DEPARTMENT OF
COMPUTER SCIENCE

DINA DOS SANTOS BORREGO

Bachelor Degree in Computer Science

VERIFYING AND ENFORCING APPLICATION CONSTRAINTS IN ANTIDOTE SQL

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

DEPARTMENT OF
COMPUTER SCIENCE

VERIFYING AND ENFORCING APPLICATION CONSTRAINTS IN ANTIDOTE SQL

DINA DOS SANTOS BORREGO

Bachelor Degree in Computer Science

Adviser: Carla Ferreira
Associate Professor, NOVA University Lisbon

Co-adviser: Nuno Manuel Ribeiro Preguiça
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

Verifying and Enforcing Application Constraints in Antidote SQL

To my parents.

Acknowledgements

This thesis would not have been possible without the support of my supervisors, Professor Carla Ferreira and Professor Nuno Preguiça, who helped me greatly throughout the year by providing constant feedback and continuous guidance. Thank you for the patience, perseverance, and helpful insights that enabled me to overcome this challenge.

I would like to thank Kevin de Porre for his time on providing me with all the help to understand and use the verification tool used in this work. Thank you as well for your opinions, ideas, support, and advises.

I would also like to thank some close and special friends, as well as my sisters, for all the good moments that made the elaboration of this thesis much easier. A very special thanks also goes to Afonso Vilalonga, who has always supported and put up with me throughout the process.

Finally, I would like to thank my parents, Maria Inês Borrego and Alfredo Borrego, for all their support at the good and bad moments during my academic journey, for their patience, and also for their love and emotional support. Without them, I could never have reached this stage in my life. Thank you for everything.

Abstract

Geo-replicated storage systems are currently a fundamental piece in the development of large-scale applications where users are distributed across the world. To meet the high requirements regarding latency and availability of these applications, these database systems are forced to use weak consistency mechanisms. However, under these consistency models, there is no guarantee that the invariants are preserved, which can jeopardise the correctness of applications. The most obvious alternative to solve this problem would be to use strong consistency, but this would place a large burden on the system.

Since neither of these options was feasible, many systems have been developed to preserve the invariants of the applications without sacrificing low latency and high availability. These systems, based on the analysis of operations, make it possible to increase the guarantees of weak consistency by introducing consistency at the level of operations that are potentially dangerous to the invariant.

Antidote SQL is a database system that, by combining strong with weak consistency mechanisms, attempts to guarantee the preservation of invariants at the data level. In this way, and after defining the concurrency semantics for the application, any operation can be performed without coordination and without the risk of violating the invariant. However, this approach has some limitations, namely the fact that it is not trivial for developers to define appropriate concurrency semantics.

In this document, we propose a methodology for the verification and validation of defined properties, such as invariants, for applications using Antidote SQL. The proposed methodology uses a high-level programming language with automatic verification features called VeriFx and provides guidelines for programmers who wish to implement and verify their own systems and specifications using this tool.

Keywords: Antidote SQL, Invariants, Static analysis, Weak consistency

Resumo

Os sistemas de armazenamento geo-replicados são atualmente uma peça fundamental no desenvolvimento de aplicações de grande escala em que os utilizadores se encontram espalhados pelo mundo. Com o objetivo de satisfazer os elevados requisitos em relação à latência e à disponibilidade destas aplicações, estes sistemas de bases de dados vêem-se obrigados a recorrer a mecanismos de consistência fracos. No entanto, sob estes modelos de consistência não existe qualquer tipo de garantia de que os invariantes são preservados, o que pode colocar em causa a correção das aplicações. A alternativa mais óbvia para resolver este problema passaria por utilizar consistência forte, no entanto esta incutiria uma grande sobrecarga no sistema.

Sendo que nenhuma destas opções é viável, muitos sistemas foram desenvolvidos no sentido de preservar os invariantes das aplicações, sem contudo, abdicar de baixas latências e alta disponibilidade. Estes sistemas, baseados na análise das operações, permitem aumentar as garantias de consistência fraca com a introdução de consistência ao nível das operações potencialmente perigosas para o invariante.

O Antidote SQL é um sistema de base de dados que através da combinação de mecanismos de consistência fortes com mecanismos de consistência fracos tenta garantir a preservação dos invariantes ao nível dos dados. Desta forma, e depois de definidas as semânticas de concorrência para a aplicação, qualquer operação pode ser executada sem coordenação e sem perigo de quebra do invariante. No entanto esta abordagem apresenta algumas limitações nomeadamente o facto de não ser trivial para os programadores definirem as semânticas de concorrência adequadas.

Neste documento propomos uma metodologia para a verificação e validação de propriedades definidas, como os invariantes, para aplicações que usam o Antidote SQL. A metodologia proposta utiliza uma linguagem de programação de alto nível com capacidade de verificação automática designada por VeriFx, e fornece as diretrizes a seguir para que o programador consiga implementar e verificar os seus próprios sistemas e especificações, utilizando a ferramenta.

Palavras-chave: Antidote SQL, Invariantes, Análise estática, Consistência fraca

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Context

Today, many applications are user centered and are used by thousands of users around the world. Examples include social networks such as Facebook, Instagram and Twitter, and e-commerce applications such as e-Bay and Amazon. In turn, users are becoming increasingly demanding and require applications to have high availability, short response times (i.e., low latency) and good scalability [7].

To meet these requirements, many modern distributed systems are developing microservices applications where data storage is delegated to databases replicated on multiple machines in several globally distributed data centers [22, 28].

Data replication in itself enables the development of applications with high availability guarantees, even in the presence of failures. However, when replicas are scattered all over the world, in addition to high availability, it is also possible to significantly improve the latency of responses to users by redirecting them to the geographically closest replicas [17, 5, 4].

Despite these benefits, geo-replication presents a trade-off between low latency coupled with high availability and strong consistency [20, 14]. To achieve consistency in replicated systems, there is the need to coordinate the replicas. Nevertheless a low latency and high availability, is not compatible with coordination. For that reason, programmers have to choose what best suits the application they are developing.

As far as databases are concerned, they can be divided into two major groups: SQL and NoSQL.

For many years, the databases used were mostly SQL databases. SQL databases provide a relational model in which the structure of the database is organized into tables. This type of database emphasizes data integrity and consistency, and enforces the ACID[1] properties. It also provides a structured query language (SQL) for querying the databases. However SQL databases are not optimized to scale horizontally, that is, to scale by using more machines.

[1]Atomicity, Consistency, Isolation, Durability

With the emerging need for highly available and scalable applications where data is replicated across multiple locations around the world, NoSQL databases have become more attractive. This type of database follows a non-relational model and supports multiple structural and data modeling options, such as graphs, document-oriented, key-value, and others. NoSQL databases are mainly used for their flexibility, availability, scalability, and performance. However, they also have some drawbacks: they only provide weak consistency and many of these systems provide a very simple and limited interface that makes it difficult to model and access data efficiently [19].

1.2 Motivation

In geo-replicated settings, clients can access the data by contacting one of the replicas (usually the closest one). Ideally, regardless replication, the system should behave as if there were only one replica, i.e., it should provide strong consistency. However, strong consistency requires synchronization between replicas, to enforce a total order of operations in each replica [22]. This would have a large impact on the response times of replicated systems, and for large geo-replicated systems, the complete ordering of all operations becomes too expensive due to the large number of replicas and the distance between them.

In order to avoid the cost of synchronization, many systems opt for weak consistency models [17] because operations can be performed without coordination. In these models, operations are propagated in the background, which can lead to different execution orders in different replicas [10]. Although these models provide a better performance, they can lead to temporary or permanent state divergence [4] between replicas and sometimes violate application-defined properties.

Permanent divergence is not admissible by users in this type of application, but this problem can be solved by conflict resolution mechanisms such as CRDTs [25, 23]. These mechanisms are defined at the data level and ensure that replicas have the same state after applying the same set of operations. Nonetheless, as long as the operations are not replicated to all replicas, clients may see different states.

The second problem mentioned was the violation of data integrity properties defined by the application. These properties, also called invariants, are properties that must be checked throughout the execution of the program, but can be violated by the concurrent execution of some operations. Since the problem lies in concurrency, many systems today try to combine strong and weak consistency mechanisms by coordinating only the necessary operations and letting the others execute without synchronization.

Many proposals have emerged in an attempt to strengthen invariants, mostly based on analysing code and introducing consistency mechanisms. Some examples are RedBlue [17], IPA [4], and ECROs [10]. However, these solutions are not easy to put into practice, since they are defined at the level of operations and the behaviour of each operation must be thoroughly analysed.

Antidote SQL [20] has emerged as an alternative to these proposals and takes a different approach: Antidote SQL defines the properties that applications must maintain associated with the data (unlike other approaches that define these properties at the operation level) and guarantees that the application properties are maintained regardless of the executed operations. In addition, Antidote SQL has developed an interface that provides SQL functionalities for the NoSQL database, solving the problem of a limited and inefficient interface mentioned in Section 1.1.

Nevertheless, Antidote SQL poses some problems. The first problem is the correct definition of invariants by developers, which can be a challenging task, especially for complex applications. Once invariants are defined, it is difficult to understand how the defined invariants apply to Antidote SQL and what conflict resolution strategies are appropriate. The third problem occurs when more than one application needs to use the same database at the same time. In this case, it is necessary to verify that the applications have mutually compatible invariants.

1.3 Proposed Solution

The main goal of this dissertation was to address some of the limitations of Antidote SQL. The Antidote SQL presents an innovative approach with respect to the treatment of invariants. However, this approach requires experience on the part of programmers to choose the right semantics for the application. Even after being defined it is not always clear that the semantics chosen are the appropriate ones for the application.

One possible solution to this problem is to use an analysis tool that allows developers to automatically check whether the defined concurrency levels and conflict resolution strategies are correct for an application using Antidote SQL, i.e. whether the invariants are preserved throughout the execution. Unfortunately, and taking into account the differentiating approach of Antidote SQL, none of the analysis tools available in the literature allow the analysis to be performed immediately, and many of them require a high level of expertise in verification to be used.

Therefore, in this work we propose the use of a sophisticated programming language called VeriFx [9] to verify the concurrency levels defined for tables in Antidote SQL. To enable programmers to implement their applications in this language, we have implemented in VeriFx all the CRDTs used in Antidote SQL that are not included in the VeriFx CRDT library and developed a methodology that describes the guidelines that the programmer must follow to be successful in the verification task.

After analysing the weaknesses and limitations of the developed methodology, we also propose the automation of the methodology and present a workflow of the proposed analysis tool.

1.4 Contributions

The main contributions of this work are:

- Mapping between the different types of invariants that exist in the literature, and mechanisms for strengthening invariants in Antidote SQL (these mechanisms attempt to exploit the concurrency levels of the system whenever possible).

- A library that implements all the CvRDTs used by Antidote SQL, such as counters, registers, tables, and flags (some of the CvRDTS were provided in [9]) in VeriFx. These CvRDTs allow developers to implement their most complex systems by composing the provided CRDTs.

- Implementation and verification of some of the invariant maintenance mechanisms used by Antidote SQL, namely the bounded counter, the referential integrity mechanism for the *update-wins* policy, and exclusive and shared locks.

- Implementation of a running example and verification of convergence properties and correctness of invariants for the implemented system.
- Methodology for implementing and verifying applications defined in Antidote SQL in VeriFx.
- Generic model for the implementation of referential integrity relations in VeriFx whose foreign key resolution policy is *update-wins*.
- Generic model of a proof for verifying the maintenance of invariants in CvRDTs.
- Publication of a research paper for the INForum conference ("Verificação e Reforço de Invariantes Aplicacionais no Antidote SQL"). This paper presents some of the results of this dissertation, namely an overview of the types of invariants and mechanisms for maintaining different types of invariants while exploiting concurrency, as well as illustrative examples of how to proceed in software verification.

1.5 Document Structure

The remaining of the document is structured as follows:

- Chapter 2: In this chapter, we present related work. First, we introduce some of the existing consistency models and their properties. This is followed by a brief presentation of some conflict resolution mechanisms. Finally, we conclude this chapter by presenting some of the analysis tools available in the literature.
- Chapter 3: In this chapter we provide the necessary background for this dissertation and describe the main systems and tools that support this work. First we give a summary overview of Antidote SQL, then we set the context to VeriFx and explain some important details.
- Chapter 4: In this chapter, we present a theoretical study of the different types of invariants. For each type, a description and examples are presented to understand how they are defined. For the types supported by Antidote SQL, the mechanisms used by the system to ensure that the invariants are not violated are also presented. On the other hand, for the types that are not currently supported, suggestions are made for possible mechanisms that could be used if Antidote integrates them in the future.
- Chapter 5: In this chapter we discuss the implementations of CRDTs and invariant reinforcement mechanisms used by Antidote SQL in VeriFx. We also discuss some details of the verification process of the implemented data types.
- Chapter 6: In this chapter we describe a methodology for implementing and verifying systems that use Antidote SQL in VeriFx. The chapter ends with a critical analysis of the limitations of the approach.
- Chapter 7: In this chapter we conclude this thesis with a review of the work done and discuss some possible future work.

2 Related Work

In this chapter, we present relevant related work, taking into account the conducted work in this dissertation. First, we discuss consistency models and present some types of strong, weak, and mixed consistency (Section 2.1). Then, we introduce some conflict resolution protocols (Section 2.2). Finally, we present some of the existing invariant analysis systems (Section 2.3).

2.1 Consistency Models

A consistency model can be viewed as a contract between the processes and the database that specifies precisely the results of write and read operations when they execute concurrently, and specifies under what conditions different results can be obtained. Each of the consistency models provides a trade-off between the levels of consistency and availability provided due to the CAP theorem impossibility result.

The CAP [6] theorem states that it is impossible for a distributed system to simultaneously provide the following guarantees:

- **Strong Consistency:** the multiple replicas of the system contain the same value after an operation is performed, which means that clients observe the most recent write regardless of which node they connect to.

- **High Availability:** although replicas may fail, the system is still accessible through other nodes and continues to respond to clients.

- **Partition Tolerance:** the system maintains the correct execution of operations in the presence of network partitions.

Distributed systems cannot avoid partitions, so they must choose between strong consistency or availability. While strong consistency models prioritise consistency over availability, weak consistency models provide weaker consistency guarantees to ensure availability.

2.1.1 Strong Consistency

With strong consistency, systems behave exactly like a single-threaded system, giving the illusion that only a single replica exists, even though operations are performed in multiple replicas. As a result of this behaviour, a total order of the operations must be defined, meaning that each operation must be executed by every replica in the system in the same order, and users see the same state of the system regardless of which replica the request was made to.

Since the system evolves synchronously under this model, reasoning about the behaviour of the system becomes simpler and more intuitive. However, developers must consider the scalability and performance of their application, as this level of coordination limits the concurrency between operations, which affects the response time for the client [27].

Thus, this level of consistency is important for applications where the different replicas must always achieve agreement and contain the latest version to work correctly.

Below we will explain three types of strong consistency and their guarantees, as well as the differences between them.

Linearizability

Linearizability is the strongest form of consistency that a distributed system can have and it provides guarantees for single operations on single objects.

Under this model, all replicas execute operations in the same order, and this order must match the real-time order in which they were issued, implying a behaviour equivalent to a non-replicated system. Write operations should appear to be instantaneous, i.e., as soon as a write operation completes, all subsequent operations should see a state in which the write operation was performed.

Linearizability can be implemented using state machine replication with Paxos. State machine replication ensures that all correct replicas follow the same sequence of state transitions by performing operations in the same order. Therefore, Paxos can be used to determine the order in which the state machine must execute operations.

Serializability

Serializability considers groups of one or more operations over one or more objects (i.e., transactions) and guarantees that the execution of a set of transactions over multiple objects is done in compliance with a total order.

One of the differences of this model compared to linearizability is that the order is not constrained with respect to real-time. That is, all replicas must execute operations in the same order, but this order may be different from the order of operations with respect to a global wall clock.

Sequential Consistency

Sequential consistency, also known as timeline consistency, is very similar to linearizability. This consistency model guarantees that operations appear to take place in a particular overall order. However, unlike linearizability, the order does not have to match the global real-time order in which the operations were actually issued by the clients.

Despite that, it is important to note that the order in which the operations are executed must be consistent with the order in which the operations are executed in each process, i.e., if op_1 is issued before op_2 in process A, op_1 must be executed before op_2 in all replicas.

The difference between this consistency model and serializability lies in the granularity: the granularity of sequential consistency is a single operation, while that of serializability is a transaction. So if a program satisfies serializability, it also satisfies sequential consistency. The opposite is not true.

2.1.2 Weak Consistency

Under this consistency model, the operations are executed only on a small set of replicas (eventually one) and then the response are sent to the client. The operations are then propagated asynchronously to the other replicas. The fact that the replication mechanism is asynchronous means that coordination overhead is minimised, which in turn is key to a high-performance and scalable database design.

In contrast to strong consistency, weak consistency models allows the state of replicas to diverge, meaning that the same read can return different values on different replicas.

The downside, however, is that it becomes more difficult to reason about the system due to the complex interleavings of execution.

Causal Consistency

Causal consistency captures the potential causal relations between operations and guarantees that all processes observe causally-related operations in an order that respects their causality relations, i.e., in an order that respects the happens-before relation.

The happens-before relation was defined by Leslie Lamport [16] as follows:

Definition 1. *The relation "happened-before" on the set of events of a system is the smallest relation satisfying the following three conditions:*

(i) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.

(ii) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.

(iii) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two distinct events a and b are said to be concurrent if $a \nrightarrow b$ and $b \nrightarrow a$.

Concurrent operations are not ordered in causal consistency and can be replicated in any order across replicas. Thus, with causal consistency, all processes execute causally-related operations in the same order, but may disagree about the order of causally independent operations. Since this consistency model allows operations to be executed in different orders, the final state of the replicas can diverge after executing the same set of operations without violating any causal property.

Causal+ Consistency

Causal+ consistency [18] is a consistency model that provides stronger guarantees than causal consistency by combining causal consistency with convergent conflict handling, thus solving the aforementioned problem.

The causal property guarantees that causal dependencies between operations are preserved. On the other hand, the convergent conflict handling component ensures that the state of replicas does not diverge forever and that concurrent updates executed in different order in different replicas result in the same state in all replicas. This can be implemented using a handler function, which must be associative and commutative to achieve convergence.

Eventual Consistency

Eventual consistency is the weak consistency model that offers the fewest guarantees, promising only that eventually, when there are no more writes, all replicas of the system will converge to the same state. In the meantime, the system may be inconsistent.

Once this model does not coordinate operations, it is the one that provides better performance to the systems. However, for many applications, eventual consistency is a model with guarantees that are too weak.

2.1.3 Mixed Consistency

An invariant is a property that must always be maintained in the context of an application. A system may have multiple invariants, and to ensure its correct behaviour, all invariants must be maintained throughout the application's lifetime.

For instance, in an application that manages a bank account, one possible invariant is that the balance must not be negative [4, 12, 2]. Suppose the initial balance is 100€ and two withdrawals of 60€ are made simultaneously in different replicas.

In an application with a strong consistency model, the two operations would be ordered and one would be executed before the other. After the execution of the first operation, the balance would be 40€ and for this reason the second operation would be cancelled, since the balance would not be sufficient for the second withdrawal. In this model, the invariant holds and the final state of the system would be correct.

On the other hand, under weak consistency models (e.g., eventual consistency), the following situation could be verified. Since the balance is initially 100€ and once $60 < 100$, the two operations would be validated and executed locally in each replica. However, when the operations were propagated, the invariant would be broken since the final balance would be -20€.

As the example shows, the concurrent execution of some operations can affect the correctness of the application when weak consistency models are used. However, the problem no longer exists with strong consistency models. While weak consistency models provide high availability and performance, the fact that they allow concurrency can lead to undesirable semantics. These anomalies do not occur in stronger consistent systems, but sorting all operations incurs on a large overhead [5].

Moreover, not all operations require strong consistency: only the operations that may violate the invariants of the application need to be coordinated, while the remaining operations can be safely executed without synchronization [17].

Mixed consistency, also called hybrid consistency, allows combining multiple levels of consistency to ensure that the invariants defined by the applications are not violated.

Per-record Timeline Consistency

Per-record timeline consistency [7] is the consistency model of PNUTS and is a model that combines strong consistency with weak consistency.

Under this model, all replicas of a given record apply all updates to that record in the same order. On the other hand, updates can be performed on different records at the same time, and the order in which each replica executes them is irrelevant. This means that strong consistency is guaranteed for each record, while only weak consistency is guaranteed between records. For example, if a set of operations (op_1 and op_2) are performed on record rec_1 in replicas A and a set of operations (op_3 and op_4) are performed on rec_2 in replica B, then in both replica A and B the operations for rec_1 and rec_2 must be executed in the same order, that is, op_2 cannot be executed before op_1 and op_4 cannot be executed before op_3. However, the order in which operations are executed on different records is not subject to any restriction, so op_3 could be executed before op_1.

In this model, each record is assigned to a master replica to which all update operations for that record are forwarded. For efficiency reasons, the master replica is the one that receives the majority of write operations for a given record, and this replica may change depending on the workload. After execution in the master replica, the operation is asynchronously propagated to the other replicas.

RedBlue Consistency

RedBlue Consistency [17] is a consistency model that aims to ensure low response times while maintaining data consistent. To this end, it divides operations into two types: red and blue. Red operations require coordination because they can break invariants and must be executed in the same order on all replicas. Blue operations, on the other hand, are immediately executed locally and replicated asynchronously, which means that the order of execution on replicas can be different.

RedBlue consistency besides to consider a partial order of operations (taking into account the definition of each operation in red or blue), also considers a set of local causal serializations that define site-specific total orders in which the operations are locally applied.

From the definition of each operation type, we can conclude that systems in which all operations are marked as red provide serializability, while systems in which all operations are blue can be considered as providing eventual consistency.

Although some operations are not commutative and therefore cannot be executed in different order on replicas, it is sometimes possible to make changes so that the system state commute. For this purpose, each operation is divided into two components: a generator and a shadow operation. The generator operation has no side-effects, is executed only on the replica where the request was made, and generates the shadow operation. The shadow operation, in its turn, should have the same effects as the original operation and is executed on all replicas, including the one on which the generator operation was performed. It is the generator operation that decides which state transitions to execute, while the shadow operations simply apply the transitions, regardless of the state.

With this approach, it is possible to define commutative shadow operations, even for original operations that are not commutative. For example, an operation that increments the value of X by 20% is a non-commutative operation because the result varies depending on the value of X. However, if the generator operation computes the value corresponding to 20% of X, and the shadow operation adds only that value to X in each replica, the operation becomes commutative.

Nevertheless, there may be non-commutative shadow operations. Therefore, the operations are classified as blue or red according to the following three criteria:

- For each pair of non-commutative shadow operations, the two operations are marked as red.
- Any shadow operation that might violate an invariant is marked as red.
- Any operations that are not marked as red are marked as blue.

Explicit Consistency

Explicit consistency [5] is a consistency model that strengthens eventual consistency by combining it with guarantees that the properties defined by applications will be maintained.

This consistency model is defined in terms of application properties, which means that systems are free to order the execution of operations on different replicas as long as the invariants hold.

The developers define application-specific correctness rules in the form of invariants over database state. Systems based on the invariants identify unsafe operations, i.e., operations that can lead to invariant violation under concurrent execution. More sophisticated implementations, instead of coordinating all operations deemed critical, use approaches that allow systems to avoid coordinating these operations.

2.2 Conflict Resolution Protocols

In weaker consistency models, where concurrent operations can be executed in a different order in each replica, replicas can diverge eternally. For example, consider a replicated register that can be modified by any of the replicas and whose initial state is 1. In replica A, an operation is performed that increases the register value by 2 units. At the same time, in replica B, an operation is performed that multiplies the register value by 5. When the operations are replicated to the other replicas, the register value in replica A would be 15, while in replica B it would be 7, i.e. the state of the system would not be consistent.

In consistency models that guarantee that the state of replicas converges to the same state when writes cease, as is the case with eventual and causal+ consistency, this behaviour is unacceptable. Therefore, systems that adopt these types of consistency must include mechanisms for merging concurrent updates that guarantee that all replicas converge to the same state after the same set of operations is applied to each.

Depending on the specifics of the system, different merging strategies can be used. Three different conflict resolution strategies are presented below: last-writer-wins, CRDTs, and Dynamo's approach.

2.2.1 Last-Writer-Wins

Last-writer-wins is the most famous conflict resolution protocol, and as the name implies, when concurrent writes are applied to a given data object, the last write prevails according to a specified overall order, and all other concurrent writes are discarded. This behaviour can cause updates to be lost, which should not happen in certain situations [2].

For example, if the application in question manages a bank account, the use of last-writer-wins is inappropriate. If two withdrawals were made to the same bank account at the same time, only the last one would prevail under this policy, meaning that only one of the values would be considered for the final state of the database. Therefore, despite the fact that the value of two withdrawals left the account, only one would be deducted from the balance.

An example of a system that uses the last-writer-wins merge policy is Cassandra [15].

2.2.2 CRDTs

To avoid anomalies caused by lost updates, many databases today use CRDTs that provide merge functions to ensure that all updates are reflected in the final database state [2].

A Conflict-Free Replicated Data Type (CRDT) [25] is an abstract data type designed to be replicated across multiple replicas and based on mathematical properties that guarantee state convergence without synchronization across replicas. When using a CRDT, update operations can be applied immediately in each replica without coordination, and replicas are guaranteed to reach equivalent states when they have seen the same set of operations.

Designs of these data types include registers, sets, counters, maps, etc., and each type has concurrency semantics that define how the CRDT handles concurrent operations. Below are presented some examples of CRDTs.

There are two different CRDTs that can be used to implement a register. The last-writer-wins register CRDT uses the last-writer-wins policy. Thus, if the register is updated concurrently, only the value of the last write (in a given total order) is retained. The multi-value register CRDT, on the other hand, is designed to preserve all values from concurrent writes and for this reason, a read operation returns a set containing all values written simultaneously.

Another type of CRDT is the counter CRDT. Counters are data types that can be modified by increasing or decreasing them by one or more units. Since these operations commute with each other, the natural concurrency semantics for the counter CRDT is to have a state that reflects all updates made, i.e., the sum of all increments minus the sum of all decrements. The counter CRDT is a useful data type for numeric data, but it cannot be used when that data is subject to constraints (e.g., $x > 0$), as this could lead to invariant violations. To allow some degree of concurrency even for operations that might violate numeric invariants, a CRDT called bounded counter CRDT is used.

The bounded counter CRDT defines a counter that never reaches negative values and encapsulates an implementation of escrow techniques. This CRDT assigns a number of allowed decrements to each replica, where the sum of all decrements must not exceed the value of the counter. As long as the allocated decrements are not used up, the replicas can accept decrements without coordination. When a decrement operation is performed and the replica has exhausted its allowed decrements, the replica tries to get decrements from other replicas. If this is not possible the operation is aborted.

According to the system synchronization model, there are two basic types of CRDTs: state-based CRDTs and operation-based CRDTs.

State-based CRDTs. In state-based CRDTs, operations are executed locally and then the local state of the replica is propagated (this state reflects the effects of the operations performed). When a replica receives the state of another replica, it merges the received state with its own local state so that the resulting state contains the updates of the two states. To achieve convergence in this case, it is necessary that the operations change the state of the replicas by inflation (i.e., produce a state equal to or greater than the previous state), that all possible states form a join-semi-lattice (i.e., that the sequence of all possible states is partially ordered according to $\leq$), and that the defined merge function produces the join (least upper bound) of the two states.

Operation-based CRDTs. In operation-based CRDTs, update operations are propagated to the other replicas, which also execute them. Therefore, the same set of operations is performed in each replica allowing the replicas reach the same state. Convergence is achieved only if the concurrent updates are commutative. In an operation-based specification, the update operations are executed in two phases: the prepare phase and the effect phase. The prepare phase is executed on the replica on which the update was submitted and has no effects. The effects of the operation are encoded in the effector function, which is created depending on the state of the original replica. Then, the operation is replicated by broadcast the effector function to all replicas including itself. The effect phase applies the incoming messages and updates the state, by executing the effector function.

2.2.3 Dynamo

Dynamo [11] is a highly available key-value storage system used by some of Amazon's core services to provide an "always-on" experience. To this end, Dynamo uses an eventual consistency model in conjunction with a conflict resolution strategy.

In Dynamo, the conflict resolution task is delegated to the application rather than to the database: the divergent state is exposed to the clients, which in turn merge these values and write the new value back to the system. This approach is called application-dependent policy and allows the application to choose the conflict resolution method that best fits the user's intended experience.

Consider an application that manages a shopping cart in which a product A is added to the cart and, at the same time, a product B is simultaneously placed in the cart in another replica. Dynamo's approach keeps the two products in the cart until the next read. On the next read, all values are returned and the application merges the concurrent operations and sends the new state of the cart (which would contain the two products) to the database.

2.3 Invariant Analysis Systems

Minimizing coordination among replicas is key to maximizing the scalability, availability, and performance of database systems [2]. As explained above, and with this goal in mind, applications currently attempt to combine weak and strong consistency mechanisms, using the latter for operations that could potentially compromise application invariants.

However, reason about the interleavings in the execution of concurrent operations and distinguish which operations should be coordinated and which should not, can be a very complicated task for programmers.

Therefore, several tools have been developed to analyse the specification of applications and their invariants, and to identify which operations should be coordinated. The analysis tools help programmers to exploit mixed consistency as much as possible [22] and sometimes suggest alternative mechanisms for coordination.

To address the question of which operations must be coordinated, Bailis et al. formalized a sufficient and necessary condition for invariant-preserving and coordination-free execution of an application's operations: invariant confluence, or $\mathcal{I}$-confluence [2].

To define $\mathcal{I}$-confluence, we must first define what an I-valid state and an I-T-reachable state are. A state is said to be I-valid if the invariant in that state is true. As for an I-T-reachable state is a state S_i which, given an invariant I and a set of transactions T (with merge function $\sqcup$), contains a (partially ordered) sequence of transaction and merge function invocations that yields S_i, and any intermediate state generated by transaction execution or merge invocation is also I-valid. These previous states are called ancestor states.

We can now introduce the definition for the $\mathcal{I}$-confluence property:

Definition 2 ($\mathcal{I}$-confluence)**.** *A set of transactions T is $\mathcal{I}$-confluent with respect to invariant I if, for all I-T-reachable states D_i, D_j with a common ancestor state, $D_i \sqcup D_j$ is I-valid.*

Informally, an object is $\mathcal{I}$-confluent with respect to an invariant if all replicas of the object are guaranteed to preserve the invariant, even if the different replicas can be modified or merged simultaneously.

Using the definition of $\mathcal{I}$-confluence, the following theorem was derived:

Theorem 1. *A globally I-valid system can execute a set of transactions T with coordination-freedom, transactional availability, convergence if and only if T is $\mathcal{I}$-confluent with respect to I.*

Theorem 1, states that if the operations in an application are $\mathcal{I}$-confluent, the database can execute the operations correctly without requiring coordination. Thus, $\mathcal{I}$-confluence can be used to define which operations need to be coordinated.

Next, we present some static analysis tools, some of which use the $\mathcal{I}$-confluence definition to derive conflicting operations.

2.3.1 Indigo

Indigo [5] is a middleware system designed to provide explicit consistency on a causally-consistent geo-replicated data store. It proposes a three-step methodology for deriving a safe version for applications. In the first step, the static analysis tool is used to determine which operations can be performed safely without coordination. The unsafe operations, i.e., the operations that can lead to invariant violation if executed concurrently, are referred to as I-offender sets. Once the I-offender sets are identified, programmers must decide how to handle such operations, with two alternatives available to them: violation avoidance and invariant repair. Finally, the application code is instrumented with the appropriate middleware library calls.

For the analysis, programmers must provide the model of the effects of operations (postconditions) in the form of annotations that indicate the changes that each operation makes to the state. This information, in combination with the invariants, enables the derivation of the I-offender sets using a static verification technique. The need for a static verification technique arises from the fact that it is not possible to analise all reachable states and, for each state, all sets of possible operations.

To identify the I-offender sets, the algorithm statically determines the pair of conflicting operations, i.e., the operations that may violate the invariants if executed concurrently.

Definition 3 (Conflicting operations)**.** *Operations $op_1, op_2, ..., op_n$ conflict with respect to invariant I if, assuming that I is initially true and the preconditions for op_1 and op_2 to produce side effects are initially true, the result of substituting the postconditions of both operations into the invariant is not a valid formula.*

Indigo uses Z3, a Satisfiability Modulo Theory (SMT) solver to check which operations are conflicting. First, it checks whether the operations lead to opposite postconditions, which would violate the invariant since a predicate cannot have two different values at the same time. To this end, it checks whether the operations are self-conflicting, i.e., operations that violate the invariant when performed simultaneously with different arguments and then, for all pairs of distinct operations, it checks whether they can lead to opposite effects. The remainder of the analysis tests the effects of concurrent execution of pairs of operations on the invariant.

After computing the I-offender sets, the programmer must choose between invariant repair and violation avoidance.

Invariant Repair

The invariant repair technique allows simultaneous execution of operations, even conflicting operations, and repairs the invariant when it is broken. To this end, conflict resolution protocols must include mechanisms to repair invariants in the case of a violation. Indigo supports multiple CRDTs to automate the process of invariant repair.

Violation Avoidance

Violation avoidance is a preventive technique and restricts concurrency to avoid invariant violations. In many cases, it is not necessary to impose coordination on replicas to avoid invariant violation. Indigo offers several reservation techniques for this purpose, which are presented below.

UID generator. The generation of unique identifiers can be easily solved without requiring coordination between replicas by statically partitioning the space of identifiers for each replica. The Indigo approach is to append a replica-specific suffix to the locally generated identifier.

Multi-level lock reservation. The multi-level lock restricts the execution of concurrent operations that could break invariants. There are three types of multi-level locks, each of which provides the following privileges:

- **Shared forbid:** assigns the right to forbid the execution of a particular operation.
- **Shared allow:** grants the right to allow the execution of a specific operation.
- **Exclusive allow:** grants the exclusive right to execute an operation, which means that no other operation can be executed at the same time.

If a replica has a lock, no other replica can have a different lock type at the same time.

This type of reservation allows the invariants of each application to be strengthened. However, in some cases it is possible to maintain invariants while having some concurrency degree.

Multi-level mask reservation. Multi-level masks are used for invariants that are the disjunction of multiple predicates. For these invariants, it is sufficient that one of the predicates be true for the invariant to be satisfied.

A multi-level mask reservation can be viewed as a vector of multi-level locks, where, for example, for the invariant $P_1 \vee P_2 \vee ... \vee P_n$ a multi-level mask with n entries would be created (the i^{th} entry would control the operations that can make P_i false). To execute an operation that makes P_i false, the replica must get a shared allow right for the i^{th} entry and a shared forbid right for an entry where the predicate is true.

Escrow reservation. This technique can be used with numeric invariants. For example, considering an invariant of the form $x \geq k$, the escrow reservation technique allows some decrements to be performed without coordination. Initially, there are $x_0 - k$ rights to execute decrements, since x_0 is the initial value of x. These rights can be dynamically shared between replicas, and when a decrement is executed, the operation consumes n rights. If the replica does not have n rights locally, then try to obtain additional rights from the other replicas. If this is not possible, the operation is aborted. The increment operation, in turn, creates rights for the system.

If the numeric invariant is expressed in the form of $x \leq k$, the increment operation consumes rights, while the decrement operation generates them.

Partition lock reservation. Allows replicas to obtain an exclusive lock on an interval of real values. For example, in a system for booking medical appointments, two concurrent operations scheduling an appointment break the invariant if the appointments overlap in time. However, the invariant is not broken if the appointment times do not overlap. With partition lock reservation, this operation would lock the appointment interval so that other appointments can be scheduled at the same time for different hours.

2.3.2 IPA

IPA [4] is a static analysis tool that identifies conflicting operations and proposes changes to preserve invariants without requiring coordination. Whenever possible, IPA attempts to augment operations with updates that preemptively guarantee the preservation of invariants in the presence of concurrency. However, the additional updates should have no visible effect when no conflicting operations are performed concurrently, and should only be applied when it is necessary to correct an undesirable effect caused by concurrency. For cases where this condition cannot be not met, IPA supports compensation mechanisms that are applied only when violations are detected.

IPA follows the notions of $\mathcal{I}$-confluence and presents a methodology that can be divided into three phases: specification, IPA analysis, and code modification.

IPA, just like Indigo, uses the information about operations and application invariants to perform the analysis. Thus, the first step is to create the application specification, which defines the invariants and the effects of each operation on the system state.

Using the specification and the conflict resolution strategies originally defined by the programmer, the IPA analysis is performed.

The analysis step returns a new specification to the application that contains the selected changes, i.e., both the appropriate conflict resolution rules for each object and the necessary changes to the operations to avoid breaking the invariants. With this information, the programmer must apply the selected options in the application code.

IPA Analysis

This phase is performed by the IPA analysis tool and consists of an iterative process that analyses the specification and the conflict resolution strategies originally defined by the programmer. In each iteration, the tool identifies the pairs of conflicting operations - conflict detection - and also proposes modifications that ensure that the invariants are preserved under concurrent execution - conflict repair. These changes comprise new versions of the operations and appropriate conflict resolution mechanisms. Considering these suggestions, the programmer chooses which conflict resolution protocol he prefers, and a new iteration begins. Iterations continue until no more conflicts are detected.

Conflict Detection. The conflict detection algorithm identifies the pairs of operations that, if executed simultaneously, could violate the invariants of the application. For this purpose, all pairs of operations in the specification are considered.

The conflict detection begins by checking for each pair of operations whether they have opposite effects. For each of the operations that have an opposite effect, the algorithm uses the rule established by the initial conflict resolution protocol to change the predicate value in one of the operations. If no rule exists for the predicate, the programmer is prompted for the rule it pretends to use (*add-wins* or *remove-wins*).

Then the algorithm checks whether the simultaneous execution of two operations breaks the invariant, and it consider their execution in all valid states. It is enough that the invariant does not hold for one of the states for the operations to be considered as conflicting. To efficiently search for a valid execution, IPA uses an SMT solver that employs various optimizations and heuristics to avoid exhaustively testing all cases.

Conflict Repair. The conflict repair algorithm tries to find modifications to the operations and to the conflict resolution rules for the pairs of operations that break the invariants. The purpose of these changes is to guarantee that the invariants are preserved when the operations are executed concurrently.

Numeric Invariants

The numeric invariants are difficult to preserve by modifying operations and are therefore treated with a compensation mechanism. The idea of the compensation mechanism is to check whether the precondition of the operations is true when the operation is performed in the initial replica, and to check whether the invariants are preserved when the operation is replicated. If the invariants do not hold, a compensation is applied.

In a flight booking application where two customers buy the last available seat for a flight at the same time, a possible compensation could be to refund the money to one of the customers.

The compensation mechanisms in IPA are implemented using a CRDT developed for this purpose. If compensation is required, the analysis tool simply detects the violation of the invariant and does not propose any change. The compensation mechanism is entirely up to the programmer.

2.3.3 CISE

CISE consists of a formal proof rule [12] and a static analysis tool that encodes the proof rule [22] and whose goal is to help programmers explore consistency models. With this tool, application developers can verify that their applications maintains invariants on a replicated database with a particular replication protocol.

CISE assumes an asynchronous consistency model, where client operations are executed immediately on the local replica (called the origin replica) and the response is returned to the client immediately. After that, the replica sends to the other replicas a function that defines the operation effector describing the updates to the database state. Upon receiving this message, the replicas apply the effector to their local state. Another assumption of CISE is that the consistency model guarantees causality, which means that the effectors of causally dependent operations must be executed in the same order in all replicas.

The tool automates the proof rule using an SMT solver. When an obligation[2] fails, a counterexample is provided by the tool which allows the developer to understand the problem in order to fix it.

The CISE analysis can be divided into three proof obligations: safety analysis, commutativity analysis, and stability analysis. Safety analysis checks whether the effector of each operation preserves the invariants when applied in any state where the preconditions are verified. In turn, commutativity analysis checks whether all pairs of effectors of all operations commute with each other, i.e., whether executing these effectors in any order leads to the same state, regardless of the initial state in which the effector is applied. Finally, the stability analysis checks whether precondition of operation o is stable under the execution of the effector of any operation o'.

[2]A proof obligation is a theorem stating that a defined property must hold in order for a formal specification to be internally consistent [26].

2.3.4 Hamsaz

Hamsaz [14] is a tool that automates the static analysis process of sequential objects, instantiates protocols, and synthesizes replicated objects. The analysis process allows to determine in a static way the conflicting and dependent operations, information with which it tries to avoid coordination between replicas. To minimize coordination between replicas this tool uses an approach based on a condition for the integrity and convergence of replicated objects called well-coordination. Well-coordination is a condition that requires coordination between conflicting operations and causality between dependent operations. A replicated execution is said to be well-coordinated if the permissibility of the calls in the origin replica is verified (permissibility requires that the invariant holds before and after the execution of the operation), if the conflicting calls are synchronized, and if the dependencies are preserved.

The Hamsaz tool first determines, with a SMT solver, the pairs of conflicting operations and the methods that have dependency relationships. The analysis returns graphs representing the conflict and dependency relationships. The graphs are used to avoid coordination between replicas and to instantiate the protocols that synthesize the replicated objects, which corresponds to the second phase of Hamsaz's approach.

Hamsaz presents two possible protocols for synthesizing the replicated objects. One of them is non-blocking and based on Total Order Broadcast (TOB). The other one is blocking but allows the execution of some conflicting methods without synchronization.

Non-blocking Protocol

With this protocol, the crash of one replica does not prevent the other replicas from progressing. The idea behind the non-blocking protocol is to synchronize the conflicting calls in each set. As mentioned earlier, graphs containing the conflict and dependency relationships are returned as the result of the analysis. To minimize coordination between replicas, this protocol uses the graph representing the conflict relations and finds the maximal cliques[3] of this graph. The methods belonging to each of the maximal cliques found must be synchronized with each other, while methods from different maximal cliques can be executed simultaneously.

To deliver the method calls in the same order in all replicas, a variant of the total order broadcast protocol is used. Since methods need only be ordered relative to the other methods in the clique, one instance of TOB is used for each maximal clique. If a method belongs to more than one clique at a time, a method call must be ordered for all cliques to which it belongs. Method calls are broadcast to each of the TOB instances to which they belong, and are not executed until they have been sorted and delivered by all replicas to ensure that all replicas perform operations in the same order. However, if the TOB instances are not properly coordinated, deadlocks can occur. Thus, to prevent deadlocks, a variant of TOB is used, the Multi-Total Order Broadcast Protocol (MTOB).

[3]A clique is a subset of vertices where each two distinct vertices are adjacent. In turn, a maximal clique is a clique that cannot be extended by the inclusion of another adjacent vertex.

Blocking Protocol

When a non-blocking algorithm is faced with a pair of conflicting methods, both methods must be synchronized. The blocking protocol, in turn, avoids using synchronization and, in the same situation, only synchronizes one of the methods. Suppose there are two conflicting methods m and m' and it is intended that calls to method m are executed without synchronization. When calls to method m' are executed, the other replicas block the execution of calls to m (in this way calls to m are not accepted). However, some replicas may have already received calls from m that they have not yet propagated. These operations must be propagated and executed, to guarantee that the same set of calls of m has been executed on all replicas, and then m' can be safely executed. Finally, the calls to m are unblocked. To minimize the required synchronization, the protocol finds the minimum vertex cover[4] of the conflict graph and synchronizes only when methods in the cover are invoked.

To optimize the responsiveness of the system, methods can be assigned with weights that are inversely proportional to how often they are invoked. This would mean that the methods that block the system are called less often, improving system performance.

With this protocol, the replicas wait for each other, so that the crash of one replica can prevent the progress of the other replicas.

Dependency Tracking Protocol

The above description of the protocols assumed that the operations are independent of each other, and only described the behaviour when the operations conflict with one another. In this section, the behaviour of the protocols is presented considering dependency relationships.

If a call does not require synchronism, it is simply transmitted and executed by the replicas immediately upon arrival. However, if the method call has dependencies, these should be tracked in the originating node and transmitted along with the call. The nodes should not execute the call until the dependencies are applied.

On the other hand, if synchronization is required, it is used a protocol that is the inverse of the classic atomic protocol. After the call is synchronized across all replicas, it may or may not be permissible on different nodes. If the call is permissible on one of the nodes, that node propagates the dependencies to the other nodes so that they all become permissible. This means that the call will be aborted only if all replicas vote to abort.

2.3.5 Blazes

Blazes [1] is a cross-platform program analysis framework that identifies program locations that require coordination to ensure consistent execution, and automatically synthesizes application-specific coordination code that can significantly improve system performance that would otherwise require global coordination.

[4] A minimum vertex cover of a graph is a smallest subset of the vertices such that each edge has at least one endpoint in the cover.

$$(1)\ \frac{\{\textbf{Async}, \textbf{Run}\} \qquad \textbf{OR}_{gate}}{\textbf{NDRead}_{gate}}$$

$$(2)\ \frac{\{\textbf{Async}, \textbf{Run}\} \qquad \textbf{OW}_{gate}}{\textbf{Taint}} \qquad (3)\ \frac{\textbf{Inst} \qquad \textbf{CW}, \textbf{OW}_{gate}}{\textbf{Taint}}$$

$$(4)\ \frac{\textbf{Seal}_{key} \qquad \textbf{OW}_{gate} \qquad \neg\ \text{compatible}(gate, key)}{\textbf{Taint}}$$

Figure 2.1: Reduction rules for component paths.

$$\text{protected}(\textbf{NDRead}_{gate}) \equiv \forall l \in \texttt{Labels}\ \ l = \textbf{NDRead}_{gate} \vee \exists key\ l = \textbf{Seal}_{key} \wedge \text{compatible}(gate, key)$$

$$\frac{\textbf{Taint} \in \texttt{Labels}}{Rep\ ?\ \textbf{Diverge} : \textbf{Run}}$$

$$\frac{\exists gate \exists \textbf{NDRead}_{gate} \in \texttt{Labels} \qquad \neg\text{protected}(\textbf{NDRead}_{gate})}{Rep\ ?\ \textbf{Inst} : \textbf{Run}}$$

Figure 2.2: Reconciliation rules.

Blazes can be automatically applied to existing platforms based on distributed stream or dataflow processing. Moreover, Blazes takes advantage of declarative programming languages once programmers do not need to annotate these cases. However, when programs are developed using non-declarative languages, programmers must annotate paths through components and input streams. Blazes uses this information to derive labels for the output streams of each component. Blazes views a stream as a collection of unbounded and unordered messages connecting components, which in turn are logical computation and storage units that process input streams and generate output streams.

There are 4 annotations for components: CR, CW, OR_{gate}, OW_{gate}. CR indicates that a path through a component is confluent and stateless. That is, it produces a deterministic output regardless of the order of the inputs, and these inputs do not change the state of the components. CW indicates a path that is confluent and stateful. The annotations OR_{gate} and OW_{gate} denote non-confluent paths that are stateless and stateful, respectively. The *gate* is optional and consists of a set of attribute names that specify the partitions of the input streams over which the non-confluent component operates.

The programmer may also specify stream annotations to describe the semantics of the streams. However, these are not mandatory. The annotation **Rep** indicates that the stream is replicated and $\textbf{Seal}_{key}$ denotes that the stream is punctuated on the *key* subset of the stream attributes.

The other stream labels are derived by Blazes for the output streams from the component annotations and the input stream labels. **Taint** and $\textbf{NDRead}_{gate}$ are internal labels used by the analysis system. **Taint** indicates that the internal state of the component may be corrupted by unordered inputs, and $\textbf{NDRead}_{gate}$ indicates that the output stream may temporarily have non-deterministic content. The **Async** label corresponds to streams with deterministic content, where the order may be different in different runs or in different stream instances. Streams marked as **Run** may have cross-run non-determinism, where a component generates different output stream content in different runs over the same inputs. **Inst** exhibits cross-instance non-determinism, that is, replicated instances of the same components in the same run generate different output stream content over the same inputs. Finally, the **Diverge** label may exhibit persistent divergence across replicas. Some services may tolerate temporary inconsistencies between streams, but permanent divergence between replicas is never desirable.

Blazes uses component and stream annotations to determine whether a given data flow is guaranteed to produce deterministic results; if this guarantee cannot be made, the program is augmented with coordination code. The analysis process can be divided into two steps: inference and reconciliation. In both steps, the labels are derived according the rules. The inference rules are shown in Figure 2.1 and the reconciliation rules in Figure 2.2.

Inference. For each component whose input streams are labeled, Blazes first performs an inference step for each path through the component. Then, each of the component's output interfaces is associated with a set of inferred stream labels.

For non-confluent components with sealed input streams, the inference step should verify that the component preserves the independence of the sealed partitions. If so, Blazes can guarantee a deterministic result by delaying the processing of the partitions until their entire contents are known.

Reconciliation. The reconciliation step may add more labels, and finally the labels for each output interface are merged into a single label. This output stream becomes an input stream for the next component in the data flow, and so on until all output streams are labeled.

Based on the analysis results, Blazes automatically generates code that prevents consistency anomalies with minimal coordination by restricting the way messages are delivered to specific components. Whenever possible, Blazes tries to avoid global coordination by using a seal-based strategy. Otherwise, it enforces the entire message delivery order for those components.

Sealing Strategies

If the programmer has provided a seal annotation $\textbf{Seal}_{key}$ that is compatible with the non-confluent component annotation, we can use a synchronization strategy that avoids global coordination. Once a component never combines the inputs of different partitions, the order in which the component learns about the partitions, their contents, and the corresponding seals has no effect on the output.

The protocol used to implement the sealing strategies must allow the consumer to determine when the contents of the partition are complete. In this way, the consumer must (i) participate in a protocol with each of the producers to ensure that the partition generated by that producer is complete, (ii) and perform a vote to ensure that it has received the data from each of the producers.

Once the consumer has determined that the contents of the partition are complete and therefore immutable, it can process the partition without coordination.

2.3.6 Repliss

Repliss [28] is a verification tool that uses a technique that augment an existing sequential programming language with primitives for concurrent interaction with a highly available database, combining axiomatic consistency models (used to formalize database semantics using event graphs) with operational semantics (used to formalize programming language semantics). This combination allows the verification problem of a distributed and replicated program to be reduced to the verification problem of a sequential program with non-deterministic steps that simulate the possible effects of concurrent invocations.

The tool receives a program and a specification of functional properties as input. The functional properties can use history invariants that describe causal relationships between different calls or the effects of calls on the state of the database. In addition, the user must specify additional invariants to guide the tool. The tool uses random tests to find counterexamples and symbolic execution to verify that there are no errors.

Repliss uses single-invocation semantics, allowing only steps in a single invocation. The effects of different invocations are handled with non-deterministic steps in the rules for starting a procedure invocation and starting a transaction. In these cases, the rules for the semantics of the single-invocation assume an arbitrary state change, the preservation of the invariant, a new state well-formed, and assume that the history of the new state is an extension of the previous history. When a single invocation occurs, the effects of the other concurrent invocations need only be considered at certain points, namely at the following steps: immediately after a procedure invocation, after the end of a transaction, and after a procedure invocation returns to the client.

2.3.7 ECROs

ECROs [10] (Explicitly Consistent Replicated Objects) are Replicated Data Types (RDTs) derived from sequential data types based on a specification that declares application semantics in terms of invariants over replicated state. The goal of an RDT is to provide the same interface as the corresponding sequential object with embedded mechanisms to reinforce the correctness of the application.

Unlike the other approaches where conflicting operations are coordinated due to potential conflicts, ECRO resolves the conflict by imposing an order between the operations, i.e., ECRO attempts to resolve conflicts by reordering the operations rather than coordinating the operations. To identify conflicts and understand their causes, ECROs perform a static analysis of the specification using Ordana.

ECROs differ from the other approaches in that: (i) RDTs are correctly derived from sequential data types and their associated specification, (ii) they avoid coordination between replicas by applying full order in the execution of non-commutative operations, (iii) and they ignore causality between unrelated commutative operations.

To find an appropriate approach for each conflict, ECRO divides the conflicts into four categories. The first category includes the conflicts where the replicas execute non-commutative operations concurrently. In this case, the replicas perform the operations in different order, resulting in different states when the state is replicated to the replicas. In such cases, ECRO orders these operations deterministically in all replicas, which guarantees convergence. The second category includes operations that transition the system state to a state where the execution of a concurrent operation is no longer possible. This type of conflict is resolved by placing the unavailable operations before the operation that brings the transaction to the new state (in which it would be impossible to execute the other operation). The third category, in turn, includes numeric invariants, and the approach consists in coordinating the problematic operations. Finally, the fourth category find the conflicts caused by the mutual exclusion of concurrent operations (i.e., by the generation of unique identifiers). In this case, ECRO also opts for the coordination of operations.

Ordana is a tool that performs a static analysis of the distributed specifications over the sequential object to detect the conflicting operations and find a solution for them. The information inferred by Ordana is used during runtime to serialize the calls in an efficient way, i.e., with as little coordination as possible. Ordana's global analysis can be divided into three types: **dependency analysis**, which reveals dependencies between sequential method calls; **commutativity analysis**, which detects commutativity between concurrent calls; and **safety analysis**, which reveals conflicts and finds solutions by reordering calls locally. In addition to these three analyses, dependency analysis and commutativity analysis are combined to detect **commutativity in sequential calls**. In all three analyses, is used an SMT solver that determines whether the operations are dependent, commutative, or safe for each of the analyses.

Dependency Analysis. The dependency analysis traces potential dependencies between method pairs and determines under what conditions these dependencies occur. The dependency analysis returns the function `dep :: C x C` $\rightarrow \mathbb{B}$, which returns true for two calls if the first call depends on the second, and false otherwise.

Commutativity Analysis. To detect non-commutative method calls, Ordana analyses all method pairs and checks if there are two concurrent method calls that are not commutative. Concurrent calls are commutative if executing operations in different order on different replicas results in the same state. As a result, the function `commutative :: C x C` $\rightarrow \mathbb{B}$ is returned, which returns true if two calls are commutative and false otherwise.

Deriving Sequential Commutativity. Sometimes causal relations do not imply dependencies. For example, if two sum operations on different objects are launched one after the other in a replica, there is a causal relation, and yet they are independent because one does not affect the other.

ECRO allows sequential calls to be executed in a different order in different replicas as long as the operations are commutative and independent, i.e., sequentially commutative. To derive sequential commutativity, Ordana combines dependency analysis with commutativity analysis. Ordana provides a function to check sequential commutativity: `seqCommutative :: C x C` $\rightarrow \mathbb{B}$. This function receives two calls and returns true if the first call is sequentially commutative with the second call, false otherwise.

Safety Analysis. Safety analysis detects pairs of operations that may violate invariants if executed concurrently. To identify methods that are not safe, Ordana analyses the invariants on all the pairs of methods and checks whether a serialization of concurrent calls could violate the invariants. Two functions are returned from the safety analysis: `restrictions :: C` $\rightarrow R$ and `resolution :: C x C` $\rightarrow \{<, >, \top, \perp\}$. The first function receives a call c and returns a set R of restrictions. The restrictions are a set of methods that must be coordinated because they can break the invariant if executed concurrently with c. The second function receives two concurrent calls and returns one of four values: $\top$ if the calls are considered safe, $<$ or $>$ if the calls are not safe but there is a safe serialization, and otherwise $\perp$, which means the calls must be coordinated.

Replication Algorithm

An ECRO is represented as a tuple $\langle \Sigma, \sigma_0, \mathtt{M}, \mathtt{G}, \mathtt{t}, \mathtt{F} \rangle$, where Σ is the set of possible states, σ_0 is the initial state, `M` is the set of methods, `G` is the execution graph of the object, `t` is the current topological ordering of `G`, and `F` is the set of functions generated by Ordana. The graph is a labeled directed acyclic graph where the vertices (`C`) correspond to method calls and the edges (`E`) express the relationships between calls.

Three types of edges can be considered in the graph, which guarantee that any topological arrangement of the graph is a safe serialization that preserves dependencies and guarantees strong convergence:

- **Happened-before edges (hb-edges):** Hb-edges represent the causal relations, and for this reason an edge is added to the graph for each pair of causally related calls. Once the sequentially commutative calls can be executed simultaneously, these calls are ignored.
- **Conflict-order edges (co-edges):** Co-edges guarantee the preservation of invariants in a distributed specification. For each pair of concurrent operations, the algorithm checks if there is a safe serialization, and in the positive case, a co-edge is added between the calls.
- **Arbitration order edges (ao-edges):** Ao-edges are used for concurrent operations that do not commute. These operations must be executed in the same order in all replicas to achieve convergence. Therefore, replicas order the calls deterministically based on the unique identifiers of the calls.

When a local call c is received, a function named `execute_local` is executed. The function first checks if the call is safe, and if not, the call must be coordinated. To do this, the necessary locks are acquired according to the result of the `restrictions` function. Then, the call c is added to the graph and an hb-edge is added between c and all calls that have the happens-before relation but do not sequentially commute with c. Next, the call c is added to the topological order, it is applied to the current state σ, and the stable causal calls are committed (so that the graph does not grow infinitely). Finally, the call is propagated and if locks have been acquired, they are released.

Upon receiving a remote call c, the replicas execute a function called `execute_remote`, which begins by adding c to the vertices and the necessary edges between the vertices. For calls that have a happen-before relationship, an hb-edge is added between c and all calls that do not sequentially commute with c. Concurrent calls that are not safe but have safe serialization use the result of the `resolution` function to determine the direction of the co-edge. For concurrent calls that are safe but do not commute, the function uses the identifiers to determine the direction of the ao-edge to add between them. In the end, a dynamic topological sorting of the graph is performed and the stable causal relations are committed.

2.3.8 Lucy

Lucy [27] is a prototype that was developed with two main goals. The first goal was to develop a system capable of checking confluence. To this end, two different approaches were developed: (i) an interactive decision procedure that requires user interaction, (ii) and merge reducibility. The second goal was to implement a mechanism that minimizes coordination overhead when operations conflict, which in this case is called segmented invariant confluence.

Checking whether an object is invariant confluent is impossible because it would require analysing a myriad of states. To circumvent this impossibility, some systems check whether the object is invariant closed. An object is said to be invariant closed with respect to an invariant, I-closed for short, if the invariant is satisfied after the states are merged. Whittaker et al. have defined that an object $O = (S, \sqcup)$

(distributed object, consisting of a set S of states and a binary merge operator $\sqcup$) is invariant closed with respect to an invariant I, if invariant satisfying states are closed under merge. That is, for every state $s_1, s_2 \in S$, if $I(s_1)$ and $I(s_2)$, then $I(s_1 \sqcup s_2)$.

Theorem 2. *Given an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I, if $I(s_0)$ and if O is I-closed, then O is (s_0, T, I)-confluent.*

From the Theorem 2 it can be concluded that if an object is I-closed, it is also $\mathcal{I}$-confluent. However, if an object is not invariant closed, nothing can be inferred about $\mathcal{I}$-confluence.

Interactive Decision Procedure

In the interactive decision procedure, the user must provide some input. Namely, the user interacts with the algorithm to iteratively eliminate the unreachable states of the invariant. This allows that after the elimination of all unreachable states, the problem is reduced to the invariant closure problem.

In each iteration, the algorithm either concludes that an object is $\mathcal{I}$-confluent or not, or it provides counterexamples to help the user eliminate the unreachable states.

A state is said to be reachable if there is an expression corresponding to a valid system execution that causes the system to transition to that state. For an object to be confluent, it must satisfy the invariant for all reachable states.

Theorem 3. *Consider an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I. If the invariant is a subset of the reachable states (i.e. $I \subseteq \{s \in S \mid reachable_{(s_0,T,I)}(s)\}$), then*

$$(I(s_0) \textit{ and } O \textit{ is } I\textit{-closed}) \iff O \textit{ is } (s_0, T, I)\textit{-confluent}$$

This approach has its limitations, especially because it requires interaction with the user to identify the unreachable states. For example, for large and complex transaction sets, or even if the merge operator is complex, reasoning about unreachable states can become a difficult task.

Merge Reduction

Invariant closure is not necessary for invariant confluence because it does not contain a notion of reachability. Merge reducibility, in turn, covers some cases that invariant closure does not.

As defined by Whittaker et al. an expression $e = t_1(t_2(...(t_n(s))...))$ is merge-free if does not contain any merges. An object $O = (S, \sqcup)$ is merge-reducible with respect to a start state $s_0 \in S$, a set of transactions T, and an invariant I, abbreviated (s_0, T, I)-merge reducible, if for every pair e_1 and e_2 of merge-free (s_0, T, I)-reachable expressions, there exist some merge-free (s_0, T, I)-reachable expression e_3 that evaluates to the same state as $e_1 \sqcup e_2$. That is, if O is merge-reducible, we can replace $e_1 \sqcup e_2$ (which has one merge) with e_3 (which has no merges) to obtain an equivalent expression with fewer merges.

Theorem 4. *Given an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I, if $I(s_0)$ and if O is (s_0, T, I)-merge reducible, the O is (s_0, T, I)-confluent.*

Merge reducibility is a sufficient condition for invariant confluence, but as with invariant closure, it is not straightforward to automatically determine whether an object is merge-reducible.

To circumvent this difficulty, Theorem 5 introduces a sufficient condition for merge-reducibility that is easy to determine automatically. As invariant closure, the criteria presented in the theorem are not necessary for invariant confluence, but can be used to prove that some objects which are considered non-closed are invariant confluent.

Theorem 5. *Given an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I, if the following criteria are met, then O is (s_0, T, I)-merge reducible (and therefore (s_0, T, I)-confluent).*

1. *O is a join-semilattice.*
2. *For every $t \in T$, there exists some $s_t \in S$ such that for all $s \in S$, $t(s) = s \sqcup s_t$. That is, every transaction t is of the form $t(s) = s \sqcup s_t$ for some constant s_t.*
3. *For every pair of transactions $t_1, t_2 \in T$ and for all states $s \in S$, if $I(s)$, $I(t_1(s))$, and $I(t_2(s))$, then $I(t_1(s) \sqcup t_2(s))$.*
4. *$I(s_0)$.*

Segmented Invariant Confluence

As mentioned earlier, the system tries to minimize coordination by using a property called segmented invariant confluence. The idea of this approach is to divide the set of states satisfying the invariant into segments, with a restricted set of transactions in each segment, making each segment $\mathcal{I}$-confluent (even if it is not globally confluent). In this way, servers can execute uncoordinated code within each segment and only need to coordinate when transitioning from one segment to another.

While $\mathcal{I}$-confluence characterizes objects that can be replicated without any kind of coordination, segmented invariant confluence allows replication of non-$\mathcal{I}$-confluent objects with only occasional coordination.

3 Background

3.1 Antidote SQL

Antidote SQL [20, 19] is a NoSQL database system that provides a SQL-like interface to applications and combines both strong and weak consistency to enforce database invariants in the presence of concurrent updates.

It requires programmers to specify (i) the data items that can be modified concurrently; (ii) the concurrency semantics (i.e., the outcome of concurrent updates to the data items); (iii) the integrity constraints (i.e., the database constraints that should be maintained); (iv) and the degree of concurrency that is allowed while enforcing these constraints. With this information, the database system is responsible for efficiently enforcing the defined data model and minimizing coordination overhead.

3.1.1 System Model

Antidote SQL is designed to operate in geo-replicated environments such as cloud infrastructures. The data stored in the database is replicated in multiple data centres, and in each data centre this data is sharded. Each shard is in turn replicated on a small number of nodes.

Applications interact with the database via transactions, where each transaction is a sequence of SQL statements (e.g., `insert`, `update`, `delete`, and `select`).

Transactions are executed under parallel snapshot isolation (PSI) semantics. PSI extends snapshot isolation (SI) for geo-replicated environments, and as such does not allow conflicts between concurrent writes unless the writes are to data types that have a merge mechanism. Under this isolation level, transactions can be executed in different orders across replicas as long as the causal order between transactions is respected, unlike SI, which requires a total order of transactions.

In addition, Antidote SQL extends PSI to enforce integrity, which means that the defined integrity constraints are maintained in all snapshots.

3.1.2 Concurrency Semantics

As mentioned above, in Antidote SQL programmers must define the behaviour of the database in case of possible conflicts due to concurrency, i.e. they must specify the semantics of concurrency. This information must be specified when the tables are created.

```sql
CREATE UPDATE-WINS TABLE Artists (
  Name VARCHAR PRIMARY KEY,
  Country LWW VARCHAR
)
```

Listing 3.1: Table definition in Antidote SQL.

In this section we explain how to define tables in AQL[5], using the example of a system for managing albums and their artists, and we describe all the semantics supported by Antidote SQL. First, we present the semantics for controlling concurrency associated with each table. Then we present the semantics for controlling concurrent accesses that can lead to constraint violations.

Database Model

In Antidote SQL, for each table created, the programmer must specify the concurrency semantics of the table, i.e., how conflicts between concurrent updates and deletions should be resolved, and how concurrent updates in each of the table columns should be handled. Listing 3.1 shows how to define a table in Antidote SQL.

Semantics for update-delete. The semantics for update-delete allows to resolve possible conflicts between insertions, updates and deletions of objects with the same primary key. In the example, the table was defined with the *update-wins* policy. However, programmers can choose between three different semantics:

- **Update-wins:** ignores the effects of concurrent delete operations.
- **Delete-wins:** the delete takes precedence over concurrent updates and the row is deleted.
- **No concurrency:** concurrency between update and delete operations for the same data item is not allowed. This is the default update-delete semantics of Antidote SQL. Thus, if the programmer does not specify a conflict resolution policy for the table (with the `UPDATE-WINS` or `DELETE-WINS` token), concurrent inserts and deletes are not allowed.

To implement the *update-wins* and *delete-wins* semantics, a hidden column (visibility column) is used in each row to indicate whether the row has been deleted or not. Since it may be necessary to resolve conflicts, Antidote SQL does not support permanent deletion of records from the database. Instead, it uses the visibility column to determine the visibility of the record to the end user. The visibility column is implemented using a multi-value register CRDT that stores visibility tokens. The visibility tokens are assigned according to the operation performed: a delete operation stores a `D` in the register, while an insert/update operation assigns a `I` to the register.

Depending on the semantics used, the visibility column is used to determine whether the row has been deleted or not. In *update-wins* semantics, a row is considered deleted if there is no `I` in the visibility column. For *delete-wins* semantics, a row is considered deleted if one of the values in the visibility column is `D`.

[5]AQL is the SQL-like interface provided by Antidote SQL.

The *no concurrency* semantics is implemented using multi-level locks (MLLs) [13]: insert/update operations need a shared lock on the row's primary key to be executed; on the other hand, delete operations must acquire an exclusive lock on the primary keys of the rows to be deleted.

Semantics for update-update. The update-update semantics allow to specify which attributes of an object can be updated simultaneously. These semantics are defined for each of the table columns, and there are four possible semantics:

- **Last-writer-wins:** the value of the last update is kept according to a global order.
- **Multi-value:** the database stores all values from the concurrent updates.
- **Additive:** the values from all updates are combined into one numeric value.
- **No concurrency:** concurrent updates in the same columns of the same object are not allowed. Therefore, the transaction must acquire an exclusive lock before being executed. An exclusive lock blocks all other update operations for the same column of the object. After the transaction completes, the lock is released and the blocked operations are executed. Note that simultaneous operations on the same column of different objects do not need to be coordinated.

Table 3.1 summarizes for each of the update-update semantics the modifier used in the table definition and the CRDT used to implement the corresponding concurrency semantics.

Table 3.1: CRDTs supporting the different update-update semantics.

	AQL Modifier	**Implementation**
Last-writer-wins	`LWW`	LWW Register CRDT
Multi-value	`MULTI-VALUE`	Multi-value Register CRDT
Additive	`ADDITIVE`	PN-Counter CRDT/Bounded Counter CRDT

In the example of Listing 3.1, the *Name* column has no conflict resolution policy associated with it, so the *no concurrency* semantics is applied. For the *Country* column, it was chosen the last-writer-wins policy.

When defining columns, the programmer also have to specify the data type of the column. Antidote SQL supports four data types: `VARCHAR` for general text/string columns; `INTEGER/INT` for integer columns; `BOOLEAN` to assign boolean values to columns; and `COUNTER_INT` for integer counters.

Integrity Constraints

The main concern of Antidote SQL is to enforce database invariants during concurrent database updates. In this section we explain how the programmer can define invariants and integrity constraints for a table. The mechanisms used by Antidote SQL to preserve invariants and handle these constraints are explained in detail in Chapter 4.

```sql
CREATE UPDATE-WINS TABLE Albums (
 Title VARCHAR PRIMARY KEY,
 Year LWW VARCHAR,
 Likes COUNTER_INT CHECK (Likes ≥ 0),
 Artist VARCHAR FOREIGN KEY UPDATE-WINS REFERENCES Artists(Name) ON DELETE CASCADE
)
```

Listing 3.2: Definition of Integrity Constraints in Antidote SQL.

Listing 3.2 shows the definition of the *Albums* table, where all types of constraints supported by Antidote SQL have been expressed: primary key constraints, check constraints, and foreign key constraints.

The **primary key constraint** expressed in the *Title* column, is used to guarantee the uniqueness of the primary key, i.e. to ensure that there is no more than one row with the same primary key in the table.

Check constraints allows to set conditions that the values in the columns must satisfy. In the example, it was defined a check constraints on the *Likes* column, specifying that an album must not have a negative number of likes.

Finally, a **foreign key constraint** was defined in the *Artist* column. These constraints are used to prevent the relationship between the tables from being destroyed during concurrent operations. To define a foreign key, the programmer must specify the name of the referenced column and table (AQL supports references only to primary keys) and the conflict resolution policy to resolve any referential integrity issues. Antidote SQL supports three concurrency semantics for foreign key constraints: *update-wins*, *delete-win*s, and *no concurrency*.

In the example, the *Artist* column references the *Name* column of the *Artists* table (which was presented in Listing 3.2) and the foreign key policy chosen was the *update-wins* policy.

When defining a foreign key constraint, the programmer can specify the behaviour of the record when its parent records are deleted by using the notation `ON DELETE CASCADE`. This notation states that a record must be deleted if its parent record is also deleted. If the `ON DELETE CASCADE` notation is omitted, a parent record cannot be deleted if one or more records exist that reference it.

3.2 VeriFx

VeriFx [9] is a high-level programming language that allows to define system's safety properties and automatically verify them using an SMT solver. VeriFx allows programmers to develop replicated data types (RDTs) that once verified can be translated into languages such as Scala and JavaScript.

VeriFx is a Scala-like programming language that combines both object-oriented and functional paradigms and has built-in collections for tuples, sets, maps, vectors, and lists. These collections are immutable, which means that they return an updated copy of the object when changes are made.

Developers can create their own RDTs by composing collections and/or other RDTs. To validate their implementation, they must express the correctness properties they wish to validate in the form of proof constructs that are automatically checked. For each proof, VeriFx derives proof obligations

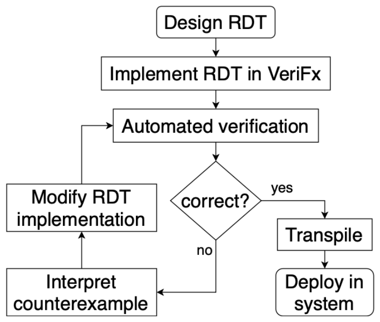


Figure 3.1: Workflow for developing RDTs (taken from [9]).

and discharges them using the Z3 [8] SMT solver. This solver try to determine (automatically) whether a given formula is satisfiable or not.

The development of RDTs in VeriFx is an iterative process represented in Figure 3.1. The programmer begins by implementing the RDT in VeriFx, which automatically checks it without requiring a separate formalization. If the implementation is not correct, VeriFx returns a counterexample. Then the programmer must analyse and interpret the counterexample to understand the error in its implementation, in order to correct it. After correcting the RDT, the implementation is checked again and the process repeats until the implementation is validated. Once correct, VeriFx transpile the RDT implementation into a language such as Scala or JavaScript and the RDT can be integrated into the system.

In order to simplify the development of distributed systems that use replicated data, VeriFx provides a library for automatic implementation and verification of CRDTs. This library facilitates the development of CRDTs and provides the necessary proofs to prove the correctness of a CRDT. However, it is always possible for developers to define their own proofs, for example, to verify invariants.

This library supports several families of CRDTs, including state-based and operation-based.

State-based CRDTs

To implement a CvRDT, the programmer can use the trait that is depicted on Listing 3.3. The programmer must extend the trait, specifying the type of CRDT he is implementing. He must also provide an implementation for the `merge` and `compare` methods. For the `merge` method, the programmer must specify how to join two states (the state of the local replica with the incoming state) by computing the least upper bound (LUB) between them. For the `compare` method, it must be defined how the state `this` is less than or equal to the state `that`.

All other methods of the trait have a default implementation, but may be overridden by the concrete CRDT. The `reachable` method allows to define the reachable states, i.e. the states that can be reached through an initial state by applying only the operations supported by the CRDT and it must return true if the state is reachable, and false otherwise. For instance, for a counter whose initial state is 0 and which supports only increment operations, all states in which the counter has a value greater than or equal to 0 are reachable states. In contrast, states in which the counter has a negative value are unreachable. The `compatible` method, in turn, defines whether two states are compatible,

```
trait CvRDT[T <: CvRDT[T]] {
 def merge(that: T): T
 def compare(that: T): Boolean
 def reachable(): Boolean = true
 def compatible(that: T): Boolean = true
 def equals(that: T): Boolean = {
  this.asInstanceOf[T].compare(that) && that.compare(this.asInstanceOf[T])
 }
}
```

Listing 3.3: Trait for the implementation of CvRDTs in VeriFx (taken from [9]).

```
trait CvRDTProof[T <: CvRDT[T]] {
 proof mergeIdempotent {
  forall (x: T) { x.reachable() =>: x.merge(x).equals(x) } }
 proof mergeCommutative {
  forall (x: T, y: T) {
   (x.reachable() && y.reachable() && x.compatible(y)) =>:
   (x.merge(y).equals(y.merge(x)) && x.merge(y).reachable()) } }
 proof mergeAssociative {
  forall (x: T, y: T, z: T) {
   (x.reachable() && y.reachable() z.reachable() &&
   x.compatible(y) && x.compatible(z) && y.compatible(z)) =>:
   (x.merge(y).merge(z).equals(x.merge(y.merge(z)))
   && x.merge(y).merge(z).reachable()) } }
 proof equalityCheck {
  forall (x: T, y: T) {
   (x.reachable() && y.reachable() && x.compatible(y)) =>:
   (x.equals(y) == (x == y)) } }
}
```

Listing 3.4: Trait for the verification of CvRDTs in VeriFx (adapted from [9]).

that is, whether the two states represent different replicas of the same CRDT object. Consider a system in which there are replicas of different counters, each of which has a unique identifier. In this case, two states are compatible only if they both represent the same counter, simply put, if the counter identifier is the same in both states. Finally, the `equals` method defines the equivalence of states. By default, equivalence is derived using the `compare` function.

For verification of CvRDTs implementations the CvRDTProof trait shown in Listing 3.4 can be used. This trait is defined for a parameter of type `T` which must be a CvRDT (the CvRDT defined by the programmer) and defines tests to verify that the merge function is idempotent, commutative, and associative, i.e., the properties required to guarantee that a CvRDT converges.

In addition to the presented traits, the CRDT library also provides traits with the correctness proofs for polymorphic CvRDTs expecting 1, 2 or 3 type arguments.

Operation-based CRDTs

Listing 3.5 shows the trait that must be extended by programmers to implement CmRDTs. When extending this trait, the programmer must specify the concrete type of the supported operations, the exchanged messages, and the CRDT type itself. The implementation of the `prepare` and `effect` methods is also required.

```
trait CmRDT[Op, Msg, T <: CmRDT[Op, Msg, T]] {
 def reachable(): Boolean = true
 def compatible(x: Msg, y: Msg): Boolean = true
 def compatibleS(that: T): Boolean = true
 def enabledSrc(op: Op): Boolean = true
 def prepare(op: Op): Msg
 def enabledDown(msg: Msg): Boolean = true
 def effect(msg: Msg): T
 def tryEffect(msg: Msg): T = {
  if (this.enabledDown(msg))
   this.effect(msg)
  else
   this.asInstanceOf[T] // operation executes only if its downstream precondition is true
 }
 def equals(that: T): Boolean = this == that
}
```

Listing 3.5: Trait for the implementation of CmRDTs in VeriFx (taken from [9]).

The `prepare` method prepares a message to be sent to all replicas with the effector function. The `effect` method in turn applies an incoming message (downstream) and returns the updated state. The trait also provides the `tryEffect` method, whose implementation applies operations only if the downstream precondition is met.

The remaining operations of the trait allow the encoding of some assumptions required to check some CRDTs, and although they have a default implementation, they can be overridden. The `reachable` method allows the definition of reachable states, as in state-based CRDTs. The method `compatible` checks whether two operations can be invoked concurrently. In the `compatibleS` method, the programmer can define whether two states are compatible. For example, if several replicas generate unique identifiers by adding the replica prefix to a sequential identifier, two states are compatible if the identifiers of the replicas are different. The `enableSrc` method checks whether the specified operation can be generated in this state. For example, in an application that manages a bank account whose balance is less than the amount the operation is trying to move, it does not make sense to generate the effector function and send it to all replicas. The `enabledDown` method in turn checks whether the specified operation can be applied (downstream) to the state. Finally, the `equals` method checks whether two states are equivalent.

Similar to state-based CRDTs, the VeriFx CRDTs library also provides traits for the verification of operation-based CRDTs. These traits define a general correctness proof that verifies that all operations are commutative.

4 Invariants Mapping

This chapter presents a study of the different types of application invariants in the Antidote SQL database system. For each type of invariant, we present a brief description, a real-world example, and the mechanisms available to ensure that the invariant is maintained. We also present, to some of these classes, extensions to the existing mechanisms that allow to explore a higher concurrency degree.

This chapter first introduces the types of invariants supported by Antidote SQL (i.e., uniqueness, referential integrity, and numeric invariants). For the invariants that are not supported by Antidote SQL, we have proposed mechanisms that can be implemented in this database system in order to facilitate a possible extension of the system.

This study shows that is very important to obtain a concrete mapping between the types of invariants and the mechanisms to handle each type. This mapping is important to better understand which mechanisms can be used in the different cases. Additionally, since we have analysed invariants that are not supported by Antidote SQL, this mapping facilitates a possible integration of new invariants into Antidote SQL.

4.1 Uniqueness

Uniqueness is a class of invariants that includes all the constraints regarding the uniqueness of a given attribute and is often used to denote primary keys or identifiers.

Antidote SQL supports two different approaches to deal with primary key constraints. The former consists in disallowing concurrent insertions, which can be achieved by selecting the *no concurrency* semantics in one of the columns in the table (other than the column representing the primary key). The second approach, can be applied when all columns, except the column representing the primary key, have a conflict resolution policy. In this case Antidote SQL allows multiple concurrent insertions and the final value of each column is determined according to the defined concurrency semantics. Figure 4.1 shows this situation. In this example each column has a conflict resolution policy: *Country* follows the last-writer-wins policy while *Age* follows the first-writer-wins policy. As the *Name* column is the table primary key, it does not require a conflict resolution policy. Consider that two artists with the same primary key are inserted simultaneously. Then, after replicas synchronize, the final state of the database is defined by the resolution policies of each column, so the *Country* column gets the last

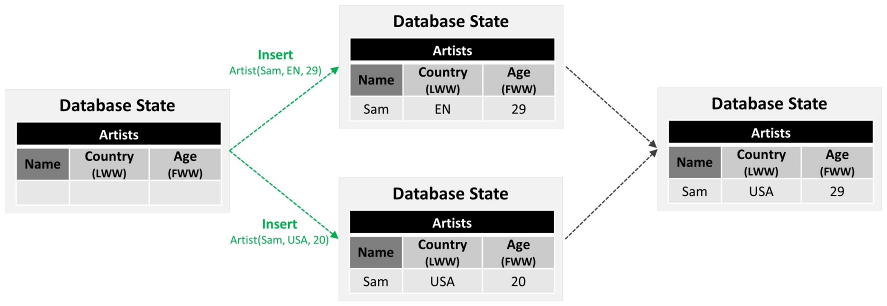


Figure 4.1: Concurrent inserts on Antidote SQL.

inserted value according to a global order, while the *Age* column gets the first value according to the same global order.

Depending on the application, unique identifiers may be defined by users or by the database itself. The identifiers generated by the database can be distinguished in two subclasses: sequential identifiers and unique identifiers. The difference between these two types is that the former must guarantee that identifiers are sequential (i.e. if the identifier of the last element inserted is 100, the element inserted immediately after must have the identifier 101, and so on), whereas the latter only has to guarantee uniqueness. Thus, when using sequential identifiers, a strong consistency mechanism must be used to guarantee that identifiers are sequential, i.e. any insertion must be coordinated. However, most applications that require a unique identifier do not need it to be sequential. As unique identifiers do not need to maintain an order, they can be generated without coordination by attaching to a sequential identifier a prefix associated with the replica that generated it, or by splitting the identifier domain space per replica in some other way.

Antidote SQL only supports uniqueness for primary keys. However, it is common in many systems to have secondary keys, also known as alternate keys. These attributes are unique in the database and identify the object unequivocally and can be used, such as primary keys, to locate specific data. For example, in a table with information about students in a college, where the primary key is the student number there are several secondary keys such as email, citizen card number, or NIF. In this example all the secondary keys are already unique for each person. For this reason, the use of locks would not have much impact on the performance of the system, as it is unlikely that two different rows with the same NIF would be inserted, unless by mistake.

4.2 Referential Integrity

Referential integrity is used to establish dependency relationships between objects, and it is imperative that these relationships are not destroyed during the execution of the applications. In Antidote SQL, these relationships are expressed in terms of foreign keys, and the constraints imposed by these relationships are not $\mathcal{I}$-confluent [2].

To illustrate this type of invariant and its violation, consider the example in Figure 4.2. In this example, the database contains two tables, *Artists* and *Albums*, where *Albums* has a foreign key in the *Artist* column that points to the *Artists* table. Initially, the database has one artist and two concurrent transactions are executed, one of which deletes the artist and the other adds an album to that artist.

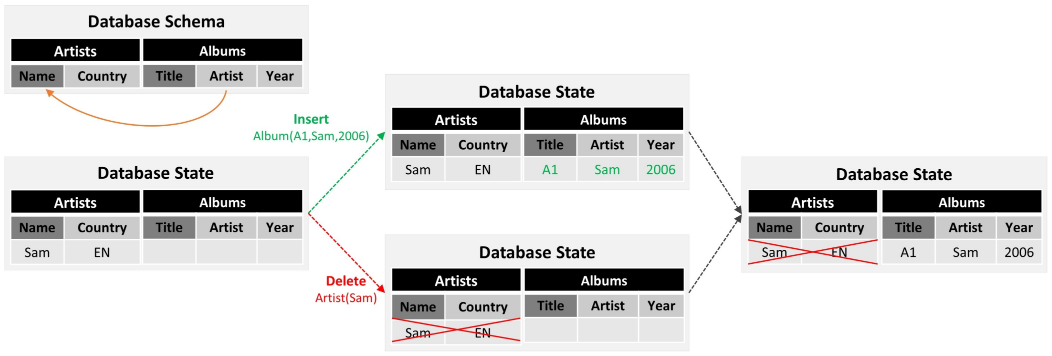


Figure 4.2: Example of foreign key constraint violation.

However, after merging the two new states, the foreign key constraint is violated because the database contains an album for an artist that no longer exists in the database.

Antidote SQL supports three concurrency semantics to deal with foreign key constraints:

- **Update-wins:** if one operation deletes row r and another operation inserts/updates a row that references row r, the deletion has no effect on the state of the database.
- **Delete-wins:** if one operation deletes row r and another operation inserts/updates a row that references row r, the insert/update operation is not reflected in the final state of the database.
- **No concurrency:** concurrent operations that violate the referential constraint are not allowed. The system uses MLLs to control concurrent accesses. To delete a row in the parent table, it is necessary to acquire an exclusive lock, while to insert/update a row in the child table, it is necessary to acquire a shared lock. This allows insert/update operations can proceed concurrently, which is expected to allow a high degree of concurrency in most applications, where deletes are expected to be rare.

Similar to the update-delete semantics (see Chapter 3), the foreign key constraints implements the *update-wins* and the *delete-wins* conflict resolution policies using a visibility column.

***Update-wins* semantics.** To implement the *update-wins* semantics, in addition to the `D` and `I` flags, there is also the `T` flag. This flag indicates that a new object referencing the row has been added. Under this semantics, the `T` and `I` flags are considered stronger than `D` and therefore the element is considered deleted only if the only value in the visibility column is `D`. Figure 4.3 illustrates the behaviour of this semantics. Initially, the database is composed by one artist (*Sam*) and one album (*A0*), and two concurrent operations are issued: one inserts a new album of the existing artist, the other deletes the artist. After merging the two states, *Sam* remains in the database since at the time of the insertion of the new album this record is marked as touched.

When the programmer defines the foreign key policy to *update-wins*, the `T` flag takes precedence over all other flags. That is, if a parent table is set to *delete-wins* and the foreign key policy to *update-wins*, and the visibility column contains a `D` and a `T`, the record exists in the database. Otherwise there would be a foreign key violation.

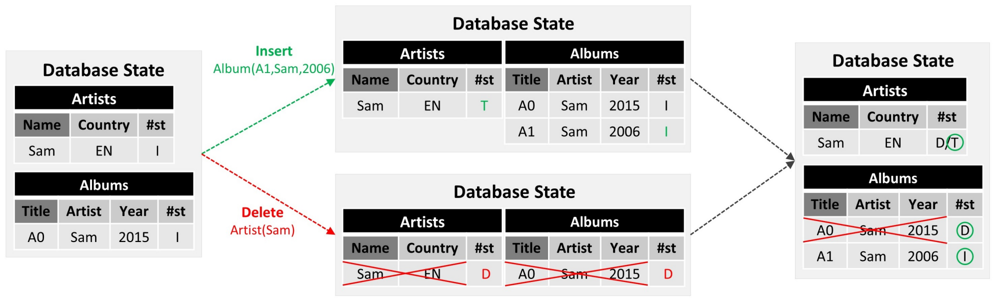


Figure 4.3: *Update-wins* semantics. (taken from [20]).

***Delete-wins* semantics.** In *delete-wins* semantics, an element that has the D flag in its visibility column must be deleted from the database, as well as all elements that reference it (otherwise there would be a referential integrity violation). However, the inserted child record may not be known at the time of deletion. As can be seen in Figure 4.4, album *A1*, after replicas synchronise, only has the I flag in its visibility column.

To solve this problem, the read operation has been extended to check whether the parent row has been deleted or not. However, this verification cannot be limited to checking the visibility column of the parent record. Otherwise, we have no guarantee that a deleted child element is visible because an element in the parent table was re-inserted. In this way, in addition to the visibility flags, the *delete-wins* semantics also needs to maintain the version of the objects. The version is an integer associated with each record that is incremented when it is inserted an object that already belonged to the database. Each child maintains the version of its parent. For instance, consider a register with primary key x and version 1 (version 1 means that the object was inserted for the first time) and another register in the child table that refers to version 1 of the object with primary key x. If an operation that deletes the element with primary key x from the database is executed (the operation deletes all its children) and sequentially a new operation inserts a new object with primary key x, this object is stored in the database with version 2. In this way, the child element that would be considered visible if versions were not used is not visible because the version stored by the child record does not match the version in the parent record.

Thus, in this semantics, determining the visibility of a record involves three steps: (i) checking the record's visibility column (the record must not have a D in this column to exist); (ii) checking the version of the parent record (the record exists only if the two versions are the same); (iii) checking the status of the parent record (whether it is visible or not).

The example only cover a scenario where the concurrent operations involve different objects. However, if in addition to the two operations a third operation that updated the *Sam* record was executed, the visibility column would contain a D and an I. In this case, the conflict is resolved by the table's conflict resolution policy, so if the policy was *update-win*s the object would be kept in the database, while if the policy was *delete-wins* the opposite would occur and the object would be deleted from the database.

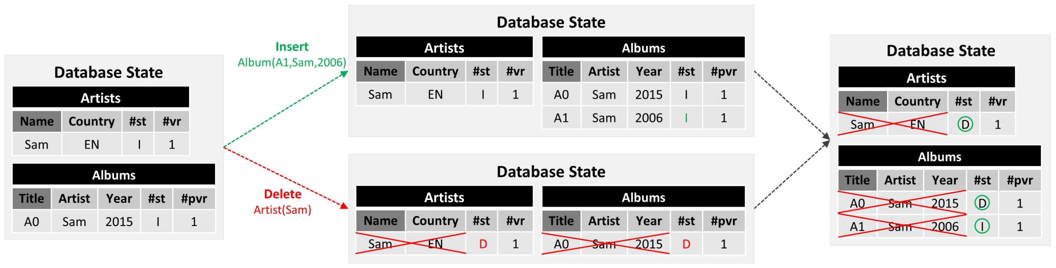


Figure 4.4: Delete-wins semantics. (taken from [20]).

4.3 Numeric Invariants

Numeric invariants are usually associated with attributes that are manipulated by increment or decrement operations, and where there must be some control over the values of the column/attribute, since there is a lower or upper bound. For instance, restricting the maximum number of movie tickets sold given the maximum allowed capacity, or guaranteeing that the stock of a product does not go bellow zero, are two real-world examples of numeric invariants.

Consider an example in which a lower bound (i.e., $x \geq k$) is defined. While increments can be executed immediately without synchronization, since these operations pose no risk to the preservation of the invariant, the same does not apply when the operations are decrements. Consider that the value of x is equal to 1 at a given time of the program execution and that two operations decrement the value of x in one unit simultaneously. After these operations are propagated, the value of x would be negative, resulting in a violation of the invariant. The same situation can occur with upper bounds, but in this case, it is the increment operations that may cause a violation. For this reason, numeric invariants are not $\mathcal{I}$-confluent [2].

To allow a higher degree of concurrency, a special type of CRDT can be used: the *bounded counter* [3]. The bounded counter is a CRDT based on the escrow technique [24] that manages information to guarantee the preservation of numeric invariants without requiring coordination between operations in most cases.

The key idea is to assign a certain number of rights to each replica (the total number of rights in the system must be equal to the difference between the value of the counter and its limit). For example, for a counter x whose limit is $x \geq k$ and which is initially x_0, the system generates $x_0 - k$ rights that are distributed among the replicas. The execution of decrements consumes rights (a decrement of n units consumes n rights), and these operations can be executed locally immediately if the replica has enough rights. If the replica's number of rights is insufficient, there are two possible approaches. The former consists of having the replica trying to contact other replicas to obtain enough rights to execute the operation. However, this process can be slow and there is no guarantee that the total number of rights in the system will be sufficient to execute the operation. The second approach is to abort the operation if the replica does not have enough rights locally. For simplicity, Antidote SQL takes this approach. The increment operation, in turn, creates new rights for the system. Thus, if x is incremented by n units, n new rights are generated.

If the bounded counter is defined with an upper bound ($x \leq k$), it is the increment operations that consume rights, while the decrement operations are responsible for generating new rights. Table 4.1

Table 4.1: Management of rights depending on limit type and operation performed.

	Increment(n)	**Decrement**(n)
Column $\leq X$	Generates n rights	Consumes n rights
Column $\geq X$	Consumes n rights	Generates n rights

summarises the way in which rights are generated depending on the operation performed and the type of limit defined.

So far in Antidote SQL it is only possible to define a limit per column. However, a possible solution to support simultaneously a lower and an upper bound for the same column ($k_0 \leq x \leq k_1$) is to use two bounded counters: one to manage the lower bound and another to manage the upper bound.

4.4 Attribute Equality/Inequality

Attribute equality/inequality is a type of invariant used when an attribute must take only a value from a set of possible values (i.e., equality) or must take a value that does not belong to a set of impossible values (i.e. inequality). For instance, in a system that manages the transactions of a bank account, the value of each transaction must be different from 0 ($transactionValue \neq 0$). The transaction value can be negative if it is a debit, or positive if it is a credit, but it can never be 0.

This type of invariant is only used with attributes that can be manipulated exclusively by assignment operations. Therefore, as Bailis et al. have shown [2], these invariants are $\mathcal{I}$-confluent, meaning that there is no risk of breaking the invariant in concurrent transactions.

Although Antidote SQL does not support these types of invariants, they could be easily integrated, since they only require that the condition defined for the column be checked before executing operations that manipulate that column.

4.5 Aggregation Constraints

Aggregation constraints allow to set limits on the size of collections. For example, to define a maximum number of students enrolled in a class (e.g. $nrEnrolled(c) \leq 30$), an invariant of this type is defined. In this example, a limit is defined on the size of a collection of students.

At first glance, aggregation constraints are similar to the numeric invariants introduced in Section 4.3, and just like them, these invariants are not $\mathcal{I}$-confluent. Consider there are 29 students enrolled in a class. If two different students enroll in this class at the same time, and each operation is performed in a different replica, the invariant would be broken after synchronization, since the number of students would be 31.

To take advantage of concurrency while ensuring the correctness of these invariants, we propose that Antidote SQL take an approach similar to that used for numeric invariants. The idea is to keep an additional column (e.g., in a metadata table) to represent and control the limit of the size of the collection. This column would be implemented with a bounded counter so that some operations can be executed without coordination as long as there are enough rights to do so.

However, this approach can lead to problems in specific situations. If two teachers remove the same student at the same time, two new rights would be generated, even though only one student was removed. This would allow the invariant to be violated, because there would be more rights than vacancies in the class.

To work around this problem, the rights cannot be created immediately, but this task must be delegated to a master replica. So if a student is removed from the class, no right is generated. Periodically, the master replica recalculates the rights, generates the missing rights and distributes them among the replicas.

A similar problem occurs when the same student is enrolled in the class twice, as more rights are consumed than vacancies occupied. Apart from the fact that this situation does not pose a risk to the invariant, the rights are eventually replenished by the master replica.

For invariants that set a lower bound on the size of the collection, the same approach is used with the necessary adjustments.

4.6 Aggregation Inclusion

Aggregation inclusion is a type of invariant that is used when there is a column that represents the size of a collection and the programmer wants to guarantee that the operations that add and remove elements from the collection are correctly reflected in the value of that column. This type of invariant can be expressed as $x = collection.size()$.

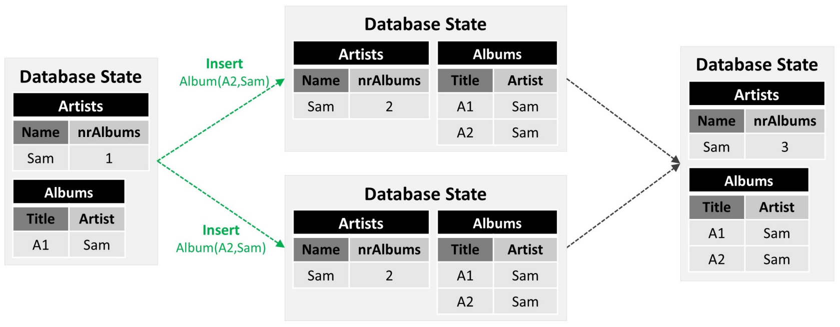
Despite the similarities of this type of invariant with aggregation constraints, the column that keeps the size of the collection cannot be implemented with a counter, since using this data type make the invariant non-$\mathcal{I}$-confluent.

With this approach, the counter would be incremented whenever an element is added to the collection and decremented whenever an element is removed from the collection. However, concurrent transactions can lead to a violation of the invariant. Consider the example in Figure 4.5(a). In this example, two concurrent transactions add album *A*2 to *Sam*. After combining the effects of the two transactions, the database would reach a state where the counter value does not match the actual number of *Sam's* albums, resulting in a violation of the invariant. This problem can also occur for two concurrent removals.

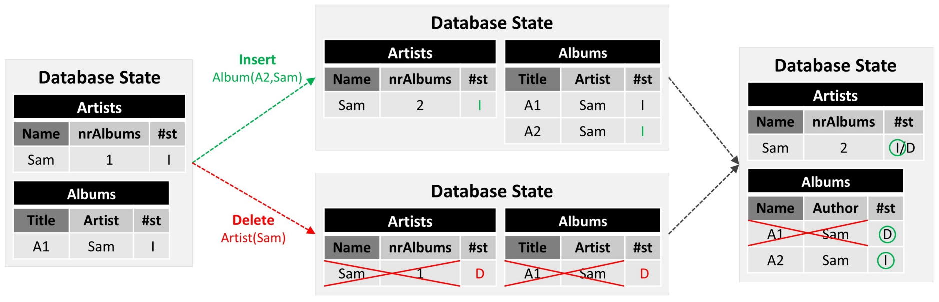
Besides this problem, there is also another issue that is presented in Figure 4.5(b), and that only occurs when there are dependency relationships between database objects (that is, when referential integrity must be maintained) and the defined semantics is the *update-wins*. In this scenario, two concurrent transactions are executed, one that inserts a new album for *Sam* and another that deletes *Sam* from the database. After synchronization, *nrAlbums* would remain 2 even though there is only 1 album in the database.

When the defined semantics for the foreign key constraint is the *delete-wins*, this issue does not occur since the artist would be deleted.

Both problems can be solved if the column *nrAlbums* is implemented with a set with the same conflict resolution policy of the one used to resolve referential integrity. Figure 4.6 presents an example with this solution. A "normal" set is sufficient to solve the first problem and can be used



(a) Result of insertions of the same object concurrently.



(b) Result of concurrent insert of child record and delete of parent record.

Figure 4.5: Problems of using a counter with aggregation inclusion invariants.

for invariants when objects do not have referential integrity relations. However, when referential integrity comes into play, a set CRDT is needed to resolve any conflicts that may arise.

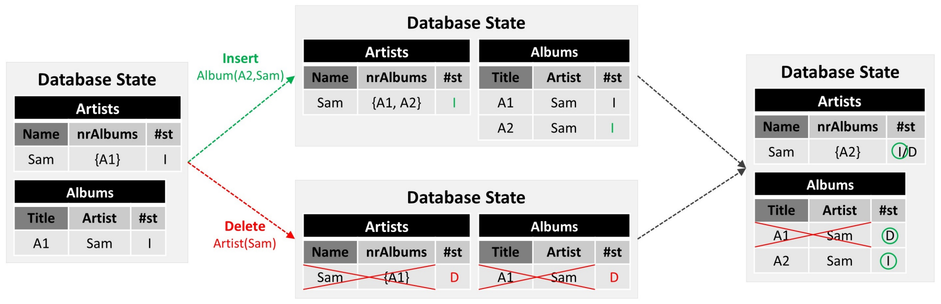


Figure 4.6: Example of a solution using a set CRDT to maintains the "number" of albums.

4.7 Disjunctions

Disjunctions are a type of invariant used to specify that at least one of several conditions must be met. This class of invariants can be divided into three subtypes: disjunction, exclusive disjunction and implication.

In both disjunctions and exclusive disjunctions at least one of the predicates must be true, and their difference is on the number of predicates that can be true. Whereas in disjunction several predicates can be true, in exclusive disjunction only one of them can take this value.

For example, a student having to be enrolled in at least one course is considered a disjunction, because the student can be simultaneously enrolled in several courses. In its turn a student having to be enrolled in one of the existing theoretical classes of each course is an exclusive disjunction because the student can only be enrolled in one of them.

To ensure that a disjunction is not violated we only have to ensure that at least one of the predicates remains true (the student maintains at least a course registration). Therefore we propose the use of a bounded counter to control how many predicates hold true. Considering that n predicates are true, $n-1$ rights can be generated, allowing $n-1$ predicates to become false. Operations that change a predicate from true to false would have to consume rights, while operations that change the value of a predicate from false to true would generate a new right. This approach in addition to allowing some degree of concurrency, ensures that the invariant is not violated, since one of the predicates would never be able to transit from true to false due to lack of rights. For the reason explained in Section 4.5, also in this situation the rights cannot be generated immediately, and a master replica is needed for that task (if rights were generated immediately two concurrent operations with the same effect could generate more rights than should).

Since in an exclusive disjunction one and only one predicate can be true at a time, it is required the use of exclusive locks to maintain these invariants.

Implications can be considered disjunctions since the rules of logic allow us to turn an implication into a disjunction ($(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$). However, while in a disjunction the two predicates cannot be simultaneously false, in an implication A cannot be true while B is false. Examples of this type of invariant are, for instance, guarantee that when a tournament is active it has a minimum number of participants, or in an auction system guarantee that when an auction is closed the value of the winning bid corresponds to the value of the maximum bid for that auction.

After an analysis of the possible solutions for an implication invariant, it was concluded that the most advantageous would be the use of locks. Thus, changing the value of one predicate would be done with an exclusive lock while changing that of the other would be done with a shared lock. To maximise concurrency in the system, Antidote SQL could apply a mechanism to check which operations are executed most often, so that the most frequent operation would be the one to use the shared locks.

When the implication involves a numeric invariant or an aggregation constraint in the right side of the implication, it is defined a boundary that only needs to be verified when the left side predicate is true. This can be exemplified in the tournament example, where the number of players enrolled in a tournament must be greater than 1 if the tournament is active. However if the tournament is not active the number of players enrolled could be less than this value.

One possible approach that can be implemented in Antidote SQL to optimize systems under this circumstances, is to use a bounded counter when limits have to be imposed. While the left side predicate is false, a normal counter can be used, since there is no limit. At the moment of the execution of the operation that makes the predicate true (which would have to be synchronised by obtaining a lock) the counter would be converted into a bounded counter.

4.8 Linear Resources

Linear resources is a type of invariant used on objects/resources that can be partitioned, and its goal is to ensure that there is no overlap. For instance, ensuring that two users do not reserve a room for overlapping times, is an example of this type of invariant.

In this scenario, two operations reserving a room can break the invariant. However, using an exclusive lock in these cases is too restrictive, since reserving rooms for intervals that do not overlap would not violate this type of invariants.

Partition lock reservation [5] is a mechanism that allows replicas to acquire exclusive locks on intervals of values. Therefore, multiple replicas can acquire locks as long as they do not reserve locks on intersecting ranges. Whereas exclusive locks require all reserving operations to be coordinated, with this mechanism the reservation operation acquires a lock on a specific time interval, and other reserving operations can be performed concurrently as long as they are for not overlapping intervals, minimizing the coordination overhead.

4.9 Materialized Views

A materialized view is a precomputed data set that contains the results of a query and is stored on disk to improve database performance when the query is executed. Whenever data is modified, all materialized views must be modified according to the new data so that the state of the view matches the state of the database. Therefore, invariants can be specified to indicate that the materialized views reflect the primary data. Bailis et al. have shown that updates to materialized views are $\mathcal{I}$-confluent [2].

Currently, Antidote SQL does not provide support for materialized views, so this type of invariant was not analysed.

4.10 Summary

In this chapter we have presented a study of different types of invariants and made a mapping between classes of invariants and the mechanisms that can be used to enforce them. We explain for each invariant supported by Antidote SQL the mechanisms that the system uses to enforce them. For the invariant classes that are not supported by Antidote SQL, we propose mechanisms to guarantee that the invariants are maintained.

To conclude this chapter, we provide an overview of the invariants mapping in Table 4.2.

Table 4.2: Overview of the invariants mapping.

		Mechanism	**Antidote SQL**
Uniqueness	**Sequential Identifier**	Exclusive locks	Ⓔ
	Unique Identifier	Locally generated sequential identifier + replica prefix	Ⓔ
Referential Integrity		*Update-wins*, *delete-win*s, *no concurrency*	Ⓔ
Numeric Invariants		Bounded counter	Ⓔ
Attribute Equality/Inequality		-	-
Aggregation Constraints		Bounded counter with master replica	Ⓐ
Aggregation Inclusion		Set CRDT with the same conflict resolution policy as the object table	Ⓝ
Disjunctions	**Disjunction**	Bounded counter with master replica	Ⓝ
	Exclusive Disjunction	Exclusive locks	Ⓝ
	Implication	Exclusive locks + shared locks	Ⓝ
Linear Resources		Partition lock reservation	Ⓐ
Materialized Views		—	—

Ⓔ Mechanism already implemented in Antidote SQL. Ⓐ Mechanism proposed in the literature to deal with this type of invariants (type not supported by Antidote SQL). Ⓝ Mechanism we propose for dealing with the invariant type (type not supported by Antidote SQL).

5 Implementation

This chapter focuses on the implementation aspects of some mechanisms of Antidote SQL in VeriFx and is divided into two main sections. In Section 5.1 we discuss some CRDTs used to ensure state convergence: Last-Writer-Wins Register (Section 5.1.1), First-Writer-Wins (Section 5.1.2), Multi-Value Register (Section 5.1.3), Enable-Wins Flag (Section 5.1.4), Positive-Negative Counter (Section 5.1.5), and an implementation of the *update-wins* and *delete-wins* tables (Section 5.1.6). On the other hand, Section 5.2 presents the Antidote SQL mechanisms that enable invariants strengthening, namely the bounded counter (Section 5.2.1), the referential integrity preservation mechanism (Section 5.2.2), and the exclusive and shared locks mechanism (Section 5.2.3). Finally, in Section 5.3 we present an example in which we have tested the impact of the choice of conflict resolution policies on the preservation of defined invariants.

Except for the implementation of the two examples with locks, where the operation-based replication model was used, all implementations correspond to state-based CRDTs. We chose this type because several CRDTs used by Antidote SQL were already implemented with this synchronization model in VeriFx, which made our work easier. For this reason, all these classes extend the CvRDT trait that is provided by VeriFx's CRDT library for building state-based CRDTs (see Chapter 3).

Due to space limitations, sometimes only parts of the implementations are presented and not all methods are shown. However, the omitted methods are methods that are not very relevant for the reader's understanding of the implemented object.

5.1 Data Convergence Mechanisms

Antidote SQL is a highly available geo-replicated database system that provides stronger semantics than eventual consistency. To ensure that the system eventually converges, i.e., that all replicas are in the same state after performing the same set of operations (possibly in completely different order), Antidote SQL resorts to CRDTs. These data types encapsulate the replication logic and provide concurrency semantics that can be customised to meet the needs of developers, guaranteeing convergence under asynchronous replication models.

5.1.1 Last-Writer-Wins Register

The last-writer-wins register (LWW register for short) is a CRDT that, as its name implies, applies the last-writer-wins policy. Thus, if the register is updated concurrently, only the value of the last write is retained according to a global order. This CRDT supports only two operations: assign, a write operation that allows to change the value stored in the register and value, a read operation that returns the value of the register. Specification 5.1 shows the design of a state-based LWW register.

Specification 5.1 State-based Last-Writer-Wins Register (taken from [23]).

```
1:  payload $X\ x$, timestamp $t$                          ▷ $X$: some type
2:      initial $\perp$, 0
3:  update assign ($X\ w$)
4:      $x, t := w, now()$                                 ▷ Timestamp, consistent with causality
5:  query value () : $X\ w$
6:      let $w = x$
7:  compare ($R$, $R'$) : boolean $b$
8:      let $b = (R.t \leq R'.t)$
9:  merge ($R$, $R'$) : payload $R''$
10:     if $R.t \leq R'.t$ then
11:         $R''.x,\ R''.t = R'.x,\ R'.t$
12:     else
13:         $R''.x,\ R''.t = R.x,\ R.t$
```

To determine which assignments should be discarded, this CRDT associates a timestamp to each assign operation. The timestamps must be unique, fully ordered and consistent with causal order, i.e. if one event occurs causally before a second event, the timestamp of the second event must be greater than that of the first.

In Listing 5.1 is presented the implementation of the LWW Register CRDT in VeriFx, which is the direct translation of the specification. Note that the implementation was made for a generic value V and this value can be replaced by any data type. In this implementation, Lamport clocks [16] were used. Although Lamport clocks only define a partial order between events, they can be used to "artificially" build a total order.

```
class LWWRegister[V](value: V, stamp: LamportClock) extends CvRDT[LWWRegister[V]] {
 def value() = this.value

 def assign(x: V, timestamp: LamportClock) = new LWWRegister(x, timestamp)

 def compare(that: LWWRegister[V]) = this.stamp.smallerOrEqual(that.stamp)

 def merge(that: LWWRegister[V]) = {
   if (this.stamp.smallerOrEqual(that.stamp))
     that
   else
     this
 }
}
```

Listing 5.1: Last-Writer-Wins Register implementation in VeriFx (taken from [9]).

Unlike in the specification, where the `assign` function calculates the timestamp using a *now()* function, in the implementation the value of the timestamp is received as an argument. In this way, we consider that the system generates the timestamp before calling the `assign` function, and we can assume that the timestamp has the properties mentioned above.

The `LamportClock` object consists of two integers, one of which is the identifier of the replica and the other is a counter that is incremented each time the replica performs a write operation on the object to which the timestamp is associated. When comparing two Lamport clocks c_1 and c_2, c_1 is considered smaller than c_2 if the value of the counter of c_1 is smaller than the value of the counter of c_2. If the counters are equal, the tiebreaker is done by the identifier of the replica, so the counter is considered smaller if the replica has a lower identifier.

We have subjected the implementation to all the proofs provided by the CRDTs library for state-based CRDTs, and we have verified that the implementation is commutative, associative, and idempotent, i.e. that the LWW register guarantees convergence, regardless of the type of element it contains. However, the proof of equivalence of states was rejected. The returned counterexample consisted of two states with exactly the same timestamp but with two different values (`s1.stamp == s2.stamp && s1.value != s2.value`). After re-analysing the implementation and the specification, we realised that this case could not be verified because the same timestamp can never be assigned to two different write operations due to the timestamp properties. So we corrected our implementation by redefining the method `compatible` so that two states with the same timestamp and different values are not considered compatible states:

```
override def compatible(that: LWWRegister[V]) =
  (this.stamp == that.stamp) =>: (this.value == that.value)
```

The implementation was checked again and with this change all the proofs were accepted.

5.1.2 First-Writer-Wins Register

The difference between the Last-Writer-Wins and the First-Writer-Wins registers lies in the way the operation that prevails in the register is selected. In this case, the first operation is selected according to a global order and all others are discarded. So the only difference between the implementations of these two registers is the `merge` function, which is exactly the opposite of that of the LWW register.

This CRDT is not one of the CRDTs supported by Antidote SQL. However, it was developed to perform some validation tests of the conflict resolution policies presented later in Section 5.3.

```
def merge(that: FWWRegister[V]) = {
 if (this.stamp.greaterOrEqual(that.stamp))
   that
 else
   this
}
```

Listing 5.2: `merge` method for FWW Register.

5.1.3 Multi-Value Register

The multi-value register CRDT (MV register) is a CRDT designed to preserve all values from concurrent writes. For this reason, a read operation returns a set containing all concurrently written values. Specification 5.2 shows the design of a state-based multi-value register.

Specification 5.2 State-based Multi-Value Register (taken from [23]).

```
1:  payload set $S$                                  ▹ set of $(x, V)$ pairs; $x \in X$; $V$ its version vector
2:      initial $\{(\perp, [0, ..., 0])\}$
3:  query $incVV$ () : integer[$n$] $V'$
4:      let $g = myID()$
5:      let $\mathcal{V} = \{V | \exists x : (x, V) \in S\}$
6:      let $V' = [max_{V \in \mathcal{V}}(V[j])]_{j \neq g}$
7:      let $V'[g] = max_{V \in \mathcal{V}}(V[g]) + 1$
8:  update $assign$ (set $R$)                          ▹ set of elements of type $X$
9:      let $V = incVV()$
10:     $S := R \times \{V\}$
11: query $value$ () : set $S'$
12:     let $S' = S$
13: compare $(A, B)$ : boolean $b$
14:     let $b = (\forall (x, V) \in A, (x', V') \in B : V \leq V')$
15: merge $(A, B)$ : payload $C$
16:     let $A' = \{(x, V) \in A | \forall (y, W) \in B : V \parallel W \vee V \geq W\}$
17:     let $B' = \{(y, W) \in B | \forall (x, V) \in A : W \parallel V \vee W \geq V\}$
18:     let $C = A' \cup B'$
```

In the MV register, it is necessary to capture the notion of concurrency between writes, so Lamport clocks cannot be used (they only allow the definition of a partial order between writes). In this way, version vectors are used. A version vector is a vector of integers, where each entry in that vector represents a replica. When a replica performs a write operation, it updates the version vector by incrementing the corresponding entry. Version vectors are therefore very complex objects, as comparing and synchronising two version vectors requires multiple cycles to compare all entries of each version vector. This places a considerable time constraint on objects that use them and often leads to proofs of more complex objects being aborted by the SMT solver.

This situation occurred in the proof of the MV register, where it was only possible to prove commutativity and idempotence of the `merge` function. On the other hand, the associativity proof was always aborted due to the high complexity of the operations that had to be performed to prove this property.

To get around this problem, the implementation of MV Register replaced the vector clock by a time abstraction, i.e., it uses a generic time with established properties. When programmer creates an object of this type, they must specify not only the type of data to be stored in the register, but also what type of time to use (e.g. a version vector) and the function that compares times (called `before` function).

Listing 5.3 shows the function that encodes the properties of the `before` function. The first property (line 2) states that for any time `t1` and `t2`, if `t1` is before `t2`, `t2` cannot be before `t1`; the

```
 1 private def beforeAssumptions() = {
 2   forall (t1: Time, t2: Time) {
 3     this.before(t1, t2) =>: !this.before(t2, t1)
 4   } &&
 5   forall (t1: Time) {
 6     !this.before(t1, t1)
 7   } &&
 8   forall (t1: Time, t2: Time, t3: Time) {
 9     (this.before(t1, t2) && this.before(t2, t3)) =>: this.before(t1, t3)
10   }
11 }
```

Listing 5.3: Time assumptions (taken from [9]).

second property (line 5) says that a time `t1` is not before itself; and the last one (line 8) encodes the transitivity property, i.e. if `t1` is before `t2` and `t2` is before `t3`, then necessarily `t1` is before `t3`.

When using abstractions, it is important to ensure that the properties defined are properties that are verified in the concrete data types that define the abstraction. Otherwise, the verification of the objects may not be correct. For example, if a condition has been added to the previous function that causes the function to always return false (an example would be `forall(t1: Time){this.before(t1, t1)}`), any proof that calls this function in the left-hand side of the implication would return an accept (if the left-hand side of an implication is false, any value assigned to the right-hand side will cause the global expression to be true).

In addition to this problem, it is important to emphasise that objects that use abstractions with defined properties are proved according to those properties. Therefore, the programmer can only assume that a multi-value register converges with implementations of time that respect all the properties specified in the `beforeAssumptions` function. Thus, if the programmer uses an implementation of time that does not satisfy all properties, there is no guarantee that such an object will actually converge and be a CRDT.

To verify whether the three properties hold in version vectors, we used the version vector implementation provided by the VeriFx CRDT library.

Listing 5.4 presents the implementation of the multi-value register. The `reachable` function calls the `beforeAssumptions` function shown above to encode the assumptions that must be true for the logical clocks used. The second condition of this method (lines 14-16) encodes the fact that all values in a register must be either concurrently written values or equally clocked (since multiple values can be written at the same time).

Since an abstraction is used to define the time, the SMT solver cannot infer equality from the `compare` function. Therefore, it was necessary to override the `equals` method.

5.1.4 Enable-Wins Flag

A flag is a data structure that contains a boolean value and supports two types of operations: enable, which sets a flag value to true, and disable, which sets a flag value to false.

The enable-wins flag CRDT is a flag that resolves possible conflicts between enable and disable operations by leaving the flag in the enabled state. This means that a flag that is enabled and disabled concurrently is considered enabled after synchronisation of the states.

```
class MVRegister[V, Time](before: (Time, Time) => Boolean,
                 values: Set[Tuple[V, Time]] = new Set[Tuple[V, Time]]())
                 extends CvRDT[MVRegister[V, Time]] {

 private def concurrent(t1: Time, t2: Time) =
   !this.before(t1, t2) && !this.before(t2, t1) && t1 != t2

 private def beforeOrEqual(t1: Time, t2: Time) = this.before(t1, t2) || t1 == t2

 private def afterOrEqual(t1: Time, t2: Time) = this.before(t2, t1) || t1 == t2

 override def reachable() = {
   this.beforeAssumptions() &&
   this.values.forall((t1: Tuple[V, Time]) => {
     this.values.forall((t2: Tuple[V, Time]) => {
       t1.snd == t2.snd || this.concurrent(t1.snd, t2.snd)
     })
   })
 }

 override def compatible(that: MVRegister[V, Time]): Boolean =
   this.before == that.before // replicas must have the same notion of time

 def assign(v: V, timestamp: Time) = {
   val newValue = new Set[Tuple[V, Time]]().add(new Tuple(v, timestamp))
   new MVRegister(this.before, newValue)
 }

 def assignMany(vs: Set[V], timestamp: Time) = {
   val stampedValues = vs.map((v: V) => new Tuple(v, timestamp))
   new MVRegister(this.before, stampedValues)
 }

 private def keepLatest(that: MVRegister[V, Time]) = {
   this.values.filter((x: Tuple[V, Time]) => {
     val v = x.snd
     that.values.forall((y: Tuple[V, Time]) => {
       val w = y.snd
       this.concurrent(v, w) || this.afterOrEqual(v, w)
     })
   })
 }

 def merge(that: MVRegister[V, Time]) = {
   val myLatestValues = this.keepLatest(that)
   val hisLatestValues = that.keepLatest(this)
   val mergedValues = myLatestValues.union(hisLatestValues)
   new MVRegister(this.before, mergedValues)
 }

 def compare(that: MVRegister[V, Time]) = {
   this.values.forall((x: Tuple[V, Time]) => {
     that.values.forall((y: Tuple[V, Time]) => {
       this.beforeOrEqual(x.snd, y.snd)
     })
   })
 }

 override def equals(that: MVRegister[V, Time]): Boolean = this == that
}
```

Listing 5.4: Multi-Value Register implementation in VeriFx (taken from [9]).

In a first phase, we tried to implement this CRDT using the `merge` function presented in Listing 5.5. However, after submitting the implementation to the convergence proofs, we found that the `merge` function was not associative and the implementation was therefore wrong.

The `merge` function is quite trivial: if one of the times associated with one of the flags is higher than the other, the state with the higher time state is maintained, since it represents an inflation of the other state (lines 5-8); if the times are concurrent, they are synchronised using a function provided by the programmer, whose properties are encoded in the function `syncTimeAssumptions`, and the flag value is considered enabled if one of the flags is in that state, or disabled if both flags are in that state (line 11).

```
1  class EWFlag[Time](enabled: Boolean, time: Time, before: (Time, Time) => Boolean,
2                    syncTime: (Time, Time) => Time) extends CvRDT[EWFlag[Time]] {
3   // This method is incorrect
4   def merge(that: EWFlag[Time]) = {
5    if(this.afterOrEqual(this.time, that.time)) // this.time >= that.time
6     this
7    else if(this.afterOrEqual(that.time, this.time)) // that.time >= this.time
8     that
9    else // this.time and that.time are concurrent
10    new EWFlag(this.enabled || that.enabled, this.syncTime(this.time, that.time),
11               this.before, this.syncTime)
12  }
13
14  private def syncTimeAssumptions() = {
15   forall(t1: Time, t2: Time) { // syncCommutative
16    this.syncTime(t1, t2) == this.syncTime(t2, t1)
17   } &&
18   forall(t1: Time) { //syncIdempotent
19    this.syncTime(t1, t1) == t1
20   } &&
21   forall(t1: Time, t2: Time, t3: Time) { //syncAssociative
22    this.syncTime(this.syncTime(t1, t2), t3) == this.syncTime(t1, this.syncTime(t2, t3))
23   } &&
24   forall(t1: Time, t2: Time) {
25     (this.before(t1, t2)) =>: (this.syncTime(t1, t2) == t2)
26   } &&
27   forall(t1: Time, t2: Time, t3: Time) {
28     (this.before(t1, t3) && this.before(t2, t3)) =>:
29     (this.beforeOrEqual(this.syncTime(t1, t2), t3))
30   } &&
31   forall(t1: Time, t2: Time, t3: Time) {
32     (this.concurrent(t1, t2) && this.concurrent(t1, t3) && this.before(t3, t2)) =>:
33     (this.before(t3, this.syncTime(t1, t2)))
34   } &&
35   forall(t1: Time, t2: Time, t3: Time) {
36     (this.concurrent(t1, t2) && this.concurrent(t1, t3) && this.concurrent(t2, t3)) =>:
37     (!this.before(this.syncTime(t1, t2), t3))
38   } } }
```

Listing 5.5: `merge` method of the first implementation of the EW Flag CRDT.

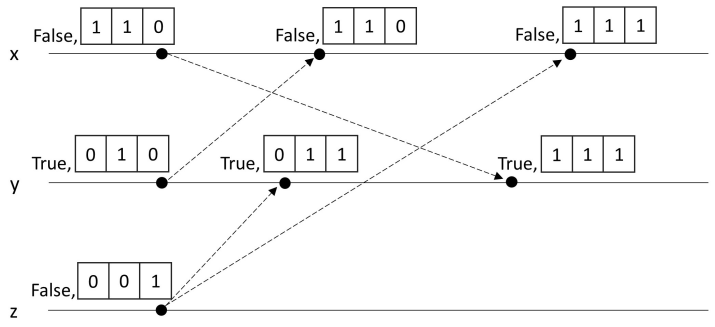


Figure 5.1: Counterexample returned by the associativity proof.

Originally, the function `syncTimeAssumptions` had only the first three properties (commutativity, idempotency and associativity), but with the counterexamples obtained it was easy to understand that the function should be more restrictive and that it would be necessary to define more precisely how time synchronisation would work. All conditions expressed in the function were derived from the counterexamples received and checked against the implementation of version vectors to ensure that incorrect properties were not inadvertently added.

With all these constraints imposed on the `syncTime` function, the counterexample of Figure 5.1 was returned. Since the object implementation resorts to a generic time, the counterexample also returns generic times. However, to make it easier for the reader to understand, we have adapted it to present a concrete example with version vectors.

The example shows three states, where state `x` is concurrent with state `z` and is an inflation of state `y`. The states `y` and `z` are concurrent. In the example, the arrows represent the propagation of the state, and the circle they point to represents the state after the local state has merged with the incoming state. In this example, we want to prove that the `merge` function is associative, so we test whether the final result of merges in different orders is the same. So in the top row we execute `x.merge(y).merge(z)` and in the middle row we execute `y.merge(z).merge(x)`.

The execution of `x.merge(y)`, is equal to `x`, because `x` has a later time than `y`. This value is then merged with `z` and since the two states are concurrent and the value of both is false, the flag is disabled.

Merging `y` and `z`, on the other hand, results in a state where the flag is enabled because the two states are concurrent and the flag is set to true in one of the them. The new state is concurrent with `x`, so the merging of the two states leads to a state in which the flag is enabled.

From the analysis of this counterexample, we can conclude that although the `merge` function appears to be correct, it is not associative because the order in which the merges are performed affects the final state of the flag, so this implementation is not a CRDT.

Finally, the EW flag was implemented using a multi-value register (Listing 5.6). In our implementation, the flag is basically a multi-value register that stores boolean values. The flag is considered enabled if the register contains a true value. Since the multi-value register stores all values corresponding to concurrent writes, only one true value is needed to assume that an enable operation has occurred. Note that almost all functions implemented in this class simply call the corresponding method of the MV register.

```
class EWFlag[Time](flags: MVRegister[Boolean, Time]) extends CvRDT[EWFlag[Time]] {
  def enable(t: Time) = new EWFlag(this.flags.assign(true, t))

  def disable(t: Time) = new EWFlag(this.flags.assign(false, t))

  def isEnabled(): Boolean = this.flags.contains(true)

  override def reachable() = this.flags.reachable()

  override def compatible(that: EWFlag[Time]) = this.flags.compatible(that.flags)

  def merge(that: EWFlag[Time]) = new EWFlag(this.flags.merge(that.flags))

  def compare(that: EWFlag[Time]) = this.flags.compare(that.flags)

  override def equals(that: EWFlag[Time]) = this == that
}
```

Listing 5.6: Enable-Wins Flag implementation in VeriFx.

Although we have not implemented the disable-wins flag because it is not one of the CRDTs supported by Antidote SQL, it could easily be implemented using the approach used for the EW flag. The difference between the two CRDTs would be the function `isEnabled()`, which would return false if the multi-value register contains a false.

5.1.5 Positive-Negative Counter

A counter is an integer that supports three types of operations: increment and decrement, which are update operations, and value, a query operation that returns the number of increments minus the number of decrements.

The specification of a state-based counter CRDT is not entirely straightforward. If we only consider that the counter supports increment operations, we are dealing with a grow-only counter [23]. The payload of this CRDT is a vector of integers, where each entry of the vector corresponds to a replica that can only be incremented by it. The counter value corresponds to the sum of all entries in the vector, and the merge function calculates the maximum of each entry. The specification of this CRDT can be found in [23].

This approach cannot be used for counters that support both increments and decrements, as the decrements would be lost in the merge. Therefore, to support both types of operations, the positive-negative counter was proposed. The PN-Counter, whose design is presented above in Specification 5.3, combines two vectors of integers: P to register the increments and N to register the decrements. The value of the counter is equal to the sum of all entries of the vector P minus the sum of all entries of the vector N.

The implementation of the PN-Counter in VeriFx can be found in Listing 5.7. In the implementation, instead of integers vectors, grow-only counters were used to represent the increment and decrement vectors. Also, the increment and decrement operations take the value to be incremented/decremented as an argument, so it is possible to update the counter value by more than one unit at a time.

Specification 5.3 State-based PN-Counter (taken from [23]).

```
1: payload integer[n] P, integer[n] N                          ▷ One entry per replica
2:     initial [0,0,...,0], [0,0,...,0]
3: update increment ()
4:     let $g = myID()$                                          ▷ g: source replica
5:     $P[g] := P[g] + 1$
6: update decrement ()
7:     let $g = myID()$
8:     $N[g] := N[g] + 1$
9: query value () : integer v
10:    let $v = \sum_i P[i] - \sum_i N[i]$
11: compare (X, Y) : boolean b
12:    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \bigwedge \forall i \in [0, n-1] : X.N[i] \leq Y.N[i]$
13: merge (X, Y) : payload Z
14:    let $\forall i \in [0, n-1] : Z.P[i] = max(X.P[i], Y.P[i]$
15:    let $\forall i \in [0, n-1] : Z.N[i] = max(X.N[i], Y.N[i]$
```

```
class PNCounter(p: GCounter, n: GCounter) extends CvRDT[PNCounter] {
 def value() = this.p.value() - this.n.value()

 def increment(replica: Int, value: Int) =
  new PNCounter(this.p.increment(replica, value), this.n)

 def decrement(replica: Int, value: Int) =
  new PNCounter(this.p, this.n.increment(replica, value))

 def merge(that: PNCounter) = new PNCounter(this.p.merge(that.p), this.n.merge(that.n))

 def compare(that: PNCounter) = this.p.compare(that.p) && this.n.compare(that.n)
}
```

Listing 5.7: PN-Counter implementation in VeriFx (taken from [9]).

5.1.6 Table

Tables are the database objects that store all the data in the database. Tables consist of several rows, each row representing a single record. Each row has several columns that represent a field of a record. For example, if we look at a table of artists, each row represents an artist and each column represents information about that artist, such as their name or country.

Considering that tables are a central element of Antidote SQL, we have implemented a data type in VeriFx that represents a table. In this implementation, we also use another data type that represents a row of a table. Therefore, this section is divided into two parts: a first part that deals with the implementation of a data type that represents table elements and a second part that focuses on the implementation of the table data type.

Table Element

The need to implement a data type to represent the table elements arose from the fact that Antidote SQL uses a visibility column to implement table conflict resolution policies and foreign key policies.

```
//0: I flag (insert), 1: T flag (touch), 2: D flag (delete)
class TableElem[V, Time](value: V, flags: MVRegister[Int, Time], mergeValues: (V, V) => V)
                        extends CvRDT[TableElem[V, Time]] {

 def touchFlag(t: Time) =
  new TableElem(this.value, this.flags.assign(1, t), this.mergeValues)

 def deleteFlag(t: Time) =
  new TableElem(this.value, this.flags.assign(2, t), this.mergeValues)

 def isVisible(tablePolicy: Boolean): Boolean = {
  if(this.flags.contains(1)) //T
   true
  else {
   if(tablePolicy) //tablePolicy == update-wins
    this.flags.contains(0)
   else //tablePolicy == delete-wins
    !this.flags.contains(2)
  }
 }

 override def reachable(): Boolean =
  this.mergeValuesAssumptions() && this.flags.reachable()

 override def compatible(that: TableElem[V, Time]): Boolean =
  this.mergeValues == that.mergeValues && this.flags.compatible(that.flags)

 def merge(that: TableElem[V, Time]): TableElem[V, Time] =
  new TableElem(this.mergeValues(this.value, that.value), this.flags.merge(that.flags),
                this.mergeValues)
}
```

Listing 5.8: Implementation in VeriFx of the Table Element data type.

The implementation presented in Listing 5.8 only considered the requirements of the *update-wins* foreign key policy. To support the *delete-wins* foreign key policy, it would be necessary to extend this class to maintain versions (see Section 5.2.2). Therefore, the table implementation presented here does not allow the implementation of systems that define foreign keys with the *delete-wins* policy.

This data type consists of three arguments: `value` represents the value stored in the table row, represented by a generic type `V` for extensibility reasons; `flags` a multi-value register representing the visibility column; and `mergeValues`, a merge function abstraction that must be specified by the programmer when creating objects of this type (in VeriFx, this function only needs to be specified if the programmer wants to perform proofs for time-specific implementations). This abstraction is necessary to merge the values of two different rows because the value is generic.

To prove that the `TableElem` object converges, it was necessary to define the properties of the `mergeValues` function. Since this is a state-based replication model, these properties are the commutativity, idempotency and associativity of the function, which are defined in the `mergeValuesAs-sumptions` function, depicted on Listing 5.9.

Strictly speaking, the visibility column (flags) should contain strings instead of integer values. However, strings are "heavier" objects than integers and SMT solvers may be unable to complete a proof if the proof object are strings. In our implementation, we therefore assign an integer to each flag: `I`, the flag indicating that the element has been added/updated is represented by 0; `T`, the flag

```
private def mergeValuesAssumptions() = {
  forall(v1: V, v2: V) { // commutativity
   this.mergeValues(v1, v2) == this.mergeValues(v2, v1)
  } &&
  forall(v1: V) { // idempotency
   this.mergeValues(v1, v1) == v1
  } &&
  forall(v1: V, v2: V, v3: V) { // associativity
   this.mergeValues(this.mergeValues(v1, v2), v3) ==
   this.mergeValues(v1, this.mergeValues(v2, v3))
  }
}
```

Listing 5.9: `mergeValues` assumptions.

indicating that an element referencing the object has been added/updated is represented by 1; and `D`, the flag indicating that an element has been deleted is represented by 2.

Methods `touchFlag` and `deleteFlag` are used to change the value of the flag when an element referencing it has been added or when the object is removed.

Remember that in Antidote SQL table elements are not actually removed from the database. What indicates whether the element exists in the database is the status of the visibility column. The `isVisible` method is the method used to check whether an element is present in the database or not. This method receives as an argument a boolean value that represents the conflict resolution policy of the table. If the value is true, the *update-wins* policy is taken into account and if the value is false, the *delete-wins* policy is considered. The first condition of the `isVisible` method (line 12) only affects tables that have been defined as the parent table of another table (since the `T` flag is only used in these situations) and checks whether the visibility column contains the `T` flag. In this case, it is assumed that the element is present in the table, as otherwise a foreign key constraint would be violated. Otherwise, the decision whether the element exists in the table depends on the conflict resolution policy defined for the table: if the policy is *update-wins*, then the element exists in the table if the multi-value register contains the `I` flag; otherwise, the element exists in the table if the multi-value register does not contain the `D` flag.

The `compatible` function defines that two states are compatible if they have the same merge function for the objects and if the multi-value register of one state is compatible with that of the other state. The `merge` function uses the `mergeValues` function defined by the programmer to merge the two values and merges the flags with the merge function of the multi-value register itself.

Table

Like the data type that represents each row of the table, the `Table` class was implemented as a polymorphic CvRDT to be extensible and allow developers to use this CRDT when implementing their own systems with VeriFx with any data type. This CRDT can be thought of as a map in terms of data structures, with the key corresponding to the primary key of the objects stored in the table and the value corresponding to an object of type `TableElem`. Although it was possible to use an array to implement this CRDT, the map was chosen for efficiency in accessing the objects stored in the table.

The implementation of the class `Table` can be found in Listing 5.10. This class has four arguments:

```
class Table[V, Time](mergeValues: (V, V) => V, before: (Time, Time) => Boolean,
         tablePolicy: Boolean, elements: Map[String, TableElem[V, Time]])
         extends CvRDT[Table[V, Time]] {

  def add(id: String, v: V, t: Time) = {
    if(this.isVisible(id)) //there is already an object with the primary key id
      this
    else {
      val flags = new MVRegister[Int, Time](this.before).assign(0, t) //I -> 0
      val elem = new TableElem(v, flags, this.mergeValues)
      new Table(this.mergeValues, this.before, this.tablePolicy, this.elements.add(id, elem))
    }
  }

  def update(id: String, v: V, t: Time) = {
    if(this.isVisible(id)) { //update only works if the object exists in the table
      val flags = new MVRegister[Int, Time](this.before).assign(0, t) //I -> 0
      val elem = new TableElem(v, flags, this.mergeValues)
      new Table(this.mergeValues, this.before, this.tablePolicy, this.elements.add(id, elem))
    } else
      this
  }

  def remove(id: String, t: Time) = {
    if(this.isVisible(id)) {
      val elem = this.elements.get(id).deleteFlag(t)
      new Table(this.mergeValues, this.before, this.tablePolicy, this.elements.add(id, elem))
    } else
      this
  }

  def touch(id: String, t: Time) = {
    val elem = this.elements.get(id).touchFlag(t)
    new Table(this.mergeValues, this.before, this.tablePolicy, this.elements.add(id, elem))
  }

  def isVisible(id: String): Boolean =
    this.elements.contains(id) && this.get(id).isVisible(this.tablePolicy)

  override def compatible(that: Table[V, Time]): Boolean = {
    this.before == that.before && this.mergeValues == that.mergeValues &&
    this.tablePolicy == that.tablePolicy && this.elements.zip(that.elements).values().forall(
       (t: Tuple[TableElem[V, Time], TableElem[V, Time]]) => t.fst.compatible(t.snd))
  }

  def merge(that: Table[V, Time]): Table[V, Time] = {
    val mergedSet = this.elements.combine(that.elements,
       (e1: TableElem[V, Time], e2: TableElem[V, Time]) => e1.merge(e2))
    new Table(this.mergeValues, this.before, this.tablePolicy, mergedSet)
  }
}
```

Listing 5.10: Implementation in VeriFx of the `Table` data type.

```
forall(table: Table[V, Time], id1: String, v2: V,
       t2: Time, t3: Time) {

 val before = table.before

 ( table.reachable() && table.isVisible(id1) &&
  table.tablePolicy == true &&
  !before(t2, t3) && !before(t3, t2) && t2 != t3 ) =>: {

    val tableUpd = table.update(id1, v2, t2)
    val tableRem = table.remove(id1, t3)

    tableUpd.isVisible(id1) &&
    !tableRem.isVisible(id1) &&
    tableUpd.merge(tableRem).isVisible(id1)
  } }
```

Listing 5.11: Proof of the correct functioning of concurrent updates and removals in an *update-wins* table.

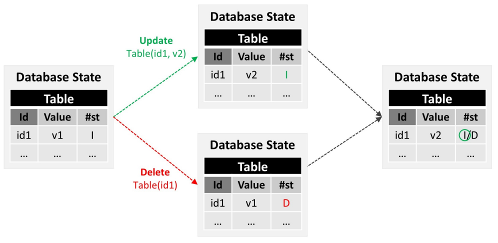


Figure 5.2: Test scenario.

the first two correspond to the `merge` function and the `before` function. These functions are not used directly by the `Table` object, but are needed to create the `TableElem` and the `MVRegister` objects, as can be seen in lines 9 and 10. The `mergeValues` function is used in the `TableElem` class to merge the values stored in the table row (line 30 of Listing 5.8), and the `before` function is needed to merge the multi-value registers representing the visibility column (see Listing 5.4). The `tablePolicy` argument is a boolean value that allows the programmer to specify the conflict resolution policy of the table, i.e. whether it is *update-wins* or *delete-wins*. The last argument corresponds to the map in which the table data is stored.

Although Antidote SQL still supports the *no concurrency* semantics, implemented with locks, this semantics has not been implemented in VeriFx. Its implementation could make it possible to check whether locks allow referential integrity to be maintained.

The `Table` class supports four write operations: one that adds elements to the table, one that updates items already in the table, one that allows items to be removed from the table, and one that marks an item as touched when another element is added that references it and the conflict resolution strategy is *update-wins*.

The `compatible` method restricts the compatible states to states where the `before` and `mergeVal-ues` functions are the same, where the table policy is the same, and where all table elements in both tables are compatible.

The `merge` method calculates a new map with all entries of the two maps. For the entries that are in both maps, it is necessary to merge these elements because the objects may have been updated in only one of the replicas.

In order to check whether the `Table` object works correctly, several tests were used, which were carried out for the two possible configurations (*update-wins* and *delete-wins*). These tests checked several execution scenarios, namely concurrent insertion of elements, sequential additions and deletions, and concurrent additions and deletions, so that we could verify that the implementations behaved as specified. Listing 5.11 and Figure 5.2 contain a proof and an illustration of the scenario verified in the proof.

In this scenario, a table with the *update-wins* policy is considered in which there is one element whose primary key is id_1. Two operations are concurrently executed. The former updates the object with id_1 and the second deletes the object with id_1 from the database. The proof checks whether the object exists in the table in the state in which the element was updated and whether the object no longer belongs to the table in the state in which it was deleted. Finally, it checks whether merging these two states results in a state in which the instance is visible in the table. As expected, the proof was accepted.

In addition to the various tests where correct scenarios were defined and the proofs were accepted, incorrect scenarios were also tested and rejected by the SMT solver.

5.2 Invariant Maintenance Mechanisms

Antidote SQL is a database system that allows programmers to specify which database constraints should be maintained. Depending on the constraints established and the level of concurrency defined for these constraints, the database system uses some mechanisms to enforce the defined data model efficiently, i.e., to keep the concurrency used as low as possible.

In Chapter 4 we presented the mechanisms that Antidote SQL uses to handle the supported invariant types. In this section, we show the implementation of some of these mechanisms in VeriFx and the results of their verification.

5.2.1 Bounded Counter

The behaviour of a bounded counter is very similar to that of a PN-Counter described in Section 5.1.5. However, unlike the PN-Counter, the bounded counter allows limits to be defined for the counter. For example, if a programmer defines a column with a limit greater than or equal to 0, Antidote SQL implements that column with a bounded counter that does not allow the counter to decrement below zero.

The specification of the bounded counter, capable of holding invariants of type *greater than or equal to K*, is in Specification 5.4.

In addition to the increment and decrement operations, the bounded counter supports another write operation: the transfer operation. This operation is used to transfer rights from one replica to another.

To support the information about these three write operations, the payload of the bounded counter consists of a matrix R in which the entry $R[i][j]$ contains the number of rights transferred from replica i to replica j and the entry $R[i][i]$ indicates the number of rights that the replica retains locally, as well as a vector U containing the number of rights consumed by each replica. An integer (*min*) corresponding to the limit set by the programmer is also required.

Since Antidote SQL does not support the process of transferring rights between replicas, our implementation in VeriFx (Listing 5.12) is much simpler than the specification shown above.

Since we do not support the transfer of rights, we do not need to implement the bounded counter with a matrix, as two integer vectors are sufficient. In this case, we use two grow-only counters, one containing the rights held by each replica and the other the rights consumed by each replica. These two structures could be replaced by an object of type PN-Counter. In our implementation, the

Specification 5.4 State-based Bounded Counter for invariant *greater or equal to K* (taken from [3]).

```
payload integer[n][n] R, integer[n] U, integer min
    initial [[0,0,...,0], ..., [0,0,...,0]], [0,0,...,0], K
query value () : integer v
    $v = min + \sum_{i \in Ids} R[i][i] - \sum_{i \in Ids} U[i]$
query localRights () : integer v
    $id = repId()$                                              ▷ Id of the local replica
    $v = R[id][id] + \sum_{i \neq id} R[i][id] - \sum_{i \neq id} R[id][i] - U[id]$
update increment (integer n)
    $id = repId()$
    $R[id][id] = R[id][id] + n$
update decrement (integer n)
    pre-condition $localRights() \geq n$
    $id = repId()$
    $U[id] = U[id] + n$
update transfer (integer n, replicaId to): boolean b
    pre-condition $b = (localRights() \geq n)$
    $from = repId()$
    $R[from][to] := R[from][to] + n$
merge (S)
    $R[i][j] = max(R[i][j], S.R[i][j]), \forall i, j \in Ids$
    $U[i] = max(U[i], S.U[i]), \forall iIds$
```

bounded counter has two other attributes, one of which is the defined limit and the other is the initial value of the counter. In the specification of the bounded counter, the initial value of the counter was equivalent to the limit. However, we considered that separating the two concepts would make the implementation more comprehensive.

In the `reachable` function we have encoded some properties of the reachable states that are necessary to prove this kind of object. The first condition (line 34) states that the initial value of the counter must be greater than or equal to the limit, otherwise the invariant would be broken when the object is created. Lines 38 and 39 expresses the condition that the number of rights consumed by each replica must have a value between zero and the number of rights maintained by the replica. Line 40, on the other hand, encodes that the sum of the rights present in the system must be greater than or equal to the number of rights originally created (remember that this value increases when new rights are created). Due to the great complexity that these conditions brought to the proofs, it was necessary to limit the number of replicas in the system. So we specified that the maximum number of replicas in the system was five replicas. Finally, we had to specify that the two vectors with integers have the same dimension, since there is one entry for each replica in each vector (lines 35-37).

Note that when the PN-Counter was implemented, the `reachable` function was not defined and therefore it was not specified that the size of the two vectors must be the same. Although the reachable states were not restricted to these states, since the object was proved for an even more general case, one can conclude that the object would also be proved correctly for more restrictive states.

To prove that the implementation of the bounded counter does not violate numeric invariants, i.e. that it does not allow the counter value to become smaller than the defined lower bound, the proof

```
class BCounterGeq(rightsHold: GCounter, rightsConsumed: GCounter, bound: Int,
                  initialValue: Int = 0) extends CvRDT[BCounterGeq] {

  def value() =
    this.bound + this.rightsHold.value() - this.rightsConsumed.value()

  def localRights(replica: Int) =
    this.rightsHold.valueOfEntry(replica) - this.rightsConsumed.valueOfEntry(replica)

  def increment(replica: Int, n: Int) =
    new BCounterGeq(this.rightsHold.increment(replica, n), this.rightsConsumed, this.bound,
          this.initialValue)

  def decrement(replica: Int, n: Int) = {
    if(this.localRights(replica) >= n)
      new BCounterGeq(this.rightsHold, this.rightsConsumed.increment(replica, n), this.bound,
       this.initialValue)
    else
      this
  }

  def invariant(): Boolean = this.value() >= this.bound

  def merge(that: BCounterGeq): BCounterGeq = {
    val rightsHoldMerged = this.rightsHold.merge(that.rightsHold)
    val rightsConsumedMerged = this.rightsConsumed.merge(that.rightsConsumed)
    new BCounterGeq(rightsHoldMerged, rightsConsumedMerged, this.bound, this.initialValue)
  }

  override def reachable() = {
    val rHoldEntries = this.rightsHold.entries
    val rConsumedEntries = this.rightsConsumed.entries

    this.initialValue >= this.bound &&
    this.rightsHold.wellFormed() &&  this.rightsConsumed.wellFormed() &&
    this.rightsHold.networkSize() == this.rightsConsumed.networkSize() &&
    this.rightsHold.networkSize() <= 5 &&
    rHoldEntries.positions.keys().forall((idx: Int) => rConsumedEntries.get(idx) >= 0 &&
      rConsumedEntries.get(idx) <= rHoldEntries.get(idx)) &&
    this.rightsHold.value() >= this.initialValue - this.bound
  }

  override def compatible(that: BCounterGeq) =
    this.rightsHold.networkSize() == that.rightsHold.networkSize() &&
    this.bound == that.bound && this.initialValue == that.initialValue
}
```

Listing 5.12: Implementation in VeriFx of the bounded counter CRDT for invariants of the type *greater or equal to K*.

presented in Listing 5.13 was written and tested. This proof verifies that the merge function does not violate the invariant. In addition to this proof, several other proofs were written to verify that the operations behave as expected (the proofs check the counter status after executing increment and decrement operations).

```
proof BCounterGeq_merge_holds_invariant {
  forall (s1: BCounterGeq, s2: BCounterGeq) {
    (s1.invariant() && s2.invariant() && s1.reachable() && s2.reachable() &&
    s1.compatible(s2)) =>: s1.merge(s2).invariant()
  }
}
```

Listing 5.13: Proof to verify that the bounded counter maintain the invariant.

Since the programmer can set lower bounds in Antidote SQL, we have also implemented a version of the bounded counter for upper bounds. This version is very similar to the one shown in this section and can be seen in the Appendix A.

5.2.2 Referential Integrity

The verification of the referential integrity mechanism had to be done using concrete examples. Implementing the mechanism using a generic example, where the parent table and child table could be of any data type, required the use of table merge abstractions. However, the merge function abstraction only encodes the properties required to prove that the system converges. By using these abstractions to merge the tables, relevant information required to prove that referential integrity is not violated is lost. That is, after two tables have been merged using an abstract merge function, it is impossible for the SMT solver to determine whether the element is still in the table or not, because the merge function says nothing about it.

In this way we implemented a concrete example to verify this mechanism. Consider again the management albums system. This system consists of two tables: an *Artists* table, which stores information about the artists registered in the system, and an *Albums* table, which keeps information related to the albums. The *Albums* table has a foreign key that refers to the primary key of the *Artists* table. Therefore, one of the invariants of the system is to ensure that referential integrity is not violated, i.e. that for each album in the system there is also the artist to whom the album belongs.

In order to verify this system in VeriFx, i.e. to check whether the invariant has been maintained and whether the system converges with the defined conflict resolution strategies, we implemented a class that represents the entire system. The implementation of this class can be seen in Listing 5.14 and Listing 5.15. The system consists of two tables, `albums` and `artists`, and two functions, `mergeAlbums` and `mergeArtists`. These two functions correspond to the functions for merging the *Albums* table and the *Artists* table respectively.

In the system implementation presented here, we only consider the *update-wins* foreign key policy. We have also implemented a version in which the *delete-wins* policy was used. However, since this policy needs to control object versions, they became more complex and the SMT solver aborted several tests. Since we could not verify that the implementation of the mechanism was correct, we do not show its implementation.

```
class AlbumsSystem[Time](albums: Table[Album, Time], artists: Table[Artist, Time],
            mergeAlbums: (Table[Album, Time], Table[Album, Time]) => Table[Album, Time],
            mergeArtists: (Table[Artist, Time], Table[Artist, Time]) => Table[Artist, Time])
            extends CvRDT[AlbumsSystem[Time]] {

 def insertAlbum(a: Album, t: Time) = {
  if(this.containsArtist(a.artist) && !this.containsAlbum(a.title)) {
   new AlbumsSystem(this.mergeAlbums, this.mergeArtists, this.albums.add(a.title, a, t),
    this.artists.touch(a.artist, t))
  } else
   this
 }

 def updateAlbumYear(title: String, newYear: Int, t: Time, stamp: LamportClock) = {
  val album = this.getAlbum(title).value
  val newAlbum = album.updateYear(newYear, stamp)
  new AlbumsSystem(this.mergeAlbums, this.mergeArtists,
   this.albums.update(title, newAlbum, t), this.artists.touch(album.artist, t))
 }

 def deleteAlbum(title: String, t: Time) =
  new AlbumsSystem(this.mergeAlbums, this.mergeArtists, this.albums.remove(title, t),
   this.artists)

 def insertArtist(a: Artist, t: Time) =
  new AlbumsSystem(this.mergeAlbums, this.mergeArtists, this.albums,
   this.artists.add(a.name, a, t))

 def updateArtistCountry(name: String, newCountry: String, t: Time, stamp: LamportClock)= {
  val artist = this.getArtist(name).value
  val newArtist = artist.updateCountry(newCountry, stamp)
  new AlbumsSystem(this.mergeAlbums, this.mergeArtists, this.albums,
   this.artists.update(name, newArtist, t))
 }

 def deleteArtist(name: String, t: Time, delCascade: Boolean) = {
  if(delCascade) {
   val albumsElems = this.albums.elements.mapValues((elem: TableElem[Album, Time]) =>
     (this.deleteChildren(name, elem, t)))
   val newAlbumsTable = new Table(this.albums.mergeValues, this.albums.before,
    this.albums.tablePolicy, albumsElems)
   new AlbumsSystem(this.mergeAlbums, this.mergeArtists, newAlbumsTable,
    this.artists.remove(name, t))
  } else
   this
 }

 private def deleteChildren(name: String, elem: TableElem[Album, Time], t: Time) = {
  if(elem.value.artist == name)
   elem.deleteFlag(t)
  else
   elem
 }

 def containsArtist(name: String) = this.artists.isVisible(name)

 def containsAlbum(title: String) = this.albums.isVisible(title)
```

Listing 5.14: Implementation in VeriFx of the album management system (part 1).

```
override def compatible(that: AlbumsSystem[Time]): Boolean =
  this.mergeAlbums == that.mergeAlbums && this.mergeArtists == that.mergeArtists

override def reachable(): Boolean =
  this.mergeAlbumsAssumptions() && this.mergeArtistsAssumptions()

def merge(that: AlbumsSystem[Time]) = {
  new AlbumsSystem(this.mergeAlbums, this.mergeArtists,
    this.mergeAlbums(this.albums, that.albums),
    this.mergeArtists(this.artists, that.artists))
}

def reachableConcreteMerges(): Boolean =
  this.artists.reachable() && this.albums.reachable()

def mergeWithConcreteMerges(that: AlbumsSystem[Time]) = {
  new AlbumsSystem(this.mergeAlbums, this.mergeArtists, this.albums.merge(that.albums),
    this.artists.merge(that.artists))
}
```

Listing 5.15: Implementation in VeriFx of the album management system (part 2).

In the implementation of this system, since the tables have a relationship among themselves, it was necessary to define some operations taking this relationship into account. The `insertAlbum` method, for example, only inserts the album if the system contains the artist to whom the album belongs and if the album does not exist in the database. In case of insertion, the artist to whom the album belongs is marked as touched. Although the `add` method of the `Table` class only produces effects on the table if the element to be added does not already exist in the table, in this case we check first if the album exists to avoid marking an artist as touched if the album hasn't been added. The method `updateAlbum` updates the album and marks its parent record as touched. It is the fact that the parent record is marked as touched that makes it possible to resolve conflicts that may lead to the referential integrity invariant being broken. The `deleteAlbum` method simply removes the album from the database (by setting the `D` flag in the visibility column).

The methods `insertArtist` and `updateArtistCountry` in turn insert/update the element (note that the element is only inserted if there is no element with the same primary key in the table, and that the update operation only takes effect if the element exists in the table). The `deleteArtist` method is more complex due to the definition of a foreign key and behaves differently depending on whether or not the `ON DELETE CASCADE` setting is enabled. If disabled, elements can only be deleted from the parent table if they do not have an element in the child table that references them. However, this behaviour would add an extra overhead to the method, which is why we decided not to allow elements to be deleted from the parent table in this configuration. If, on the other hand, the setting is enabled, the method will not only delete its own record, but also all records that refer to it. In this case, all albums of an artist are deleted when the artist is deleted.

The `containsArtist` method checks whether the artist is visible in the database, i.e. whether there is a record with this primary key in the artists table and if so, whether the visibility column of the record indicates that the row is visible. Since this implementation takes into account the *update-wins* foreign key policy, the method `containsAlbum` checks the same conditions. If the foreign key policy were *delete-wins*, the method would look similar to the one stated on Listing 5.16. In addition

```
def containsAlbum(title: String) = {
  if(this.albums.isVisible(title)) {
    val album = this.getAlbum(title)
    if(this.containsArtist(album.value.artist)) {
      album.fkVersion == this.getArtist(album.value.artist).version
    } else
      false
  } else
    false
}
```

Listing 5.16: Adaptation of the `containsAlbum` function for the *delete-wins* foreign key policy.

to checking whether the record is visible, it would have to check whether the parent record exists and whether the version stored in the album and the version of the parent record match. Only if all these conditions are met can the album (i.e. the child record) be considered visible under the *delete-wins* foreign key policy.

In Listing 5.15 we present the methods that define the properties of compatible states, reachable states, and the merge function. In this example, we define two compatible states as two states in which the `mergeAlbums` and `mergeArtists` functions are the same.

For both the definition of the reachable states and the merge function, it was necessary to define two methods. The `reachable` method encodes the reachable states as the states in which the merge function abstractions for the albums table and the artists table have the properties necessary to guarantee that a state-based CRDT converges: commutativity, associativity and idempotency. These abstractions are necessary to reduce the complexity of the convergence proofs, because without them these proofs would be aborted, so we cannot determine the system convergence and the system implementation correctness. The `merge` method is used in convergence proofs to join states. For the reasons mentioned, this method uses abstractions to merge each of the tables. The convergence of the system, as with the other data types presented here, was checked using the proofs provided by the VeriFx CRDTs library.

To prove referential integrity, one must prove that some particular records remain in the database. However, when using abstractions such as the `mergeValues` function and the `mergeArtists` function, information about what remains in the table is lost, so it was impossible to prove referential integrity using this approach. Therefore, we defined two methods that can be used to prove referential integrity. The method `reachableConcreteMerges` encodes reachable states as those in which each of the tables is in a reachable state. The `mergeWithConcreteMerges` function uses the merge functions of the `Table` object itself. In addition to these methods, we encoded the property we wanted to check (referential integrity) in the `refIntegrityInvariant` function shown in Listing 5.17.

The `refIntegrityInvariant` function checks whether, for all visible elements in the *Albums* table (child table), the artist to which each album refers exists in the *Artists* table (parent table). The proof defined to verify this property is also in the listing above and checks whether referential integrity is preserved after merging two states that are compatible and reachable and for which the invariant holds. Unfortunately, due to the high complexity of the objects representing tables and the operations performed in the proof, the SMT solver cannot prove or reject and return an abort. Since the definition of abstractions is not a solution to prove referential integrity, we had to resort to

```
def refIntegrityInvariant() =
 this.albums.elements.forall((title: String, album: TableElem[Album, Time]) =>
  album.isVisible(true) =>: this.containsArtist(album.value.artist))

proof genericReferentialIntegrity {
 forall(s1: AlbumsSystem[VersionVector], s2: AlbumsSystem[VersionVector]) {
  ( s1.reachableConcreteMerges() && s2.reachableConcreteMerges() && s1.compatible(s2) &&
   s1.refIntegrityInvariant() && s2.refIntegrityInvariant() ) =>: {
    s1.mergeWithConcreteMerges(s2).refIntegrityInvariant()
  }
 }
}
```

Listing 5.17: Generic proof to verify the preservation of referential integrity.

defining more specific and simpler proofs that define concrete situations.

In Listing 5.18 we present one of these proofs, which corresponds to the scenario shown in Figure 5.3. This proof defines a scenario where the two tables have been specified with the *update-wins* conflict resolution policy and the foreign key policy is also the *update-wins* policy. An important detail in this proof is that it specifies the type of time used (in this case an implementation of the version vectors) as opposed to what happens in the convergence proofs where generic time is used. In a first phase we tried to prove referential integrity for a generic time, but the proofs were aborted. When we tried to specify a type of time the proofs started to be accepted. Through these tests we were able to conclude that the best approach depends on the specific use case being verified. Another detail of this proof is the need to specify that the system starts with the tables empty. Without this condition, the proof is also aborted. So, to simplify the proof, we only prove cases where the system starts in an empty state and define small examples from there. Since the presence of other elements in the table does not affect referential integrity and would not lead to a different result, we can generalise from an example where we consider the system to be empty that the same thing happens when the system contains multiple elements.

The proof starts with the insertion of an artist into the system, which leads to the state `s`, and then an album is inserted into the system (`s1`). In this state, two operations are concurrently executed: one that deletes the artist (`s2`) and another that inserts a new album for that artist (`s3`). Finally, state `s23` represents the state after merging state `s2` and state `s3`. The test checks for each state whether it contains specific objects. For the final state, it checks whether it contains the artist, the second album, if it does not contain the first inserted album, and whether the values stored in the table in this state match those added in the previous states.

To check whether the table resolution policies and foreign key policies actually work correctly, we tested different combinations of policies for the tables. Our original goal was to also test with different foreign key policies, but this was not possible because our implementation only takes into account the *update-wins* foreign key policy (we tried to extend the `Table` and `TableElem` classes to support the *delete-wins* foreign key strategy, but the proofs were aborted). In this way, we tested all possible combinations of resolution policies for the albums and artists tables with several different tests: one that verified a simple example of referential integrity, another that tested the `ON DELETE CASCADE` setting (example in Listing 5.18), and tests that mixed update operations with insert and remove

```
forall(system: AlbumsSystem[VersionVector], t1: VersionVector, t2: VersionVector,
       t3: VersionVector, t4: VersionVector, c1: LamportClock, c2: LamportClock,
       c3: LamportClock, artistName: String, country: String, title1: String,
       year1: Int, title2: String, year2: Int) {

  val artist = new Artist(artistName, new LWWRegister(country, c1))
  val album1 = new Album(title1, artistName, new LWWRegister(year1, c2))
  val album2 = new Album(title2, artistName, new LWWRegister(year2, c3))

  (system.reachableConcreteMerges() && system.isEmpty() &&
   // specification of before function to be used
   system.albums.before == ((x: VersionVector, y: VersionVector) => x.before(y)) &&
   system.artists.before == ((x: VersionVector, y: VersionVector) => x.before(y)) &&
   t1.before(t2) && t2.before(t3) && t2.before(t4) && t3.concurrent(t4) &&
   album1.reachable() && album2.reachable() && title1 != title2 &&
   system.albums.tablePolicy == true && system.artists.tablePolicy == false ) =>: {

     val s = system.insertArtist(artist, t1)
     val s1 = s.insertAlbum(album1, t2)
     val s2 = s1.deleteArtist(artistName, t3, true)
     val s3 = s1.insertAlbum(album2, t4)
     val s23 = s2.mergeWithConcreteMerges(s3)

     s.containsArtist(artistName) && !s.containsAlbum(title1) && !s.containsAlbum(title2) &&
     s1.containsArtist(artistName) && s1.containsAlbum(title1) &&
     !s2.containsArtist(artistName) && !s2.containsAlbum(title1) &&
     s3.containsArtist(artistName) && s3.containsAlbum(title1) && s3.containsAlbum(title2) &&
     s23.containsArtist(artistName) && !s23.containsAlbum(title1) &&
     s23.containsAlbum(title2) && s23.getAlbum(title2).value == album2 &&
     s23.getAlbum(title1).value == album1 && s23.getArtist(artistName).value == artist
  }
}
```

Listing 5.18: Proof with a specific scenario to prove referential integrity with the `ON DELETE CASCADE` setting enabled.

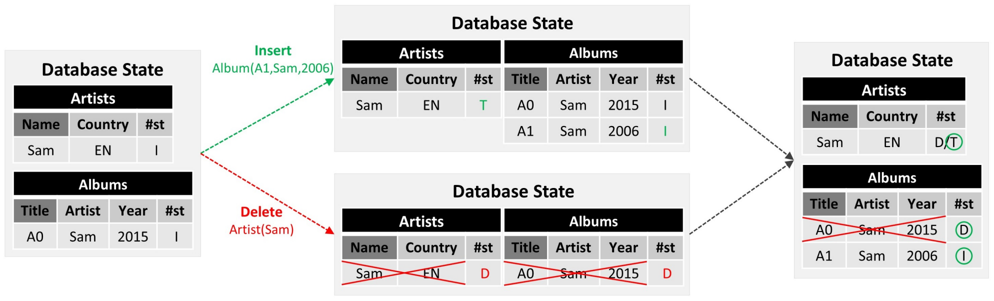


Figure 5.3: Representation of the scenario checked in the Listing 5.18.

operations to create conflicts both in the preservation of the invariant and in the tables themselves.

5.2.3 Locks

Locks are a fundamental mechanism of Antidote SQL, as they allow exclusive access to shared resources. Thus, they can be used to ensure that there are no conflicts between operations (e.g. if a conflict resolution policy is not chosen for a table, locks are used to ensure that an insert/update is not performed at the same time as a delete) or even to ensure that invariants are not violated (e.g. in the *no concurrency* semantics for foreign key constraints, locks are used to ensure that the relationship between two tables is not destroyed).

In addition to their current use in Antidote SQL, through the analysis of invariants (summarised in Chapter 4), we were able to determine that locks are essential for maintaining several other types of invariants that may be included in Antidote SQL in the future.

Antidote SQL supports two types of locks: exclusive locks and shared locks. In this section, we present the VeriFx implementation of two small examples that use locks, one with shared locks and another with exclusive locks. With these examples we show that locks are able to hold invariants and can be used for this purpose.

Exclusive Locks

Exclusive locks can only be held by one transaction at a time and give the transaction the exclusive right to execute. Therefore, if a transaction holds a lock (regardless of whether it is an exclusive or shared lock) and another transaction needs to acquire an exclusive lock, the second transaction must wait until the first transaction has executed and released the lock. Only when all locks have been released can the exclusive lock be acquired by another transaction.

As mentioned in Chapter 4, Antidote SQL uses this type of lock, for example, when it needs to create sequential identifiers for objects stored in the database. In this case, an insert operation must acquire a lock, and any insert operation that tries to execute at the same time will be blocked until it gets the lock.

To check if it is possible to maintain invariants with exclusive locks, we have implemented a small example in VeriFx that can be seen in Listing 5.19. The example consists of an object that stores two integers, `v1` and `v2`. The only operation supported by the object is the `addTransaction` operation, which changes the value of `v1` and `v2` according to the value passed as an argument in the function. In this example, we want to maintain an invariant: the value of `v1` must always be equal to the value of `v2`.

Our implementation of a state-based exclusive lock is based on the specification of distributed locks presented in [21]. For ease of reading, we created a class to represent the exclusive lock in a state-based synchronisation model. The lock can be viewed as an integer containing the identifier of the replica that holds it and a timestamp. This timestamp is needed when states with different locks are merged. Since the SMT solver explores the entire result space, states with different locks are generated. For this reason, the merge function needs to know which lock is the most recent to know which object was last changed.

The `reachable` function encodes reachable states as those where `v1` equals `v2`. Since the only operation supported by the object always assigns the same value to `v1` and `v2`, this property is true. In the `compatible` function, we have coded two conditions. The former states that two locks that have the same timestamp must also have the same owner (line 17). The second condition is necessary to prove convergence (line 18). Without it, a counterexample would be returned where two states `x` and `y` with the same lock have different values for `v1` and `v2`. That is, in state `x`, `v1` and `v2` were equal to 1 and in state `y`, `v1` and `v2` were equal to 2. These two states are incompatible because we are considering a model where the exclusive lock is released after the operation is performed. Even if

```
class ExcLock(owner: Int, stamp: LamportClock) {}

class TxWithLockSB(v1: Int, v2: Int, lock: ExcLock) extends CvRDT[TxWithLockSB] {

  def addTransaction(value: Int, replicaId: Int, stamp: LamportClock) = {
    if(replicaId == this.lock.owner)
      new TxWithLockSB(value, value, this.lock)
    else
      this
  }

  def invariant(): Boolean = this.v1 == this.v2

  override def reachable(): Boolean = this.v1 == this.v2

  override def compatible(that: TxWithLockSB): Boolean =
    (this.lock.stamp == that.lock.stamp) =>: (this.lock.owner == that.lock.owner) &&
    (this.lock == that.lock) =>: (this.v1 == that.v1 && this.v2 == that.v2)

  def merge(that: TxWithLockSB): TxWithLockSB = {
    if (this.lock.stamp.greaterOrEqual(that.lock.stamp))
      this
    else
      that
  }

  def compare(that: TxWithLockSB): Boolean = this.lock.stamp.smallerOrEqual(that.lock.stamp)
}
```

Listing 5.19: Example with exclusive locks in VeriFx.

transactions were considered, two different states with the same timestamp could never be verified because the state would only be propagated at the end of the transaction.

To prove that the defined invariant was maintained by the object, we defined the proof presented in Listing 5.20. This proof is simple and only verifies that for two compatible states in which the invariant is verified, joining the two states creates a new state in which the invariant is also preserved. In this proof, the `reachable` function has not been invoked, because the reachable states in the example are exactly the same as those that satisfy the invariant, so implicitly the states that satisfy the invariant are the same as those that are reachable.

```
proof Tx_holds_invariant {
  forall (s1: TxWithLockSB, s2: TxWithLockSB) {
    (s1.invariant() && s2.invariant() && s1.compatible(s2)) =>: s1.merge(s2).invariant()
  }
}
```

Listing 5.20: Proof to verify the preservation of the invariant using exclusive locks.

This example has also been implemented for an operation-based replication model. You can see this implementation in Appendix B.

Shared Locks

Shared locks are a less restrictive type of lock than exclusive locks. A transaction that acquires this type of lock has the right to execute transactions, but other transactions with this type of lock can

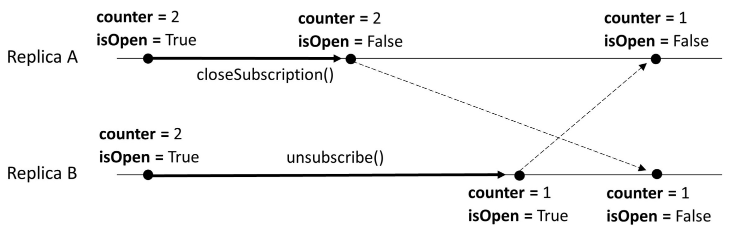


Figure 5.4: Execution scenario without using locks.

be executed at the same time. For example, if you use shared locks in Antidote SQL to implement update-delete semantics, insert/update operations must acquire a shared lock. Unlike exclusive locks, where only one transaction can be executed at a time, shared locks allow multiple objects to be inserted/updated into the table simultaneously.

Consider the following example: an application that registers the number of subscriptions for a team while the subscriptions are open. The system supports three operations: an operation that allows users to subscribe to the team, an operation that allows users to unsubscribe, and finally an operation that closes the subscriptions. This operation can only be performed if the number of subscribers is greater than 2. From the moment the registrations are closed, users can neither subscribe nor unsubscribe to the team. In this way, the following invariant can be defined as `(!this.isOpen)` ⇒ `(this.counter.ctr` ≥ `2)`.

If no mechanism is used, the scenario of Figure 5.4 may occur.

Listing 5.21 shows the example implemented with shared locks. This example was implemented for an operation-based model. The invariant defined for the system is an implication, so as said in Chapter 4, operations that make the left predicate true must use an exclusive lock (in this case operation `closeSubscription`), and operations that can break the invariant but affect the right side need a shared lock to be executed (`unsubscribe` operation). The `subscribe` operation also needs a lock, otherwise the counterexample depicted in Figure 5.5 is returned. This counterexample shows a situation where the invariant is maintained but the state does not converge when the operations in state $s3$ are performed in a different order. Hence the need to use locks in the subscribe operation.

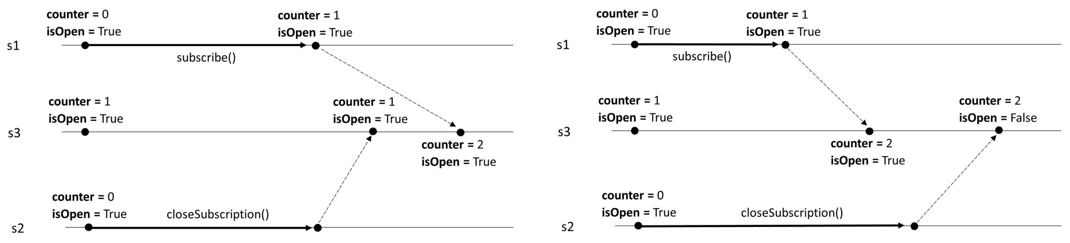


Figure 5.5: Counterexample returned if the subscribe method did not use locks.

To prove that the object converges, we used the proof provided by the VeriFx CRDT library. In Listing 5.22 we present the proof that verifies if the invariant is preserved. The proof verifies that the invariant is preserved for any three reachable states, compatible with each other, and in which the invariant is preserved regardless of the operation performed and the order in which it is performed. This proof also verifies that the state converges after the execution of the same set of operations.

```
object Op {
 enum Ops {
  CloseSubscriptions(replicaId: Int) |
  Subscribe(replicaId: Int) |
  Unsubscribe(replicaId: Int)
 }
}

class SharedLockExample(counter: Counter, isOpen: Boolean,
                        posLocks: Set[Int] = new Set[Int](), negLocks: Set[Int] = new Set[Int]())
                        extends CmRDT[Ops, Ops, SharedLockExample] {

 def closeSubscriptions(replicaId: Int) = {
  if(this.negLocks.contains(replicaId) && this.counter.ctr >= 2)
   new SharedLockExample(this.counter, false, this.posLocks, this.negLocks)
  else
   this
 }

 def subscribe(replicaId: Int) = {
  if(this.posLocks.contains(replicaId) && this.isOpen)
   new SharedLockExample(this.counter.increment(), this.isOpen, this.posLocks,
    this.negLocks)
  else
   this
 }

 def unsubscribe(replicaId: Int) = {
  if(this.posLocks.contains(replicaId) && this.isOpen)
   new SharedLockExample(this.counter.decrement(), this.isOpen, this.posLocks,
    this.negLocks)
  else
   this
 }

 override def reachable() =
  !(this.negLocks.nonEmpty() && this.posLocks.nonEmpty())

 def invariant(): Boolean =
  (!this.isOpen) =>: (this.counter.ctr >= 2)

 def prepare(op: Ops) = op // prepare phase does not add extra information

 def effect(op: Ops) = op match {
  case CloseSubscriptions(replicaId) => this.closeSubscriptions(replicaId)
  case Subscribe(replicaId) => this.subscribe(replicaId)
  case Unsubscribe(replicaId) => this.unsubscribe(replicaId)
 }
}
```

Listing 5.21: Example with shared locks in VeriFx.

```
proof SharedLockExample_holds_invariant {
 forall(s1: SharedLockExample, s2: SharedLockExample, s3: SharedLockExample,
      x: Ops, y: Ops) {
  val msg1 = s1.prepare(x)
  val msg2 = s2.prepare(y)

  ( s1.compatible(msg1, msg2) && s1.compatibleS(s2) && s1.compatibleS(s3) &&
   s1.invariant() && s2.invariant() && s3.invariant() &&
   s1.reachable() && s2.reachable() && s3.reachable() ) =>: {

    s3.tryEffect(msg1).tryEffect(msg2) == s3.tryEffect(msg2).tryEffect(msg1) &&
    s3.tryEffect(msg1).invariant() && s3.tryEffect(msg2).invariant() &&
    s3.tryEffect(msg1).tryEffect(msg2).invariant()
  }
 }
}
```

Listing 5.22: Invariant checking proof for the example with shared locks.

5.3 Verification of Conflict Resolution Policies

In order to verify that VeriFx is indeed able to detect situations where the choice of conflict resolution strategies causes the violation of the invariant, we defined a system where the choice of conflict resolution strategies determines whether the invariant is violated or not.

The implementation of the example can be found in Listing 5.23. The system contains two integers, `v1` and `v2`, implemented with register CRDTs for convergence reasons. One of the attributes, in this case `v1`, was defined with the last-writer-wins policy, and is therefore implemented with a `LWWRegister`. The other attribute (`v2`) has been implemented with a `FWWRegister` and therefore the first-writer-wins policy is applied. The system allows the execution of an update operation that changes the value stored in each of the registers to the value passed as the argument of the operation. In addition, an invariant has been defined for the system: the value stored in `v1` must always be equal to `v2`.

The implementation of the system was verified using the `Transaction_holds_invariant` proof. As expected, the proof was rejected and we got back the counterexample shown in Figure 5.6.

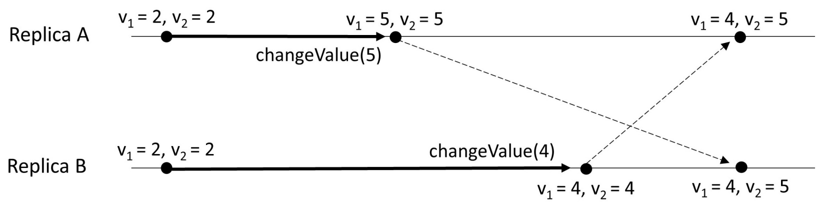


Figure 5.6: Counterexample returned by VeriFx for the `Transaction_holds_invariant` proof.

The problem with this system was the definition of the conflict resolution strategies, since with simultaneous updates each of the registers would retain a different value. To solve this problem, it

```
class ValuesEqualityExample(v1: LWWRegister[Int], v2: FWWRegister[Int])
            extends CvRDT[ValuesEqualityExample] {

 def changeValue(value: Int, timestamp: LamportClock) =
  new ValuesEqualityExample(this.v1.assign(value, timestamp),
   this.v2.assign(value, timestamp))

 def merge(that: ValuesEqualityExample) =
  new ValuesEqualityExample(
   this.v1.merge(that.v1),
   this.v2.merge(that.v2))

 def compare(that: ValuesEqualityExample) = {
  this.v1.compare(that.v1) && this.v2.compare(that.v2)
 }

 override def reachable() = {
  this.v1.stamp == this.v2.stamp
 }

 def invariant(): Boolean = {
  this.v1.getValue() == this.v2.getValue()
 }
}

object ValuesEqualityExampleProofs {
 proof ValuesEqualityExample_holds_invariant {
  forall (s1: ValuesEqualityExample, s2: ValuesEqualityExample) {
     (s1.invariant() && s2.invariant() && s1.reachable() && s2.reachable()) =>:
        s1.merge(s2).invariant()
  }
 }
}
```

Listing 5.23: Definition of a system for testing the effects of conflict resolution strategies on specific invariants.

would be sufficient to use the same conflict resolution strategy for both registers, i.e. either two LWW registers or two FWW registers.

This simple example illustrates the impact the choice of conflict resolution policies can have on the application correctness.

5.4 Summary

In this chapter, we have presented several implementations of the CRDTs used by Antidote SQL in VeriFx, as well as some mechanisms used by this database system to enforce invariants. For all implementations presented here, we have performed VeriFx proofs to verify that objects of these types converge.

For the mechanisms that enforce invariants, we also encoded the correctness properties we wanted to verify in proofs (e.g., in the case of the bounded counter, verifying that the implementation does not allow the bounds defined for the counter to be exceeded) and verified that the properties are preserved.

In this chapter, we also presented an example that shows the impact that the choice of conflict resolution strategies can have on the preservation of invariants. This example shows us the importance of using an analysis tool to validate programmer specifications in Antidote SQL.

6 Methodology

In the previous chapter, we presented the implementation and some details of the verification of all CRDTs used by Antidote SQL, as well as mechanisms for strengthening invariants. Although all mechanisms have been proven correct for the properties we wanted to verify, there are issues that only arise in specific situations and scenarios. Moreover, the correctness of a single RDT does not imply the correctness of a composition of RDTs. So if programmers want to conclude that their specifications are correct, they have to verify the system as a whole.

To help programmers verify applications, in Section 6.1 we present a methodology for implementing and verifying systems in VeriFx. In Section 6.2 we apply the methodology to a concrete case: the system for managing albums and artists. Finally, in Section 6.3 we present the limitations of the presented approach and make some suggestions for improvements.

6.1 Methodology for Implementation and Verification of Applications

By implementing the album management system example, it was possible to derive a methodology for application verification. Although this methodology focuses on applications that use Antidote SQL, it could also be adapted for the verification of other applications.

Below is a list of the different steps that the developer needs to follow when implementing and verifying their applications and systems:

1. Define the system tables in Antidote SQL by specifying which concurrency semantics are allowed for each table and column, and by identifying the system invariants.

2. For each table, implement the data type that the table should store. This implementation must be done according to the specification of the tables themselves. This data type represents the table rows without the visibility column. Therefore, for each of the columns defined in the specification, there must be an attribute in the class that corresponds to it. The data type of each attribute (in VeriFx) must also be derived from the data type and data policy defined for the column in Antidote SQL. For example, a column whose concurrency semantics is *no concurrency* and whose data type is `VARCHAR` must be implemented with the `String` data type in VeriFx. In contrast, a column defined with the same data type but with last-writer-wins semantics is implemented in VeriFx as an object of type `LWWRegister[String]`.

3. For each of these data types, check whether they converge, i.e. whether the merge function is commutative, associative and idempotent. This verification can be done using the proofs provided by the VeriFx CRDTs library and is necessary to ensure that the abstractions and assumptions defined by the `TableElem` class are correct. As explained in the previous chapter, the `TableElem` class and the `Table` class were implemented in a generic way to allow programmers to implement their own systems by composing objects of type `Table`. In addition to the implementation, it has been proved that objects of type `Table[V, Time]` converge, provided that the merge function of the values of type `V` guarantees commutativity, associativity and idempotence. In this way, having proved that the developed data types converge, the programmer can assume that objects of type `Table` containing these values also converge.

4. Implement the class system. This class must contain as many objects of type `Table` as the tables defined by the programmer. Each table must be of a type defined by the programmer in step 2. In addition, the programmer must specify the conflict resolution policy for each table, which must conform to the one defined in the specification. The programmer must also specify a table merge function abstraction for each of the defined tables. These abstractions are necessary to reduce the complexity of the properties that the SMT solver must prove, and they must express that the merge function guarantees the properties necessary for the convergence of each table. In defining the abstractions and assumptions, we can take into account that after the previous step we have concluded that these properties are true for each of the merge functions of the defined tables.

5. Verify that the system class converges by using the tests provided by VeriFx for this purpose. If the tests are aborted due to the complexity of the class (e.g. because the class contains many tables), the programmer should try to fragment this class into simpler logical parts. These parts must be tested one by one. After checking convergence, they must be gradually grouped into other classes, using abstractions to reduce the complexity of the properties to be proved. For example, consider a system with four tables (`a`, `b`, `c`, and `d`), one of which (say `b`) contains a foreign key of another table (`a`). Imagine also that the verification of the convergence properties of the implementation of the system class with these four tables is always aborted by the SMT solver. To prove the convergence properties, we suggest that the programmer implements two additional classes: one class in which the convergence of the two associated tables (`a` and `b`) is proved, called `AB`, and another class, called `CD`, in which the convergence of the remaining two tables is proved. After proving that the objects of these classes converge, the programmer can implement a new version of the system class, but instead of four tables and four abstractions to merge the tables, it would now have two objects, one of type `AB` and one of type `CD`, and two abstractions, one of which is the merge function for each of the objects. Figure 6.1 shows a diagram illustrating this situation. On the left side, the system class is formed by the four tables and four abstractions of the merge function. The right-hand side shows the approach that the programmer should take to try to simplify the objects and prove the convergence of the system. Note that a system with four tables could converge without requiring code fragmentation, and that this is just an illustrative example.

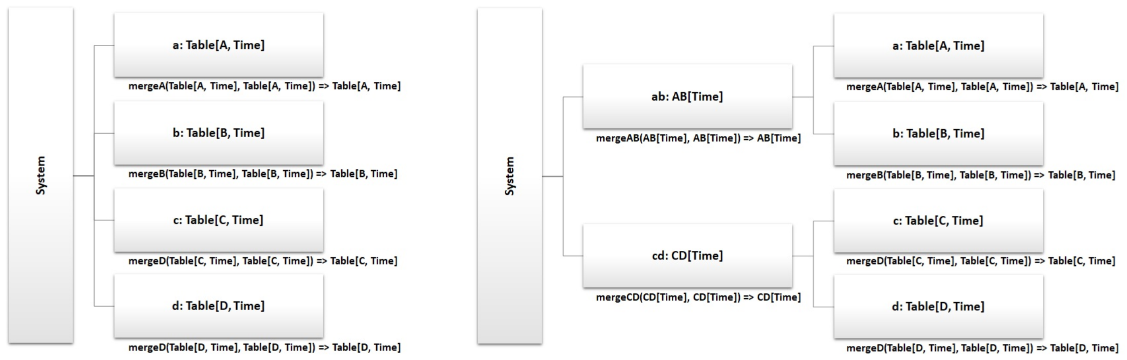

System
a: Table[A, Time]
mergeA(Table[A, Time], Table[A, Time]) => Table[A, Time]
b: Table[B, Time]
mergeB(Table[B, Time], Table[B, Time]) => Table[B, Time]
c: Table[C, Time]
mergeD(Table[C, Time], Table[C, Time]) => Table[C, Time]
d: Table[D, Time]
mergeD(Table[D, Time], Table[D, Time]) => Table[D, Time]
System
ab: AB[Time]
mergeAB(AB[Time], AB[Time]) => AB[Time]
cd: CD[Time]
mergeCD(CD[Time], CD[Time]) => CD[Time]
a: Table[A, Time]
mergeA(Table[A, Time], Table[A, Time]) => Table[A, Time]
b: Table[B, Time]
mergeB(Table[B, Time], Table[B, Time]) => Table[B, Time]
c: Table[C, Time]
mergeD(Table[C, Time], Table[C, Time]) => Table[C, Time]
d: Table[D, Time]
mergeD(Table[D, Time], Table[D, Time]) => Table[D, Time]

Figure 6.1: Fragmentation of the system class to simplify the complexity of the system and check if it converges.

6. Check that the invariants defined for the system are maintained. The invariants must be defined and proved in the class that makes the most sense. For example, referential integrity must be proved in the class that joins the two tables that maintain the integrity relationship, and a numeric invariant defined for a column must be proved in the class in which the column is defined. It is the responsibility of the programmer to define the properties to be proved in the correct classes and to write the necessary proofs to verify that the properties are proved. Listing 6.1 shows a model proof that can be used to prove invariants. Assuming that the property to be proved is encoded in the `invariant` function, the proof tests whether, for any two compatible reachable states in which the invariant is preserved, the invariant is preserved after the merge. Sometimes this model cannot be used. For example, to prove the referential integrity mechanism, it was necessary to define proofs that define concrete examples.

```
proof invariantHolds {
  forall(s1: V, s2: V) {
    (s1.reachable() && s2.reachable() && s1.compatible(s2) &&
     s1.invariant() && s2.invariant()) =>: {

      s1.merge(s2).invariant()
    }
  }
}
```

Listing 6.1: Model proof to verify the preservation of invariants in CvRDTs.

Referential Integrity. Unlike CRDTs used by Antidote SQL, or table objects where developers can create objects of these types and use them to implement their applications, this is not possible with referential integrity mechanisms. This impossibility is due to the fact that the generic implementation of the mechanism requires the use of abstractions that abstract fundamental details for proving invariant maintenance (with the abstractions it is no longer possible to check whether an element exists in the database or not).

```
class RefIntegrity[P, C, Time](pTable: Table[P, Time], cTable: Table[C, Time],
        mergePTable: (Table[P, Time], Table[P, Time]) => Table[P, Time],
        mergeCTable: (Table[C, Time], Table[C, Time]) => Table[C, Time])
        extends CvRDT[RefIntegrity[P, C, Time]] {

  def insertChild(c: C, t: Time) = {
    if(this.containsParent(c.fk) && !this.containsChild(c.pk)) {
      new RefIntegrity(this.mergePTable, this.mergeCTable, this.pTable.touch(c.fk, t),
        this.cTable.add(c.pk, c, t))
    } else
      this
  }

  def updateChild(newChild: C, t: Time, stamp: LamportClock) =
    new RefIntegrity(this.mergePTable, this.mergeCTable, this.pTable.touch(newChild.fk, t),
      this.cTable.update(newChild.pk, newChild, t))

  def deleteChild(cPk: String, t: Time) =
    new RefIntegrity(this.mergePTable, this.mergeCTable, this.pTable,
      this.cTable.remove(cPk, t))

  def insertParent(p: P, t: Time) =
    new RefIntegrity(this.mergePTable, this.mergeCTable, this.pTable.add(p.pk, p, t),
      this.cTable)

  def updateParent(newParent: P, t: Time, stamp: LamportClock) =
    new RefIntegrity(this.mergePTable, this.mergeCTable,
      this.pTable.update(newParent.pk, newParent, t), this.cTable)

  def deleteParent(pPk: String, t: Time, delCascade: Boolean) = {
    if(delCascade) {
      val childrenElems = this.cTable.elements.mapValues((elem: TableElem[C, Time]) =>
        (this.deleteChildren(pPk, elem, t)))
      val newCTable = new Table(this.cTable.mergeValues, this.cTable.before,
        this.cTable.tablePolicy, childrenElems)
      new RefIntegrity(this.mergePTable, this.mergeCTable, this.pTable.remove(pPk, t),
        newCTable)
    } else
      this
  }

  private def deleteChildren(name: String, elem: TableElem[Album, Time], t: Time) = {
    if(elem.value.fk == name)
      elem.deleteFlag(t)
    else
      elem
  }

  def containsParent(pPk: String) = this.pTable.isVisible(pPk)

  def containsChild(cPk: String) = this.cTable.isVisible(cPk)
}
```

Listing 6.2: Generic model for the implementation of referential integrity relations.

Nevertheless, the implementation of referential integrity relationships is always done in the same way, changing only the type of tables used. In this way, and with the aim of facilitating the implementation of systems that use referential integrity, we present in Listing 6.2 the code in VeriFx for a referential integrity relationship between two tables. This code works only for the *update-wins* foreign key policy.

In the listing, `P` represents the type of data stored in the parent table and `C` represents the type of data stored in the child table.

6.2 Applying the Methodology to a Concrete System

In this section, we use a concrete application to demonstrate how the programmer can use the methodology defined in the previous section to implement and verify applications that use Antidote SQL. To do this, we again use the example of the albums and artists management system.

The first step is to define the system and its tables in Antidote SQL. Although the definition of these tables has already been presented in Chapter 3, for the sake of readability we present again in Listing 6.3 the definition of the tables in Antidote SQL (the definition of the Albums table is simpler than the one in Chapter 3).

```sql
CREATE UPDATE-WINS TABLE artists (
  name VARCHAR PRIMARY KEY,
  country LWW VARCHAR
)

CREATE UPDATE-WINS TABLE albums (
  title VARCHAR PRIMARY KEY,
  year LWW INT,
  artist VARCHAR FOREIGN KEY UPDATE-WINS REFERENCES Artists(name) ON DELETE CASCADE
)
```

Listing 6.3: Definition of the albums system tables in Antidote SQL.

Once the programmer has defined all the tables in Antidote SQL, the classes representing the objects stored in each table must be implemented in VeriFx (i.e. the rows of each table except the visibility column). Their implementation must be done according to the definition of the corresponding table.

Both the `Artist` class, which represents the values stored in each row of the *artists* table, and the `Album` class, which represents the values stored in each row of the *albums* table, have been implemented by establishing a direct association between the data types and policies defined by the programmer and the data types available in VeriFx.

Artist. The *artists* table was defined to have two columns. Therefore, the VeriFx implementation of the class has two attributes, one for each of the columns. The *name* column, defined as a `VARCHAR` in the specification, is defined as a `String` in the implementation. The *country* column, in turn, has been implemented with a `LWWRegister` containing strings. The choice of a LWW register is due to the fact that the column has been defined with the last-writer-wins resolution policy to handle possible conflicts. The fact that the register contains objects of type String is due to the fact that the data type defined for the *country* column is of type `VARCHAR`. The `Artist` class is very simple, and can be found in Listing 6.4. The `updateCountry` method allows the value associated with the country attribute to be updated. The remaining methods are the methods required to define an object as a CvRDT. The `merge` method calls the `merge` method of `LWWRegister` to merge the two records, with the name column retaining the value that this state had. This does not pose any convergence

```
class Artist(name: String, country: LWWRegister[String]) extends CvRDT[Artist] {

  def updateCountry(newCountry: String, stampCountry: LamportClock) =
    new Artist(this.name, this.country.assign(newCountry, stampCountry))

  def merge(that: Artist) =
    new Artist(this.name, this.country.merge(that.country))

  def compare(that: Artist) =
    this.country.compare(that.country)

  override def compatible(that: Artist) =
    this.name == that.name && this.country.compatible(that.country)
}
```

Listing 6.4: Definition of the `Artist` data type in Antidote SQL.

problems because it is defined in the method `compatible` that two states are only compatible if the value associated with the attribute name is the same in both states. This property is true because it only makes sense to merge two objects of type `Artist` if they represent the same object, i.e. if they both have the same primary key (in this case the primary key is the *name* column).

After defining the implementation of the `Artist` class, we checked whether this type of object converges, and the SMT solver returned an *accept*. Since we have guarantees that the `Artist` object converges, i.e. that the `merge` function is commutative, associative and idempotent, we can conclude that objects of type `TableElem[Artist, Time]` and `Table[Artist, Time]` also converge.

Album. Listing 6.5 shows the implementation of the class `Album` in VeriFx. This class consists of three attributes, each of which corresponds to one of the three columns defined for the *albums* table. The attributes *title* and *artist* are of type `String` for the reasons explained above. The column *year* has been implemented with a `LWWRegister[Int]`. In this case, the register stores integers, as the column year is defined as `INT`. As with the `Artist` class, only one update operation was defined in this class and the methods used to define a CvRDT

```
class Album(title: String, artist: String, year: LWWRegister[Int]) extends CvRDT[Album] {

  def updateYear(newYear: Int, stampYear: LamportClock) =
    new Album(this.title, this.artist, this.year.assign(newYear, stampYear))

  def merge(that: Album) =
    new Album(this.title, this.artist, this.year.merge(that.year))

  def compare(that: Album) =
    this.year.compare(that.year)

  override def compatible(that: Album) =
    this.title == that.title &&
    this.artist == that.artist && this.year.compatible(that.year)
}
```

Listing 6.5: Definition of the `Album` data type in Antidote SQL.

The objects of this type were considered CRDTs by the proofs, i.e. it was verified that the objects converge. With this information, it was possible to conclude that objects of the type `TableElem[Album, Time]` and `Table[Album, Time]` also converge.

The next step would be to implement the class system and check whether the objects of this class converge. The implementation of this system has already been shown and explained in Section 5.2.2 (we used this system to prove the referential integrity mechanism). Having verified that the objects of the class system converge and that the abstractions and assumptions for the implemented objects hold, we can state with certainty that the albums and artists management system converges.

In Section 5.2.2 we also showed how to verify the referential integrity mechanism, which is an example of verifying an invariant.

Note that in the class definition of *AlbumsSystem* (Section 5.2.2) we did not define the conflict resolution policy for each table (as stated in the methodology). The reason for this was that we wanted to test different combinations of conflict resolution policies. Therefore, in each of the proofs we defined which policy to use in each of the tables. However, if you want to prove that the specifications defined by the programmer for the system are correct, you can define the conflict resolution policy for each of the tables in the system class definition.

6.3 Approach Limitations

The approach proposed in Section 6.1 has several limitations, many of which are due to the use of the SMT solver to analyse the implemented data types.

One of the major limitations of VeriFx is its limited power in verifying complex objects or in verifying proofs that must execute multiple cycles. Throughout this dissertation, we had to deal with this problem several times, and the strategy we used to mitigate this problem was to define abstractions. However, as already discussed, defining abstractions is not only a difficult task, but can also lead to the introduction of errors. Furthermore, the use of abstractions is not always feasible (e.g. abstractions cannot be used to prove the mechanics of referential integrity), and there may be cases where the definition of abstractions does not reduce the complexity of the object to a level that the SMT solver can prove (by returning either an *accept* or a *reject*).

Another problem associated with the use of an SMT solver is the butterfly effect. Sometimes simply changing a variable name will cause the SMT solver to abort a proof (if previously accepted/rejected) or accept/reject (if the proof was previously aborted). Another situation that occurred when implementing and checking objects in VeriFx was that accepted proofs were aborted after adding methods to the class that were not used by the proof in question.

Despite the great help that the counterexamples offered us, because in many cases they helped us to identify errors in the implementations, they also represent a limitation. The fact that the returned counterexamples are presented in the language of the solver (in this case Z3) makes their analysis difficult and time-consuming. Furthermore, programmers who have no prior knowledge of Z3 may not understand the counterexample at all, so this feature can be a barrier to using this approach for application verification.

Another limitation is the fact that the methodology requires the developer to implement their systems in VeriFx, as this process is time-consuming. A solution to this problem would be to automate

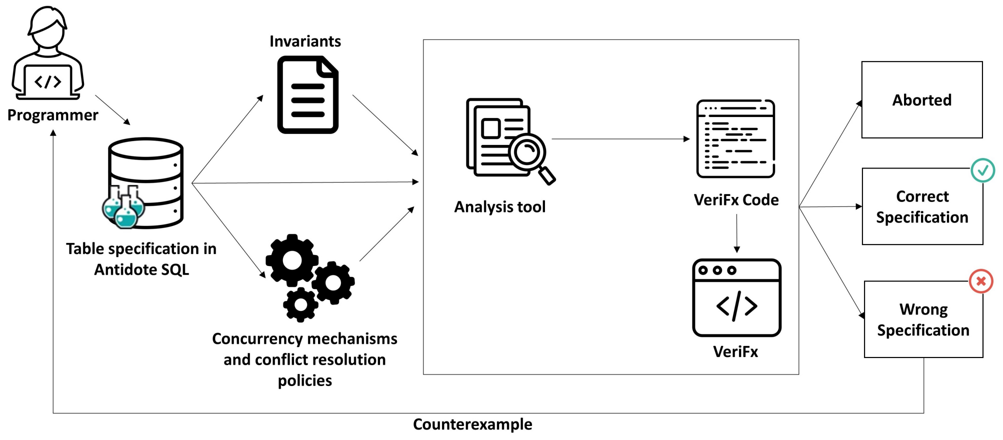


Figure 6.2: Workflow of the proposed analysis tool.

the verification process, i.e. develop an analysis tool that sits on top of VeriFx. In Figure 6.2 you can see a diagram showing the flow of the verification process with the analysis tool.

The developer would start by providing the specification of the tables in Antidote SQL. This specification would already contain information about concurrency mechanisms and conflict resolution strategies, as well as the definition of invariants. This information would be the input to the analysis tool, which would start translating the Antidote SQL table specification into VeriFx code. This code would then be analysed by the VeriFx SMT solver, which would return one of three output options: abort to indicate that the SMT solver was unable to validate/reject the proof; accept to indicate that the specification is correct; and reject to indicate that the specification does not meet the system requirements. In the case of rejection, a counterexample is returned to the programmer illustrating a situation where the property being proved is not verified. Ideally, and contrary to what is currently happening, this counterexample should be presented in high-level notation. The methodology was not automated in this work due to time constraints.

6.4 Summary

In this chapter, we presented a methodology for implementing and verifying applications that use Antidote SQL. To illustrate how application implementation and verification should be conducted, we applied the methodology to a concrete example.

Finally, we concluded the chapter with a critical analysis of the limitations of our proposed approach. For some of the limitations, we have presented alternatives that allow overcoming some shortcomings.

7 Conclusion

Antidote SQL is a geo-replicated database system that allows programmers to specify in their data model the allowable concurrency levels, invariants, and conflict resolution strategies that will resolve potential conflicts that may arise from concurrency. However, specifying applications in Antidote SQL is not an easy task, as it is not always evident which conflict resolution strategy is most appropriate for a given situation, or whether the strategy chosen by the programmer can somehow break the defined invariants.

The work developed in this dissertation aimed to fill some of these gaps and can be divided into two main phases.

In the first phase, we conducted a theoretical study of the different types of invariants, not all of which are supported by Antidote SQL. The main goal of this study was to obtain a mapping between the types of invariants and the mechanisms best suited to handle each invariant. For the invariant classes supported by Antidote SQL, we presented the mechanisms used by this database system to handle these classes. For the types of invariants not supported by Antidote SQL, we presented several suggestions for mechanisms that could be used to enforce the invariants. One of our concerns in this study has always been to propose mechanisms that attempt to combine the maintenance of invariants with some degree of concurrency.

In the second phase of the dissertation, the main goal was to verify that the mechanisms used by Antidote SQL actually work correctly and provide the promised guarantees. To this end, we implemented and verified all CRDTs supported by Antidote SQL and several mechanisms used by Antidote SQL in VeriFx. The implementation of all CRDTs later enabled us to implement a more complex system, namely the album management system. By defining this system, it was possible to understand some patterns that led us to develop a methodology for verifying applications that use Antidote SQL in VeriFx.

Future Work. Although we have verified several mechanisms used by Antidote SQL to deal with possible invariant breaks and divergence problems, some problems only manifest themselves in particular scenarios, so verifying the systems defined by programmers is important. For this reason, we have implemented in VeriFx several CRDTs used by Antidote SQL (some of them have already been provided in [9]), as well as several mechanisms to guarantee invariants. This allows programmers to implement their most complex systems in VeriFx and define their properties. However, implementing systems in VeriFx is time-consuming, especially for programmers who are not familiar with the language. Therefore, it would be important to automate the verification process of applications that use Antidote SQL. This would save developers a lot of time when implementing code in VeriFx and would encourage the use of Antidote SQL by allowing developers to validate its specifications. However, to enable a wider range of verified applications and permissible semantics, the missing mechanisms, such as the *delete-wins* foreign key policy, would need to be implemented in VeriFx.

In addition to automating the verification process, it would be interesting if the analysis tool would not only provide counterexamples defining scenarios where properties are not verified, but also suggest the best conflict resolution strategies for each case. In the example from Section 5.3, where the problem was that the conflict resolution policies were not compatible to maintain the invariant, it would be very interesting if the analysis tool suggested the two possible options to solve the problem, i.e. if it suggested that either the last-writer-wins policy or the first-writer-wins policy should be used for the two integers.

Another very interesting feature for programmers would be to add a mechanism to the analysis tool that suggests less restrictive conflict resolution strategies. Imagine, for example, that a programmer has accidentally defined a table with the semantics *no concurrency*. It would be interesting if the tool could test not only the programmer's choices but also other concurrency semantics that suggest better alternatives than the ones chosen.

As for Antidote SQL, the system could be extended in the future to support other types of invariants.

Bibliography

[1] P. Alvaro et al. "Blazes: Coordination Analysis and Placement for Distributed Programs". In: *ACM Trans. Database Syst.* 42.4 (Oct. 2017). issn: 0362-5915. doi: 10.1145/3110214. url: https://doi.org/10.1145/3110214 (cit. on p. 19).

[2] P. Bailis et al. "Coordination Avoidance in Database Systems". In: *Proc. VLDB Endow.* 8.3 (Nov. 2014), pp. 185–196. issn: 2150-8097. doi: 10.14778/2735508.2735509. url: https://doi.org/10.14778/2735508.2735509 (cit. on pp. 8, 11–13, 35, 38, 39, 43).

[3] V. Balegas et al. "Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants". In: *Proceedings of the 2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. SRDS '15. USA: IEEE Computer Society, 2015, pp. 31–36. isbn: 9781467393027. doi: 10.1109/SRDS.2015.32. url: https://doi.org/10.1109/SRDS.2015.32 (cit. on pp. 38, 60).

[4] V. Balegas et al. "IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases". In: *Proc. VLDB Endow.* 12.4 (Dec. 2018), pp. 404–418. issn: 2150-8097. doi: 10.14778/3297753.3297760. url: https://doi.org/10.14778/3297753.3297760 (cit. on pp. 1, 2, 8, 15).

[5] V. Balegas et al. "Putting Consistency Back into Eventual Consistency". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015. isbn: 9781450332385. doi: 10.1145/2741948.2741972. url: https://doi.org/10.1145/2741948.2741972 (cit. on pp. 1, 8, 10, 13, 43).

[6] E. Brewer. "CAP twelve years later: How the "rules" have changed". In: *Computer* 45.2 (2012), pp. 23–29. doi: 10.1109/MC.2012.37 (cit. on p. 5).

[7] B. F. Cooper et al. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1277–1288. issn: 2150-8097. doi: 10.14778/1454159.1454167. url: https://doi.org/10.14778/1454159.1454167 (cit. on pp. 1, 9).

[8] L. De Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. isbn: 3540787992 (cit. on p. 31).

[9] K. De Porre, C. Ferreira, and E. G. Boix. "VeriFx: Correct Replicated Data Types for the Masses". In: *arXiv preprint arXiv:2207.02502* (2022) (cit. on pp. 3, 30–33, 46, 49, 50, 54, 83).

[10] K. De Porre et al. "ECROs: Building Global Scale Systems from Sequential Code". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485484. URL: https://doi.org/10.1145/3485484 (cit. on pp. 2, 22).

[11] G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 205–220. ISBN: 9781595935915. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281 (cit. on p. 12).

[12] A. Gotsman et al. "'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 371–384. ISBN: 9781450335492. DOI: 10.1145/2837614.2837625. URL: https://doi.org/10.1145/2837614.2837625 (cit. on pp. 8, 17).

[13] J. N. Gray, R. A. Lorie, and G. R. Putzolu. "Granularity of Locks in a Shared Data Base". In: *Proceedings of the 1st International Conference on Very Large Data Bases*. VLDB '75. Framingham, Massachusetts: Association for Computing Machinery, 1975, pp. 428–451. ISBN: 9781450339209. DOI: 10.1145/1282480.1282513. URL: https://doi.org/10.1145/1282480.1282513 (cit. on p. 29).

[14] F. Houshmand and M. Lesani. "Hamsaz: Replication Coordination Analysis and Synthesis". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290387. URL: https://doi.org/10.1145/3290387 (cit. on pp. 1, 18).

[15] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922 (cit. on p. 11).

[16] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563 (cit. on pp. 7, 46).

[17] C. Li et al. "Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 265–278. ISBN: 9781931971966 (cit. on pp. 1, 2, 9).

[18] W. Lloyd et al. "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 401–416. ISBN: 9781450309776. DOI: 10.1145/2043556.2043593. URL: https://doi.org/10.1145/2043556.2043593 (cit. on p. 8).

[19] P. Lopes. "Antidote SQL: SQL for Weakly Consistent Databases". In: 2018 (cit. on pp. 2, 27).

[20] P. Lopes et al. “Antidote SQL: Relaxed When Possible, Strict When Necessary”. In: *ArXiv* abs/1902.03576 (2019) (cit. on pp. 1, 2, 27, 37, 38).

[21] S. S. Nair, G. Petri, and M. Shapiro. “Proving the Safety of Highly-Available Distributed Objects”. In: *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*. Dublin, Ireland: Springer-Verlag, 2020, pp. 544–571. ISBN: 978-3-030-44913-1. DOI: 10.1007/978-3-030-44914-8_20. URL: https://doi.org/10.1007/978-3-030-44914-8_20 (cit. on p. 68).

[22] M. Najafzadeh et al. “The CISE Tool: Proving Weakly-Consistent Applications Correct”. In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*. PaPoC ’16. London, United Kingdom: Association for Computing Machinery, 2016. ISBN: 9781450342964. DOI: 10.1145/2911151.2911160. URL: https://doi.org/10.1145/2911151.2911160 (cit. on pp. 1, 2, 13, 17).

[23] D. Navalho, S. Duarte, and N. Preguiça. “A Study of CRDTs That Do Computations”. In: *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450335379. DOI: 10.1145/2745947.2745948. URL: https://doi.org/10.1145/2745947.2745948 (cit. on pp. 2, 46, 48, 53, 54).

[24] P. E. O’Neil. “The Escrow Transactional Method”. In: *ACM Trans. Database Syst.* 11.4 (Dec. 1986), pp. 405–430. ISSN: 0362-5915. DOI: 10.1145/7239.7265. URL: https://doi.org/10.1145/7239.7265 (cit. on p. 38).

[25] N. M. Preguiça. “Conflict-free Replicated Data Types: An Overview”. In: *CoRR* abs/1806.10254 (2018). arXiv: 1806.10254. URL: http://arxiv.org/abs/1806.10254 (cit. on pp. 2, 11).

[26] A. Ribeiro and P. G. Larsen. “Proof Obligation Generation and Discharging for Recursive Definitions in VDM”. In: *Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering*. ICFEM’10. Shanghai, China: Springer-Verlag, 2010, pp. 40–55. ISBN: 3642169007 (cit. on p. 17).

[27] M. Whittaker and J. M. Hellerstein. “Interactive Checks for Coordination Avoidance”. In: *Proc. VLDB Endow.* 12.1 (Sept. 2018), pp. 14–27. ISSN: 2150-8097. DOI: 10.14778/3275536.3275538. URL: https://doi.org/10.14778/3275536.3275538 (cit. on pp. 6, 24).

[28] P. Zeller, A. Bieniusa, and A. Poetzsch-Heffter. “Combining State- and Event-Based Semantics to Verify Highly Available Programs”. In: *Formal Aspects of Component Software: 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23–25, 2019, Proceedings*. Amsterdam, The Netherlands: Springer-Verlag, 2019, pp. 213–232. ISBN: 978-3-030-40913-5. DOI: 10.1007/978-3-030-40914-2_11. URL: https://doi.org/10.1007/978-3-030-40914-2_11 (cit. on pp. 1, 21).

A
Bounded Counter for Upper Bounds

The bounded counter for invariants of type less than or equal to K is very similar to the bounded counter that defines lower bounds. In Listing A.1 we present the implementation of methods that differ from the bounded counter for lower bounds.

In this counter, it is the increment operations that consume rights and the decrement operations that generate new rights. Therefore, the increment and decrement functions have been adapted to this behaviour.

The counter value is also calculated differently. The number of rights generated by the system must be subtracted from the bound value and the number of consumed rights must be added to it. For example, consider a bounded counter whose upper limit is 5, the initial value is 1, the number of rights generated by the system is 4 and the number of rights consumed by the system is 1 (an increment operation has already been performed). The counter value is 2 $(5-4+1)$, and this value makes sense since the initial value has been incremented by one unit. When a decrement operation is performed, the counter value becomes 1 $(5-5+1)$.

```
class BCounterLeq(rightsHold: GCounter, rightsConsumed: GCounter, bound: Int,
                  initialValue: Int = 0) extends CvRDT[BCounterLeq] {

  def value() = this.bound - this.rightsHold.value() + this.rightsConsumed.value()

  def increment(replica: Int, n: Int) = {
    if(this.localRights(replica) >= n)
      new BCounterLeq(this.rightsHold, this.rightsConsumed.increment(replica, n), this.bound,
       this.initialValue)
    else
      new BCounterLeq(this.rightsHold, this.rightsConsumed, this.bound, this.initialValue)
  }

  def decrement(replica: Int, n: Int) =
    new BCounterLeq(this.rightsHold.increment(replica, n), this.rightsConsumed, this.bound,
     this.initialValue)

  def invariant(): Boolean = this.value() <= this.bound
```

Listing A.1: Implementation in VeriFx of the bounded counter CRDT for invariants of the type *less or equal to K*.

B

Operation-based Exclusive Locks

The implementation of an example with exclusive locks in an operation-based replication model, is presented in Listing B.1. In this implementation, the same example as for the state-based replication model was used. In this model, the lock can only be represented by an integer.

Two operations are considered compatible if the identifiers of the replicas that have the lock are different. Otherwise, the value assigned in each operation must be the same in both operations (as the two operations must be the same).

```
class TxWithLockOpBased(v1: Int, v2: Int, exLock: Int)
        extends CmRDT[TxOp, TxOp, TxWithLockOpBased] {

  def addTransaction(value: Int, replicaId: Int) = {
    if(replicaId == this.exLock)
      new TxWithLockOpBased(value, value, this.exLock)
    else
      this
  }

  def invariant(): Boolean = this.v1 == this.v2

  def prepare(op: TxOp) = op // prepare phase does not add extra information

  def effect(op: TxOp) = op match
    case AddTx(value, replicaId) => this.addTransaction(value, replicaId)

  override def compatible(x: TxOp, y: TxOp) = x match {
    case AddTx(v1, id1) =>
      y match{
        case AddTx(v2, id2) => ((id1 == id2) =>: (v1 == v2))
      }
  }
}
```

Listing B.1: Example with exclusive locks in VeriFx (operation-based replication model).

The proof that is performed to check whether the invariant has been maintained is also listed below (Listing B.2). The proof checks whether the execution of operations in different sequences converges,

```
proof Tx_holds_invariant {
 forall(s1: TxWithLockOpBased, s2: TxWithLockOpBased, s3: TxWithLockOpBased, x: TxOp,
           y: TxOp) {

   val msg1 = s1.prepare(x)
   val msg2 = s2.prepare(y)

   (s1.compatible(msg1, msg2) && s1.compatibleS(s2) && s1.compatibleS(s3) &&
   s1.invariant() && s2.invariant() && s3.invariant()) =>: {

    s3.tryEffect(msg1).tryEffect(msg2) == s3.tryEffect(msg2).tryEffect(msg1) &&
    s3.tryEffect(msg1).invariant() && s3.tryEffect(msg2).invariant() &&
    s3.tryEffect(msg1).tryEffect(msg2).invariant()
   } }
}
```

Listing B.2: Proof to verify the preservation of the invariant using exclusive locks (operation-based replication model).

i.e. it checks whether the operations are commutative, and it checks whether the intermediate state and the final state hold the invariant.