Master Thesis

Verifying Serializability Protocols With Version Order Recovery

Author(s):
Clark, Jack

Publication Date:
2021

Permanent Link:
https://doi.org/10.3929/ethz-b-000507577

Rights / License:
In Copyright - Non-Commercial Use Permitted
Verifying Serializability Protocols
With Version Order Recovery

Master Thesis
Jack Clark
August 23, 2021

Advisors: Prof. Dr. Zhendong Su, Dr. Manuel Rigger
Department of Computer Science, ETH Zürich
Abstract

A core feature of many database systems is the ability to group operations into transactions. An isolation level defines the extent to which operations within a transaction interact with operations from other concurrent transactions. The serializable isolation level provides correctness guarantees that many programmers implicitly assume, since it provides the illusion of transactions running sequentially. However, implementing serializable transactions, particularly in a distributed setting, has proven to be a challenging task, with many systems failing to live up to their guarantees.

One method of providing assurance of the correctness of concurrency control protocols purporting to offer serializability is by verifying that the execution histories they produce are serializable. However, existing verifiers either have no guarantees on their worst-case execution time or require constraints on the type of operations they support, for example no verifier currently supports verifying histories containing predicate operations except in very special cases.

We introduce the concept of version order recovery, which recovers the version order from database systems while imposing fewer constraints than existing verifiers, that provides the means for efficient verification of serializable concurrency control protocols. We implement version order recovery for both PostgreSQL and TiDB, and show that it has minimal overhead on execution performance. We also implement a serializability verifier Emme and show that its performance is competitive with or better than existing verifiers. Additionally, when object visibility information can be recovered on a per-operation basis, we show that Emme supports the verification of moderate size histories containing predicate operations.

Due to the significant improvements in efficiency, broader testing applicability, and moderate implementation effort, we expect version order recovery to be widely adopted in practice.
Acknowledgements

I would like to thank Dr. Manuel Rigger for his supervision and willingness to provide detailed feedback on drafts of this thesis. I would also like to thank him for his career advice and his encouragement to pursue research opportunities that I wouldn’t otherwise have considered. I am thankful to Prof. Dr. Zhendong Su for providing me with the opportunity to work on this thesis in his research group.

Thanks must also go to my family, for supporting me through my time at ETH Zürich. Last but not least, thanks to my wonderful girlfriend Lou, for your constant support and putting up with some very unreasonable sleep schedules.
# Contents

1 Introduction 1

2 Background 4
   2.1 Serializability Theory 4
   2.2 The Adya Model 5
      2.2.1 The Database Model 6
      2.2.2 Transaction Histories 7
      2.2.3 Predicates 8
      2.2.4 Conflicts and Serialization Graphs 9
      2.2.5 Isolation Level Definitions 11
   2.3 Elle Checker 12
   2.4 Cobra Verifier 14

3 Version Order Recovery 16
   3.1 Deriving a Version Order 16
   3.2 Version Order Recovery API 18
   3.3 Implementing Version Order Recovery 19
      3.3.1 Log-based Change Data Capture 20
      3.3.2 Heap Scanning for MVCC Systems 23
   3.4 Summary 27

4 Test Clients and Histories 28
   4.1 Client Histories 28
      4.1.1 Emme Histories 30
      4.1.2 Elle Histories 30
      4.1.3 Cobra Histories 32
   4.2 The Test Client 33
      4.2.1 Supporting Predicate Operations 35

iii
## Contents

4.3 Summary ......................................................... 35

5 Verification of Serializability ................................. 37
  5.1 Emme ......................................................... 37
    5.1.1 Item Dependencies .................................... 38
    5.1.2 Predicate Dependencies ............................... 40
    5.1.3 Verifying Serializability ............................. 42
  5.2 Extending Cobra ............................................ 43

6 Evaluation ....................................................... 45
  6.1 Version Order Recovery ...................................... 45
    6.1.1 Correctness ........................................... 45
    6.1.2 Performance ........................................... 46
    6.1.3 Implementation Effort .................................. 46
  6.2 Emme ......................................................... 46
    6.2.1 Correctness ........................................... 46
    6.2.2 Performance ........................................... 47
  6.3 Cobra ......................................................... 49
    6.3.1 Correctness ........................................... 49
    6.3.2 Performance ........................................... 50
  6.4 Summary ....................................................... 52

7 Related Work .................................................... 53
  7.1 Formal Models of Isolation Levels ......................... 53
  7.2 Testing and Verification of Isolation Levels ............ 54

8 Conclusions and Future Work ................................... 57
  8.1 Conclusions ................................................ 57
  8.2 Future Work ............................................... 58

Bibliography ....................................................... 60
Chapter 1

Introduction

Database systems are widely used and form a core component of many critical software systems. They are typically large and complex, providing a wide array of features for their users. Whilst these systems typically already allow a significant degree of concurrency, many are now becoming distributed systems, introducing further complexity.

A core feature of many database systems is the ability to group operations into transactions [13]. All operations within a transaction are executed in sequential order; however, they may be interleaved with operations from other concurrent transactions. An isolation level defines the extent to which a transaction interacts with other concurrent transactions. To enforce its isolation level guarantees, a database system implements a concurrency control protocol [7].

One of the strictest isolation levels, serializability [6, 36], provides the level of isolation that many programmers implicitly assume since it provides the illusion of transactions running sequentially i.e. transactions are oblivious to modifications from other concurrent transactions. We refer to a concurrency control protocol guaranteeing serializability as a serializability protocol. At a weaker isolation level, many applications can suffer data loss, data corruption [24] or even serious security vulnerabilities [54]. Therefore, errors in a database system’s serializability protocol can cause significant harm, so assurance must be provided that it is correct.

Formal verification of database systems remains a challenging topic. Whilst there has been some progress, formal verification is still not practical for real-world database systems due to their sheer size and complexity [30]. However, formal verification methods have been more successfully applied to verifying distributed algorithms [18, 26, 28, 34, 57], including concurrency control protocols [16], and are seeing adoption in industry [32].

Testing plays an important role in providing assurance in real-world soft-
ware systems [15]. Automated randomized testing has proven to be an effective method of testing database systems [3, 43, 44, 45, 49]. A key challenge for randomized testing is to come up with an effective test oracle. For verifying serializability, such an oracle would take as input an execution history and verify that the history complies with the definition of serializability.

Verifying that a history is serializable is NP-complete [36]; similar results apply to some other isolation levels [8]. This limits naive randomized testing to very small histories. Existing verifiers [3, 8, 51] sidestep this problem either by placing constraints on the types of histories that can be generated or by aiming to achieve acceptable performance in practice but providing no guarantees for worst case behavior. In particular, no verifier is able to support verification of serializability for histories with predicate operations such as the SQL statement `SELECT * FROM t0 WHERE c0 > 7`. This is a significant limitation, as predicate conditions are a fundamental feature of many database systems.

Adya et al. [1, 2] provide a definition of serializability that can be efficiently checked against an execution history if additional information, namely the version order, is available. The version order is the per-object total order that a database system assigns to committed versions for that execution. However, this information is not readily exposed by database systems.

Recovery of the complete version order would allow efficient verification of the execution history produced, and by extension the correctness of the database system’s serializable isolation level for that execution. Elle [3] is a tool to verify isolation level implementations. It is able to recover the version order from a database system by encoding version order information into write operations. However, this imposes certain restrictions on the functionality that Elle can test.

In this thesis, we focus on the goal of efficiently verifying the correctness of serializability protocols. We demonstrate that for any execution of a serializability protocol it is possible to recover a version order that must lead to a serializable Adya history, otherwise there is an error in either the protocol or its implementation. We call this process version order recovery. To achieve this, we derive the version order from a protocol’s method of ordering versions. The insight is that for a serializability protocol to be correct, it must guarantee that some version order (consistent with its method of ordering versions) will lead to a serializable Adya history.

We show that the information necessary to derive the version order for many commonly used serializability protocols is simple and demonstrate that it can be recovered from PostgreSQL [40] and TiDB [19], using change data capture techniques, which provide a stream of changes to the state of a database. Furthermore, we show that it imposes far fewer constraints on the type of execution histories that can be generated by a test client, which
allows us to expose more database functionality to verification. Additionally, we show that the change data capture techniques we use have minimal overhead for the database system.

We implement a new verifier Emme based on the transaction isolation level definitions introduced by Adya et al. [1, 2]. We show that Emme can verify non-predicate histories with 2x better performance on large histories than existing verifiers. We illustrate the effectiveness of Emme by demonstrating that it can find a previously discovered error in PostgreSQL’s serializable isolation level [20, 38].

We also improve upon existing techniques by verifying predicate operations in execution histories when it is possible to recover per-operation object visibility information. We demonstrate that Emme is capable of verifying moderately sized (2500 transactions) PostgreSQL histories containing predicate operations in under 5 minutes. We are not aware of any other verifier that supports predicate checking for multi-version systems or single-version systems using a serializability protocol that does not rely on locking.

Finally, we demonstrate that it is possible to use version order recovery with other verifiers. We convert histories that fail to comply with some of the constraints of Elle into compatible Elle histories. Additionally, we show that version order recovery information can be used to improve the execution time of the Cobra [51] verifier, which can be useful in scenarios when only part of the version order can be recovered.
In this chapter we introduce the background material necessary to understand the contributions of this thesis. In Section 2.1, we give a brief overview of serializability theory. In Section 2.2, we start by introducing a model of transaction isolation levels by Adya et al. [1, 2], referred to as the Adya model hereafter. This is the most commonly used model for specifying isolation levels. It is the model used by both the Elle checker [3] introduced in Section 2.3 and the Cobra verifier [51] introduced in Section 2.4.

2.1 Serializability Theory

Multi-version serializability theory formally describes the notion of a history and what it means for a serialization protocol to be correct. We give a brief informal overview of multi-version serializability theory [6, 55] which lays the foundation for understanding our method of deriving a version order from the execution of a serialization protocol, which is described in Chapter 3.

A version function $h$ is a function which maps every read operation on an object in a traditional single-version history to a previous write operation on the same object. In other words, the version function is responsible for determining which version of an object is read by each read operation.

A multi-version history is a pair $m = (\text{op}(m), <_m)$, where $\text{op}(m)$ contains the result of applying the version function to all operations. $<_m$ is a partial order on $\text{op}(m)$ which requires that operations respect their transaction ordering and that for each read operation $r(x_i)$, the corresponding write operation $w(x_i)$ must come before the read in the history.

A multi-version history is a monoversion history if its version function maps each read object version to the last preceding write on the same object version. Roughly, multi-version histories are view equivalent if for both
histories identical read operations read the same object version. A multi-
version history is multi-version view-serializable (MVSR) if it is view equiv-
alent to some serial monoversion history. Note that single-version view-
serializability (VSR) is a strict subset of MVSR, therefore, this definition han-
dles that case.

A multi-version conflict in a multi-version history is a pair of operations
\( r_i(x_j) \) and \( w_k(x_k) \) such that \( r_i(x_j) <_m w_k(x_k) \). A multi-version history is
multi-version reducible if it can be transformed into a serial monoversion
history by exchanging the order of adjacent steps other than conflicting pairs.
A multi-version history is multi-version conflict-serializable (MCSR) if it is
multi-version reducible to a serial monoversion history. Note that single-
version conflict-serializability (CSR) is a strict subset of MCSR, therefore,
this definition handles that case. Additionally, MCSR is a strict subset of
MVSR.

Verifying if a multi-version history is in MCSR or MVSR is NP-complete [6,
36]. The complexity comes from the fact that checking the history means
searching for a monoversion serial history (which is totally ordered) among
many possible total ordered histories that can occur from the partial order.
In the next section, we will see that having access to a version order allows
efficiently verifying if a multi-version history is in MCSR.

2.2 The Adya Model

Adya et al. [1, 2] introduce a formal model for specifying transaction iso-
lation levels. Verifying that a system correctly implements an isolation level is
one of the main contributions of this thesis, so the Adya model provides the
basis for our work. Some of their definitions, with minor stylistic changes,
are reproduced in italics to distinguish their contribution from our own.

In the Adya model, a database consists of objects and the system allows
transactions to perform operations on those objects. A transaction’s oper-
ations may conflict with the operations of other transactions. Transactions
and their conflicts can be modeled using a direct serialization graph (DSG).
The nodes of the DSG are the transactions and the edges are the inter-
transactional conflicts. Isolation levels are defined mostly in terms of cycles
that can occur in a DSG.

Roughly, an isolation level specifies the extent to which a transaction in a
system will be isolated from other, potentially concurrent transactions. For
example, a transaction running at the strictest isolation level will execute as
if it is the only transaction running in a system.
2.2. The Adya Model

2.2.1 The Database Model

The Adya model defines a database to consist of abstract objects. This abstraction provides flexibility as it is possible to use the model with many different types of systems. For example, relational database systems (RDBMS) can be modeled using rows as objects, and key-value systems can be modeled with key-value pairs as objects. Without loss of generality, we will model rows as objects in the examples given throughout this section.

The Adya model assumes that a system supports transactions (as it is specifying transaction isolation levels) and that all operations are performed within transactions. However, single statement auto-commit can be supported by wrapping every operation outside of a transaction in its own transaction. Operations within a transaction are executed sequentially, however transactions may run concurrently. Transactions are able to interact with a system by issuing read and write operations. This is sufficient to capture most transactional behavior within a system, with the exception of predicate-based operations which are introduced in Subsection 2.1.3.

Each read and write operation is associated with a single object. A system will choose a specific version of an object in order to execute a particular read or write operation. Suppose a transaction \( T_i \) writes to an object \( x \). All writes produce a new version of the object \( x \). The final modification of \( x \) by transaction \( T_i \) is denoted \( x_i \).

Every object within a system has at least one version. An object \( x \) can have three different types of versions — the unborn type, the visible type and the dead type. \( x \) has an initial version, \( x_{\text{init}} \), also called its unborn version. The unborn version of every object exists in the system before any operations have been performed. The creation of all initial versions is handled by a virtual transaction \( T_{\text{init}} \). In a real system, no such transaction need take place. When \( x \) is inserted into the system, this is modeled as a visible version being created. If \( x \) is deleted from the system, this is modeled as its dead version, \( x_{\text{dead}} \), being created.

Object identity is not related to the concrete values of an object’s versions. All objects have a unique identity. This provides flexibility when considering the insertion of duplicate tuples. In this case, it is up to the system to determine whether duplicate values are allowed and if so, the system will select a unique object for each tuple.

Furthermore, consider the case where the system contains the dead version of an object \( x \), whose last visible version was \( x_i \). Suppose a transaction \( T_j \) wishes to insert a new tuple with identical values to \( x_i \). This is modeled as the creation of a new object \( y_j \).

Transactions may be in one of three possible states — uncommitted, committed or aborted. When a transaction commits, its final modifications become...
2.2. The Adya Model

part of the committed state of the system and the transaction is said to have installed these versions. A transaction may read versions created by committed, uncommitted or aborted transactions. Isolation levels are free to constrain this behavior, for example, some isolation levels forbid reading versions created by aborted transactions.

2.2.2 Transaction Histories

The Adya model defines the history of a system as follows:

Definition 2.1 (Adya History)
An Adya history $H$ over a set of transactions consists of two parts — a partial order of events $E$ that reflects the operations (e.g., read, write, abort, commit) of those transactions, and a version order, $<<$, that is a total order on committed object versions. Each event in a history corresponds to an event of some transaction, i.e., read, write, commit or abort.

The Adya model imposes three constraints on $E$, however they are not relevant to this thesis, so we do not reproduce them in full. Essentially, the three constraints ensure that a transaction’s events are ordered the same way w.r.t each other within $E$ as they are within the transaction.

The second part of an Adya history $H$ is the version order, $<<$. Understanding the version order is essential for understanding this thesis, as one of our main contributions is recovering the version order from real systems. Without the version order, verifying the serializability of an Adya history is NP-Complete [36]. The Adya model defines the version order as follows:

Definition 2.2 (Version Order)
The version order specifies a total order on object versions created by committed transactions in $H$. There is no constraint on versions due to uncommitted or aborted transactions. Versions created due to committed transactions in $H$ are referred to as committed versions. The following constraints are imposed upon the committed versions in $H$’s version order:

- the version order of each object $x$ contains exactly one initial version, $x_{\text{init}}$, and at most one dead version, $x_{\text{dead}}$.
- $x_{\text{init}}$ is $x$’s first version in its version order and $x_{\text{dead}}$ is its last version (if it exists). All visible versions are placed between $x_{\text{init}}$ and $x_{\text{dead}}$.

Additionally, systems are constrained to only permit reads of visible versions. The Adya model uses the following notation for read and write operations:

- Write operations: A write operation on object $x$ by transaction $T_i$ is denoted by $w_i(x_i)$. If the value $v$ is written into $x_i$, then the notation, $w_i(x_i, v)$ is used.
2.2. The Adya Model

- **Read operations:** \( r_j(x_i) \) is used to denote a read operation by transaction \( T_j \) that read the version \( x_i \) installed by transaction \( T_i \). To indicate that \( T_j \) has read \( x_i \)'s value to be \( v \), the notation \( r_j(x_i, v) \) is used.

2.2.3 Predicates

Many database systems support operations that are performed on all objects that match a predicate. Adya defines a predicate \( P \) to be a condition that returns true or false. It is worth noting that SQL uses ternary logic; predicates can return true, false or null. Therefore, we broaden the definition of a predicate to include predicates returning null. For an example of a predicate, consider the following query

\[
\text{SELECT * FROM } t_0 \text{ WHERE } c_0 > 7
\]

This query selects all rows in the relation \( t_0 \) whose \( c_0 \) column value is greater than seven. The predicate within this query is \( c_0 > 7 \).

The Adya model does not make any assumptions about the language used for specifying predicates, however it does assume that predicates are used within the context of a relational database system. However, this assumption is not strictly necessary as the notion of a relation can be extended to support many different system types.

In Adya’s model, the database is divided into relations and each object (with all its versions) exists in some relation. All predicates identify the relations on which they are applied. All objects that match the predicate are read or modified depending on whether a predicate-based read or write is issued.

We now introduce the Adya model’s definition of the version set of a predicate, its notion of a predicate-based operation and its definition of overwriting a predicate-based operation. These definitions will play an important role in defining predicate dependencies in the next subsection.

**Definition 2.3 (Version set of a predicate-based operation)**

When a transaction executes a read or write based on a predicate \( P \), the system selects a version for each object in \( P \)'s relations. The set of selected versions is called the **version set** of this predicate-based operation and is denoted by \( Vset(P) \).

We emphasize that the version set of a predicate-based operation includes versions of objects that do not match the predicate. Furthermore, the version set will possibly contain unborn and dead versions of some objects.

**Predicate-based Operations**

Consider a predicate-based operation with predicate \( P \), performed by a transaction \( T_i \). Conceptually, the system selects versions for all objects in the relations specified by \( P \). This is the version set. The system then evaluates the predicate \( P \) on the version set and performs the operation on the matching versions. Note that unborn and dead versions can never match a predicate.
2.2. The Adya Model

To represent predicate-based operations in a history, performed by transaction $T_i$, the following notation is used:

- **predicate-based reads:** $r_i(P: Vset(P))$.
- **predicate-based writes:** $w_i(P: Vset(P))$.

For both predicate-based read and write operations, the read and write events of matched objects are represented separately from the predicate-based operation itself. For example, consider a predicate-based read operation where versions $x_j$ and $y_k$ match the predicate. The history would contain: $r_i(P: Vset(P)) r_i(x_j) r_i(y_k)$.

**Definition 2.4 (Overwriting a predicate-based operation)**

A transaction $T_j$ overwrites an operation $r_i(P: Vset(P))$ (or $w_i(P: Vset(P))$) based on predicate $P$ if $T_j$ installs $x_j$ such that $x_k << x_j$, $x_k \in Vset(P)$ and $x_k$ matches $P$ whereas $x_j$ does not match $P$ or vice-versa. That is, $T_j$ makes a modification that changes the set of objects matched by $T_i$’s predicate-based operation. The notion of a write operation overwriting a predicate-based operation can be defined similarly.

We emphasize that a transaction only overwrites a predicate-based operation if it changes the set of objects that match the predicate. It does not overwrite the operation if it just modifies a version in the version set.

### 2.2.4 Conflicts and Serialization Graphs

The Adya model defines three different types of direct conflicts, and their associated dependencies, that can occur in a system. A direct conflict occurs when two different committed transactions perform an operation on the same object or intersecting predicates. These are read-dependencies, anti-dependencies and write-dependencies. The Adya model separates the definitions into predicate-based dependencies and item dependencies. This is an important distinction, as some isolation level definitions allow cycles with predicate-based dependencies but disallow cycles with item dependencies.

**Definition 2.5 (Directly Read-Depends)**

A transaction $T_j$ directly read-depends on transaction $T_i$ if it directly item-read-depends or directly predicate-read-depends on $T_i$. This is also called a write-read edge (denoted by $T_i \xrightarrow{wr} T_j$).

**Directly item-read-depends**

A transaction $T_j$ directly item-read-depends on $T_i$ if $T_i$ installs some object version $x_i$ and $T_j$ reads $x_i$.

**Directly predicate-read-depends**

A transaction $T_j$ directly predicate-read-depends on $T_i$ if $T_j$ performs an operation $r_j(P: Vset(P))$ and $x_i \in Vset(P)$. 

2.2. The Adya Model

The Adya model states that a transaction $T_j$ directly predicate-read-depends on the initialization transaction $T_{\text{init}}$ since $T_j$ observes the unborn versions of objects that have not yet been inserted in $P$’s relations. However, for finite histories this may not always be the case, since all objects may have been inserted before $T_j$ performs its predicate-based read, which would mean that the version set would not contain any unborn versions. If $T_j$ observes a dead version of some object, it directly read-depends on the transaction that deleted that object.

**Definition 2.6 (Directly Anti-Depends)**

A transaction $T_j$ directly anti-depends on transaction $T_i$ if it directly item-anti-depends or directly predicate-anti-depends on $T_i$. This is also called a read-write edge (denoted by $T_i \xrightarrow{\text{rw}} T_j$).

**Directly item-anti-depends**

A transaction $T_j$ directly item-anti-depends on transaction $T_i$ if $T_j$ reads some object version $x_k$ and $T_j$ installs $x$’s next version (after $x_k$) in the version order. Note that the transaction that wrote the later version directly item-anti-depends on the transaction that read the earlier version.

**Directly predicate-anti-depends**

A transaction $T_j$ directly predicate-anti-depends on $T_i$ if $T_j$ overwrites an operation $r_i(P: \text{Vset}(P))$. That is, if $T_j$ installs a later version of some object that changes the matches of a predicate-based read performed by $T_i$.

For example, suppose that there is a Vehicle relation where every tuple represents a type of vehicle. Transaction $T_j$ inserts a tuple $y_j$ that represents a bicycle. A further two transactions, $T_i$ and $T_k$, are issued. Transaction $T_i$ checks for all vehicles with two wheels, but reads $y_j$’s unborn version. Transaction $T_k$ checks for all vehicles with three wheels. This time, $y_j$ exists in the version set but it does not match the predicate, since a bicycle does not have three wheels. In this example, $T_j$ directly predicate-anti-depends on $T_j$ since it changed the matches of the predicate “all vehicles with two wheels”, however $T_j$ does not directly predicate-anti-depend on $T_k$ since it did not change the matches of the predicate “all vehicles with three wheels”.

**Definition 2.7 (Directly Write-Depends)**

A transaction $T_j$ directly write-depends on $T_i$ if it directly item-write-depends or directly predicate-write-depends on $T_i$. This is also called a write-write edge (denoted by $T_i \xrightarrow{\text{ww}} T_j$).

**Directly item-write-depends**

A transaction $T_j$ directly item-write-depends on transaction $T_i$ if $T_j$ installs a version $x_i$ and $T_j$ installs $x$’s next version (after $x_i$) in the version order.

**Directly predicate-write-depends**

A transaction $T_j$ directly predicate-write-depends on $T_i$ if either:
2.2. The Adya Model

1. $T_j$ overwrites an operation $w_i(P: Vset(P))$, or

2. $T_j$ executes an operation $w_j(Q: Vset(Q))$ and $x_i \in Vset(Q)$

Consider again the Vehicle relation. Suppose transaction $T_i$ inserts version $y_i$ that represents a bicycle. Transaction $T_j$ issues a predicate-based write with the predicate $Q = \text{“all vehicles with three wheels”}$ and updates all matching objects to be tricycles. Suppose $y_i \in Vset(Q)$. Transaction $T_j$ predicate-write-depends on $T_i$. Suppose Transaction $T_k$ issues a write operation $w_k(y_i, \text{wheels}=3)$. This represents updating the object $y_i$ to have three wheels. Since this changes the matches of predicate $Q$, $T_k$ predicate-write-depends on $T_j$.

Now that the definitions of the three direct dependencies have been explained, it is possible to introduce the definition of a Direct Serialization Graph (DSG).

**Definition 2.8 (Direct Serialization Graph)**

The direct serialization graph arising from a history $H$, denoted $DSG(H)$, is defined as follows. Each node in $DSG(H)$ corresponds to a committed transaction in $H$ and directed edges correspond to different types of direct conflicts. There is a read/write/anti-dependency edge from transaction $T_i$ to transaction $T_j$ if $T_j$ directly read/write/anti-depends on $T_i$.

One implication of the definition of a DSG is there can be at most one edge of a particular type from node $T_i$ to $T_j$ since the edges do not record the objects that gave rise to the conflict. This is important in reducing the number of edges that exist in a DSG.

### 2.2.5 Isolation Level Definitions

The Adya model defines Isolation levels in terms of various phenomena that must be avoided. We do not repeat the definitions for all phenomena or isolation levels, as they are not necessary in order to understand this thesis. However, we will briefly discuss the general form of these phenomena and then discuss two isolation levels whose definitions are important for understanding the contribution of this thesis.

All but two of the forbidden phenomena are defined in terms of directed cycles that occur in a DSG. The type of the directed cycle is determined by the type of the dependency edges that it contains, for example, a write-dependency cycle contains only write-dependency edges. The two non-cycle based phenomena are Aborted Reads and Intermediate Reads. Aborted Reads occur when a committed transaction reads from an aborted transaction. Intermediate Reads occur when a committed transaction reads a version of an object that was not the final version installed by a committed
transaction. Isolation levels are distinguished by the directed cycles that they forbid and the presence of Aborted Reads and Intermediate Reads.

We focus on the definitions of the PL-2.99 and PL-3 isolation levels. The PL-2.99 isolation level is equivalent to the ANSI Repeatable Read isolation level [4], while the PL-3 isolation level provides conflict serializability. At level PL-3, all phenomena are disallowed. This means that the DSG(H) of a PL-3 history H must be acyclic when considering all dependency edges. Additionally, H must not contain Aborted Reads or Intermediate Reads. The only difference between PL-3 and PL-2.99 is that PL-2.99 permits cycles with at least one predicate-anti-dependency edge.

Without knowing the predicate-dependency edges, it is not possible to distinguish between the Repeatable Read isolation level and the Serializable isolation level. Additionally, it is not possible to fully verify isolation level implementations, since predicate-dependency edges form a part of all cycle phenomena. This has important ramifications for verifiers and checkers.

It is worth considering the computational complexity of verifying that an Adya history H is conflict serializable. In order to verify that H is conflict serializable, it is necessary to check that the DSG(H) is acyclic and that H contains neither the Aborted Read or Intermediate Read phenomena. Checking that the DSG(H) is acyclic can be performed in time $O(V + E)$ by first using Tarjan’s algorithm [52] to find the strongly connected components of the DSG(H) and finally by using breadth-first search to check for directed cycles within each strongly connected component. Additionally, we need to check that H contains no Aborted Reads or Intermediate Reads. This can be done in time $O(W + R)$, where W and R are the number of committed writes and reads respectively, by creating a set of installed committed versions and checking that all committed reads produce values belonging to this set.

### 2.3 Elle Checker

Elle [3] is a tool designed to check that an Adya history is compliant with a specified isolation level according to the definitions introduced by Adya. It can check a wide variety of isolation levels and, when used in conjunction with the Jepsen testing framework [21], has been very successful in detecting bugs in real world systems, claiming that it has found at least one bug in every system that it has tested. In Chapter 5, we will show that version order recovery can be used to convert a history not compatible with Elle’s constraints into one that Elle can check.

Elle is a black-box checker, meaning that it deals with histories generated entirely in terms of client operations. This presents a challenge in mapping the Adya specification of dependency-edges, which are defined in terms of versions that may not be observable outside of the system’s internals, onto
Elle Checker

the observable values read and written by clients. To do this, Elle operates using a key-value model. Each read or write is associated with a key, which are unique across a history.

Elle is independent of the client and system used to generate a history. It lets the test designer decide how to map a particular system model, for example the relational model, onto its own key-value model. As is common with transaction isolation level checkers and verifiers, Elle places restrictions on the history that the client can generate. For Elle to work most effectively, it requires two properties in a history:

- **Recoverability**: this states that all read operations must be able to uniquely identify the write operation that produced the value read. This allows Elle to identify write-read edges and is a common requirement in most other checkers.

- **Traceability**: this states that for all committed write operations, it is possible to uniquely identify the previously committed write value that was overwritten. This allows Elle to identify read-write edges by finding all operations that read the previous value and adding a dependency between those transactions and the overwriting transaction. Additionally, it allows Elle to identify write-write edges, which are simply between the transaction that wrote the previous value and the overwriting transaction. Traceability amounts to having the version order.

A common pattern for enforcing recoverability is to ensure that the set of values written to a key contains no duplicate values. Enforcing traceability is slightly more complicated, as it requires recovering the version order, however it is key to enabling efficient checking of a history. To support traceability, Elle requires a system to support append operations for data types used in the test. Instead of issuing write operations, Elle requires all writes to be append operations. This ensures that the version order of a key is encoded in its value.

For example, SQL has the TEXT data type that supports the concat operation, which allows appending to a TEXT value in-place. Imagine a database with a single table that has an integer primary key column c0 and a TEXT column c1. Imagine we want to run a test where clients will read and write integer values. To make this compatible with Elle, write operations are simply replaced with concat operations that append a comma, followed by the TEXT representation of the integer value. Read operations will now read a TEXT value, which consists of a list of comma separated integer values. Recovering the value that would have been read in a traditional read/write model requires simple string manipulation to get the last value in the list. For example, imagine a read operation r(3, "1, 2, 3, 4"). This reads the key 3
and recovers the TEXT value, containing the latest value 4 as the last element of the list, along with the entire committed version order for key 3.

If a history has both the traceability and recoverability properties, then checking a given isolation level reduces to the same linear computational complexity described at the end of Section 2.1. Elle does not support predicate operations, so it is unable to add predicate edges. This is an important point, as the PL-2.99 (ANSI Repeatable Read) and the PL-3 (conflict serializability) isolation levels are distinguished by the presence of cycles containing predicate-anti-dependency edges. Therefore Elle is unable to distinguish between these two isolation levels. A client can enable Elle to add item-read-dependency edges for predicate reads by treating each result of a predicate read as its own read operation.

Elle is also incomplete or, in other words, can give false negatives. This is due to indeterminate operations. An indeterminate operation exists when the client does not know if it successfully completed or not. This can occur if there are failures in the system. For example the client may lose its connection with the database part-way through an operation. When all operations are determinate, Elle is complete and is able to verify (ignoring predicate operations) the history.

### 2.4 Cobra Verifier

Cobra [51] is an SMT solver based approach for verifying the serializability of key-value stores. In Chapter 5, we use Cobra to support the verification of histories with only partial version order information. Cobra can be used to verify a one-off history fragment, but it also supports the verification of a continuously running system by verifying history fragments in rounds. We focus on the one-off verification process.

Like Elle [3], Cobra is based on the Adya formalism for specifying isolation levels. It only supports read and write operations, therefore it cannot check the serializability of histories containing predicates. It requires that produced histories have the recoverability property, however it does not require traceability. Cobra introduces an additional constraint that every transaction reads and writes a key at most once.

To verify the serializability of a history H, Cobra tries to find an acyclic DSG(H). However, it does this quite differently to Elle. Cobra infers the write-read edges in H using the recoverability property. From this, Cobra constructs a directed graph, called the known graph. In the known graph, transactions are nodes and the inferred write-read edges are added.

Without traceability, Cobra is unable to immediately infer the write-write and read-write edges. Instead, Cobra introduces a set of constraints C, that
captures all possible, but unknown, read-write and write-write edges. A constraint represents a pair of edges $c = \{e_1, e_2\}$, which express that either $e_1$ is in the DSG(H) or $e_2$ is in the DSG(H). The known graph, combined with the set of constraints $C$, forms a polygraph $P$.

Cobra introduces the notion of compatibility. A directed graph $G$ is compatible with a polygraph $P$ if $G$ has the same known nodes and edges as $P$, and $G$ chooses one edge from each constraint. The Cobra authors demonstrate that there exists an acyclic directed graph that is compatible with the polygraph associated to a history $H$, iff there exists an acyclic serialization graph $G$ of $H$. If there is an acyclic directed serialization graph $G$ of $H$, then $H$ is serializable [48]. However, searching for a compatible graph is not efficient, as there are $2^{|C|}$ choices when considering which edges to include from the set of constraints $C$.

To improve its efficiency in practice, Cobra introduces a number of optimizations. We focus on the pruning optimization as this will be relevant in later chapters. Given a constraint $c = \{e_1, e_2\}$, Cobra recognizes that if choosing edge $e_1$ would cause a cycle in the graph, then it can remove the constraint and add the constraint’s other edge $e_2$ to the known graph. If this would also cause a cycle, then Cobra can immediately reject the history as not serializable. Once Cobra has performed all of its optimizations, it encodes the polygraph into an SMT verification problem instance. The encoding used is not relevant to this thesis.

A significant drawback of Cobra is that it still needs to perform an NP-Complete computation in order to verify a history. Cobra depends on its optimizations to reduce real-world histories to verification instances that are small enough to verify efficiently. When there are a significant number of blind writes (when a transaction writes to a key that it did not read first), Cobra’s performance degrades significantly, due to the fact that many of its optimizations rely on there being a sufficient number of read operations in the history.

Cobra does not support indeterminate operations, so by default it is complete. We point out that Cobra actually verifies strong session serializability [11]. If a history is strong session serializable, then it is serializable. However, the converse is not true. This enables Cobra to use session order edges to improve its performance.
Chapter 3

Version Order Recovery

Deriving and recovering version order information is a key contribution of this thesis and is crucial for achieving efficient verification of transactional database system histories. The version order is defined as a total order on committed object versions. Adya et al. [1, 2] define transaction isolation levels in such a way that database system execution histories can be efficiently verified when the version order can be recovered. However, database systems do not typically expose the version order. In fact, a database system may not even explicitly produce a version order internally.

In this chapter, we will show that for any serializable concurrency control protocol (serializability protocol hereafter) it is possible to derive a unique version order from an execution of the protocol, based on the proof of correctness of the serializability protocol. This version order must lead to a serializable Adya history if the serializability protocol and its implementation are correct.

We show how to derive and recover the version order for any concrete execution, which we call version order recovery. This requires two tasks: (1) recovering all of the versions in the system, and (2) ordering the versions identically to the system. We show how to achieve (1) using different methods of change data capture and also that the information required to perform (2) is often available from change data capture information. Finally, we demonstrate that version order recovery is achievable for real world systems, specifically PostgreSQL and TiDB.

3.1 Deriving a Version Order

A version order is a total order on the committed versions. It can be used to check for the existence of a monoversion serial history, by building a graph and checking if it is acyclic. The way the graph is constructed depends
on which type of serializability is being checked. For example, the Adya model defines edge types in such a way that MCSR is checked. Therefore, for a given version order, an efficient check exists for the serializability of a multi-version history.

However, a database system may choose any version order to use, therefore, with no further knowledge we must test all possible version orders to know if a multi-version history is serializable. Yet the manner in which database system order versions is well-defined and is a function of its serializability protocol. Ultimately, a serializability protocol is responsible for: (1) deciding whether a particular object version can be accessed or modified, and (2) whether to allow a transaction to commit its changes [56]. For any concrete execution, the serializability protocol must guarantee the existence of at least one version order that can be used to make the multi-version history equivalent (either view or conflict) to a serial monoversion history.

Therefore, we can derive a version ordering function for any serializability protocol. When applied to the versions resulting from a concrete execution $E$ of the serializability protocol, the version ordering function must produce a version order, which we call the recovered version order. This recovered version order guarantees that the multi-version history resulting from $E$ is serializable.

If the recovered version order leads to a serializable multi-version history, then the serializability protocol is correct for that execution (by definition). If the recovered version order does not lead to a serializable multi-version history, then the serializability protocol or its implementation is incorrect. This is true because its proof of correctness relies on the recovered version order making the multi-version history serializable for that execution. With this we can use the recovered version order to verify that a serializability protocol is behaving as expected.

For many protocols, the version ordering function resulting from the proof of correctness is simple. For example, for protocols offering commitment ordering [41], such as strict two-phase locking, the version ordering function orders versions in the commit order. For timestamp ordering protocols [42], the version ordering function orders versions in timestamp order. For optimistic concurrency protocols [25, 27], the version ordering function orders the versions in order of the timestamp given in the validation phase (i.e. the part of the protocol that serializes the execution). For certification protocols [14], the version ordering function orders versions in certification order.

PostgreSQL uses serializable snapshot isolation (SSI) [14, 37] to provide serializability. This is a certification based protocol, therefore the version order can be derived from the certification order. O’Neill and O’Neill [33] provide an algorithm to recover the serialization order (which gives the certification
order), however this requires modifications to the protocol. Instead, we observe that the version order derived from the certification order matches the version order derived from the commit order. Therefore, we can use the commit order to recover the version order in a manner consistent with the certification order.

TiDB [19] is a modern distributed HTAP database system. Unlike PostgreSQL, TiDB does not provide serializability. Instead, it offers snapshot isolation. Whilst we have only discussed serializable systems so far, we demonstrate that the concept can be extended to other isolation levels by recovering the version order for TiDB. Like SSI [14], snapshot isolation’s commit order implies its recovered version order. TiDB uses two-phase commit as part of its concurrency protocol. During this process, a “commit_ts” timestamp is generated and is used to determine the serialization order of transactions. It is attached to each committed version, therefore, we can use this “commit_ts” timestamp to order versions in their version order.

We do not aim to provide an exhaustive list of version ordering functions for different protocols. To aid verification efforts, we hope that database systems can be encouraged to explicitly declare their internal version order, or even better their serialization order, in some manner. As we have shown above, this is not much of an additional burden for most systems as the version order follows naturally from their serializability protocol. In the remainder of this chapter, we define a version order recovery API and explore the different ways in which version order information can be recovered from database systems that do not explicitly share their internal version order.

3.2 Version Order Recovery API

We define a simple API that supports the functionality demonstrated in Chapters 4 and 5. It abstracts away the details of any particular version order recovery technique and can be used by all of the verifiers covered in this thesis. The Python API is shown in Figure 3.1. The API consists of a single class, VersionOrder, and a function recover_version_order, which returns a VersionOrder object. The recover_version_order function has three arguments:

1. A function that recovers all versions (version_recovery_fn).

2. A function to uniquely identify the object associated with every version (object_identifier_fn). The notion of a unique object is abstract, since the API should flexibly handle different database system types. For relational database systems, we use the primary key, since this should be unique for each row.

3. A function to order versions (version_ordering_fn).
3.3. Implementing Version Order Recovery

The `VersionOrder` class defines a number of methods that operate on the underlying version order. The `get_prefix(version)` returns a prefix of the version order up to and including the provided version. The `precedes` method returns true if `version_1` comes before `version_2` in the version order. The `next_version(version)` and `previous_version(version)` methods return the next and previous version in the version order respectively. Finally, the `get_all_versions()` method returns a map of TID to a list of versions created by that transaction. All methods in the `VersionOrder` class are required to operate in constant time w.r.t the number of versions, with the exception of `get_all_versions()` and `get_prefix(version)`, which scale with the size of their output.

```python
def recover_version_order(version_recovery_fn, object_identifier_fn, version_ordering_fn) -> VersionOrder

class VersionOrder:
    def add_version(self, version)
    def get_prefix(self, version) -> [Value]
    def precedes(self, version_1, version_2) -> Boolean
    def next_version(self, version) -> Value
    def previous_version(self, version) -> Value
    def get_all_versions(self) -> {integer: [Value]}

Figure 3.1: Version order recovery API shown in Python code.
```

3.3 Implementing Version Order Recovery

This section demonstrates how change data capture (CDC) techniques can be used to implement version order recovery. We show that in addition to implementing the `version_recovery_fn` part of the version order recovery API, CDC often recovers sufficient information to implement the `version_ordering_fn` too. However this is not a requirement and is determined by both the underlying database system’s serializability protocol and the CDC implementation.

CDC tools provide a real-time stream of changes to the state of the database in a format that external applications can easily access. Figure 3.2 provides
3.3. Implementing Version Order Recovery

Figure 3.2: CDC architecture. Clients submit updates to a database and the CDC tool streams out the changes in an accessible format.

an overview of this process. CDC is a general concept not tied to any specific implementation. CDC tools can recover information in many different ways, however, common techniques include log-based CDC, polling and dual-writes.

A CDC tool is responsible for interfacing with different database systems and providing a coherent interface to external clients irrespective of the underlying database system. CDC has a number of use cases including cache-invalidation, populating secondary databases, building external indexes etc. The usefulness of this technique is demonstrated by its growing popularity, with LinkedIn building Databus [10], Facebook building Wormhole [47] and Debezium [12] providing an open source CDC platform.

The next two sections will demonstrate how we can implement the `version_recovery_fn` for PostgreSQL and TiDB using two different CDC techniques. Additionally, we show that in all cases, CDC provides the necessary information required to implement the `version_ordering_fn`. We believe many CDC tools go to extra effort to provide the information necessary to order versions because it is a desirable feature outside of the version order recovery context. Having a means of ordering the changes as they occurred in the database system simplifies the construction of applications that consume CDC data. For example, consider the cache population and invalidation use case. Without an ordering consistent with that of the database system, a cache may reach an invalid state if an earlier update was applied after a later update. There are ways for applications to sidestep this, for example using their own timestamps, but it is additional effort.

3.3.1 Log-based Change Data Capture

Database systems typically provide durability guarantees. Whilst other schemes exist, almost all real-world systems provide this using an ARIES based recovery protocol [31] that uses a write-ahead log (WAL). Whenever
3.3. Implementing Version Order Recovery

A change is made to the state of a database, the change is first written to the write-ahead log before it is written to the main database storage. If the database system crashes, the database state can be recovered from the WAL.

WALs are not confined to providing durability. Many database systems that support distributed operation use their write-ahead logs to provide replication from a leader node to follower nodes. Essentially, the leader node streams the changes to its WAL to follower nodes and they use this to replicate the state of the leader.

Many CDC tools implement log-based CDC. Log-based CDC tools recover changes by implementing a database specific WAL decoder or by directly subscribing to its replication protocol. Log-based CDC has many advantages. Firstly, it allows recovery of all committed changes to the state of the database. If any change is committed, then it will appear in the WAL. Secondly, it is fault-tolerant. If the database system crashes, then it will recover directly from the WAL using its recovery protocol. If the CDC tool crashes, it can restart and recover all committed changes by re-reading the WAL or re-reading the replication stream from its last known good position. This solves the indeterminate operation issue that Elle [3] suffers from. Since log-based CDC will report all committed changes, we will always know if a transaction committed or not, despite any client or database system failures.

For database systems whose commit order implies the recovered version order, version order recovery is simple using this approach.

**PostgreSQL**

In this subsection, we describe how we implement version order recovery using log-based CDC for PostgreSQL. We use theDebezium [12] CDC tool as it is open source and features detailed documentation, however, there is nothing Debezium provides that cannot be replicated in other implementations. The general architecture of Debezium CDC is very similar to that shown in Figure 3.2, with the exception of change events being output to Kafka [23] for later processing rather than directly to an application. Debezium can be used to both recover versions and order them.

To setup Debezium for PostgreSQL, no code modifications are required. However, PostgreSQL must be configured to use logical replication. In addition to logical replication, PostgreSQL needs to be configured to decode its WAL files into a format that consumers of the logical replication protocol can handle. This is known as logical decoding. There are a number of plugins that provide logical decoding. We use the pgoutput plugin, which is the standard logical decoding output plugin for PostgreSQL version 10 onward.

Debezium provides a Kafka connector that implements the subscriber side of PostgreSQL’s logical replication protocol. It retrieves change events via
3.3. Implementing Version Order Recovery

the protocol, transforms them into its own change event format and stores the change events in Kafka for further processing. To recover the change events stored within Kafka, it is necessary to implement a Kafka consumer, which will consume messages from Kafka and make them available to an application. Kafka uses the concept of topics to divide messages into logical streams. Debezium stores all changes from each table in a separate topic, therefore, it is necessary to consume messages from all relevant topics.

Debezium uses a JSON self-describing message format that includes its own schema. This format is consistent across database systems, so while the contents may change from system to system, the method of parsing does not. Figure 3.3 shows the important parts of an update event, other events such as inserts and deletes have a similar format. We omit the schema as it simply describes the “payload” field. The value of “u” in the “op” field indicates that this is an update event. The “source” field contains database system specific metadata related to the update.

The two fields that are most important for ordering versions are the “before” and “after” fields. These fields are not specific to Debezium’s PostgreSQL connector and Debezium includes this for most supported database systems, since it can be derived from the log-sequence number (LSN) in the WAL. The “before” field shows a row’s original column values before the update operation, and the “after” field shows the new column values for that row. If any column values were unchanged by an update, then they are identical. The important point is that the “before” and “after” fields respect the commit order. Therefore, the value in the “before” field appears directly before the value in the “after” field in the version order. This suggests a simple method of extracting the version order by simply sorting the updates so that changes are ordered w.r.t their “before” and “after” fields. In general, this is not quite enough since it would be ambiguous if there are duplicate values. Fortunately, all verifiers require unique writes, so test clients do not generate duplicate values.

Verifiers need to know the value written by each write operation. This is straightforward when issuing simple update statements such as UPDATE t0 SET c1 = 5 WHERE c0 = 2. In this example, c0 is the primary key, so we know that this statement will update exactly one row, which is the row identified by the primary key 2. Additionally, it is also clear which value is being written. However, consider the following update statement UPDATE t0 SET c1 = max(t0) WHERE c0 > 4. Without tracking the state of the database in a test client, it is impossible to know ahead of time which rows will be updated. Similarly, the result of max(t0) is not known. Current test clients are unable to provide this information without reading every value immediately after it is written.

Log-based CDC provides a means of recovering this information. Firstly, it is
possible to recover the value of complex update statements such as \text{max(t0)}. This is because it will be included in the CDC information by default. Associating modifications to a predicate operation is more complicated. This is because the Debezium CDC information does not contain information about which operation within a transaction the modification came from.

Typically, WALs use an LSN to uniquely identify every entry. Since transactions execute their operations in sequential order, each WAL entry caused by an operation from a transaction has a monotonically increasing LSN. Ordering changes from the same transaction by their LSN gives the order they were applied in w.r.t other changes from the same transaction. This is almost enough to associate a modification with its original operation. In addition, the test client should check the number of rows affected by every statement. With this information, it is possible to associate each modification with an original operation. The algorithm for this is straightforward. Iterate over each operation in a transaction in the order in which they were executed. For each operation, let \( n \) be the number of rows it changed. The next \( n \) modifications in the LSN sorted list of the transaction’s modifications belong to this operation.

Implementing the abstract version order API for PostgreSQL log-based CDC is simple. Version recovery is provided by definition, unique objects are identified by their primary key and the version ordering can be done using the “before” and “after” fields as described above.

### 3.3.2 Heap Scanning for MVCC Systems

An alternative CDC implementation technique is polling. This is where a CDC tool repeatedly queries a database to recover its state. In this section, we introduce a technique called heap scanning which recovers CDC information from Multi-Version Concurrency Control (MVCC) [6] database systems. CDC polling typically queries the database system explicitly. Instead, heap scanning repeatedly polls the database system’s underlying storage to recover changes. Traditionally, the on disk area where a database system stores its data is called its heap file, hence the term heap scanning.

MVCC is a common architectural pattern in database systems. Despite its name, MVCC is not actually a method of concurrency control and needs a concurrency control protocol to handle write operations. In a single-version system, each logical object has one physical version associated with it. In an MVCC system, each logical object can have multiple physical versions associated with it. For example, in a relational database system, each logical row may have multiple physical row versions. This is typically done to provide greater performance through increased concurrency. For read operations, despite having multiple physical rows, the database system must select only one (if any) of these rows to be used. This introduces the prob-
3.3. Implementing Version Order Recovery

```json
{
  "schema": {},
  "payload": {
    "before": {
      "id": 1,
      "name": "Anne"
    },
    "after": {
      "id": 1,
      "name": "Anne Marie"
    },
    "source": {
      "version": "1.6.1.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
      "snapshot": false,
      "db": "postgres",
      "schema": "public",
      "table": "customers",
      "txId": 556,
      "lsn": 24023128,
      "xmin": null
    },
    "op": "u",
    "ts_ms": 1465584025523
  }
}
```

Figure 3.3: Debezium change event format. This particular change event is for an update. It shows an update of the name column in the customers table from Anne to Anne Marie. This is a modified version of the example given in the Debezium documentation [12].

Problem of determining which (if any) physical row is visible to a particular operation. To achieve this, MVCC systems use logical timestamps and a set of visibility rules. In the next two subsections, we will show how we can use timestamp information collected via heap scanning to implement the `version_ordering_fn`.

**PostgreSQL**

PostgreSQL is an MVCC system. It uses two metadata columns, `xmin` and `xmax`, the status of transactions (i.e. committed, running or aborted) and a set of visibility rules to determine row visibility. For committed row versions, the `xmin` column contains the transaction identifier (TID) that wrote the
3.3. Implementing Version Order Recovery

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
<th>xmin</th>
<th>xmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.4: Example of PostgreSQL MVCC row versions.

version. The xmax column is slightly more complicated, so we simplify it here to capture the essence of how it works. The xmax column either contains 0, which marks the fact that this is the latest version written by any transaction, or it contains the TID of the transaction that wrote the next version. The notion of a next version suggests that the versions are written in a particular order. In fact, this is the version order. Therefore, by recovering all versions along with their associated xmin and xmax (and transaction statuses), it is possible to order the versions belonging to each unique object by using the chain of xmin and xmax TIDs.

Consider a table with an integer primary key column c0 and an integer column c1 (for simplicity we illustrate logical rows using their primary key values, however, primary keys are not a requirement). For a transaction Ti, assume i is the TID. Figure 3.4 gives an example of a PostgreSQL table containing multiple physical row versions. Object 1, identified by the primary key 1, has two physical versions. The version order for object 1 is [(1, 1), (1, 2)]. This can be seen by inspecting the xmin and xmax columns for the two versions. The physical row version with values (1, 1) has xmin 5 and xmax 7. Since its xmax field is not 0, there must be a later version with its xmin equal to the current version’s xmax. This later version is (1, 2). Since the xmax of the later version is equal to 0, there are no other versions after the later version. It is possible to deduce that the version with values (1, 1) is the earliest version since there are no versions with an xmax of 5.

We use heap scanning to recover the versions along with their associated xmin and xmax fields. PostgreSQL stores all physical tuple versions in a heap file. A heap file resides on disk and is essentially a collection of all physical tuple versions within the database. The heap file is just a regular file, so we can read through it to recover all physical versions. However, writing a heap scanner for a PostgreSQL heap file requires knowledge of PostgreSQL’s internal heap file format and will require updating whenever the format changes. Fortunately, the pageinspect [35] PostgreSQL module provides an interface to read through heap files. We can leverage this module to recover physical row versions.

In PostgreSQL, the heap file is broken up into fixed size blocks. Blocks are identified by integer indices, with the first being 0 and continuing consecu-
3.3. Implementing Version Order Recovery

tively. The pageinspect module provides an API which will retrieve all row versions belonging to a single block, and returns an error when it is supplied with a block index that is out of range. Therefore, to find all blocks and by extension all versions, it is possible to iterate from 0 onward until an error is returned. The pageinspect API can be called from within PostgreSQL, so it is possible to issue normal PostgreSQL queries and get the results in the usual relational format.

However, there are some issues with heap scanning PostgreSQL. MVCC implementations must eventually remove physical versions when it is impossible for any transaction to view them. Typically, this is done via garbage collection. If we are unlucky, physical row versions could be removed via garbage collection before we are able to scan them. This means that we may only be able to recover a subset of the versions and therefore the version order.

This problem can be avoided by turning off PostgreSQL’s garbage collector, however, this means that we are not capturing the behavior of an important part of PostgreSQL. Additionally, PostgreSQL actually relies on its garbage collector for correctness. The garbage collector is responsible for tuple freezing, which avoids issues with the wraparound of transaction IDs. Whilst this only happens for very long running systems and isn’t something that would cause issues when testing most features, it is nevertheless an imperfection.

In theory, it is possible to modify the garbage collector to output all physical versions that it scans, however, this requires modifying a complex piece of functionality. Additionally, it is possible for the garbage collector to remove physical row versions without having to scan them, due to certain optimizations. Therefore, it may be necessary to completely change the design of the garbage collector to facilitate this.

PostgreSQL implements an optimization called heap only tuples (HOT). Understanding the optimization is not necessary, other than to know that it causes pageinspect to fail to report the MVCC TIDs for certain rows. Whilst it may be possible to modify pageinspect to recover this information, it would only take a small change in PostgreSQL’s implementation to potentially break this, for example by introducing a new optimization.

In summary, it is possible to use heap scanning via the pageinspect module to recover version order information from PostgreSQL, however, it is brittle and relies on disabling important features of PostgreSQL.

**TiDB**

Many aspects of TiDB’s design are different to PostgreSQL, however, like PostgreSQL it uses MVCC. Whilst the details of TiDB’s MVCC implementation are different to PostgreSQL, fundamentally it is based on timestamps.
3.4. Summary

As discussed in Section 3.1 TiDB uses two-phase commit as part of its concurrency protocol. In TiDB, this is marked with a “commit_ts” timestamp, which is attached to each committed version. Therefore, we can use this “commit_ts” timestamp to order versions in their version order.

TiDB does not implement its own storage manager, choosing instead to use RocksDB [46] to provide its persistent storage. TiDB stores its MVCC row versions inside RocksDB as regular key-value pairs. This makes it easy to recover the MVCC row information, as it can be recovered by directly querying the underlying RocksDB database. TiDB provides an internal API to do exactly this. We wrote a small script to poll this API for all row versions in TiDB. Once we have all row versions, we can sort them by their “commit_ts” to get the version order.

Like PostgreSQL, TiDB must also use garbage collection to remove old row versions. This introduces the same type of garbage collection issues encountered with PostgreSQL heap scanning. We can disable garbage collection, which allows this method of version order recovery to work well. Unlike PostgreSQL, we didn’t find any issues with optimizations that caused issues recovering the version order.

3.4 Summary

In this chapter, we defined what it means to recover the version order from a database system. We introduced an abstract version order recovery API and demonstrated how we could implement this API using different forms of change data capture for various database systems.
A history captures information about an execution of a database system. Serializability theory [1, 2, 6, 36] defines histories precisely but abstractly. To be useful for the purposes of verification, we must more concretely define what we call a history and how they are generated, which is the focus of this chapter.

In the context of this thesis we assume that histories result from the execution of some test clients. These test clients exercise a database system by issuing transactions and observing the results. To allow for verification of the resulting history, test clients must constrain their behavior to match the limitations of the verifier that they will use. This gives rise to different types of histories, for example, the Cobra verifier [51] supports a key being read and written at most once in a transaction, and requires that transactions from the same client be ordered in the same order they were executed.

This thesis uses the Adya model [1, 2] of database systems. In the Adya model, a history is defined over a set of transactions and consists of two parts. The first is a partial order of events $E$, which captures the operations that transactions can submit to the system. We capture this partial order with the notion of a client history. The second part is the version order, which captures the internal ordering of versions chosen by the system. Version order recovery is the focus of Chapter 3. We will see that it is necessary to combine a client history with the version order to produce an Elle history.

### 4.1 Client Histories

Before describing the specific history types that are required by the verifiers considered in this thesis, we describe a History API that any specific type of history must implement. Figure 4.1 shows the first fragment of the API, the `History` class, in pseudo-Python syntax. The `History` class has three
simple methods. The first allows adding transactions, the second gets all transactions that have been added to the history and the third allows merging two histories. Merging histories allows test clients to generate histories separately and then combine them.

```python
class History:
    def add_transaction(self, transaction)
    def get_transactions(self) -> [Transaction]
    def merge(self, other_history)
```

Figure 4.1: The History class.

Figure 4.2 shows the Transaction class on which the History class depends. The Transaction class captures the essence of a transaction while preserving sufficient implementation flexibility. Transactions have a unique ID, an ordered collection of operations (in order of execution) and a commit status. When used as part of an Emme history, transactions also have a version set and a set of unborn versions, which will be explained in Chapter 5.

```python
class Transaction:
    def add_operation(self, operation)
    def get_operations(self) -> [Operation]
    def get_transaction_id(self) -> int
    def get_status(self) -> {COMMIT, ABORT, RUNNING}
    def set_status(self, commit_status)
```

Figure 4.2: The Transaction class.

The Operation class shown in Figure 4.3 represents the execution of some operation. An Operation has a query, a collection of results, a type and a flag to track if it is a predicate operation. A query is represented by the Query class and must be able to produce a string representation of the query. The type of an operation can be anything, but we must be able to tell if it is either a read or a write. Operations can be item operations or predicate operations (as defined by the Adya model).

We represent the results of an operation with the KVResult class. Using
4.1. Client Histories

A key-value model allows modeling flexibility and therefore support for a wide range of systems. Both keys and values are represented using the Value class. The key must uniquely identify an object in the database system through its get_value(self) method. Values can represent values from a range of different systems. For example, a value could be a single value in a key-value system, it could be multiple columns in a relational database or it could even be a JSON string representing a document in a document database. This can be done by refining the Value class to store an internal representation that matches the values of the database system under test, and then converting this into some string representation in the get_value(self) method.

4.1.1 Emme Histories

The Emme verifier, described in Chapter 5, supports a range of database system functionality. It produces the least restrictive histories out of all the verifiers we will consider. The major constraint that Emme histories must obey is ensuring that there are no duplicate values for a given key. This is a constraint for every history type that we will cover. As we will see in Chapter 5, this constraint allows every write operation to be identified.

A key feature of Emme is that it supports predicate operations. Therefore, an Emme history must also support predicate operations. In fact, the Operation class already supports predicates. However, Emme requires that all predicate operations in a history support evaluation.

We introduce a new class PredicateQuery shown in Figure 4.4. It adds an evaluate_predicate(self, kv_result) method which given a KVResult, will check if that KVResult would match the predicate condition. The PredicateQuery class is database system specific, as different systems have different operators and even different semantics for seemingly identical operators.

We use an abstract syntax tree (AST) interpreter to evaluate the queries. This is similar to the approach used in pivoted query synthesis [45]. However, instead of modifying the query to match the version, the predicate is evaluated on the version to see if it would be returned in its result set. Instead of using the AST approach, it is possible to send a modified version of the predicate to the database system for evaluation, however this approach is very inefficient and significantly bottlenecks the verification process.

4.1.2 Elle Histories

Elle [3] is a verifier with more constraints than Emme. Like Emme, it doesn’t allow duplicate values for a key. Figure 4.5 gives an example of an Elle history. Each transaction is represented within a pair of curly braces. Each
class Value:
    def get_value(self) -> String
    def get_name(self) -> String
    def get_type(self) -> String

class Query:
    def get_query_string(self) -> String

class KVResult:
    def get_key(self) -> Value
    def get_value(self) -> Value

class Operation:
    def add_results(self)
    def get_results(self) -> [KVResult]
    def is_read(self) -> Boolean
    def is_predicate(self) -> Boolean
    def get_query(self) -> Query
    def set_query(self, query)

Figure 4.3: The Operation, KVResult, Query and Value classes

A transaction has a :process identifier which gives the identifier of the client process that executed it, a :type value which is :ok means the transaction committed and a :value list, which lists the operations within the transaction. The operations can be read operations, represented by :r or append operations, represented by :append.

For the history shown in Figure 4.5, the first transaction has one operation, which appends 1 to the key :x. Keys can be any valid Clojure symbol. The second transaction has two operations, an append and a read. The append works similarly to the first append operation. The read operation returns a list of values that have been appended to :x, in this case [1 2].
4.1. Client Histories

class PredicateQuery:

    def get_query_string(self) -> String

    def evaluate_predicate(self, kv_result) -> Boolean

Figure 4.4: PredicateQuery class, which supports evaluating predicate conditions against a supplied KVResult.

Elle requires that all read operations return a prefix of the version order up to and including the current value, which is represented by a list of values. This means that a database system must support a data type with an append operation to be compatible with Elle.

{:process 1 :type :ok, :value [:append :x 1]}  
{:process 1 :type :ok, :value [:append :x 2 [:r :x [1 2]]]}

Figure 4.5: Example of Elle’s history format.

It is possible to convert an Emme history into an Elle history. This requires the version order. Figure 4.6 shows the function signature. Putting their representations aside, the only difference between an Emme history and an Elle history is that in an Emme history, read operations don’t have the version order encoded into their results. Therefore we can model an Elle history using the History class from the history API. Then, for every read operation in the history, we look up the keys and values returned (if any) in the version order and substitute the prefix of the version order (including the returned value) in place of the returned value. Once this is done, all that is left to be done is to convert the History object into Elle’s actual string format, which can be done easily.

def to_elle_history(emme_history, version_order) -> History

Figure 4.6: A function that converts an Emme history into an Elle history.

4.1.3 Cobra Histories

The Cobra [51] verifier imposes the most stringent history constraints when compared to Elle and Emme. Cobra requires that each transaction may read and write a particular object at most once. Cobra also requires that each transaction knows the previous transaction executed by the same client. This information is used as part of an optimization in Cobra. Additionally, every write operation must have a unique write ID (WID), every read operation must know the WID of the value that it returned and every read operation must also know the transaction ID (TID) of the transaction that wrote the
4.2 The Test Client

Whilst it is possible to assign these at execution time in the test client, it is also possible to assign them to a history once execution has finished. This is the approach we take. This can be done by iterating through all of the write operations in a history and assigning them a unique WID and keeping a map of WID to the TID of the write operation. Then, for every read operation, we add the WID and TID to it.

4.2 The Test Client

We implement a test client that is able to generate Emme, Elle and Cobra histories. By default it generates Emme histories, which can be converted to Elle histories. The test client must be specifically configured to generate Cobra histories as these impose constraints on the types of queries executed by the test client. The logic is independent of the database system under test and it uses a database specific query generator and executor to handle differences between systems.

The test client uses a key-value model, where each operation acts on a primary key and has one value column. This is a limitation of the test client only. We chose to do this because it simplifies the implementation of the test client and the test client is not a contribution of the work. We will see that for handling duplicate values, we actually introduce a second value column which proves that we can support more than a key-value model if needed.

```python
class TestClient:
    def execute_transaction(self, ops_per_txn, executor, query_generator):
        transaction = Transaction()
        executor.begin()
        shadow_txn = create_shadow_txn()
        transaction.version_set = shadow_txn.recover_version_set()
        for _ in range(ops_per_txn):
            operation = choose_operation(query_generator)
            operation.add_results(executor.execute(operation))
            transaction.add_operation(operation)
        status = executor.commit()
        transaction.set_status(status)
        return transaction
```

Figure 4.7: The main logic of the test client.

A simplified version of the test client logic is shown in Figure 4.7. A test client is configured to run a certain number of transactions, with every transaction having a fixed number of operations. The test client runs in a loop, executing transactions until it has no more work to do. The basic logic consists of choosing an operation type along with a suitable key (and value if
it is a write), executing the operation and then adding the operation to the currently executing transaction. Once a transaction has finished executing (either committed or aborted) it is added to the test client’s history.

The test client supports a number of different operation types. It supports inserts, updates, increments (for integer values) and reads. It also supports predicate updates and reads too. The exact ratio of operations is determined by the configuration settings of the test client, for example it is possible to specify that 10% of operations are inserts, 20% are updates and the rest are reads. The `choose_operation(self)` method ensures that the ratio of operations executed matches the configured ratio.

Once an operation is chosen, a particular object must be chosen for that operation. If the operation is an insert or an update, then a value should also be chosen. Choosing a value is potentially challenging, since duplicate values cannot be written to the same key. We avoid tracking duplicate values in the client by using a shadow column for each value. A shadow column is a normal `Value` type, however it always has an integer value which starts at 0 and it has an identical name to the normal column but with a "_shadow" suffix. For key-values stores and other systems which only allow a single value column, it is possible to pair another key-value pair with a value to act as the shadow column. For every operation that involves a write, the shadow value is incremented. When reading a value with a shadow column, we combine the normal value with the shadow value in the string representation, ensuring there is always a unique value. Shadow values allow multiple clients to be created and run simultaneously with no coordination since each client only operates using its own data.

If the test client is generating a Cobra history, it must ensure that a key is read and written at most once per transaction. As Cobra does not support predicates and therefore we do not generate them, it is very unlikely that a key will be chosen twice if the keyspace is large. Nevertheless, we keep track of a read and write key set for each transaction and if we encounter a duplicate key in one of the sets, we abort the transaction.

A database specific query generator is responsible for generating queries. The query generator we built is quite limited since it is not a contribution of the thesis. Currently, the query generator supports generating queries for an integer column only. This is sufficient to demonstrate the differences between Emme and Elle, since Elle does not support using integer types. We leave it to future work to integrate a more sophisticated query generator, such as the one from SQLancer [50].
4.2.1 Supporting Predicate Operations

To support verification of predicate operations, it is necessary to recover the version set of each transaction. The version set is defined in Chapter 2. We define a function that must be implemented for every database system called 
\texttt{recover\_version\_set\_info(self)}. In order to implement this function, it must be possible to recover per-operation object visibility information from the underlying database system. The version set does not need to be recovered directly by the test client during execution, so long as sufficient information is recorded to enable offline recovery.

For PostgreSQL, the test client recovers the version set directly during execution. We use PostgreSQL’s snapshot export functionality [39] to accomplish this. PostgreSQL uses a snapshot mechanism to determine which object versions are visible to a particular operation. At lower isolation levels such as read committed, this changes on a per-statement basis, however, at the serializable level this snapshot is consistent for the whole transaction.

We introduce the notion of a shadow transaction. A shadow transaction is paired with a normal transaction and uses the snapshot exported from the normal transaction and is therefore able to see a consistent view of the database. The shadow transaction can recover a superset of the initial version set of the normal transaction, by issuing a \texttt{SELECT} operation for all tables in the database. In a more sophisticated test client, it would be possible to know which relations are accessed by predicate operations and only recover the data for those relations. The benefit of using shadow transactions is that they avoid biasing the operations of the normal transactions. For example, with a shadow transaction it is possible to have the normal transaction issue write operations only, and have the shadow transaction issue the necessary reads. This allows a wider range of transaction patterns to emerge. Shadow transactions are not quite enough though, since there may be unborn or dead versions in the version set too. These can be recovered by taking the set difference of the keys recovered in the shadow transaction and all keys ever written to the database system.

Note that version set recovery does not depend on this feature of PostgreSQL. It is possible to recover snapshot information from PostgreSQL using other mechanisms and then use this information offline to determine the version set by applying PostgreSQL’s visibility rules to the recovered versions. PostgreSQL’s visibility rules are reasonably complicated, so to avoid re-implementing them, we leverage the snapshot export feature.

4.3 Summary

In this chapter, we have demonstrated the different types of histories that we can generate, along with the test client that generates them. We have shown
4.3. Summary

how it is possible to combine recovered version order information with an Emme history to convert it into an Elle history. Finally, we have shown how to recover version set information from PostgreSQL directly.
Chapter 5

Verification of Serializability

The Adya model [1, 2] introduces a database system model and isolation level definitions in terms of that model. In this chapter, we consider the verification of the serializable isolation level defined in the Adya model. We introduce a verifier Emme, which is able to verify Adya histories that include predicate operations, a feature that current verifiers do not support. We also show how the Cobra [51] verifier can be extended to use recovered version order information to improve its execution time, which is particularly useful in cases when only partial version order information can be recovered.

5.1 Emme

Emme is designed to verify the serializability of an Adya history. An Adya history consists of both a partial order of events $E$ and a version order. Emme is implemented in Python and depends on the Version Order Recovery API described in Chapter 3, which captures the version order, and the History API described in Chapter 4, which captures the partial order of events $E$.

An Adya history $H$ is serializable if the direct serialization graph $DSG(H)$ is acyclic and $H$ does not contain certain anomalies. The nodes of the $DSG(H)$ are committed transactions and the edges result from dependencies between transactions. The full list of dependencies is described in Chapter 2.

There are also two anomalies that the Adya model defines that are not defined in terms of cycles in the $DSG(H)$. These anomalies are aborted reads and intermediate reads. An aborted read occurs when a committed transaction includes an operation that reads a value from an aborted transaction. An intermediate read occurs when a committed transaction reads the version of an object for which there is a later version installed in the same transaction.
Like Elle [3], we include new anomalies that must be avoided at the serializable isolation level. The first is a garbage read anomaly, which occurs when a read operation reads a version that was not produced by any write operation in the history. The second is a duplicate write anomaly, which occurs when a version is written more than once. The third is an internal inconsistency anomaly, which occurs when a read operation observes a version of an object that was not the version written by a previous write operation within the transaction, if such a write operation exists.

Before Emme starts to infer dependencies, it builds a map where the keys are versions and the values are the transaction ID that wrote version. During this process, Emme checks for duplicate writes. Additionally, Emme filters all uncommitted transactions from the history, which reduces the workload of the next stages. After this, Emme can start to build the DSG(H). It adds all committed transactions as nodes, identified by their TID, as well as adding a transaction that represents T\textsubscript{init}, which is identified by the string “init”. Once all of the nodes have been added, dependencies can start to be inferred.

We split dependencies into two categories: item dependencies and predicate dependencies. To aid understanding, we explain how to infer the dependencies for each category separately, however, in reality we combine the inference of both categories into a single function to improve performance. Inference of item dependencies is explained in Subsection 5.1.1 and inference of predicate dependencies is explained in Subsection 5.1.2. Both categories of inference are performed on a per-transaction basis, which enables a high degree of parallelism in the inference process.

### 5.1.1 Item Dependencies

Figure 5.1 shows \texttt{get\_item\_dependencies} function, which infers item dependencies for a transaction and returns those dependencies as a set. It iterates over every KVResult that is produced by an operation in the transaction. This is necessary because the Adya model defines dependencies in a data-driven fashion. There are three types of item dependencies to infer; read-dependencies (wr), write-dependencies (ww) and anti-dependencies (rw). There must also be a check for each of the anomalies that are forbidden.

An item read-dependency from T\textsubscript{i} \(\xrightarrow{wr} T\textsubscript{j}\) occurs when an operation in a transaction T\textsubscript{j} reads a version written by a transaction T\textsubscript{i}. These can only occur from read operations. Each read version is checked to ensure that it is not an intermediate, aborted or garbage read. Garbage and aborted reads can be checked by ensuring that the version exists in the version\_to\_tid map, which contains all committed versions, and if not an error is reported. The version\_to\_tid map tracks if a version was the last version written for the same object within the same transaction. Intermediate reads can be
def get_item_dependencies(transaction, version_order, version_to_tid):
    edges = set()
    for operation in transaction.get_operations():
        for result in operation.get_results():
            if operation.is_read():
                is_garbage, err = is_garbage_or_aborted_read(result, version_order)
                is_intermediate, err = is_intermediate_read(result, version_order)
                if is_garbage or is_intermediate:
                    raise err

                read_from_tid = version_to_tid[result.key][result.value]
                if transaction.tid != read_from_tid:
                    edges.add([read_from_tid, transaction.tid])

                next_version = version_order.next_version(result.key, result.value)
                if next_version is not None:
                    anti_dep_tid = version_to_tid[result.key][next_version]
                    if anti_dep_tid != transaction.tid:
                        edges.add([transaction.tid, anti_dep_tid])
                else:
                    previous_version =
                    version_order.previous_version(result.key, result.value)
                    if previous_version is None:
                        edges.add(["init", transaction.tid])
                    else:
                        overwritten_tid = version_to_tid[result.key, result.value]
                        if overwritten_tid != transaction.tid:
                            edges.add([overwritten_tid, transaction.tid])

    return edges

Figure 5.1: The algorithm for inferring item dependencies for a transaction.

detected by looking up this information. If there are no anomalies, then the read version’s TID is retrieved from the version_to_tid map and a dependency is created.

An item anti-dependency from $T_i \rightarrow T_j$ occurs when an operation in transaction $T_i$ reads a version $x_k$ and transaction $T_j$ writes the version $x_j$ that appears directly after $x_k$ in the version order. Assume a read operation in transaction $T_i$ results in a read of version $x_k$. The version order recovery API provides a method next_version(version_key, version_value) which will return the next version $x_j$ in the version order after $x_k$. This can be used to determine that there is an anti-dependency between $T_i$ and $T_j$.

An item write-dependency from $T_i \leftarrow T_j$ occurs when a write operation in a transaction $T_j$ writes the next version $x_i$ in the version order after the version $x_i$ produced by transaction $T_i$. The version order recovery API provides a method previous_version(version_key, version_value) that can be used to find the version $x_i$ given a write operation that produces the version $x_j$. This makes it easy to create the dependency between $T_i$ and $T_j$. 

39
5.1. Emme

Finding all item dependencies in a history has time complexity $O(V)$ where $V$ is the number of versions in the history. This is because we iterate over every version and every operation that we perform on a version has constant time complexity.

5.1.2 Predicate Dependencies

There is a predicate version of each item dependency. We split these into two categories: version set dependencies and overwrite dependencies. This better matches the way in which these dependencies are computed. In the Adya model the version set of a predicate operation contains a version of each object that the database system selects for evaluation against the predicate, irrespective of whether or not it matches the predicate. Both read and write predicate operations produce version set dependencies. Overwrite dependencies occur when a write operation changes the matches of another predicate operation.

Figure 5.2 shows the `get_predicate_dependencies` function. First, it computes the unborn versions and attaches them to the transaction. It then iterates over each operation in a transaction inferring the predicate dependencies resulting from each operation. Finally, to reflect the changes made by each operation within the transaction so far, the version set is updated. Changes from other transactions are irrelevant, since PostgreSQL uses a fixed snapshot on a per-transaction basis, so every operation sees the same snapshot. For PostgreSQL histories containing predicates, we capture object visibility information on a per-transaction basis, which we convert to a per-operation basis via manually updating the version set after each operation.

```python
def get_predicate_dependencies(transaction, version_order, version_to_tid):
    edges = set()
    if not transaction.contains_predicate:
        return edges
    transaction.unborn_versions = compute_unborn_version()

    for operation in transaction.get_operations():
        if operation.is_predicate():
            edges.add(get_version_set_edges(transaction, operation, version_to_tid))
            edges.add(get_overwrite_edges(transaction, operation, version_order, version_to_tid))
            update_version_set(transaction, operation)

    return edges
```

**Figure 5.2:** The algorithms for inferring predicate dependencies for a transaction.

Computing the version set dependencies for an operation is simple. Figure 5.3 shows the `get_version_set_dependencies` function which does this.
First, if the operation’s version set has unborn versions, a dependency is created on $T_{\text{init}}$. Then, for each version $x_i$ in the version set, a dependency is created from $T_i$ to the transaction that the operation belongs to. Every version is checked to ensure that it is not an intermediate read, since intermediate versions should never be visible to another transaction.

```python
def get_version_set_dependencies(transaction, operation, version_order, version_to_tid):
    edges = set()
    if transaction.has_unborn_versions():
        edges.add(["init", transaction.tid])
    for key, value in transaction.version_set:
        vset_tid = version_to_tid[key][value]
        is_intermediate_read, err = is_intermediate_read(key, value)
        if is_intermediate_read:
            raise err
        if transaction.tid != vset_tid:
            edges.add([vset_tid, transaction.tid])
    return edges
```

Figure 5.3: The algorithms for inferring version set predicate dependencies for an operation.

Overwrite dependencies occur when a write operation produces a version that would change the matches of a predicate operation. This is called overwriting a predicate operation. For every predicate operation, we must consider every version in the system, which makes computing the overwrite dependencies an expensive operation, however this is inherent in the definition of overwriting provided by the Adya model.

Figure 5.4 shows the `get_overwrite_dependencies` function. For each written version $x_j$, it first evaluates if $x_j$ matches the predicate operation’s predicate condition. It then checks if the version $x_i$ in the version set of the predicate operation, which is associated with the same object as $x_j$, has the same matching status. If either both versions match or neither version matches, then there is no dependency, since overwrite dependencies only occur when a written version changes the match of a predicate operation. If $x_j$ does change the match and $x_i$ precedes $x_j$ in the version order, then an overwrite dependency is created.

The computational cost of finding all predicate dependencies in a history is higher than for item dependencies. For every predicate operation the function `get_predicate_dependencies` is executed once, and iterates through all versions produced in an execution. Therefore, the time complexity is $O(P \times V)$ where $P$ is the number of predicate operations in the history and
def get_overwrite_dependencies(transaction, 
    operation, 
    version_order, 
    version_to_tid):
    edges = set()
    for tid, txn_versions in version_order.get_all_versions():
        if tid == transaction.tid:
            continue
        for version_key, version_value in txn_versions:
            originally_unborn = version_key in transaction.unborn_versions
            originally_matches = version_key in operation.get_results()
            new_version_matches = operation.get_query().evaluate_predicate(version_value)
            if originally_matches == new_version_matches:
                continue
            if originally_unborn and new_version_matches:
                overwritten_tid = version_to_tid[version_key][version_value]
                edges.add((transaction.tid, overwritten_tid))
                continue
            if originally_unborn and not new_version_matches:
                continue
            original_value = transaction.version_set[version_key]
            new_value_precedes_original = 
                version_order.precedes(version_key, version_value, original_value)
            if not new_value_precedes_original:
                overwritten_tid = version_to_tid[version_key][version_value]
                edges.add((transaction.tid, overwritten_tid))
    return edges

Figure 5.4: The algorithms for inferring overwrite predicate dependencies for an operation.

\( V \) is the number of versions in the history.

5.1.3 Verifying Serializability

When all dependencies have been inferred and added to the DSG(H), it is possible to verify the serializability of the history H. It is assumed that any non-cycle anomalies have been discovered in the inference process. Therefore, all that is left to do is to check if the DSG(H) is acyclic. This is a straightforward operation which can be completed efficiently using a number of different algorithms. If a cycle is found, then the history is not serializable. The total time complexity of verification is \( O(P \times V) \) where \( P \) is the number of predicate operations in the history and \( V \) is the number of versions in the history.
Currently, Emme does not try to produce a small counterexample when it finds a cycle. In the future, we plan to support this functionality to aid in the debugging process. As shown in the Elle [3] paper, this is achievable by finding strongly connected components and performing a breadth first search on each component to find small cycles. Emme does not label the edges resulting from dependencies with their dependency type, since all cycles are disallowed at the serializable isolation level. In the future, dependency types will be added to aid in providing understandable counterexamples.

## 5.2 Extending Cobra

Cobra [51] is an SMT solver based approach for verifying the serializability of key-value stores. Cobra can be used to verify a one-off history fragment, but it also supports the verification of a continuously running system by verifying history fragments in rounds. We focus on the one-off verification process. By default, Cobra uses the order of transactions within a client to infer client order edges, which it uses to improve its execution time. However, this causes Cobra to verify strong session serializability instead of serializability. A history is serializable if it is strong session serializable, however the converse is not necessarily true. We modified Cobra to optionally turn off client order edges so that it could verify serializability.

We outline how Cobra works in Section 2.4. To verify the serializability of a history $H$, Cobra tries to find an acyclic $DSG(H)$. Cobra infers the write-read edges in $H$ using the recoverability property, however it is unable to directly recover the read-write and write-write edges because it does not have access to the version order. To handle this, Cobra introduces the notion of a constraint, which expresses the uncertainty in the ordering of transactions that arises from the lack of these edges.

We modify Cobra to be able to use recovered version order information. This is particularly useful in scenarios where it is not possible to recover all the version order information from a system. Without all version order information, Emme is unable to fully verify the serializability of a history. However, Cobra is able to do exactly this.

To export version order information in a format that Cobra easily understands, we define a simple file format, where every line contains version order information for a single key. Each line starts with the key, followed by the number of versions and then a list of versions. Cobra uses a hash of both a key and its value, so we must output the key and versions in this format.

Our modification to Cobra is fairly simple. First, all version order information is read into a map. Then, for each adjacent pair of versions in a single key’s version order, a version order edge is added to Cobra’s known graph. There is nothing special about these edges. This explicitly creates all of the
write-write conflicts that are known from the version order. However, there are also read-write conflicts that arise. We do not require any further modifications to handle these, as Cobra already includes an optimization which is able to identify these.

Just adding the write-write and read-write edges is not enough to improve Cobra’s execution time. In addition, it is necessary to remove the associated constraints, as it is the number of constraints that determines Cobra’s execution time. Fortunately, Cobra contains a pruning optimization that is able to do exactly this. It identifies redundant constraints (constraints that are already encoded in the known graph) and removes them from the graph. This is enough to realize the performance improvements that the write-write and read-write edges offer.
Chapter 6

Evaluation

In this section, we evaluate the correctness and performance of both version order recovery and Emme. Unless stated otherwise, all experiments are carried out on a machine with a quad-core i7-4800MQ processor, 16GB RAM, SATA III SSD and an Ubuntu 18.04.5 LTS operating system. We use PostgreSQL version 13.3 and TiDB version 5.1.1.

6.1 Version Order Recovery

In this section, we evaluate the correctness and performance of the version order recovery implementations described in Chapter 3. There are three implementations: (1) log-based change data capture for PostgreSQL, (2) heap-scanning for PostgreSQL [40] and (3) heap-scanning for TiDB [19].

6.1.1 Correctness

We developed a test suite that is compatible with all implementations. We created a suite of basic hand-written examples to serve as a basic correctness test. More extensive testing was carried out using two types of randomized testing: (1) increment tests and (2) Elle differential tests.

The increment tests create a random history containing read operations and increment operations, including predicate conditions, acting on integer columns. Increment tests generate a known version order (i.e. a sequential list of integers). They are valuable because they can find errors in both the version ordering process and the version recovery process. We ran hundreds of randomized increments tests for various history sizes and number of clients. All tests passed successfully. Tests used a single table only, but support multiple columns. The tests support an arbitrary number of clients, allowing for a high degree of concurrency. To implement increment opera-
6.2. Emme

6.2.1 Correctness

We implemented a number of different test strategies to check the correctness of Emme. To test that Emme can find anomalies, we implemented twenty unit tests where each test had a unique type of anomaly. Emme was able to find all anomalies. To test that Emme could successfully verify histories that were serializable, we generated fifty histories that were known to be serializable. These histories had various different sizes, different numbers of clients and different mixes of operation types. Emme successfully verified all fifty histories.

In addition to the unit tests, we carried out randomized differential testing of Emme. We used the test client described in Chapter 4 to generate histories from PostgreSQL. The version order was recovered from PostgreSQL using the log-based change data capture approach. Then, an Elle history was created from the Emme history using the to_elle_history function. Finally, we ran both Emme and Elle on the histories and ensured they agreed on the result.
We were able to demonstrate that Emme, combined with version order recovery, was able to detect a known error in PostgreSQL 12.3’s serializable isolation level [20, 38]. The error is a well-known isolation anomaly that can occur in serializable snapshot isolation implementations [37]. This shows that Emme can detect serializability violations in real-world database systems.

6.2.2 Performance

This subsection details the performance characteristics of Emme. A single table with a key, value and shadow column is used for all experiments. Insert operations use an ON CONFLICT ... DO UPDATE clause, which will update a key if it already exists in the table. Increment operations are implemented as a read operation followed by an update operation, which captures a read-modify-write pattern that is common in real transactions.

Figure 6.1: Demonstrates the effect of increasing the number of transactions in a history on the execution time of item dependency verification in the Emme and Elle verifiers. Each transaction executed five non-predicate operations.

Figure 6.1 compares how both Emme and Elle scale with the history size when verifying histories without predicate operations. The execution time is solely comprised of the verification time. Elle histories are first written to disk and then read using Elle’s read-history function, whereas Emme histories are in-memory. To avoid penalizing Elle for this, we only measure the time Elle spends verifying the history once it has loaded it into memory. The experiment used a mix of 30% updates, 40% reads, 20% increments and 10% inserts, which gives a 50/50 read/write ratio. Emme performs similarly to Elle for smaller history sizes, but then starts to outperform Elle as the history size increases. We believe this is mostly due to Elle’s list
format for reads, which requires that each read stores its version order. As
the history increases in size, each key accumulates more and more versions,
which makes dealing with each read progressively more expensive.

Figure 6.2: Demonstrates the effect of increasing the number of transactions in a history on the
execution time of mixed item and predicate dependency verification in the Emme verifier. Each
transaction executed a mix of five predicate and non-predicate operations. Different ratios of
predicate to non-predicate operations were chosen.

Figure 6.2 shows the performance of mixed item dependency and predicate
dependency verification in Emme. The experiment explored how changing
both the ratio of predicate operations to key-value operations and the num-
ber of transactions in the history affected performance. The experiment used
an operations mix of 10% insert operations and a 50/50 ratio of reads and
updates for both key-value and predicate operations. The number of keys
was set to a maximum of 100. Predicate checking has $O(P \times V)$ complexity,
where $P$ is the number of predicates in the history and $V$ is the total num-
ber of versions. The graph shows this empirically, with both the number of
transactions (and therefore the number of versions) and the number of pred-
icate operations causing an increase in execution time as they themselves
increase. This highlights the performance limitations of predicate checking
due to the Adya model’s data-driven definitions of predicate anomalies.

Figure 6.3 shows how Emme’s verification time increases as the number of
rows matching predicates and the write/read ratio of operations change.
This experiment considers only predicate read and update operations. One
hundred keys are inserted before any transactions are executed, to allow for
a fixed number of rows to be returned by each operation. Then one hundred
transactions are executed. Emme’s verification time scales linearly with the
number of rows matched. This is mostly due to update predicates, as they
will produce more versions when they match more rows. Similarly, the veri-
6.3 Cobra

We modified the Cobra verifier [51] to use version order edges, which are inferred from the recovered version order, to improve its execution time. By default, Cobra uses the order of transactions within a client to infer client order edges, which it uses to improve its execution time. However, this causes Cobra to verify strong session serializability instead of serializability. A history is serializable if it is strong session serializable, however the converse is not necessarily true. We modified Cobra to optionally turn off client order edges so that it could verify serializability.

We show that adding partial version order information to Cobra allows it to efficiently verify the serializability of workloads that it would have previously been unable to verify within any reasonable time limit. Furthermore, we show that even when verifying strong session serializability, partial version order information leads to an improvement in verification time.

6.3.1 Correctness

We used similar testing approaches to test our modifications of Cobra as we did to test Emme. There were two main differences, the first was to
restrict the test client to produce Cobra compatible histories and the second was to only use non-predicate operations, as Cobra does not support them. All tests successfully passed, which gave us confidence that our changes to Cobra had not introduced any errors.

6.3.2 Performance

![Effect of client and version order on Cobra verification time](image)

**Figure 6.4:** Demonstrates the effect on verification time of Cobra when version order edges and client order edges are included and excluded.

All performance experiments in this subsection were carried out using a p3.2xlarge Amazon EC2 instance. This has an NVIDIA Tesla V100 GPU, an 8-core CPU, and 64GB memory. We generated three types of workloads that we used in both of the performance experiments considered. All workloads contain ten thousand transactions generated by twenty clients. We set a maximum verification time limit of five minutes, after which we aborted the verification process. We enabled five phases of pruning to be carried out, which was necessary to fully take advantage of the version order information.

The first workload is a read-modify-write (RMW) workload. A read-modify-write is modeled as a read of a key followed by a write to the same key within the same transaction. This workload has 25% read operations, 25% write operations and 50% read-modify-write operations. Cobra performs a number of optimizations that benefit from read-modify-write operations, so this workload shows the effect of these optimizations.

The second workload, which we call the read/write workload, contains a 50/50 split of read and write operations. This workload is designed to evaluate the extent to which Cobra’s performance relies on read-modify-write operations.
operations, as there are no such operations in this workload.

The third workload, which we call the heavy write workload, has a mix of 10% read operations, 10% read-modify-write operations and 80% write operations. Aside from its client order edge optimization, most of Cobra’s optimizations rely on read-modify-write operations and read operations, so this workload aims to stress Cobra’s performance when there are only a limited number of those operations.

The first performance experiment explores the effect of disabling and enabling both client order edges (CO) and version order edges (VO). One of the main use cases for Cobra that we consider is partial version order recovery, which occurs when not all version order information can be recovered. In this experiment, we include only 90% of the version order information, which tries to emulate this scenario and limits the number of version order edges that can be added by our optimization.

The results are shown in Figure 6.4. For the RMW workload, Cobra performs well across the board. Disabling client order edges and version order edges only causes a 2x slowdown compared to when they are both enabled. However, for both the read/write and heavy write workloads, disabling both the client order edges and the version order edges causes the verification time to exceed the maximum time limit. This shows that without Cobra’s primary two types of optimizations, client order edges and read-modify-write optimizations, Cobra struggles. This is also demonstrated in the heavy write workload.

When enabling the version order edges, but keeping the client order edges disabled, Cobra performs well across all workloads. This demonstrates the power of the version order edges optimization. Without version order edges, but with client order edges, Cobra performs slightly better on the read/write workload than with just version order edges. However, in the heavy write workload it performs considerably worse than with just version order edges. In the read/write workload, adding the client order allows Cobra’s pruning optimization to remove many of the constraints it would have otherwise missed and since there aren’t too many write operations that aren’t ordered by the client order edges, it is able to perform well. However, in the heavy write workload this is not the case. There are too many write operations that are not ordered by the client order edges, which means Cobra is unable to remove enough constraints to finish quickly. In both these cases, Cobra with only version order edges performs well since it can infer the order of these inter-client writes and remove enough constraints to be efficient.

The second performance experiment, shown in Figure 6.5, demonstrates how the amount of version order information that Cobra has access to affects its performance. For the RMW workload, Cobra performed well irrespective of how much version order information was included, since it was
6.4 Summary

We evaluated both the correctness and performance of version order recovery. We found that it had little impact on the running time of database systems and its execution time could be hidden behind that of the test clients should it be required. We also illustrated the correctness and effectiveness of Emme, demonstrating that it can find both synthetic and real world isolation anomalies. Furthermore, we show that when per-operation visibility information was available Emme was able to verify PostgreSQL histories containing predicate operations, something that no other verifier is able to achieve outside of very limited cases. Finally, we evaluated the performance improvements possible when using recovered version order information in the Cobra verifier. We show that with version order information, Cobra is able to verify the serializability of histories that it previously timed out on, and that version order information could significantly speed up Cobra’s default strong session serializability verification too.
7.1 Formal Models of Isolation Levels

ANSI SQL-92 [4] formally defines four transaction isolation levels that compliant systems can offer: read uncommitted, read committed, repeatable read and serializable. These levels are defined in terms of the presence of three phenomena: dirty reads, non-repeatable reads and phantom reads. Any of the phenomena may occur at the read uncommitted level, dirty reads are disallowed at the read committed level, non-repeatable reads are disallowed at the repeatable level and phantom reads are disallowed at the serializable level. The serializable level is additionally defined using the standard serializability definition.

Building on previous work [17] Berenson et al. [5] show that the absence of the three phenomena defined in ASNI SQL does not guarantee serializable execution. Stricter versions of the ANSI SQL phenomena are defined and they also define new phenomena that should be avoided at certain isolation levels. Additionally, they define new isolation levels called cursor stability and snapshot isolation. Cursor stability is somewhere between read committed and repeatable read, and snapshot isolation does not permit any of the phenomena described in the ANSI SQL standard, but does permit some of the newly defined stricter phenomena. Many commercial systems implement snapshot isolation and have offered it as the serializable isolation level, leading to debates around the original serializable definition in the ANSI SQL standard.

Both the ANSI standard and Berenson et al. define their models in terms of a history of events that operate on single versions of database objects. This maps well to single version systems, however it is problematic for multiversion concurrency control (MVCC) systems, as these models do not account for multiple versions of an object being active at a given time.
Adya et al. [1, 2] define an abstract history that consists of an event order and version order. This gives rise to the notion of a directed serialization graph where the nodes are transactions and edges are conflicts between transactions. Isolation levels are defined in terms of graph phenomena that should be avoided, for example at the read committed level, no cycles consisting of certain types of edges may occur. This model works for MVCC systems as well as single version systems.

Crooks et al. [9] define a formalism based upon client observable behavior. This approach provides intuitive definitions of isolation levels compared to the models described so far and allows for reasoning about isolation levels in terms of client observable states as opposed to abstract histories that may or may not physically exist. However, this model does not aid in finding an efficient approach to checking serializability.

7.2 Testing and Verification of Isolation Levels

In general, serializability checking is NP-Complete [36]. Biswas and Enea [8] consider the case where a system provides session guarantees [53] and specify isolation levels using this assumption. They show under this model that checking serializability, snapshot isolation and prefix consistency are NP-Complete, whilst read committed, read atomic and causal consistency are polynomial time checkable. Importantly, they provide a serializability checker that runs in polynomial time in the number of sessions, when the number of sessions is fixed.

Hermitage [22] is an attempt to better understand the isolation levels on offer by database systems. Hermitage provides a fixed set of manually written transactions and an execution order. When executed in the given order, the transactions should maintain hand proven invariants. If an invariant is broken, it indicates an error in the implementation of the tested isolation level. This approach has drawbacks, as each system requires its own set of manually generated transactions and like any non-exhaustive testing approach only shows errors that manifest by executing this small set of transactions.

PostgreSQL [40] has a suite of isolation tests that are run by a tool called Isolationtester. Like Hermitage, Isolationtester has manually generated sets of transactions to test. However, unlike Hermitage it can automatically execute different orderings of the transactions, where each unique ordering is called a permutation. Testing all possible permutations is at best computationally expensive and at worst intractable, so by using knowledge of the system, a test author can define “interesting” permutations that they believe are likely to contain errors and have Isolationtester only execute those permutations. This approach relies on the test author’s knowledge to find “interesting”
permutations. Random testing has been able to find bugs that occur due to certain transaction patterns that weren’t considered in Isolationtester [20].

Elle [3] is a checker based upon the Adya model. It can check a wide variety of isolation levels and has been very successful in finding bugs in real-world systems, claiming that it has found at least one bug in every system that it has tested. For Elle to work most effectively, it requires two properties: (1) recoverability and (2) traceability. Recoverability allows Elle to identify write-read edges and can be accomplished by enforcing that values can be written at most once in the system, which is a common requirement in most other checkers. Traceability allows Elle to identify both read-write and write-write edges and is where Elle’s real power comes from. In order to identify these edges, Elle must recover the version order from the system. To do this, Elle requires that the system supports data types with an append operation, such as the TEXT type in SQL. Currently, Elle is unable to check predicate operations which means that it is unable to distinguish between the repeatable read and serializable isolation levels.

Histex [29] is a gray-box approach to testing isolation level implementations. Histex requires that a system be single versioned, uses a locking protocol approach to concurrency control and that the input history can be executed in a totally ordered fashion. Under these assumptions, it can be shown that the presence of certain patterns of input history guarantee errors in the implementation of a given isolation level. Histex provides a means of building such an input history using a template approach and provides a checker that examines the input history for the presence of the known error patterns.

To the best of our knowledge, Histex is the only system that is able to fully check the serializability of a relational database system (under the restrictions listed) as it is able to reason about predicate conditions, whereas most other approaches treat the database as a key-value store and do not support testing of predicates. It is interesting to note that the authors of Histex considered testing non-locking systems but found that it would be too much work to recover the version order, which is something this thesis aims to overcome.

Cobra [51] is an SMT solver based approach for verifying the serializability of key-value stores and aims to support continuous checking of a running system. Cobra is based on the Adya formalism and like many other checkers requires an execution to have the recoverability property in order for it to be able to discover write-read dependencies between transactions. Unlike Elle, Cobra does not attempt to recover the version order from the system and instead, it creates a polygraph with transactions as nodes, known write-read dependencies as edges and bipaths which act as constraints that essentially express possible read-write and write-write transaction dependencies.
Checking whether an execution is serializable then amounts to finding a compatible directed graph in the polygraph. A compatible directed graph can be thought of as the solution to the constraint satisfaction problem posed by the polygraph, where the directed graph contains the same nodes and edges as the polygraph but chooses one edge from each constraint. Since Cobra only supports basic key-value stores, it cannot check full serializability of modern SQL systems, as it has no support for predicate operations. Cobra also scales poorly when there are many write-write conflicts, as this significantly increases the number of constraints in the polygraph. Currently, Cobra does not support verification of any isolation level other than serializable.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we show that it is possible to derive a unique version order from any serializable concurrency control protocol, which must lead to a serializable Adya history if the protocol is implemented correctly. We provide examples of how to derive a version order for a range of common serializable protocols, including timestamp ordering protocols, optimistic concurrency protocols and certification-based protocols.

We introduce the notion of version order recovery, which is the process of recovering version order information directly from database systems. We introduce a number of techniques to implement version order recovery, such as log-based change data capture and heap-scanning. We implement heap-scanning for both PostgreSQL [40] and TiDB [19], and log-based change data capture for PostgreSQL. We show that both techniques have minimal performance impact on the database system. Importantly, we show that these approaches to version order recovery allow a greater degree of database functionality to be exercised than existing verifiers, which enables a greater degree of randomized testing.

We implement a verifier Emme, that can verify the serializability of execution histories that conform to the Adya Model, using recovered version order information. For item-dependencies, the type of dependencies supported by existing verifiers, Emme has equal or better performance and can scale to hundreds of thousands of transactions. Additionally, when per-operation visibility information can be recovered from a database, Emme is able to verify histories containing predicate operations, something no existing verifier can achieve outside of very special cases. We show that Emme can verify PostgreSQL histories containing 2500 transactions with a mix of predicate operations in under 5 minutes.
We show that the utility of version order information is not tied to Emme. We show that it is possible to convert Emme histories into the format required by the Elle verifier. This can be useful if verification of weaker isolation levels is required, which is not something currently supported by Emme. Additionally, we modify the Cobra [51] verifier to use recovered version order information and show that this can significantly improve the execution time, particularly in workloads with a high ratio of write operations.

8.2 Future Work

We have demonstrated that it is practical to implement version order recovery for both PostgreSQL and TiDB, which are systems with two very different designs. To emphasize the generality of version order recovery, we would like to expand the range of database systems that our version order recovery implementation supports, including non-relational systems. This shouldn’t require any fundamental changes to the techniques presented.

Version order recovery has been shown to work for systems supporting serializability and snapshot isolation, however, there are a number of weaker isolation levels that are commonly used. We would like to prove that version order recovery will work for those isolation levels. Additionally, we would like to investigate supporting stronger versions of serializability such as strict serializability and strong session serializability, both in Emme and for version order recovery. We believe it should be straightforward to support these additional constraints as the additional information required is minimal and the techniques for implementing a verifier for these isolation levels is well established [1, 2, 3].

Currently, Emme simply outputs an acceptance or rejection of a history. A good verifier should also be able to provide a counterexample, and ideally the simplest one it can find. Due to time constraints, Emme does not support this. However, the techniques for achieving this are well understood [3], so it should be straightforward to implement this for Emme.

The test client used throughout this thesis is very limited. To take advantage of the freedom offered by version order recovery, a more sophisticated test client is needed. We hope to integrate version order recovery and Emme into the SQLancer project [50], which provides very good query generation facilities for a range of different databases. It also contains an AST interpreter for many SQL dialects, which is important for achieving good performance when verifying predicate histories. A major benefit of integrating with SQLancer is that it amortizes many of the implementation costs associated with randomized database testing across a number of different randomized testing strategies. For example, SQLancer support the PQS [45], TLP [44] and NoREC [43] strategies. We believe that the integration of ver-
Version order recovery and Emme into SQLancer will broaden its appeal as a “one-stop shop” for finding logic bugs in database systems via randomized testing.

Version order recovery is an approach which requires knowing the concurrency control protocol under test. Whilst we demonstrate that the version order derivation is simple for most major protocols, if the protocol used is not known, then it is not possible to use version order recovery. It would be nice to have a more black-box approach to verifying the serializability of database systems that retains the benefits of version order recovery. We believe one approach is for database systems to explicitly provide their serialization order to external clients, as a proof of correctness, which would allow them to verify that the database system is providing serializability.
Bibliography


Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verifying Serializability Protocols With Version Order Recovery

Authoried by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):
Clark

First name(s):
Jack

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Signature(s)