CRR-layer $\tilde{R}_1$ and $\tilde{R}_2$. For tuple $\tilde{t}_v$ in CRR-layer $\tilde{v}$, tuple t_v is in AR-layer v, if the L values of both $\tilde{r}_1$ and $\tilde{r}_2$ are odd and the σ values of both $\tilde{r}_1$ and $\tilde{r}_2$

are TRUE, i.e.,

$$\mathsf{in\_ar?}(\tilde{t}_v) \overset{\text{def}}{=} \mathsf{odd?}(\tilde{t}_v(L_{r_1})) \wedge \mathsf{odd?}(\tilde{t}_v(L_{r_2})) \wedge \tilde{t}_v(\sigma_{r_1}) \wedge \tilde{t}_v(\sigma_{r_2})$$

In Fig. 7, there is only one update in the view, $v_3\langle \text{Lovesong}, \text{Paris} \nearrow \text{Disintegration}, 4 \nearrow 7\rangle$. Since the album-attribute is part of the primary key of the view, the update is interpreted as a deletion $-v_3\langle \text{Lovesong}, \text{Paris}, 4\rangle$ and an insertion $+v_3\langle \text{Lovesong}, \text{Disintegration}, 7\rangle$.

For the deletion, we delete the corresponding tuple in the root base relation. Hence the tuple $\langle \text{Lovesong}, \text{Paris}\rangle$ is deleted from r_{ta}.

For the insertion, we first insert $\langle \text{Lovesong}, \text{Disintegration}\rangle$ into the root relation r_{ta}. Then, since there is already a Disintegration-tuple in r_a, we set the quantity-attribute to the new value 7.

6 Implementation and experiments

We have implemented CRR and updatable views in SQLite. The structure of a replica, whether it is a source database or a view, is the same as shown in Fig. 4. In addition to the augmented CRR-layer relations, there are other relations including a history of updates, state vectors for synchronization etc. Local CRR-layer updates (as $\tilde{u}$ in Fig. 4) are implemented as triggers that are created when a database instance is CRR-augmented. This allows a database application to continue working without any modification or recompilation. The merge part is implemented in a high-level programming language. We have at present implementations in both Python and Rust.

Since only the AR-layer source database has full knowledge about integrity constraints, a temporarily violation of an integrity constraint can only detected when a replica merges a remote update and refreshes to the AR layer (Fig. 4). Both concurrent updates on source database and views could cause such a violation, due to lack of global knowledge of the source database when the updates were locally applied. Upon the detection of a violation, the replica undoes one of the offending updates [17,18].The approach guarantees that all replicas unilaterally undo the same update.

Due to space limit, we do not present here further implementation details, such as the handling of auto-increment integers as primary keys, and we only report the following experiment.

In the experiment, there are two relations $R_1(K_1, A, B, K_2)$ and $R_2(K_2, C, D)$, where K_1 and K_2 are primary keys of R_1 and R_2, and K_2 in R_1 is a foreign key referencing R_2. Attributes K_1, B, K_2 and D are integers, while attributes A and C are 200-byte character strings. For a run with N tuples in R_1, there are $N/10$ tuples in R_2. K_1 ranges over $[1..N]$ and K_2 ranges over $[1..N/10]$.

The view $V(K_1, A, B, K_2, D)$ is defined as $v = (\sigma_{B=3} r_1) \bowtie (\pi_{K_2,D} r_2)$. K_2 is hence the primary key. With B ranging over [1..100], there are $N/100$ tuples in the view. In every run, we update the D value of 10 different tuples. We ran experiments with the Rust implementation in Windows 10, with Intel-8700k CPU, 32GB RAM and 6 Gb/s 8TB HDD with 400GB m.2 SSD as cache. We range N, the number of tuples in R_1, from 20,000 to 120,000.

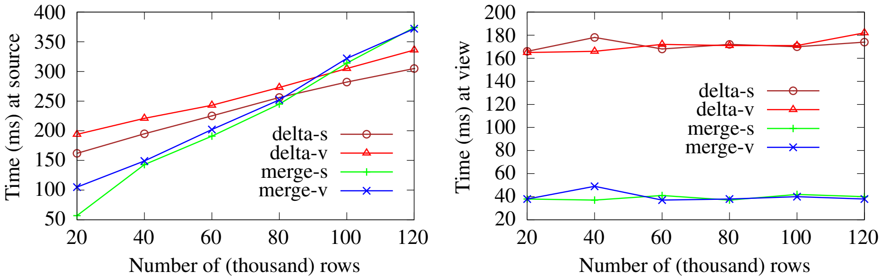


Fig. 8. Time spent for delta generation and merge

We first measured that the CRR-augmented database introduces a space overhead that is close to three times the size of the original AR-layer database. This applies to both the source database and the view. Furthermore, the size of the view is about $1/100$ the size of the source database.

We then measured the time used for generating the deltas of the updates to be sent and for merging the incoming deltas. We replicated both the source database and the view. At the source databases, we measured the time spent for generating deltas for the remote source (delta-s) and the remote views (delta-v), and the time for merging deltas from the remote source (merge-s) and a remote view (merge-v). At the views, we also measured the time for generating outgoing deltas (delta-s and delta-v) and for merging incoming deltas (merge-s and merge-v). As shown in Fig. 8, although the number of updates is constantly 10 in each run, the time spent increases linearly with the size of the source data. Since the size of the view is much smaller than the source, the data size has little impact on the time used for generating and merging deltas.

Depending on the view definition, the space overhead in the view can be significantly smaller than in the source database. For the updates that change both the source and view states, the time overhead in the view is still smaller than in the source, although the difference is less dramatic. What we have not shown in this experiment is that most updates in the source might not affect the view state and would not be sent to the view. All these justify the need for replicating updatable views in devices with limited storage, computing and communication capacity.

7 Related work

[2] and [9] study the consistency of updatable views via mapping of states between source databases and views, where a source database is modeled as the product of the view and a complementary. When a chosen complementary is kept constant (side-effect free) [2] or "shrinking" (under a partial order) [9], there is an unambiguous translation of a view update to the source database. [2] did not aim for computational algorithms that translate view updates to source databases.

To translate the updates from a view to a source database, [12] directly associates tuples and attributes in view relations with base relations in the source database. [4] makes the translation based on the tractability and functional dependency of attributes via view dependency graphs. [9] translates view programs (sequence of updates equipped with if-then-else statements) to base programs. [3] and [10] make bi-directional translation of every query operation (known as a lens) that defines the views. In most of the work on updatable views, translation of view updates is based on the attribute values. For example, since the view dependency graphs in [4] are defined on attributes, deletions are defined with predicates on attributes, for instance, "delete from V where $A = 7$". The source tuples can then be identified with queries on attributes with similar predicates. This may work well in a non-distributed system. In a distributed system where the source database and the view can be replicated, different replicas in different states may make different translations (as discussed in Section 2).

Our work is different from the previous work in that we use delta states (i.e. join-irreducible states in a join semilattice) to represent state updates. The translation is independent of the state to which the update is to be applied.

Regarding the restrictions on views that are updatable, [12] is the closest to our work, which are probably the most restrictive. There are at least two reasons for these restrictions. The first one is practical. Most related work assumes that all information about integrity constraints is available when a view is created, which is practically not true. In particular, the only functional dependencies that can be expressed in SQL is primary-key constraints. The second reason is that we are currently not able to express aggregate results (such as COUNT and MAX) as join-irreducible states.

In their seminal work [4], Dayal and Bernstein pointed out that a view update can be correctly (exactly) translated to the source relations if and only if there is a *clean source* of the update. It is possible to verify if a source is clean with the use of view dependency graphs. With the restrictions of the view that can be updated (Section 4), we guarantee that every update in a view has a clean source.

Unlike previous work, we allow translations of view updates to have side effects (Fig. 5). Avoiding side effect is probably more important in earlier work, which expects virtual (i.e. non-materialized) views. In fact, avoiding side effect is impossible without knowing all integrity constraints, such as the functional dependencies embedded in the view dependency graphs [4]. Notice that concurrent updates at different replicas may temporarily violate integrity constraints (like uniqueness and referential constraints) anyway [18]. We detect violations and repair constraints at the time of merge.

8 Conclusion

We presented an approach to asynchronously replicating both source databases and views. The local replicas of the database and the view can be updated even when they are offline. The approach guarantees eventual consistency. That is, the view updates are correctly translated to the source database, and when the replicas have applied the same set of updates, their states converge. Our experimental results justify the need for replicating updatable views at devices with limited resources.

References

1. ALMEIDA, P. S., SHOKER, A., AND BAQUERO, C. Delta state replicated data types. *J. Parallel Distrib. Comput. 111* (2018), 162–173.

2. Bancilhon, F., and Spyratos, N. Update semantics of relational views. *ACM Trans. Database Syst. 6*, 4 (1981), 557–575.
3. Bohannon, A., Pierce, B. C., and Vaughan, J. A. Relational lenses: a language for updatable views. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)* (2006), S. Vansummeren, Ed., ACM, pp. 338–347.
4. Dayal, U., and Bernstein, P. A. On the correct translation of update operations on relational views. *ACM Trans. Database Syst. 7*, 3 (1982), 381–416.
5. Enes, V., Almeida, P. S., Baquero, C., and Leitão, J. Efficient Synchronization of State-Based CRDTs. In *IEEE 35th International Conference on Data Engineering (ICDE)* (April 2019).
6. Fox, A., and Brewer, E. A. Harvest, yield and scalable tolerant systems. In *The Seventh Workshop on Hot Topics in Operating Systems* (1999), pp. 174–178.
7. Garg, V. K. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.
8. Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (2002), 51–59.
9. Gottlob, G., Paolini, P., and Zicari, R. V. Properties and update semantics of consistent views. *ACM Trans. Database Syst. 13*, 4 (1988), 486–524.
10. Horn, R., Perera, R., and Cheney, J. Incremental relational lenses. *Proc. ACM Program. Lang. 2*, ICFP (2018), 74:1–74:30.
11. Johnson, P., and Thomas, R. The maintamance of duplicated databases. *Internet Request for Comments RFC 677* (January 1976).
12. Keller, A. M. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 25-27, 1985, Portland, Oregon, USA* (1985), ACM, pp. 154–163.
13. Kleppmann, M., Wiggins, A., van Hardenberg, P., and McGranaghan, M. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, (Onward! 2019)* (2019), pp. 154–178.
14. Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. A comprehensive study of convergent and commutative replicated data types. *Rapport de recherche 7506* (January 2011).
15. Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS 2011)* (2011), pp. 386–400.
16. Vogels, W. Eventually consistent. *Comminications of the ACM 52*, 1 (2009), 40–44.
17. Yu, W., Elvinger, V., and Ignat, C.-L. A generic undo support for state-based CRDTs. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)* (2020), vol. 153 of *LIPIcs*, pp. 14:1–14:17.
18. Yu, W., and Ignat, C.-L. Conflict-free replicated relations for multi-synchronous database management at edge. In *IEEE International Conference on Smart Data Services (SMDS)* (October 2020), pp. 113–121.
19. Yu, W., and Rostad, S. A low-cost set CRDT based on causal lengths. In *Proceedings of the 7th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)* (2020), pp. 5:1–5:6.