Bringing Order to Practical Validation of Database Isolation Levels

JACK CLARK, Imperial College London, UK
MANUEL RIGGER, National University of Singapore, Singapore
JOHN WICKERSON, Imperial College London, UK
ALASTAIR F. DONALDSON, Imperial College London, UK

Transactions are a key feature of database systems and isolation levels specify the behavior of concurrently executing transactions. Ensuring the correctness of isolation levels is crucial. Recently, many isolation anomalies have been found in production database systems. Checkers can be used to validate that a particular execution history conforms to a desired isolation level. However, state-of-the-art checkers cannot handle predicate operations, which are both common in real-world workloads and essential for distinguishing between the *repeatable read* and *serializable* isolation levels.

In this work, we address this issue by proposing two techniques for efficient white-box checking. Our key idea is to use information that is easily provided by database systems to efficiently check the isolation level of a given execution history. We present the *version certificate recovery* technique and its associated checker *Emme*. Version certificate recovery is a method of recovering the *version order* and each operation's *version set* from the database system under test. To minimise execution time, we also propose the *expected serialization order* technique, along with its associated checker *Enne*, which obviates the need to define and recover a version certificate for many *serializable* concurrency control protocols. We have implemented version certificate recovery for three widely used database systems—PostgreSQL, CockroachDB, and TiDB. We demonstrate that Emme is 1.2–3.6× faster than Elle, a state-of-the-art checker. When paired with the expected serialization order technique, Enne obtains a further speedup of 34–430× when checking execution histories containing predicate operations. We show that our approaches can identify invalid execution histories that cannot be detected by Elle and also show that they can find realistic bugs purposely introduced by an engineer.

Finally, we create a new checker, *King Cobra*, by modifying Cobra, an existing black-box checker, in order to conduct an ablation study to evaluate how the performance of a black-box checker changes when provided with varying degrees of ordering information. By doing so, we highlight the fundamental role that ordering information plays in the execution time of an isolation level checker and explore the limits that this places on a truly black-box checker.

CCS Concepts: • **Software and its engineering** → **Correctness**; **Software verification and validation**; • **Information systems** → **Database transaction processing**; **Distributed database transactions**.

Additional Key Words and Phrases: database systems, isolation levels, transactions, software testing

1 Introduction

In this paper, which extends a conference paper published at EuroSys 2024 [14], we study the problem of validating the correctness of isolation level implementations in database systems.

A core feature of many database systems is grouping operations into *transactions* [19]. The extent to which operations within a transaction interact with operations from other concurrent transactions is defined by an *isolation level*. Database systems offer a range of isolation levels [2, 3, 5, 7, 12, 13, 15], with each level providing different guarantees to client applications. The guarantees that an isolation level offers are typically defined in terms of *anomalies* that any possible *execution history*—a record of the operations submitted to and the results received from the database system—must not

Authors' Contact Information: Jack Clark, Imperial College London, London, UK, jack.clark1@imperial.ac.uk; Manuel Rigger, National University of Singapore, Singapore, Singapore, rigger@nus.edu.sg; John Wickerson, Imperial College London, London, UK, j.wickerson@imperial.ac.uk; Alastair F. Donaldson, Imperial College London, London, UK, alastair.donaldson@imperial.ac.uk.

contain [3, 5, 7]. It is vital that database systems provide the isolation guarantees that they claim, because bugs leading to weaker guarantees can corrupt application state [28] and cause significant security vulnerabilities [54].

The role of *concurrency control protocols* is to prevent these anomalies from occurring. However, automated testing approaches have shown implementations of concurrency control protocols to be lacking in practice: a wide variety of bugs have been found in the implementations of isolation levels in various database systems [26]. This includes bugs in widely used and stable systems such as PostgreSQL [40], which had a bug in the implementation of its *serializable* isolation level [41]. Despite the success of these approaches, it would be desirable to obtain more confidence that isolation level guarantees are met. Formal verification approaches for mature database systems and their isolation levels are still out of reach [32]. In contrast, so-called *checkers* [4, 10, 51] enable validating that *a particular execution history* conforms to the isolation level that the database system claims to support. In other words, such checkers enable approaching the problem through a *translation validation* [35, 38] lens, allowing clients to verify that the operations performed by the database system conform to the desired isolation level. However, all of these checkers suffer from at least one of two major problems.

Problem 1: lack of support for predicate operations. Predicate operations, such as the SQL statement `SELECT * FROM t0 WHERE c0 > 7`, return all rows that satisfy a given predicate (`c0 > 7` in this case). They are widely used in most database systems, enabling complex features such as scans, joins, and aggregates. These features typically require additional implementation complexity and potentially more complex concurrency control mechanisms such as predicate locking [19]. Predicate operations can cause unique types of anomalies in execution histories, known as *predicate anomalies*, that simple key-value reads and writes cannot cause. For example, the execution history shown in Fig. 1 contains a *phantom read* anomaly [19]. These unique predicate anomalies have been observed in production database systems [34]. In fact, predicate anomalies are the sole means by which the industry-standard Adya model [3] distinguishes between *serializability* and the weaker *repeatable read* isolation level. Therefore the inability of current checkers to detect predicate anomalies is a severe limitation: they cannot distinguish between these isolation levels for execution histories containing predicate operations.

Problem 2: high execution time. For important isolation levels such as *serializability* and *snapshot isolation*, checking whether an execution history is valid is an NP-complete problem [9, 10, 37]. Supporting large and highly concurrent execution histories while minimising execution time is important because many randomized testing approaches rely on executing large numbers of transactions from many clients. Checkers in prior work either use workload-specific optimizations to increase the size of the execution histories that they support [51] or scale poorly as the concurrency in the execution history increases [10]. The exception, Elle [4], a checker for execution histories produced by the Jepsen testing framework [25], relies on an encoding scheme that works efficiently only if all writes in the execution history are atomic list-append operations; a requirement that some database systems (e.g., TiKV [52]) cannot fulfill. Furthermore, even in systems that *do* support atomic list-append operations, there is no guarantee that exclusive use of these operations can catch bugs in the implementations of other operations.

1.1 Our Contributions

Our first two contributions—the *version certificate recovery* technique and its associated checker *Emme*, and the *expected serialization order* technique and its associated checker *Enne*—both address these problems through a white-box checking approach. Although both techniques are white-box, we show that after a small amount of initial work from someone familiar with the database system, the information needed by both techniques can be exposed via a black-box API.

$$\begin{aligned} T_1 &: \ w_1(x,\ 1) \\ T_2 &: \ w_2(y,\ 2) \\ T_3 &: \ r_3(<5, \{x\}),\ r_3(<5, \{x,\ y\}) \end{aligned}$$

Fig. 1. This non-serializable execution history has three committed transactions. T_3 contains a phantom read. It issues two identical predicate operations that read all objects in the database whose value is less than five. The first operation returns the object x, but the second one returns both x and y.

Table 1. A comparison of the state of the art checker Elle with our checkers Emme and Enne.

	Elle [4]	**Emme**	**Enne**
Supports predicates	No	Yes	Yes
Execution time	3rd	2nd	1st
DB integration effort	Low	High	Medium
Black-box	Yes	No	No
Supported isolation levels	Most	*Serializability* and *snapshot isolation*	*Serializability*

This enables clients to use the checkers in a fully black-box manner. This is particularly valuable in a cloud setting where vendors may not want to reveal proprietary implementation details. Table 1 compares Emme and Enne against the state of the art checker Elle. As a final contribution, we conduct an ablation study to understand if the problem of high run-time overhead is inherent to a general-purpose black-box checker. To enable this, we create a new checker, *King Cobra*, that is a modification of an existing black-box checker, Cobra [51], that is able to optionally use additional ordering information to improve checking performance. This allows us to explore the fundamental impact that ordering information, in particular the version order, has on a checker's execution time.

Contribution 1: version certificate recovery and Emme. Our checker, Emme, when used alongside the version certificate recovery technique, has the following unique features:

(1) It minimises the execution time when checking large and highly concurrent execution histories by recovering a version certificate from the database system via a simple interface, without requiring support for atomic list-append operations (or any other particular operation).
(2) It is the first checker capable of supporting execution histories containing predicate operations, making it the only checker able to check the *serializability* of execution histories resulting from common database workloads.

Our core insight is that it is possible to use the invariant(s) that guarantee the correctness of a database system's concurrency control protocol to define and recover a so-called *version certificate*. A version certificate consists of an *expected version order* and, for every predicate operation in the execution history, an *expected version set*. Roughly, the expected version order is a per-object total order on writes to the database, while the expected version set of a predicate operation is the set of values over which it will evaluate its predicate. A checker based on the Adya model [3] of isolation levels can then use the version certificate to certify that the corresponding execution history conforms to a given isolation level or reject it. The checker *accepts* or *rejects* an execution history based on the following criteria:

- The checker will *accept* an execution history as valid if the expected version order and expected version sets form a valid version certificate to show that the execution history meets the requirements of the desired isolation level. Since the certificate was derived from the invariants that guarantee the correctness of the concurrency control protocol, we provide stronger guarantees than existing checkers: our checker will confirm an execution history as valid only if the execution history meets the desired isolation level *by design*. An execution history that meets the desired isolation coincidentally—despite violating these invariants—will be flagged as invalid, uncovering defects in the implementation of the protocol that would otherwise go unnoticed.

- The checker will *reject* an execution history as invalid if the version certificate cannot be used to show that the desired isolation level has been met. In this case, there *might* nevertheless exist *some* version certificate that could be used to validate the execution history. However, the fact that the version certificate did not lead to confirmation that the required isolation level had been met implies that either (a) the invariants of the concurrency control protocol were violated during execution or (b) the invariants were insufficiently strong to provide the required isolation level. Both of these cases indicate errors in the database system. Therefore, from the point of view of a database system developer, we argue that it is irrelevant that the execution history could be certified using some other version certificate and it is *valuable* that the checker can demonstrate these defects by rejecting the execution history.

A benefit of our core approach is that it is very general: it works for a wide range of isolation levels and database systems due to the central role that both the version order and version sets play in defining isolation levels in the widely used Adya model. We have implemented the version certificate recovery interface for three diverse and widely used systems: PostgreSQL [40], CockroachDB [50], and TiDB [23]. Despite the distinct concurrency control protocols used by these systems, we show that it takes no more than a few hundred lines of Python code to implement the interface for each system. Our checker, Emme, supports both *serializability* and *snapshot isolation*.

To demonstrate the effectiveness of version certificate recovery, we show that it can detect a faulty execution history caused by a known bug in an old version of PostgreSQL's *serializable* isolation level. Additionally, Cockroach Labs supported our work by introducing three bugs of their own choosing into a fork of CockroachDB that affect its *serializability* guarantees. We demonstrate, without any prior knowledge of the bugs, that version certificate recovery expectedly rejects invalid execution histories caused by all three bugs. We also show that our approach can detect predicate-only anomalies in execution histories produced by running PostgreSQL at the *read committed* isolation level, and that Elle—a state-of-the-art checker—is unable to detect these anomalies and will incorrectly validate these execution histories as *serializable*.

We compare the checking performance of Emme against Elle and find that for execution histories without predicate operations, Emme has up to 4× better performance. For execution histories containing predicate operations, we demonstrate that Emme can check execution histories of 2,500 transactions in under two minutes, making Emme the first checker able to support predicate operations for non-trivial execution histories.

Contribution 2: expected serialization order and Enne. The *expected serialization order* technique introduces a more specialized approach to checking that is even more effective for many *serializable* concurrency control protocols. This approach defines an expected serialization order—a total order on committed transactions such that they must be *serializable* in that order—that can be used to certify an execution history. This approach avoids the need to define and recover a version certificate and allows significantly faster checking, particularly for execution histories containing predicate operations.

We show that the more specialised expected serialization order technique, when paired with the checker Enne, significantly outperforms both Elle and Emme, with a speedup ranging from 34× to 430× compared to Emme on non-predicate execution histories, and a speedup ranging from 53× to 120× for predicate execution histories. Furthermore, we show that the expected serialization order technique scales significantly better than Emme for execution histories containing predicate operations allowing much larger execution histories to be supported.

Contribution 3: exploring the impact of ordering information on checking performance. Our final contribution explores the fundamental limits on the runtime performance of black-box vs white-box isolation level checkers by conducting an evaluation of the performance of *King Cobra*, a modified version of an existing black-box checker, Cobra [51]. King Cobra supports optionally using both session order and version order information as part of its checking process. We show that access to ordering information, particularly the version order, plays an essential role in enabling efficient checking of *serializability* and, by extension, other isolation levels where checking is NP-complete. We believe this highlights an interesting tension in the specification of isolation levels—the desire for declarative, high-level definitions that are easy to understand versus the desire to enable efficient checking.

Contribution over our prior work. This work extends a paper published at the European Conference on Computer Systems [14]. Our main additional contributions are: a checker, *King Cobra*, that is an extension of the Cobra checker [51], that is able to use version order information to improve its checking performance; and an ablation study that quantifies the impact that varying the amount of both session order and version order information has on the performance of King Cobra in order to understand the limitations of a general-purpose black-box checker. The new material appears in Section 7 and Section 8.4.

2 Example-Driven Overview

Our first contribution consists of version certificate recovery—the technique for recovering sufficient information from the database system to enable fast checking—and our checker, Emme. We imagine a scenario where a proprietary cloud database company has tasked an engineer with a) ensuring that their system correctly implements its isolation level guarantees and b) providing clients with a means of validating these guarantees.

To this end, the engineer can use our approach based on three steps: (1) defining the expected version order and expected version sets that will form the version certificate, (2) implementing mechanisms to retrieve any information from the database system that is necessary to recover the expected version order and version sets, and (3) implementing the black-box API that outputs the version certificate. Finally, for validation, both the engineer and client can retrieve the version certificate from the black-box API and input it along with the recorded execution history into our checker to verify that the history complies with the required isolation level.

Let us assume the database system uses a *multiversion concurrency control timestamp ordering protocol* (MVTO) to guarantee *serializability* [44]. In such a protocol, each transaction is assigned a unique timestamp which is associated with all of its operations. Additionally, each version of an object is assigned the timestamp of the transaction that created it. The correctness of the protocol depends on the invariant that ordering the operations of committed transactions in ascending timestamp order will produce an equivalent serial history that is *serializable*. It follows that choosing a version order consistent with timestamp order will guarantee a *serializable* history in any correct execution. A further implication of the timestamp order invariant is that it must always be possible for a transaction to execute its operations using the version of each object with the greatest timestamp that is less than or equal to its own. These facts should already be clear to a developer implementing the protocol, however, if a different developer is deriving the version certificate, then they can discover these facts by consulting a standard textbook describing MVTO [9].

With the knowledge above, it is clear that the expected version order should be defined as the total order resulting from sorting versions in ascending timestamp order (where versions written by the same transaction appear in transaction

$$\begin{aligned} T_1 &: w_1(x,\ 1) \\ T_2 &: w_2(y,\ 2) \\ T_3 &: r_3(>0, \{x,\ y\}) \end{aligned}$$

Fig. 2. A history that is *serializable* in ascending timestamp order. Transaction identifiers act as timestamps.

$$\begin{aligned} T_1 &: w_1(x,\ 1) \\ T_2 &: w_2(y,\ 2) \\ T_3 &: r_3(>0, \{x\}) \end{aligned}$$

Fig. 3. A history for which ordering transactions by their timestamps ($T_1 \rightarrow T_2 \rightarrow T_3$) does not produce a *serializable* order. There is another order ($T_1 \rightarrow T_3 \rightarrow T_2$) that is *serializable*, however, it contradicts the timestamp order. Transaction identifiers act as timestamps.

order). Each operation's expected version set can be defined to contain the version of each object with the greatest timestamp that is less than the predicate operation's timestamp, or if the operation's transaction has modified an object, the latest version of the object modified by the transaction.

To recover the expected version order and version sets for any particular execution history, the engineer needs to recover the timestamps associated with each transaction (and therefore each operation). In our experience, most timestamp-ordering database systems include this in the transaction metadata. Therefore, the engineer can use this metadata to automatically recover the timestamp associated with each transaction, generate the version certificate, and make it available via a black-box API. This is the last step that requires any knowledge of the database system internals.

Finally, the engineer or client can recover the version certificate via the black-box API and provide it as input along with the history to Emme, which will use the certificate to check that the history is *serializable*. Fig. 2 contains a history that Emme will verify as *serializable* using the trivial version order that arises from a single write to each object in a history, and the expected version set $\{x,\ y\}$ for r_3. In the examples within this section, the transaction identifiers are also their timestamps, for example, T_1 has timestamp 1. Since both the version order and version set have been derived from the timestamp order, we can be sure that the history is not only *serializable*, but respects the key timestamp ordering invariant of the MVTO protocol.

Fig. 3 demonstrates a history that does not form a *serializable* serial order when arranged in timestamp order. The expected version set of r_3 is $\{x,\ y\}$, however, r_3 only reads x despite y also matching the predicate condition *(> 0)*. Clearly, this should never happen in a database system using the MVTO protocol. However, black-box checkers will verify this history as *serializable* using the serial order $T_1 \rightarrow T_3 \rightarrow T_2$, despite the fact that this violates the timestamp order invariant, as they have no knowledge of the underlying protocol used. In contrast, Emme will reject this history as it recognizes that the invariant used to derive the version certificate has been broken, indicating either an implementation error in the database system, or the use of an incorrect invariant. This is a valuable feature of our checker, as it can detect defects in concurrency control protocols that would otherwise be missed by black-box checkers.

Our second contribution, the expected serialization order technique, can also be used for the MVTO protocol. An expected serialization order is a total ordering of transactions such that they must be *serializable* in that order. For the MVTO protocol, this is simply ascending timestamp order. There are many benefits to using the expected serialization order approach compared to generating a version certificate. Firstly, an engineer only needs to derive an expected serialization order, rather than both an expected version order and the expected version sets, which is typically much easier. Secondly, there is no need to implement version certificate recovery for the database system.

Finally, as demonstrated in Section 8, the checking performance and scalability of Emme are vastly improved when using an expected serialization order, particularly for histories containing many predicate operations.

We highlight, both through this MVTO example and the examples in Section 5, that the invariants required by our approach are high level and can be derived from the proof of correctness or formal specification of a protocol. This is particularly useful for distributed database systems, where it is becoming more common to formally specify the invariants of a concurrency control protocol in a language such as TLA$^+$ [29]. Furthermore, the invariants do not require analysis of the database system code and are robust to changes in implementation details that do not affect the correctness proof of the protocol.

3 Background

We now provide relevant background on: the Adya model [3], on which our novel checker Emme is based (Section 3.1); the existing isolation level checker Elle [4] (Section 3.2); and the existing isolation level checker Cobra [51] (Section 3.3).

3.1 Adya Model

Our checker uses the isolation level model introduced by Adya et al. [3], so this section provides an overview of the model to aid in understanding how our checker works, and also demonstrates the theoretical basis for checking histories containing predicate operations.

An Adya history H comprises a set of operations performed by transactions and a version order ($\ll$). The database consists of a set of abstract objects and operations act on a particular version of each object. Write operations introduce a new version of an object and read operations read a particular version of an object. The version order ($\ll$) is defined as a per-object total order over committed versions. $x_i \ll x_j$ means that x_i appears before x_j in the version order.

The Adya model also defines predicate operations, which operate on versions matching a particular predicate condition P. Since many versions of a particular object may exist, the system conceptually chooses a single version of each object over which to evaluate the predicate. This is called the version set, $Vset(P)$, of the predicate operation.

Isolation levels are defined within the Adya model by the different types of *anomalies* that are allowed to occur. The Adya model defines two non-cyclic anomalies, *aborted reads* and *intermediate reads* that are disallowed in every isolation level other than the *read uncommitted* (PL-1) isolation level. In addition to the non-cyclic anomalies, there are also anomalies that are defined by the cycles that can occur in a *Direct Serialization Graph*.

Given an Adya history H, it is possible to build a *Direct Serialization Graph*, $\mathsf{DSG}(H)$. The nodes of the $\mathsf{DSG}(H)$ consist of the committed transactions in H and edges between transactions occur due to dependencies that arise from operations within the history. The Adya formalism introduces two types of dependencies—*item dependencies* and *predicate dependencies*. We call a $\mathsf{DSG}(H)$ with only item dependencies an *item-only* $\mathsf{DSG}(H)$. Three types of item dependencies exist (write-read, write-write, and read-write) that occur whenever a version of a single object is read or written. Let us call a $\mathsf{DSG}(H)$ that includes only the item dependencies in H the $\mathsf{item\text{-}DSG}(H)$. Fig. 5 shows the $\mathsf{item\text{-}DSG}(H)$ that results from the Adya history in Fig. 4. The history contains three transactions, T_1, T_2 and T_3. Transactions T_1 and T_2 each write a new version of x. The third transaction, T_3, issues two operations—a predicate read operation and a read of x. The predicate read operation—$r_3(< 5 : x_2)\ \{\}$—tries to read any versions less than 5. It has a version set consisting of only x_2 and an empty result set (since x_2 is greater than 5).

An Adya history H is *serializable* if the resulting $\mathsf{DSG}(H)$ is acyclic when both item dependency edges and predicate dependency edges are considered. Notice that the history is not *serializable*, yet the $\mathsf{item\text{-}DSG}(H)$ is acyclic. Since the

$T_1 : \ w_1(x_1 = 4)$
$T_2 : \ w_2(x_2 = 6)$
$T_3 : \ r_3(< 5 : x_2) \ \{\}, \ r_3(x_1 = 4)$
$\textit{Version Order} : x_1 \ll x_2$

Fig. 4. A non-serializable Adya history consisting of four operations. T_3 contains two operations—a predicate read that reads anything less than five, has x_2 in its version set, and an empty result set, followed by a normal read of x_1.

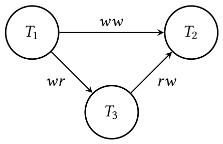


Fig. 5. An item-DSG(H) built from the history shown in Fig. 4.

item-DSG(H) includes only item dependency edges, it is insufficient for checking *serializability*. Existing checkers that use the Adya model only build an item-DSG(H) and thus cannot check *serializability* as defined in the Adya model.

Predicate dependencies. A central concept for defining predicate dependencies in the Adya model is the notion of *changing the matches* of a predicate operation. A version x_i changes the matches of a predicate operation $r_j(P\colon \mathit{Vset}(P))$ if x_i matches the predicate condition and the version x_h immediately preceding x_i in the version order does not match the condition or *vice versa*. Two types of predicate dependencies exist in the Adya model:

(1) **A predicate read dependency (pred wr)**, which occurs from T_i to T_j when T_j issues a predicate read $r_j(P\colon \mathit{Vset}(P))$, $x_k \in \mathit{Vset}(P)$, $i = k$ or $x_i \ll x_k$, and x_i changes the matches of $r_j(P\colon \mathit{Vset}(P))$.
(2) **A predicate anti-dependency (pred rw)**, which occurs from T_i to T_j when T_j overwrites a predicate read operation $r_i(P\colon \mathit{Vset}(P))$. T_j overwrites an operation $r_i(P\colon \mathit{Vset}(P))$ if T_j installs x_j such that x_k belongs to $\mathit{Vset}(P)$, $x_k \ll x_j$ and x_j changes the matches of $r_i(P\colon \mathit{Vset}(P))$.

Recall the Adya history in Fig. 4. The corresponding DSG(H) is shown in Fig. 6 once predicate dependencies have been added. The predicate read dependency from T_2 to T_3 occurs because $w_2(x_2)$ is observed in the version set of the predicate read in T_3 and it would not match the predicate condition, whereas the previous version in the version order x_1 would have matched, therefore it changes the matches. This is the dependency that makes the graph cyclic and therefore violates *serializability*.

Imagine instead of x_2, x_1 was in the version set of the predicate operation in the history in Fig. 4. This results in the DSG shown in Fig. 7. This DSG is acyclic, despite the history not being *serializable*. This is because the Adya model assumes that if a version in the version set matches the predicate, it must be in the result set. However, this is not something a checker can assume, since the system may have an error. This requires us to add an additional anomaly to the Adya model, which we call a *result set mismatch*, which occurs when a version in the version set should have been included in the result set but was not or *vice versa*.

As well as predicate reads, the Adya model allows updates based on a predicate condition (*predicate updates*). Predicate updates are modelled as predicate reads followed by a sequence of item write operations that create a new version v_{new} for each version v_{old} that matched the predicate read. This captures common operations such as `UPDATE table SET c0 = 20, c1 = 20 WHERE c1 >= 0 AND c1 <= 10`.

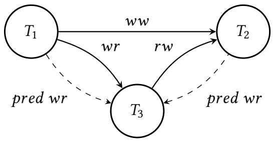


Fig. 6. The DSG that is built when including predicate dependencies from the predicate operation $r_3(< 5 : x_2)\ \{\}$.

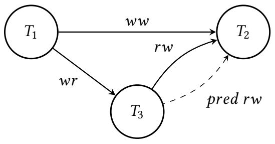


Fig. 7. The DSG that is built when including predicate dependencies from the predicate operation $r_3(< 5 : x_1)\ \{\}$.

3.2 The Elle Checker

Elle [4] is a tool designed to check that an Adya history is compliant with a specified isolation level according to the definitions introduced by the Adya model. It can check a wide variety of isolation levels and, when used in conjunction with the Jepsen testing framework [25], has been very successful in detecting bugs in real world systems.

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 externally observable, onto 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, where keys are unique across a history. 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 a history to satisfy two properties:

- **Recoverability:** this states that for each read operation it must be possible to uniquely identify the write operation that produced the read value. 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 an edge between those transactions and the overwriting transaction. Additionally, it allows Elle to identify write-write edges, which occur between the transaction that wrote the previous value and the overwriting transaction.

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 fast checking of a history. To support traceability, Elle requires that a system supports some method of encoding the version order into the values stored within the system so that it can recover the full version order during read operations. An example of how Elle achieves this for database systems that support SQL is to use a TEXT data type for storing values and changing all write operations to append the new value to the existing value rather than overwriting it. This allows Elle to recover the full version order on a read, for example, a read operation might return "1, 2, 3, 4," which would imply a version order of $[1, 2, 3, 4]$.

If a history H has both the traceability and recoverability properties, then checking a given isolation level amounts to constructing a DSG(H), which can be checked in linear time. Elle does not support predicate operations, so it is unable to add predicate edges and therefore is limited to constructing an *item-only* DSG(H). This is an important point, as the *repeatable read* and the *serializable* 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.

3.3 The Cobra Checker

Cobra is a state-of-the-art checker that supports checking whether key-value execution histories are *serializable*.[1] Like Elle, Cobra is black-box, meaning that it deals with histories generated entirely in terms of client operations. Cobra only supports read and write operations, therefore, it cannot check the *serializability* of histories containing predicates. Like Emme, Cobra requires that execution histories contain unique writes. Unlike Emme, Cobra introduces an additional constraint that every transaction reads and writes a key at most once. Cobra can be used to verify a one-off execution history fragment, but it also supports the verification of a continuously running system by verifying execution history fragments in rounds. In this paper, we focus on the one-off verification process.

To verify the *serializability* of an execution history H, Cobra tries to find an acyclic DSG(H). However, it does this quite differently to Emme. Cobra constructs a directed graph, called the *known graph*. In the *known graph*, transactions are nodes and edges represent ordering dependencies between transactions. The *known graph* contains only the write-read edges that can be inferred from the execution history.

Cobra does not immediately add write-write and read-write edges to the *known graph* as the execution history does not contain sufficient ordering information. Instead, Cobra introduces a set of constraints C, which 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 [37]. Essentially, a polygraph P represents a family of directed graphs which captures all of the possible unique graphs that can be constructed by choosing an edge from each constraint in C and adding it to the *known graph*.

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 with an execution 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. 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 execution time *in practice*, Cobra introduces a number of optimizations that aim to reduce the number of constraints in the polygraph, thereby reducing the cost of checking. One such optimization is called *pruning*. Pruning is a way to remove redundant constraints from the polygraph. Given a constraint $c = (e_1, e_2)$, Cobra recognizes that if choosing the edge e_1 causes 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 also causes a cycle, then Cobra can immediately reject the history as not serializable. Notably, Cobra relies on the use of a GPU to speed up the reachability computation that it uses during pruning. Once Cobra has performed all of its optimizations, it encodes the polygraph into an SMT verification problem instance.

A significant drawback of Cobra is that it still needs to perform an NP-complete computation in order to verify a history. Cobra relies on its optimizations to reduce real-world histories to verification instances that are small enough

[1]More accurately, Cobra supports the checking of *strict serializability* [51, Section 3.5] and *strong session serializablity* [51, Section 6.1], both of which are slightly stronger isolation levels compared with serializability.

Algorithm 1: The main checking function.

```
Def check(history, vo, isolation_spec):
    txns = get_committed_txns(history)
    deps = set()
    for txn ∈ txns do
        for op ∈ txn.ops do
            item_deps = get_item_deps(op, vo)
            pred_deps = get_pred_deps(op, vo)
            deps = deps ∪ item_deps ∪ pred_deps
    dsg = build_dsg(deps)
    return ¬dsg.has_cycle(isolation_spec)
```

to be verified 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 execution history.

An important point is that Cobra checks *strong session serializability* [16] by default. If an execution history is strong session serializable, then it is serializable. However, the converse is not true. This enables Cobra to use session order edges to further reduce the number of constraints in the graph and improve performance.

4 Emme

We now describe our checker, Emme, which, given an Adya history and an isolation level, is responsible for determining if the history satisfies a given isolation level. Our checker supports multiple isolation levels, including *serializability* and *snapshot isolation*. It is easy to modify our checker to support additional isolation levels, as long as they can be expressed using the Adya model.

Algorithm 1 gives a high-level overview of the checking process, which involves two steps: (1) computing dependencies and (2) checking the resulting DSG. The process of computing dependencies is separated into computing item dependencies which is handled by `get_item_deps` (see Section 4.1) and predicate dependencies which is handled by `get_pred_deps` (see Section 4.2). These functions also detect all non-cyclic anomalies introduced by the Adya model [3], such as intermediate and aborted reads. In practice, database systems can exhibit anomalies that violate the assumptions of the Adya model, so we also check for the non-cyclic anomalies introduced by Elle [4] and for the result set mismatch anomaly that we introduced in Section 3.1.

Once Emme has computed all dependencies, it builds a DSG and checks it for cycles. Emme's cycle-detection algorithm takes the isolation level specification as input, since isolation levels differ in the types of cycles they allow. Additional dependencies exist that need to be computed for some isolation levels, for example, start-dependencies for *snapshot isolation*. Emme does support *snapshot isolation*, however, we omit these details from our general checking algorithm for simplicity.

4.1 Item Dependencies

The algorithm for computing the item dependencies of an operation is shown in Algorithm 2. In addition to computing the item dependencies, the algorithm must also detect all non-cyclic anomalies, such as aborted reads, that can occur from read and write operations. Our algorithm handles reads and writes separately.

Algorithm 2: Compute the item dependencies from an operation.

```
Def get_item_deps(op, vo):
    if op.is_read then
        return get_read_deps(op, vo)
    else
        return get_write_deps(op, vo)

Def get_read_deps(op, vo):
    deps = set()
    if is_anomalous_read(result) then
        raise read_anomaly()
    add_read_dep(deps, op.result.tid, op.tid)
    next_write = vo.get_subsequent_write(result)
    if next_write ≠ null then
        add_anti_dep(deps, op.tid, next_write.tid)
    return deps

Def get_write_deps(op, vo):
    deps = set()
    if is_duplicate_write(op.result) then
        raise duplicate_write_anomaly()
    prev_write = vo.get_previous_write(op.result)
    add_write_dep(deps, prev_write.tid, op.tid)
    return deps
```

Read operations are handled by `get_read_deps`. The `is_anomalous_read` function tries to detect four non-cyclic anomalies: (1) aborted reads and (2) intermediate reads which are defined in the Adya model, and (3) garbage reads and (4) internally inconsistent reads which are defined by Elle [4]. If there are no non-cyclic anomalies, then read dependencies and anti-dependencies are computed. A read dependency is created with the `add_read_dep` function, which creates a dependency from the transaction that wrote the version to the transaction that read it. The `add_anti_dependency` function creates an anti-dependency from the transaction that issued the read to the transaction that writes the next version in the version order after the version read.

Write operations are handled by `get_write_deps`. There is only one non-cyclic anomaly that can occur due to a write operation, which is a duplicate write. A duplicate write occurs when two separate write operations each create a new version of an object and both versions have the same value. Since we restrict versions to be uniquely identifiable through a combination of their object identifier and their value, a duplicate indicates something has gone wrong internally. Finally, a write dependency is created between the transaction that wrote the previous version in the version order and the transaction that issued the write operation with the current version.

Computing all item dependencies has time complexity proportional to the number of read and write operations in the history since it is necessary to iterate over all read and write operations.

4.2 Predicate Dependencies

Algorithm 3 outlines how predicate dependencies are computed. It starts by checking for a result set mismatch anomaly. As defined in Section 3.1, a result set mismatch anomaly occurs when the actual result set of an operation does not equal the result set returned when evaluating the predicate on the version set. Therefore, computing the expected results requires a predicate evaluation oracle, which we call the *matches* oracle. Given a test version and a predicate, the matches oracle returns true if the test version matches the predicate. We have designed two matches oracles with different tradeoffs.

The *database matches oracle* is the most basic oracle, as it uses the database system itself to evaluate the predicate. First, the test version replaces the version of the same object that was in the version set. Then, this modified version set is loaded into the database. Finally, the predicate is executed by the database system and the oracle returns true if the test version is in the result set. The advantage of this approach is its simplicity and ease of implementation. However, it does require inserting a potentially large version set into the database for each call to the oracle, which can be expensive. The database matches oracle also assumes that the non-transactional query evaluation part of the database is correct, but it does not assume that the concurrency control protocol implementation is correct. In practice, these parts of the database system implementation have little overlap. We believe this is reasonable as there are existing techniques [1, 6, 45–49] to validate the correctness of the query evaluator/optimizer and the focus of this work is on finding bugs in the implementation of concurrency control protocols.

The second matches oracle, which we call the *interpreter oracle*, uses a SQL interpreter to evaluate the predicate. Using a SQL interpreter instead of the database matches oracle makes the evaluation of each predicate significantly more efficient because the interpreter can evaluate the predicate without communicating with the database system. A major downside of this approach is the cost to implement the interpreter, which scales with the number of database features it needs to support, and also the ongoing maintenance required to keep the implementation in sync with any changes to the specification of the database system. This approach is inspired by *pivoted query synthesis* [47], which uses a database system specific SQL interpreter to execute a predicate condition to determine if a given row would match the predicate. This has been successfully applied to find hundreds of query evaluation bugs and implementations exist for many systems. We believe the use of a SQL interpreter for other database testing approaches demonstrates that the cost of implementing the SQL interpreter is worth it and can benefit various different testing approaches. We implemented a basic query evaluator that consists of roughly 100 lines of Python code, which we reused across each database system.

Once the matches oracle has been used to confirm that the result set matches the expected result set, it is necessary to iterate over all versions ever written in order to determine whether that version could change the matches of the predicate operation. If a version does not, then no dependencies exist for that version. If a version does change the matches, then the type of dependency created depends on where that version lies in the version order compared to the version from the same object in the version set. If the version comes after the version in the version set, then a predicate anti-dependency is added and if it is equal to or comes before the version set version in the version order, then a predicate read dependency is created.

Algorithm 3 computes the predicate dependencies for a single predicate operation. To compute all predicate dependencies, the algorithm is repeated for every predicate operation. This makes the predicate dependency computation expensive compared to item dependency computation. The cost is intrinsic to the Adya model's [3] data-driven definitions. The item dependency computation runs in time $O(W + R)$ where W and R are the number of read and write operations in the history. The predicate computation runs in $O(P \cdot W)$ where P is the number of predicate operations in the history. This is exacerbated by predicate update operations causing additional individual write operations.

Algorithm 3: Compute the predicate dependencies from an operation.

```
Def get_pred_deps(op, vo):
    $deps = set()$
    $expected\_results = compute\_results(op.query,\ op.version\_set)$
    if $expected\_results \neq op.results$ then
        raise $result\_set\_mismatch\_anomaly()$
    $objects = vo.get\_all\_objects()$
    for $object \in objects$ do
        $vset\_version = op.version\_set.get(object)$
        for $version \in object.versions$ do
            $prev = vo.previous\_version(version)$
            if $\neg changes\_matches(op.query, version, prev)$ then
                continue
            if $vo.succeeds(version,\ vset\_version)$ then
                $add\_pred\_anti\_dep(deps,\ op.tid,\ version.tid)$
            else
                $add\_pred\_read\_dep(deps,\ version.tid,\ op.tid)$
    return $deps$

Def compute_results(query, version_set):
    $results = set()$
    for $version \in version\_set$ do
        if $matches(query, version)$ then
            $results.add(version)$
    return $results$

Def changes_matches(query, version, prev_version):
    $prev\_matches = matches(query,\ prev\_version)$
    $curr\_matches = matches(query,\ version)$
    return $prev\_matches \neq curr\_matches$
```

4.3 Comparison With Existing Checkers

We now summarize the main advantages, differences and limitations of our new checking approach, Emme, compared with the existing state-of-the-art checkers Elle [4] and Cobra [51]. We defer an experimental comparison with Elle to Section 8.2.

Support for predicate operations. The main advantage of Emme over these works is that Emme supports predicate operations. As discussed in the introduction, predicate operations are widely used in most database systems, enabling complex features such as scans, joins, and aggregates. Predicate operations can cause unique types of anomalies in execution histories, known as *predicate anomalies*, that simple key-value reads and writes cannot cause, and which have been observed in production database systems [34]. We emphasize again that predicate anomalies are the sole means by which the industry-standard Adya model [3] distinguishes between *serializability* and the weaker *repeatable read* isolation level.

Black box vs. white box approaches. An important difference between the approach we take with Emme vs. the existing techniques is that Elle and Cobra are entirely black-box methods, while Emme can be seen as a white-box method. This is because, as per the Adya model, Emme relies on information about the ordering of writes, and the way this information is made available varies between database systems. Thus a one-off, per-database engineering effort is required to integrate a database system with Emme. In practice database system developers take testing seriously—e.g. the amount of code associated with testing the widely-used SQLite database is reported to be 590 times more than the code associated with the database implementation[2]—therefore it is reasonable to assume that this modest effort would be acceptable in practice.

Furthermore, once the version order and version sets have been recovered from the database, they can be used without any knowledge of how they were produced or of the underlying database system. This makes it possible to abstract away the version order and version set derivation and recovery process behind an API, and subsequently work with them in a purely black-box fashion. We discuss this further in Section 5.

Working with the version order and version sets in a black-box fashion has two key advantages. Firstly, testers can be oblivious to the internals of the systems that they are writing tests for. This is particularly useful when testers are separated from the team that implemented the concurrency control protocol. Furthermore, regardless of who writes the tests, the test code itself need not be tied to the implementation details of either the version certificate recovery process or the concurrency control protocol. Secondly, this enables users of the database to gain assurance about the correctness of the system without needing access to its source code or knowledge of the concurrency control protocol used. This is particularly important when interacting with proprietary database systems and is a key motivation for the Cobra checker [51]. It is sufficient for the database system, perhaps through a verification interface, to output the version certificate, which the client can then input into our checker to validate that the database is upholding its contract. For isolation levels where checking is NP-complete, clients also obtain the guarantee that faking a version certificate, that is, outputting a version certificate that passes checking but in reality using one that does not, would require solving an NP-complete problem efficiently. We view this as a certificate of correctness from the database system.

5 Version Certificate Recovery

In order to check that an execution history conforms to some isolation level specification, it is necessary to first define the version certificate, which consists of both the expected version order and expected version sets, and then to recover it from the database system under test. To demonstrate that version certificate recovery is feasible in practice, we have implemented version certificate recovery for three real-world database systems: CockroachDB [50], TiDB [23], and PostgreSQL [40]. We chose these three systems because they are a mix of distributed and single-node systems and they each use a different concurrency control protocol. We did not modify any of the systems to enable our approach.

5.1 CockroachDB

CockroachDB is a distributed database system offering the *serializable* isolation level. CockroachDB assigns each transaction a unique hybrid logical clock timestamp [18] and guarantees that arranging transactions and their operations in ascending timestamp order will provide a valid serialization order. This is identical to the guarantees of the timestamp ordering protocol discussed in Section 2, so the expected version order can be defined as ascending timestamp order. The expected version set of an operation consists of the version of each object with the greatest timestamp less than or equal to the timestamp assigned to the reading transaction, or if the operation's transaction has modified an object, the latest version of the object modified by the transaction.

[2]https://www.sqlite.org/testing.html

With the version certificate defined, it is necessary to recover it from the database. We leverage CockroachDB's `CHANGEFEED` mechanism to recover the versions written to the database along with their timestamps. The `CHANGEFEED` mechanism is implemented by aggregating information from each node's *write-ahead log* and providing it in any easy-to-consume format. Recovering information from a database system's *write-ahead log* is a common way of implementing some, if not all of version certificate recovery. Additionally, with the increasing popularity of change data capture as a general technique for recovering information from database systems, more and more systems are implementing this functionality, which makes implementing version certificate recovery significantly easier.

CockroachDB's `CHANGEFEED` mechanism provides only the final version of each object that is written by a transaction. If an object is updated twice within the same transaction, we will not recover the version resulting from the first update. This has no impact on checking item-only histories, as every item-only write operation can record which version it is writing, so it is possible to associate a timestamp with these writes regardless of whether or not we can recover them. However, for predicate updates, there is no way of knowing which versions were created due to the update as it depends on the versions that match the predicate condition.

Missing versions due to predicate updates can cause false positives in the general case due to the way the version set of an operation is computed. Therefore, to ensure that we can always recover all versions and their timestamps, we restrict CockroachDB transactions to only contain a predicate update if it contains no other write operations. This ensures that every version written by the predicate update must be the final version of an object written within the transaction and therefore will be reported by CockroachDB's `CHANGEFEED` mechanism. There are alternative strategies for dealing with this, such as scanning the entire contents of the database after each predicate update. Of course, it may still be possible to modify CockroachDB to report intermediate writes, however, we wanted to demonstrate that it is possible to get a *useful* test setup with minimal effort.

The version certificate recovery implementation consists of roughly 200 lines of Python code, demonstrating the simplicity of the implementation.

5.2 TiDB

TiDB is a distributed hybrid transactional/analytical processing database system that offers *snapshot isolation* as its primary isolation level. For systems supporting *snapshot isolation*, such as TiDB, the version order is always chosen as the commit order of transactions [21]. The intuition for this is that updates in any *snapshot isolation* scheme are handled in one of two ways: (1) the first committer wins or (2) the first updater wins. Either scheme ensures a total order on versions that matches the commit order. This means that we can define the expected version order to be equal to the commit order of transactions.

To define the expected version sets, it is important to understand how *snapshot isolation* performs reads. In a system supporting *snapshot isolation*, all reads are performed at a start time $start(T_i)$. We can use $start(T_i)$ to define the expected version set of every operation in T_i. The expected version set of an operation performed at $start(T_i)$ is the set containing the version of each object whose timestamp is greatest and also less than or equal to $start(T_i)$, or the latest write by an operation in T_i if such an operation exists. Care has to be taken to use the correct timestamp for $start(T_i)$ as TiDB supports two transaction models that use different timestamps to perform reads: (1) optimistic, where transactions are rolled back only when there is a conflict, which defines a timestamp `start_ts` to act as $start(T_i)$ for reads, and (2) pessimistic, where transactions take locks during execution and start committing only after ensuring a transaction can be successfully executed, which uses the `for_update_ts` timestamp for reads. We set $start(T_i)$ to be either `start_ts` or `for_update_ts` depending on which model is used.

With both the expected version order and expected version sets defined, it is necessary to recover all versions written to the system, along with the `start_ts` for optimistic transactions and `for_update_ts` for pessimistic transactions. TiDB makes it easy to recover this information as it is exposed by an HTTP debugging API that will list each version and its timestamp. It does this by scanning the underlying *multiversion concurrency control* (MVCC) data store, a technique that we call heap-scanning. Similar to CockroachDB, TiDB's HTTP API reports only the final version of each object that is modified within a transaction, therefore, we also restrict a transaction to contain a predicate update only if there are no other writes within it. Alternatively, it is possible to recover version information using the TiDB change data capture tool, which operates on the *write-ahead log* (WAL) of each node in TiDB, similar to CockroachDB's `CHANGEFEED` mechanism. Our TiDB heap-scanning implementation of version certificate recovery consists of roughly 150 lines of Python, which demonstrates again the simplicity of the technique.

5.3 PostgreSQL

PostgreSQL is an MVCC RDBMS that supports multiple isolation levels. We focus on its *serializable* isolation level, which it provides using *serializable snapshot isolation* (SSI) [11, 39]. SSI is based on *snapshot isolation*, and uses the same underlying mechanisms as *snapshot isolation*, however, it adds an additional level of checks to prevent a transaction from committing if certain dangerous dependency structures [11] are present between transactions. These dependency structures are known to lead to *serializability* violations, so by preventing these, in addition to the properties provided by *snapshot isolation*, all histories produced are *serializable*.

Like *snapshot isolation*, picking the commit order of versions as the version order guarantees a *serializable* DSG in a correct execution [20]. This is because the checks introduced by SSI may prevent certain transactions from committing, but they do not alter the ordering of transactions once they have committed. Therefore, it is possible to define the expected version order as the commit order of transactions. To recover the version order, we leverage PostgreSQL's logical streaming replication and the Debezium change data capture tool. The primary benefit of using Debezium's CDC tool is that it abstracts the low-level details of PostgreSQL's WAL format and integrates with its logical replication protocol. This significantly reduces the effort required to implement version certificate recovery, with the implementation only just exceeding 300 lines of Python code.

In addition to recovering the version order, our approach also requires recovering the version set of each predicate operation. PostgreSQL uses an MVCC scheme along with a snapshot mechanism to determine the set of versions visible to an operation. At the *serializable* isolation level, each transaction uses a single snapshot to determine the set of visible versions for its operations. PostgreSQL defines a snapshot in three parts:

(1) The smallest transaction ID, T_{min}, that is still active. The versions written by transactions with a smaller ID than T_{min} are visible (modulo issues with wraparound, which we do not discuss here, but can be handled).
(2) The largest transaction ID, T_{max}, that is still active. The versions written by transactions with an ID greater than or equal to T_{max} are therefore not visible.
(3) A list of active transaction IDs, `active_txs`, between T_{min} and T_{max}, whose versions are not visible.

PostgreSQL represents these snapshots in a condensed form of T_{min}: T_{max}: `active_txs`. The snapshot $100 : 104 : 100, 102$ means that 100 is the smallest active transaction ID, 104 is the largest active transaction ID and finally, that transactions with ID 100 and 102 are active, so their versions are not visible.

In general, the visibility rules of PostgreSQL are complex. It is well-known that PostgreSQL lacks support for arbitrary logical time-travel queries, that is, the ability to ask for a consistent view of the database from the point of view of a historical transaction ID. However, we can simulate this for our specific use case of recovering an expected version set by using PostgreSQL's `pg_current_snapshot()` function [42].

For each predicate operation, we record the snapshot used by calling the `pg_current_snapshot()` function. Then, a new transaction, which we call the *shadow transaction*, is started that shares the same snapshot using PostgreSQL's `SET TRANSACTION SNAPSHOT` command. To recover the version set, we then query all rows to see the latest visible version. While simple to implement, it requires running an additional transaction for every predicate operation in the history.

5.4 Generality

With some exceptions, concurrency control protocols can be grouped into categories of similar approaches [9, 55, 56]. There are arguably four main categories: (1) locking, (2) timestamp ordering, (3) optimistic concurrency control, and (4) certifier-based approaches. We have tried to cover as many of these as possible in our three implementations. TiDB uses locking in its pessimistic mode and optimistic concurrency control in its optimistic mode; PostgreSQL uses SSI which can be considered a certifier-based approach; and finally, CockroachDB's core invariant is equivalent to that of a timestamp-ordering protocol. We believe this shows that our approach is general. There will of course be differences between concurrency control protocols even within the same category, however, we expect that the above approaches can be adapted to cover most variations.

5.5 Limitations

Primarily, the goal of using version certificate recovery to test a system is to increase confidence in its correctness. Version certificate recovery does not aim to provide a guarantee that a system is free from bugs. As well as recognizing the benefits of version certificate recovery, it is important to understand its limitations.

There were some practical limitations that we found when applying version certificate recovery to real systems. The first limitation that we encountered was that both CockroachDB and TiDB report only the last written version of each object within a transaction. This meant that we had to disallow any other write operations in transactions that contained a predicate update operation. The second limitation came from the decision to leverage existing functionality to recover some aspects of the version certificate. As a result of this, we rely on this functionality to be correct in order for our results to be valid. We believe this is a sensible trade-off as using this functionality significantly reduced the amount of effort necessary to implement version certificate recovery and also helps to decouple the version certificate recovery implementation from low-level details such as the WAL format.

In addition to the practical limitations, both Emme's soundness and completeness rely on a valid version certificate being presented. If the version certificate is created incorrectly then both false positives and false negatives can occur. This can happen either because of bugs in the mechanisms used to recover the version certificate, for example, a bug in CockroachDB's CDC functionality, or because the version certificate was specified incorrectly, for example, creating the version order in timestamp order for a *serializable snapshot isolation* system.

6 Expected Serialization Order

As discussed in Section 4, a fundamental issue with using Adya's model as the basis for a checker is the $O(P \cdot W)$ cost of computing predicate dependencies. Furthermore, any checker using the Adya model needs to know both the version order and version sets associated with a history. This is not exposed by database systems, so the version certificate recovery approach presented in Section 5 is required to recover this information. While we argue the effort to achieve this is modest, for some *serializable* concurrency control protocols we can do better.

Algorithm 4: Check the expected serialization order of the recorded transactions.

```
Def check_expected_order(txns):
    txns = sort_in_expected_order(txns)
    parent_state = {}
    for txn ∈ txns do
        for op ∈ txn.ops do
            if op.is_read then
                if ¬expected_results_match(parent_state, op) then
                    return False
            else
                new_writes = evaluate(parent_state, op)
                for write ∈ new_writes do
                    parent_state[write.key] = write.value
    return True

Def expected_results_match(parent_state, op):
    expected_results = compute_results(op.query, parent_state)
    return op.results == expected_results
```

For many *serializable* protocols, it is possible to define an expected total order on transactions such that for any execution of the protocol, arranging transactions in that order ensures that they are *serializable*. We call this an *expected serialization order*. For example, Section 2 shows a timestamp ordering protocol that guarantees transactions will always be *serializable* in ascending timestamp order.

Commitment ordering protocols [43], such as strong strict two-phase locking, have an expected serialization order equal to the commit ordering. For optimistic protocols, it is possible to define an expected serialization order by arranging transactions in the order of their validation timestamps. However, there are some *serializable* protocols, such as *serializable snapshot isolation* (SSI) where it is not possible, at least by default, to define an expected serialization order. Nevertheless, it is possible to modify many variants of SSI to record a *serializable* ordering if required [36].

We implement the expected serialization order approach for CockroachDB. CockroachDB guarantees that arranging transactions in ascending timestamp order will guarantee *serializability*, therefore we can define the expected serialization order this way. Notice that this is much simpler than defining an expected version order and expected version sets. All that is needed to recover the expected serialization order is to recover each transaction's timestamp. For CockroachDB, we modify the test client to have each transaction record its own timestamp. Finally, we can sort transactions by their timestamp and pass them to a checker for verification.

Checking an expected serialization order. We built a checker, *Enne*, that supports checking an execution history when provided with the expected serialization order. Intuitively, to check that the execution history is *serializable* when arranged in the expected serialization order, we should be able to replay each transaction and check that the results of any read operations match those that were observed in the history. Crooks et al. [15] formalize this idea and define *serializability* in terms of first-order logic predicates over a total order of transactions. Effectively, to check that a total order of transactions is *serializable*, a "current state" of the database is maintained and each transaction is replayed, with write operations updating the "current state". Each read operation is checked to ensure that it is valid at the "current state". If not, then that particular transaction ordering is rejected. Their formalization does not contain predicate operations, however, it can naturally be extended for *serializability* by handling predicate updates and reads in the same way as normal reads and writes, and their proof sketch can be modified to include predicate dependencies without changing its structure.

Our checker, Enne, uses Algorithm 4 to check an execution history. As with Emme, we require a matches oracle to evaluate predicates. We use our interpreter described in Section 4. The time complexity of checking is reduced as it is only required to evaluate each predicate on the writes in the "current state" rather than on every write. As Section 8 shows, this significantly improves both the performance and scalability of checking histories with predicate operations, and is a major advantage of the expected serialization order approach.

Whilst we have focused on *serializable* protocols in this section, Crooks et al. [15] define weaker isolation levels in terms of first-order logic predicates that must hold over some total order of transactions, so it may also be possible to define an expected total order for some weaker protocols.

7 King Cobra

There are many formal models of isolation levels [2, 3, 8, 13, 15, 37]. A common theme is that they all require exploring a large search space to decide if an execution history conforms to the definition of a particular isolation level. For many important isolation levels, such as *serializability*, this search space is exponentially large with respect to the size of the execution history, making the checking of even moderate-sized execution histories infeasible without a means of cutting down the search space. Therefore, a key determinant of the execution time of a checker is its ability to reduce this space.

Recall from Section 3.3 that Cobra [51] is a black-box checker that supports checking *strong session serializability* by default. In contrast to both white-box checkers introduced in this paper, Cobra relies only on ordering information gleaned from an execution history—i.e. no ordering information from the underlying system is recovered. In order to explore the important role that ordering information, in particular session order and version order information, plays in cutting down the search space that a checker must consider (and therefore its execution time), we create a modified version of Cobra which we call *King Cobra*. King Cobra supports optionally excluding session order information from and including version order information into its checking process. This enables us to run a series of performance experiments, presented in Section 8.4, to evaluate the impact that this ordering information has on King Cobra's execution time.

The key differences between Cobra and our King Cobra extension are summarised in Table 2. Because King Cobra depends on version order recovery, as per our Emme checker it requires some per-database engineering to extract version order information (see Section 4.3). Armed with this version order information, King Cobra then provides a strict improvement over Cobra. As an aside, an important difference between King Cobra and Emme is that King Cobra can benefit from *any* amount of version order information that can be extracted from a database, even if this information is incomplete, whereas the soundness of Emme depends on complete version order information. In Section 8.4, we investigate empirically how the performance of King Cobra improves as an increased amount of version order information is provided.

As previously discussed, King Cobra differs from Cobra in two ways. Firstly, it supports optionally disabling session order edges, allowing it to check *serializability* instead of *strong session serializability*. Secondly, it supports optionally incorporating version order information into its checking process in order to improve its execution speed.

Table 2. A comparison of King Cobra with the state of the art checker Cobra

	Cobra [51]	**King Cobra**
Execution time	2nd	1st
Black-box	Yes	No
Supported isolation levels	*Strict Serializability* and *Strong Session Serializability*	*Serializability*, *Strong Session Serializability* and *Strict Serializability*
Supports predicates	No	No

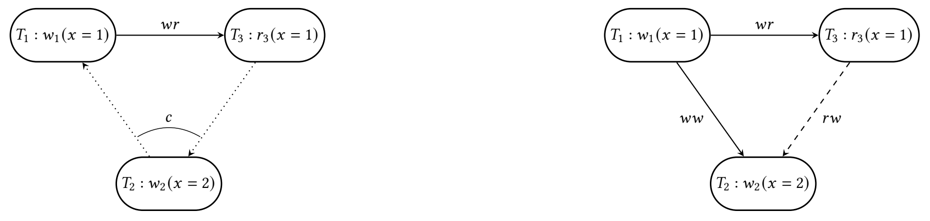
Session order edges. Cobra uses the order of transactions submitted by a client to infer session order edges. These edges allow Cobra to significantly reduce the number of constraints in the polygraph, leading to a large improvement in its execution time. In order to explore the impact of the session order on checking performance and to illustrate the relative cost of checking *serializability* vs *strong session serializability*, we added an option to turn off session order edges in King Cobra so that it could verify *serializability*.

Version order edges. Cobra lacks access to version order information when it constructs the *known graph*. The *known graph* is constructed using only the extracted write-read dependencies from the execution history. Cobra then uses optimizations to try to infer other dependencies that allow constraints to be pruned from the polygraph that it constructs. This raises the question—how much faster would Cobra be if it used version order information? Or, put another way, what is the performance penalty for relying exclusively on information within the execution history?

We added the option to allow using version order information to King Cobra. When the option to use version order information is enabled, King Cobra expects to receive a list of versions associated with each key in the execution history. Not every version needs to be present for King Cobra to function. Then, for each adjacent pair of versions in a single key's version order, a version order edge is added to the known graph. There is nothing special about these edges other than that they provide additional ordering information. This explicitly creates all of the write-write dependencies that are known from the version order. However, there are also read-write conflicts that arise. King Cobra does not require any further modifications to handle these, as there are already optimizations included in Cobra that are able to identify these given sufficient information about the version order.

Just adding the write-write and read-write edges is insufficient to improve King Cobra's execution time. In addition, it is necessary to remove the associated constraints, as it is the number of constraints that determines the 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.

To illustrate the difference between Cobra and King Cobra, Figure 8 shows how the polygraphs generated by Cobra and King Cobra differ for the same execution history H. In the example, the history H consists of three transactions each containing a single read or write operation. Each graph contains three nodes, one per transaction, and also contains an edge $T_1 \rightarrow T_3$ representing the write-read dependency that arises from $r_3(x)$. Cobra has no further information about any dependencies in the graph and therefore cannot add any additional known edges. Instead, it has to add a constraint $c = (T_3 \rightarrow T_2, T_2 \rightarrow T_1)$, which captures the fact that T_2 must have happened either after T_3 or before T_1. When Cobra cannot generate any further known edges, it begins to solve the remaining constraints until it finds an acyclic graph or exhausts all possible graphs. The complexity of this process is exponential in the number of constraints, therefore reducing the number of constraints is vital to achieve a reasonable execution time.



(a) Polygraph constructed by Cobra. (b) Polygraph constructed by King Cobra.

Fig. 8. Example demonstrating the difference between Cobra and King Cobra, showing the polygraph that each generates given the same execution history.

Unlike Cobra, King Cobra has the version order available. In this example, the version order for x is $1 \ll 2$. This allows King Cobra to add an additional write-write edge $T_1 \longrightarrow T_2$. By adding this edge, King Cobra eliminates the need for a constraint to be added as there is no uncertainty over the ordering of T_1 and T_2: T_1 must have happened before T_2. This highlights why King Cobra is faster than Cobra—it is able to significantly reduce the number of constraints that occur in the polygraph by adding additional write-write edges. Furthermore, Cobra already contains an optimisation that is able to infer read-write edges from existing edges. This optimisation allows King Cobra to make further use of the write-write edge that was added, by adding an additional read-write edge $T_3 \longrightarrow T_2$.

8 Evaluation

We evaluate (1) the effectiveness and performance of three implementations of version certificate recovery for PostgreSQL, TiDB, and CockroachDB; (2) the performance of our checker Emme on histories produced from PostgreSQL; (3) the performance of the expected serialization technique and its associated checker Enne on execution histories produced from CockroachDB; and (4) the impact of ordering information on the performance of King Cobra, our modification of the Cobra checker [51]. Unless stated otherwise, all experiments are carried out on a machine with a hexa-core Ryzen 1600 processor, 32 GB RAM, a WD Blue SN570 2 TB SSD, and an Ubuntu 20.04 LTS operating system. We use PostgreSQL 13.3 and run it on a single node. We use TiDB 5.3.0 and use the recommended cluster topology (three PD nodes, three TiKV nodes, and two TiDB nodes). We use CockroachDB v21.2.17 and run three nodes.

8.1 A Note on Evaluation Benchmarks

As is the case for prior work on Elle [4] and Cobra [51], the focus of this paper is on targeted testing to validate the correctness of database systems. The aim of Elle, Cobra's one-shot verification mode and of our own work is not to provide techniques to be used in an "online" setting, i.e. to be run in real-time against a production workload. Both Elle and Cobra include their own history generators which use a similar operation mix to the mix that appears in this paper (read-only, read-modify-write, write-heavy). This operation mix makes sense in the context of the primary use case (testing/validation) as workloads that are good at finding bugs may not be the same as standard database benchmarks. When measuring the execution time of each checker, only the checking time is included—any time spent collecting the histories or converting the histories across checker formats is not counted.

8.2 Version Certificate Recovery

We created a checker *Emme*, described in Section 4, that supports checking using the version certificate recovery technique described in Section 5. This subsection evaluates the effectiveness and performance of Emme when paired with the version certificate recovery technique. We consider all three version certificate recovery implementations: (1) log-based change data capture (CDC) for CockroachDB using their `CHANGEFEED` mechanism, (2) log-based CDC for PostgreSQL using the Debezium CDC tool [17], and (3) heap-scanning for TiDB.

Effectiveness. We demonstrate that version certificate recovery can find known bugs in existing systems. Firstly, we were able to demonstrate that version certificate recovery could find a known error in PostgreSQL 12.3's *serializable* isolation level [41]. The error is a well-known isolation anomaly that can occur in *serializable snapshot isolation* implementations [39]. Secondly, we asked a CockroachDB engineer to create three versions of CockroachDB [50], each with a bug that would violate its *serializability* guarantee. To avoid biases, we avoided inspecting the changes to CockroachDB before attempting to find each bug. Version certificate recovery was able to detect all three bugs. This shows that version certificate recovery is effective at detecting isolation level anomalies in real-world database systems. Finally, we demonstrate that our approach can find predicate-only anomalies that Elle—a state-of-the-art checker—cannot. To do this, we executed patterns of transactions that are highly likely to produce predicate-only anomalies when run at the *read committed* isolation level. We executed these using PostgreSQL and tried to validate them at the *serializable* isolation level. Emme was able to detect these predicate-only anomalies and rejected them as invalid, however, Elle could not detect these and accepted them as *serializable* histories. This clearly shows the importance of being able to check histories for predicate anomalies.

Cost of recovering a version certificate. For each version certificate recovery implementation, we measured the execution time of both the test client and the version certificate recovery process to confirm that the time spent recovering the version certificate was less than the history generation time. This demonstrates that it is possible to hide the cost of version certificate recovery behind that of the test client. However, for ease of implementation, all version certificate recovery implementations recover the version order and version sets after the test clients finish executing. Nevertheless, it would be possible to have an implementation of version certificate recovery that runs alongside the test client execution.

Performance of Emme. To examine the performance characteristics of Emme, we carry out two experiments. The first compares the performance of Emme and Elle on histories containing only non-predicate operations and the second demonstrates the performance of Emme on histories containing predicate operations. All experiments use a single table with three columns. 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. Predicate histories contain basic range queries such as `SELECT c0, c1 FROM table WHERE c1 >= 0 AND c1 <= 10`, as well as aggregate operations min and max.

Fig. 9 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. The experiment uses a mix of 30% updates, 40% reads, 20% increments, and 10% inserts, which gives a 50/50 read/write ratio. We limit the number of keys inserted to 1000. Emme performs similarly to Elle for smaller history sizes but then starts to outperform Elle as the history size increases. We believe this is due to Elle's requirement to use list-append operations, which causes all reads to contain their full version order history as a list. As the history increases in size, each key accumulates increasingly many versions, which makes processing each read progressively more expensive.

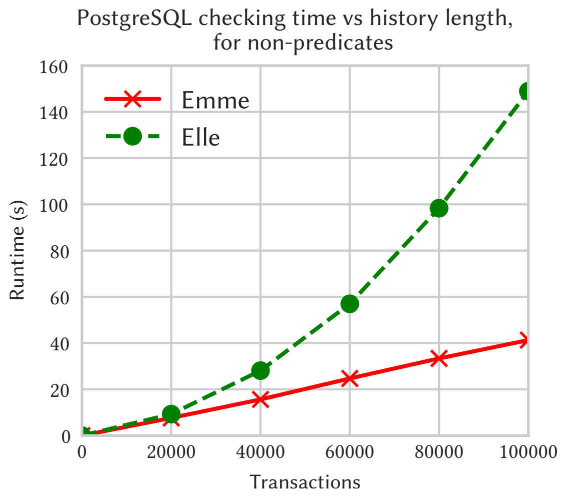


Fig. 9. Comparison of history size and the execution time of item dependency verification in the Emme and Elle verifiers. Each transaction executed five non-predicate operations.

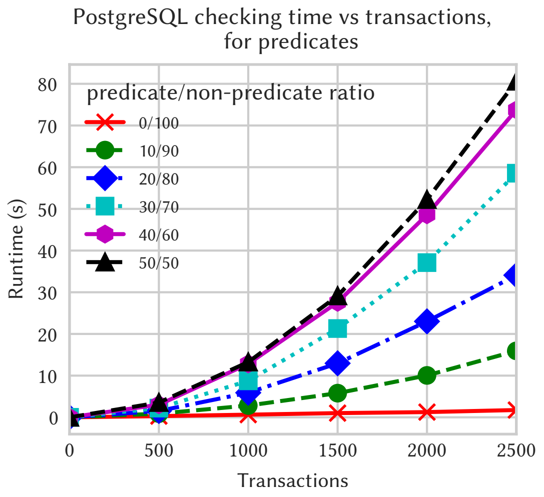


Fig. 10. Comparison of history size and the execution time of mixed item and predicate dependency verification in the Emme verifier. Each transaction executed a mix of five predicate and non-predicate operations. Different ratios of predicate to non-predicate operations were chosen.

Fig. 10 shows how the ratio of predicate and non-predicate operations in a history affects Emme's checking performance. The experiment uses 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 is set to a maximum of 100. Predicate checking has $O(P * V)$ complexity, where P is the number of predicates in the history and V is the total number of versions. The graph shows this empirically, with both the number of transactions (and therefore the number of versions) and the number of predicate operations causing an increase in execution time as they themselves increase. This highlights the

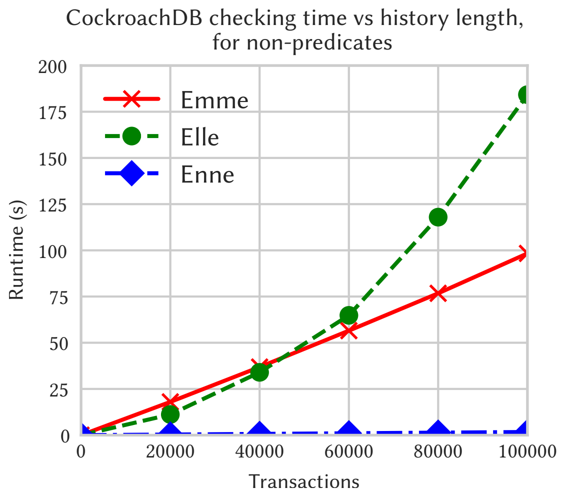


Fig. 11. Comparison of history size and checking time for the Elle, Emme, and Enne checkers using item-only histories. The same mix of operations as Fig. 9 was used.

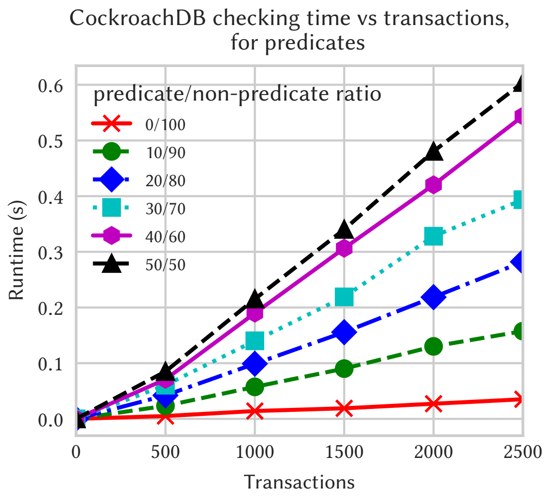


Fig. 12. Comparison of the effect of both increasing history size and increasing the mix of predicate operations on the checking time of Enne for histories produced from CockroachDB.

performance limitations of predicate checking due to the Adya model's data-driven definitions of predicate anomalies. Nevertheless, Emme is still able to verify moderately large histories in a time frame that is acceptable for testing.

8.3 Expected Serialization Order

We created a checker *Enne* that supports checking using the expected serialization order technique described in Section 6. To demonstrate that Enne is as effective as Emme, we ensured that it could detect all three errors in the modified versions of CockroachDB.

Performance. A key benefit of using Enne is its superior performance. Fig. 11 compares the performance of Enne to that of Elle and Emme using item-only histories generated from CockroachDB. Although Enne's execution time still scales linearly with the history size, its execution time grows significantly slower than that of both Elle and Emme. Enne verifies a 100,000 transactions history in only 1.7 seconds compared to 98.3 seconds it takes Emme, and 184.2 seconds it takes Elle.

Enne has the most pronounced speedup for histories containing predicate operations. It checks a history containing 2,500 transactions in only 0.6 seconds, compared to 76.2 seconds taken by Emme. Fig. 12 shows how the execution time of Enne changes as both the history size and proportion of predicate operations changes. Unlike Emme's predicate checking performance (Fig. 10), Enne's performance scales linearly with the history size when holding the proportion of predicate operations constant. This makes it suitable for checking large histories containing predicate operations.

8.4 Evaluating the impact of ordering information on the performance of King Cobra

This subsection presents a series of performance experiments run against King Cobra, our modified version of Cobra [51], a black-box *serializability* checker. Section 7 provides further details on King Cobra. By conducting these performance experiments we seek to answer the following questions:

(1) What is the performance impact of checking serializability vs strong session serializability?
(2) What impact does version order information play in checking performance?
(3) How does the amount of version order information affect checking performance?

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

The first workload, which we call the read-modify-write (RMW) workload, contains 25% read operations, 25% write operations, and 50% read-modify-write operations. A read-modify-write is modeled as a read of a key followed by a write to the same key within the same transaction. 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, 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 session 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.

We ran four performance experiments for each of the three workloads. The results are shown in Figure 13. The first experiment disabled both session order and version order edges (*No SO, No VO*); The second experiment enabled version order edges but disabled session order edges (*VO w/o SO*); The third experiment enabled session order edges but disabled version order edges (*SO w/o VO*); The fourth experiment enabled both session order and version order

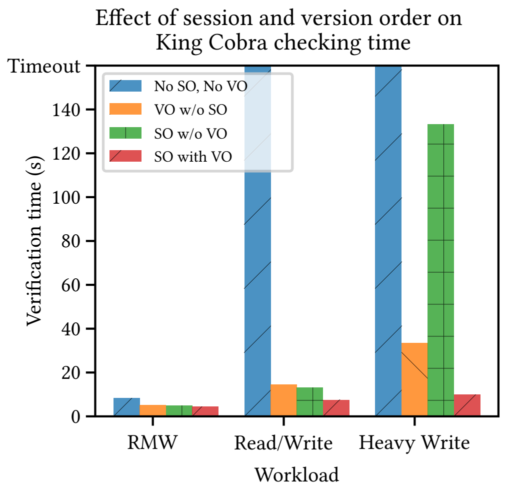


Fig. 13. The effect on the execution time of King Cobra when version order edges and session order edges are included and excluded. Recall that SO and VO refer to "session order" and "version order", respectively.

edges (*SO with VO*). For the cases where version order information is enabled, we include 90% of the version order information available.

Impact of Cobra's optimizations on checking performance. To provide a performance baseline with which to compare, we first run Cobra in its default execution mode, with session order edges enabled and all of its optimizations. This means Cobra is checking *strong session serializability* instead of *serializability*. A history is serializable if it is strong session serializable, however the converse is not necessarily true.

Cobra's default execution mode is labeled *SO w/o VO* in Figure 13. The RMW workload has the lowest execution time out of the three workloads. The read/write workload is 3× slower to check than the RMW workload. This demonstrates the effectiveness of Cobra's write-combining optimization when there are many RMW operations in an execution history. The heavy write workload is 10× slower to check than the read/write workload and 30× slower to check than the RMW workload. In this case, Cobra's optimizations are less effective at reducing the number of constraints in the polygraph as they primarily rely on RMW and read operations. Cobra also has no information about the version order (the order of writes), so there is significant uncertainty about the ordering of writes, leading to a high number of constraints that are not able to be pruned.

Impact of the session order on checking performance. Cobra uses session order edges to improve its execution time at the expense of checking *strong session serializability* instead of *serializability*. In order to explore the impact of the session order on checking execution time and to illustrate the relative cost of checking *serializability* vs *strong session serializability*, we ran all three workloads against King Cobra with the session order edges disabled.

The variation of King Cobra that does not use session order edges is labeled *No SO, No VO* in Figure 13. For the RMW workload, disabling session order edges leads to a 2× increase in execution time. This is the smallest increase out of the three workloads and shows that Cobra is already able to effectively reduce the search space without session order edges for RMW workloads. However, for both the read/write and heavy write workloads, disabling the session order edges causes the checking time to exceed the maximum time limit. This demonstrates that irrespective of Cobra's optimizations, the session order is a key factor in cutting down the search space for these workloads.

Impact of the version order on checking performance. As explained in Section 7, by default, Cobra does not have access to any version order information. King Cobra, our modified version of Cobra, does support using version order information during checking. To understand the impact of version order information on checking, we run all three workloads against King Cobra with the version order edges enabled.

The results of disabling/enabling the version order are shown in Figure 13. The variation of King Cobra that uses only version order information is labeled *VO w/o SO*. When enabling the version order edges, but keeping the session order edges disabled, King Cobra performs well across all workloads. For the RMW workload, there is an 8% improvement in execution time, which further demonstrates that Cobra's RMW optimizations are very effective. For the read/write and heavy write workloads, which both exceeded the 5 minute checking limit when neither session or version order edges were enabled, there is a significant performance improvement, with the read/write workload finishing within 15 seconds and the heavy write workload finishing within 34 seconds. This demonstrates the crucial role that version order information plays in enabling fast checking of *serializability*.

The variation of King Cobra that uses both version order and session order information is labeled *SO with VO* in Figure 13. For the RMW workload, there is a 16% performance improvement from enabling the session order edges in addition to the version order edges. For the read/write and heavy write workloads, there is a 92% and 236% performance improvement respectively. This shows that the session order carries additional information that contributes to reducing the search space compared to just using the version order information.

Without version order edges, but with session order edges, King Cobra performs slightly better on the read/write workload than with just version order edges. However, in the heavy write workload, King Cobra performs considerably worse than with just version order edges. In the read/write workload, adding the session order allows Cobra's pruning optimization to remove many of the constraints it would have otherwise missed and since there are not too many write operations that are not ordered by the session 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 session order edges, which means Cobra is unable to remove enough constraints to finish quickly. In both these cases, King Cobra with only version order edges performs well since it can infer the order of these inter-client writes and remove enough constraints to finish quickly.

Impact of partial version order information on checking performance. Figure 14 demonstrates how varying the amount of version order information to which King Cobra has access affects its execution time across the three workloads. Session order edges were disabled for all workloads. To provide King Cobra with partial version order information, we first select a percentage of the version order to discard, for example 50%, and then randomly select the corresponding number of the versions within the version order to exclude. The version order is represented within King Cobra as an ordered list of versions associated with a key, so the process of discarding a version consists of removing it from the list.

For the RMW workload, King Cobra performed well irrespective of how much version order information was included, since it was able to apply Cobra's standard optimization techniques. However, in both the read/write and heavy write workloads, 90% or more of the version order was needed to perform well. Whilst lower levels of version order information allows King Cobra to cull a significant percentage of its constraints, there is a low ceiling on the number of constraints needed to cause the time limit to be exceeded since the verification time of Cobra is exponential in the number of constraints.

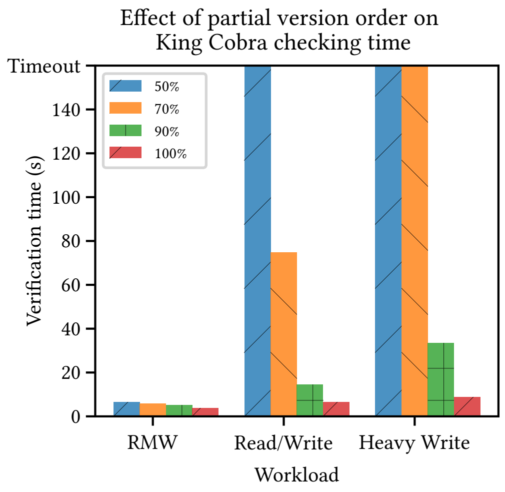


Fig. 14. The effect on King Cobra's execution time of varying the amount of version order information it can use, for three different workloads.

Summary of King Cobra's performance. This subsection has explored how the execution time of the King Cobra checker varies with the amount of ordering information it is given. Like Elle and Emme, Cobra uses the property that each write in the execution history is unique to reduce the search space that needs to be explored to check if an execution history conforms to a particular isolation level. However, with just this information, even with the various optimizations it implements, King Cobra struggles to check *serializability* for two out of the three workloads that were used within a reasonable amount of time. When checking *strong session serializability*, Cobra is able to use session order information to further cut down on the search space, which leads to significantly reduced checking time, particularly for the non-RMW workloads. When version order information is available to King Cobra it is able to use this information to further cut down the search space and in the case of checking *serializability* enables King Cobra to successfully check all three workloads within the 5 minute limit. For checking *strong session serializability*, the inclusion of version order information further improves the execution time, albeit not as drastically.

Cobra makes an important contribution in the effort to build a general-purpose black-box checker. However, for systems that do not support *strong session serializability* or *strict serializability*, Cobra is unable to check the *serializability* of moderate size execution histories for some workloads in reasonable time. The fundamental reason for this is the unavailability of the underlying version order information from the database system that produced the execution history. These results raise two interesting questions—is it possible to create a general-purpose checker that supports a range of operations and does not rely on additional ordering information, whilst also being an efficient checker for large-scale real-world execution histories? Or will it always be necessary to augment execution histories with version order information in order to facilitate efficient checking?

9 Related Work

Formal models of isolation levels. ANSI SQL-92 [5] 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. Building on previous work [22]

Berenson et al. [7] show that the absence of the three phenomena defined in ANSI SQL does not guarantee *serializable* execution. They define stricter versions of the ANSI SQL phenomena and formalize two additional isolation levels called *cursor stability* and *snapshot isolation*. Bernstein [8, 9] defines *serializability* in terms of dependency graphs and a version order. Adya et al. [2, 3] extend this model to support a wider range of isolation levels and systems. Departing from dependency graphs, recent work has moved away from defining isolation levels in terms of ordering low-level operations and instead focuses on more declarative definitions [12, 15, 53].

Testing and verification of isolation levels. Hermitage [27] provides a set of fixed test cases that demonstrate the behavior of various database systems at different isolation levels. PostgreSQL [40] uses a tool called Isolationtester to run a select set of interleavings for manually-written tests to find bugs in its implementation of various isolation levels.

In general, *serializability* checking is NP-Complete [37]. Biswas and Enea [10] provide exponential-time checkers for *prefix consistency*, *snapshot isolation*, and *serializability*, whilst providing polynomial-time checkers for *read committed* and *casual consistency*. Plume [31] offers faster polynomial-time checking for these same isolation levels, whilst the AWDIT [33] checker further improves upon this by providing provably optimal checking, achieving time complexities of $O(n^{3/2})$, $O(n^{3/2})$, and $O(n \cdot k)$, respectively, when testing execution histories of size n and k sessions.

Cobra [51] is an SMT solver based approach for verifying the *serializability* of key-value histories. It uses various heuristics to optimize checking speed on certain workloads, however, in the worst case still has exponential running time. Similarly, Viper [57] and PolySI [24] rely on SMT solver for checking, however, they support checking *snapshot isolation* rather than *serializability*.

Elle [4] is a checker based on the Adya model of isolation levels. It supports a variety of isolation levels and has successfully found bugs in real-world systems. To work efficiently, Elle requires the database system to support atomic list-append operations. Histex [30] is a gray-box approach to testing isolation level implementations. To the best of our knowledge, Histex is the only system—other than ours—that can check histories that include predicate operations. However, these histories must have been produced by a single-version system running a locking protocol, so Histex only works for a very narrow set of systems, and therefore, we do not compare our checker against it.

10 Conclusion

Our work shows that it is possible to define and recover a *version certificate* consisting of an expected version order and set of expected version sets for three widely used database systems—TiDB, PostgreSQL, and CockroachDB. Our checker, Emme, supports checking both the *serializability* and *snapshot isolation* of execution histories using the recovered version certificate, which makes Emme the first general-purpose checker that can check histories containing predicate operations. Our results show that version certificate recovery is an effective validation method, demonstrating that it can identify invalid histories caused by a known bug in an older version of PostgreSQL's *serializability* implementation and caused by three bugs in a fork of CockroachDB's *serializability* implementation that were introduced purposefully by a CockroachDB engineer.

Additionally, we introduce the notion of an *expected serialization order* and describe how it can be applied to a range of *serializable* concurrency control protocols without the need for defining and recovering a version certificate. Our results show that using the expected serialization order technique, along with its associated checker Enne, leads to faster checking performance and better scalability for checking histories with predicate operations: for CockroachDB, Enne allows predicate histories to be checked 53×–120× faster compared with Emme.

Finally, we present King Cobra, a modified version of the black-box checker Cobra, that incorporates version order information in its checking process and also supports checking *serializability*. Our evaluation of the execution time of King Cobra with varying degrees of version order and session order information demonstrate the value of version order information: *without* version order information, King Cobra is unable to check the *serializability* of two out of the three workloads tested within a five minute time limit. This highlights the vital role that ordering information, particularly the version order, plays in enabling efficient checking. We believe this presents a challenge for isolation level specifications as there appears to be a tension between high-level declarative specifications, which facilitate understanding and reasoning about the characteristics of an isolation level, and the desire for efficient checking in order to facilitate testing and verification of isolation level implementations.

Acknowledgments

We would like to thank the anonymous reviewers of both this TOCS article and the original EuroSys article, and our EuroSys shepherd Annette Bieniusa for their insightful and valuable comments that helped us improve this paper. We would also like to thank Nathan Vanbenschoten for enabling a part of our evaluation by introducing three bugs into a fork of CockroachDB. This work was supported by the EPSRC IRIS Programme Grant (EP/R006865/1).

References

[1] Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL query generation for systematic testing of database engines. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 329–332. https://doi.org/10.1145/1858996.1859063

[2] A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Technical Report. Massachusetts Institute of Technology, USA.

[3] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 67–78. https://doi.org/10.1109/ICDE.2000.839388

[4] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280. https://doi.org/10.5555/3430915.3442427

[5] ANSI [n. d.]. ANSI X3.135-1992, American National Standard for Information Systems — Database Language — SQL, November 1992.

[6] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174

[7] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 1–10. https://doi.org/10.1145/223784.223785

[8] P.A. Bernstein, D.W. Shipman, and W.S. Wong. 1979. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering* SE-5, 3 (1979), 203–216. https://doi.org/10.1109/TSE.1979.234182

[9] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (Jun 1981), 185–221. https://doi.org/10.1145/356842.356846

[10] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28. https://doi.org/10.1145/3360591

[11] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (Dec 2009), 42 pages. https://doi.org/10.1145/1620585.1620587

[12] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 42)*, Luca Aceto and David de Frutos Escrig (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

[13] Andrea Cerone and Alexey Gotsman. 2016. Analysing Snapshot Isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (Chicago, Illinois, USA) *(PODC '16)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/2933057.2933096

[14] Jack Clark, Alastair F. Donaldson, John Wickerson, and Manuel Rigger. 2024. Validating Database System Isolation Level Implementations with Version Certificate Recovery. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25,*

2024. ACM, 754–768. https://doi.org/10.1145/3627703.3650080

[15] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, Elad Michael Schiller and Alexander A. Schwarzmann (Eds.). ACM, 73–82. https://doi.org/10.1145/3087801.3087802

[16] K. Daudjee and K. Salem. 2004. Lazy database replication with ordering guarantees. In *Proceedings. 20th International Conference on Data Engineering*. 424–435. https://doi.org/10.1109/ICDE.2004.1320016

[17] debezium [n. d.]. Debezium. https://debezium.io Accessed: 2022-11-07.

[18] Murat Demirbas, Marcelo Leone, Bharadwaj Avva, Deepak Madeppa, and Sandeep S. Kulkarni. 2014. Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases.

[19] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (Nov. 1976), 624–633. https://doi.org/10.1145/360363.360369

[20] Alan D. Fekete. 2018. Serializable Snapshot Isolation. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_80774

[21] Alan D. Fekete. 2018. Snapshot Isolation. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_346

[22] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, G. M. Nijssen (Ed.). North-Holland, 365–394.

[23] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (Aug 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[24] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-Box Checking of Snapshot Isolation in Databases. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1264–1276. https://doi.org/10.14778/3583140.3583145

[25] Jepsen testing framework. [n. d.]. Jepsen testing framework. https://github.com/jepsen-io/jepsen Accessed: 2022-12-03.

[26] Kyle Kingsbury. 2013. Jepsen Analyses. https://jepsen.io/analyses

[27] Martin Kleppmann. 2014. Hermitage: Testing transaction isolation levels. https://github.com/ept/hermitage https://github.com/ept/hermitage.

[28] H. T. Kung and C. H. Papadimitriou. 1979. An Optimality Theory of Concurrency Control for Databases. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 116–126. https://doi.org/10.1145/582095.582114

[29] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. 2002. Specifying and Verifying Systems with TLA+. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (Saint-Emilion, France) *(EW 10)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/1133373.1133382

[30] Dimitrios Liarokapis, Elizabeth O'Neil, and Patrick O'Neil. 2019. HISTEX HISTory EXerciser : A tool for testing the implementation of Isolation Levels of Relational Database Management Systems. *CoRR* abs/1903.00731 (2019). arXiv:1903.00731 http://arxiv.org/abs/1903.00731

[31] Si Liu, Long Gu, Hengfeng Wei, and David Basin. 2024. Plume: Efficient and Complete Black-Box Checking of Weak Isolation Levels. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 302 (Oct. 2024), 29 pages. https://doi.org/10.1145/3689742

[32] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a Verified Relational Database Management System. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/1706299.1706329

[33] Lasse Møldrup and Andreas Pavlogiannis. 2025. AWDIT: An Optimal Weak Database Isolation Tester. *Proc. ACM Program. Lang.* 9, PLDI, Article 209 (June 2025), 25 pages. https://doi.org/10.1145/3742465

[34] MySQL [n. d.]. MySQL phantom read bug report. https://bugs.mysql.com/bug.php?id=27197 Accessed: 2023-04-11.

[35] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

[36] Elizabeth J. O'Neil and Patrick E. O'Neil. 2016. Determining serialization order for serializable snapshot isolation. *Information Systems* 58 (2016), 14–23. https://doi.org/10.1016/j.is.2016.02.001

[37] Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J. ACM* 26, 4 (1979), 631–653. https://doi.org/10.1145/322154.322158

[38] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

[39] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (Aug 2012), 1850–1861. https://doi.org/10.14778/2367502.2367523

[40] PostgreSQL. [n. d.]. PostgreSQL. https://www.postgresql.org Accessed: 2022-11-30.

[41] PostgreSQL 12.3 serializable error [n. d.]. PostgreSQL commit that fixes an error in its serializable isolation level. https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=5940ffb221316ab73e6fdc780dfe9a07d4221ebb Accessed: 2022-11-30.

[42] PostgresSQL [n. d.]. PostgreSQL `pg_current_snapshot()` documentation. https://www.postgresql.org/docs/13/functions-info.html#FUNCTIONS-PG-SNAPSHOT Accessed: 2022-11-30.

[43] Yoav Raz. 1992. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment. In *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, Li-Yan Yuan (Ed.). Morgan Kaufmann, 292–312. http://www.vldb.org/conf/1992/P292.PDF

[44] David P. Reed. 1983. Implementing Atomic Actions on Decentralized Data. *ACM Trans. Comput. Syst.* 1, 1 (Feb. 1983), 3–23. https://doi.org/10.1145/357353.357355

[45] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. https://doi.org/10.1145/3368089.3409710

[46] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428279

[47] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger

[48] Andreas Seltenreich. 2019. SQLSmith. https://github.com/anse1/sqlsmith

[49] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.

[50] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[51] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 63–80. https://www.usenix.org/conference/osdi20/presentation/tan

[52] TiKV [n. d.]. TiKV. https://tikv.org/ Accessed: 2022-11-30.

[53] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1, Article 19 (Jun 2016), 34 pages. https://doi.org/10.1145/2926965

[54] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 5–20. https://doi.org/10.1145/3035918.3064037

[55] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (Mar 2017), 781–792. https://doi.org/10.14778/3067421.3067427

[56] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8 (Nov 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[57] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 654–671. https://doi.org/10.1145/3552326.3567492