Boosting End-to-End Database Isolation Checking via Mini-Transactions

Hengfeng Wei, Jiang Xiao, Na Yang
Nanjing University
hfwei@nju.edu.cn
{mg21330063, nayang}@smail.nju.edu.cn

Si Liu, Zijing Yin
ETH Zurich
{zijing.yin, sliusi}@inf.ethz.ch

Yuxing Chen, Anqun Pan
Tencent Inc.
{axingguchen, aaronpan}@tencent.com

***Abstract*—Transactional isolation guarantees are crucial for database correctness. However, recent studies have uncovered numerous isolation bugs in production databases. The common black-box approach to isolation checking stresses databases with large, concurrent, randomized transaction workloads and verifies whether the resulting execution histories satisfy specified isolation levels. For strong isolation levels such as strict serializability, serializability, and snapshot isolation, this approach often incurs significant end-to-end checking overhead during both history generation and verification.**

We address these inefficiencies through the novel design of Mini-Transactions (MTs). MTs are compact, short transactions that execute much faster than general workloads, reducing overhead during history generation by minimizing database blocking and transaction retries. By leveraging MTs' read-modify-write pattern, we develop highly efficient algorithms to verify strong isolation levels in linear or quadratic time. Despite their simplicity, MTs are semantically rich and effectively capture common isolation anomalies described in the literature.

We implement our verification algorithms and an MT workload generator in a tool called MTC. Experimental results show that MTC outperforms state-of-the-art tools in both history generation and verification. Moreover, MTC can detect bugs across various isolation levels in production databases while maintaining the effectiveness of randomized testing with general workloads, making it a cost-effective solution for black-box isolation checking.

***Index Terms*—Transactional isolation levels, Strict serializability, Serializability, Snapshot isolation, Isolation checking**

I. Introduction

Databases serve as the backbone of modern web applications. Transactional isolation levels (the "I" in ACID – Atomicity, Consistency, Isolation, and Durability [1]), such as Strict Serializability (SSER), Serializability (SER), and Snapshot Isolation (SI) [2], are crucial for ensuring database correctness. However, recent studies [3], [4], [5], [6], [7] have uncovered numerous isolation bugs in many production databases, raising concerns about whether these databases actually uphold their promised isolation guarantees in practice. Isolation bugs are notoriously difficult to detect, as they often produce incorrect results without any explicit errors. For example, an SER violation in PostgreSQL remained undetected for nearly a decade before being identified recently [8].

In recent years, a range of sophisticated checkers [3], [4], [5], [6], [9], [10], [11] has emerged to verify database isolation guarantees. These checkers operate in a *black-box* fashion, as database internals are often unavailable or opaque. To extensively stress test databases, they rely on randomized workload generators that produce large, concurrent transaction sets, aiming to either increase the likelihood of exposing isolation bugs or to build confidence in their absence. Each generated workload is *general*, typically involving dozens of clients, tens of thousands of transactions per client, and dozens of operations per transaction. Subsequently, these checkers collect database execution histories to verify compliance with the specified isolation level. The Cobra [9] and PolySI [4] checkers represent these histories as transactional dependency graphs [12], [13] and leverage off-the-shelf solvers to identify specific cycles that indicate isolation violations.

Challenges: The randomized testing approach using general transaction (GT) workloads frequently encounters efficiency challenges in practice, primarily due to two factors.

First, isolating a large number of highly concurrent GTs imposes significant *execution overhead* on databases. This overhead is particularly exacerbated under strong isolation levels, including SSER, SER, and SI [14], [15]. For example, with optimistic concurrency control [16], databases may abort more transactions due to conflicts, such as concurrent transactions attempting to write to the same object. To obtain a history with sufficiently many committed transactions, the system may require time-intensive retries. Alternatively, under pessimistic concurrency control [16], acquiring and releasing locks to resolve conflicts among concurrent transactions also incurs significant time costs.

Second, large histories composed of GTs often result in extensive, dense transactional dependency graphs [12], [13], [5]. While such graphs are effective for detecting isolation bugs, they also impose substantial *verification overhead* on isolation checkers, which must handle the complexity of encoding and solving dependency constraints or traversing these graphs to identify cycles. This verification overhead is further amplified by the inherently high complexity of verifying strong isolation levels like SSER, SER, and SI, which have been shown to be NP-hard in general [5], [17]. Notably, a checker's efficiency can greatly impact its effectiveness in bug detection, given the often limited resources such as time and memory [4], [6], [9].

A straightforward approach to addressing the above challenges is to randomly generate smaller transaction workloads, e.g., with each transaction comprising only a few operations.

However, this presents a dilemma. On one hand, such workloads could mitigate inefficiency, as shorter transactions generally execute faster and have lower abort rates. On the other hand, these short transactions may not be as effective as large GTs in uncovering isolation bugs. In fact, randomly generated, arbitrarily short transactions may even fail to capture certain data anomalies. For example, the WRITESKEW anomaly, which violates SER, requires both concurrent transactions to read and subsequently write to both objects x and y, as illustrated in Figure 5n (see Section III-B).

Our Approach: In this paper, we propose a unified solution to this dilemma through the novel design of *Mini-Transactions* (MTs), which addresses both types of inefficiencies while preserving the effectiveness of randomized testing in detecting isolation bugs.

An MT is a compact transaction, containing no more than four read/write operations (see Section III), which we employ in both the *history generation* and *history verification* stages. As MTs are short, they can be executed much faster than GTs within a database, leading to more efficient processing and reduced overhead. Specifically, shorter transactions minimize database blocking and exhibit lower abort rates.

However, even with these improvements in execution time, efficient verification against strong isolation levels remains challenging due to the potential for large transaction numbers, which generate dense dependency graphs. To address this issue, we design MTs to adhere to the *read-modify-write* (RMW) pattern, where each write operation is preceded by a read operation on the same object. Furthermore, by ensuring that each MT writes unique values, a common practice among existing isolation checkers [5], [9], [4], [10], [3], [7], we enable highly efficient algorithms in linear or quadratic time (relative to the number of transactions) to verify strong isolation levels, including SSER, SER, and SI, over MT histories (see Section IV). Particularly, for MT histories of *lightweight transactions* (i.e., Compare-And-Set operations), we develop a linearizability [18] verification algorithm that operates in linear time. This design significantly reduces verification overhead. Notably, we also establish the NP-hardness of verifying strong isolation levels for MT histories *without* unique values.

To address the challenge on effectiveness in bug detection, we demonstrate that, despite their simplicity, MTs are semantically rich enough to capture common isolation anomalies that can occur in GT histories. These encompass all 14 well-documented isolation anomalies specified in contemporary specification frameworks for transactional isolation levels [12], [19], [13], [5], [6]. Such anomalies include THINAIRREAD, ABORTEDREAD, FUTUREREAD, NOTMYLASTWRITE, NOTMYOWNWRITE, INTERMEDIATEREAD, NONREPEATABLEREADS, SESSIONGUARANTEEVIOLATION, NONMONOTONICREAD, FRACTUREDREAD, CAUSALITYVIOLATION, LONGFORK, LOSTUPDATE, and WRITESKEW; see Figure 5 in Section III-B for illustrations. We believe this list is exhaustive with respect to the aforementioned specification frameworks.

Our insight behind the MT-based characterization of anomalies is that, while isolation violations have been detected in GT workloads of substantial size, these violations are typically due to a small subset of operations within a limited number of transactions. For example, Figure 1 illustrates an SI violation identified in MariaDB Galera (v10.7.3) using PolySI [4], depicted as a cyclic dependency graph (see Section II-D). The corresponding GT workload comprises 10 sessions, each containing 100 transactions, with each transaction performing 10 operations. However, as highlighted in red, the core of this bug involves only three transactions, each with two operations, which can be represented by our MTs. Specifically, both committed transactions T_{423} and T_{830} read the value written by T_{949} on object 2 and subsequently write different values. This leads to a LOSTUPDATE anomaly that violates SI.

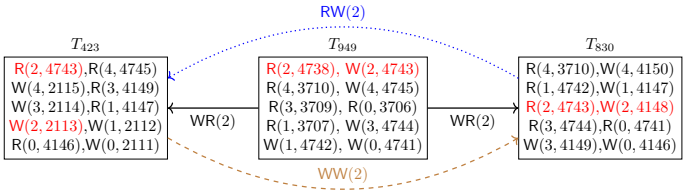


Fig. 1: An SI violation (LOSTUPDATE) detected in MariaDB Galera (v10.7.3). The core operations involved are highlighted in red. This core comprises only three transactions with two operations each, which can be represented by our MTs.

Contributions: Our approach is independent of isolation levels. Nonetheless, it is particularly effective for strong isolation levels, including SSER, SER, and SI, which impose higher execution overhead on databases and involve greater verification complexity. Our contributions are three-fold:

- At the conceptual level, we address inefficiencies in both history generation and verification stages during black-box checking of database isolation levels by proposing a unified approach centered on MTs.
- At the technical level, we propose the design of MTs, which are compact yet semantically rich, capable of capturing all 14 well-documented isolation anomalies specified in contemporary specification frameworks [12], [19], [13], [5], [6]. Furthermore, we develop a suite of highly efficient verification algorithms for strong isolation levels, leveraging the RMW pattern in MTs and unique values in MT histories. We also establish that verifying strong isolation levels for MT histories *without* unique values is NP-hard.
- At the practical level, we implement our verification algorithms, along with an MT workload generator, in a tool called MTC. Experimental results demonstrate that MTC outperforms state-of-the-art isolation checkers in the end-to-end checking process including both history generation and verification. Furthermore, we show that MTC can detect bugs across various isolation levels in production databases while maintaining the effectiveness of randomized testing with general workloads. R1-O1

II. Background

A. Database Isolation Levels and Black-box Checking

Databases are essential to modern cloud-based systems and web applications, serving as the backbone for geo-replicated data storage. Applications coordinate their highly concurrent data accesses through transactions, each composed of multiple read and write operations. To balance data consistency with system performance, databases provide various isolation levels. We focus on strong isolation levels, including SSER, SER, and SI, which are widely implemented in production databases [20], [21], [22], [23], [24], [25]. The gold standard, SER, ensures that all transactions appear to execute sequentially, one after another. SI relaxes SER but prevents undesirable anomalies such as FRACTUREDREAD, CAUSALITYVIOLATION, and LOSTUPDATE. SSER strengthens SER by enforcing the real-time order of transactions.

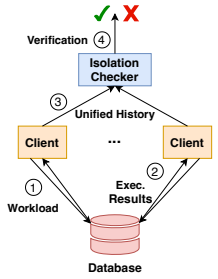


Fig. 2: Workflow of black-box isolation checking.

Black-box database isolation checking involves two stages: *history generation* (Steps ①, ②, and ③) and *history verification* (Step ④); see Figure 2. First, clients send transactional requests to the database (①), treating it as a black-box system. Each client records the requests sent to the database along with the corresponding results received (②). For example, if a client performs a read operation, it records both the requested object and the returned value. The logs from all clients are combined into a single history, which is provided to the isolation checker (③). Finally, the checker performs history verification, deciding whether the history satisfies the specified isolation level. If isolation violations are detected, some checkers offer counterexamples (④).

Following common practice in black-box isolation checking [5], [4], [9], [6], [10], we assume that every write operation assigns a unique value to each object. This guarantees that each read can be uniquely linked to the transaction responsible for the corresponding write. In practice, uniqueness can be easily enforced by combining a client identifier with a local counter for each write.

B. Objects and Operations

We consider a key-value database that manages a set of objects $\mathsf{X} = \{x, y, \dots\}$ associated with values from a set V. Clients interact with the database by issuing read and write operations on the objects, grouped into transactions. We denote by Op the possible operation invocations in database executions: $\mathsf{Op} = \{\mathsf{R}(x, v), \mathsf{W}(x, v) \mid x \in \mathsf{X}, v \in \mathsf{V}\}$, where $\mathsf{R}(x, v)$ represents a read operation that reads value v from object x, and $\mathsf{W}(x, v)$ represents a write operation that writes value v to object x.

C. Relations and Orderings

A binary relation R on a set A is a subset of $A \times A$. Let $I_A \triangleq \{(a, a) \mid a \in A\}$ be the identity relation on A. For $a, b \in A$, we use $(a, b) \in R$ and $a \xrightarrow{R} b$ interchangeably. The inverse R^{-1} of a relation $R \subseteq A \times A$ is defined as $R^{-1} \triangleq \{(b, a) \mid (a, b) \in R\}$. The reflexive closure $R?$ of a relation $R \subseteq A \times A$ is defined as $R? \triangleq R \cup I_A$. A relation R on a set A is transitive if $\forall a, b, c \in A.\ a \xrightarrow{R} b \wedge b \xrightarrow{R} c \implies a \xrightarrow{R} c$. The transitive closure R^+ of a relation $R \subseteq A \times A$ is the smallest relation on A that contains R and is transitive. A relation $R \subseteq A \times A$ is *acyclic* if $R^+ \cap I_A = \emptyset$. Given two binary relations $R, S \subseteq A \times A$, we define their composition as $R\ ;\ S \triangleq \{(a, c) \mid \exists b \in A.\ a \xrightarrow{R} b \wedge b \xrightarrow{S} c\}$. Note that $R\ ;\ S? = R\ ;\ (S \cup I_A) = (R\ ;\ S) \cup R$. A strict partial order is an irreflexive and transitive relation. A strict total order is a relation that is a strict partial order and total. We write _ for a component that is irrelevant and implicitly existentially quantified. We use $\exists!$ to denote "unique existence." R1-O6

D. Histories and Dependency Graphs

Definition 1 (Transaction)**.** *A transaction T is a pair (O, po), where O is a set of operations and $\mathsf{po} \subseteq O \times O$ is a strict total order on O, called the program order.*

For a transaction T, we let $T \vdash \mathsf{W}(x, v)$ if T writes to x and the last value written is v, and $T \vdash \mathsf{R}(x, v)$ if T reads from x before writing to it and v is the value returned by the first such read. We also use $\mathsf{WriteTx}_x$ to denote the set of transactions that write to x.

Transactions issued by clients are grouped into *sessions*, where a session is a sequence of transactions. We use a *history* to record the client-visible results of such interactions.

Definition 2 (History)**.** *A history is a triple $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{RT})$, where $\mathcal{T}$ is a set of transactions, $\mathsf{SO} \subseteq \mathcal{T} \times \mathcal{T}$ is the session order on $\mathcal{T}$, and $\mathsf{RT} \subseteq \mathcal{T} \times \mathcal{T}$ is the real-time order on $\mathcal{T}$ such that $\mathsf{SO} \subseteq \mathsf{RT}$ and $T_1 \xrightarrow{\mathsf{RT}} T_2$ if and only if T_1 finishes before T_2 starts in $\mathcal{H}$.*

The real-time order RT in the history is necessary for defining SSER, which requires a transaction to observe the effects of all transactions that finish before it starts [26]. We only consider histories that satisfy the *internal consistency axiom*, denoted INT, which ensures that, within a transaction, a read from an object returns the same value as the last write to or read from this object in the transaction. Additionally, we assume that every history contains a special transaction $\bot_T$ which initializes all objects [13], [5], [27], unless explicitly specified otherwise. This transaction precedes all the other transactions in the session order.

A *dependency graph* extends a history with three relations, i.e., WR, WW, and RW, representing three kinds of dependencies between transactions in this history [12], [13]. The WR relation associates a transaction that reads some value with the one that writes this value. The WW relation stipulates a strict total order among the transactions on the same object. The RW relation is derived from WR and WW, relating a

transaction that reads some value to the one that overwrites this value, in terms of the WW relation.

Definition 3 (Dependency Graphs)**.** *A dependency graph is a tuple* $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{RT}, \mathsf{WR}, \mathsf{WW}, \mathsf{WR})$ *where* $(\mathcal{T}, \mathsf{SO}, \mathsf{RT})$ *is a history and*

- $\mathsf{WR} : \mathsf{X} \to 2^{\mathcal{T}\times\mathcal{T}}$ *is such that*
  - $\forall x.\ \forall S \in \mathcal{T}.\ S \vdash \mathsf{R}(x, \_) \implies \exists!\ T \in \mathcal{T}.\ T \xrightarrow{\mathsf{WR}(x)} S.$
  - $\forall x.\ \forall T, S \in \mathcal{T}.\ T \xrightarrow{\mathsf{WR}(x)} S \implies T \neq S \wedge \exists v \in \mathsf{V}.\ T \vdash \mathsf{W}(x, v) \wedge S \vdash \mathsf{R}(x, v).$
- $\mathsf{WW} : \mathsf{X} \to 2^{\mathcal{T}\times\mathcal{T}}$ *is such that for every* $x \in \mathsf{X}$, $\mathsf{WW}(x)$ *is a strict total order on the set* $\mathsf{WriteTx}_x$*;*
- $\mathsf{RW} : \mathsf{X} \to 2^{\mathcal{T}\times\mathcal{T}}$ *is such that* $\forall x.\ \forall T, S \in \mathcal{T}.\ T \xrightarrow{\mathsf{RW}(x)} S \iff T \neq S \wedge \exists T' \in \mathcal{T}.\ T' \xrightarrow{\mathsf{WR}(x)} T \wedge T' \xrightarrow{\mathsf{WW}(x)} S.$

We denote a component of $\mathcal{G}$, such as WW, by $\mathsf{WW}_\mathcal{G}$.

E. Characterizing Strong Isolation Levels

We review how the SSER, SER, and SI isolation levels can be characterized by dependency graphs that are acyclic or contain only specific cycles. Specifically, a history $\mathcal{H}$ satisfies SSER, denoted $\mathcal{H} \models \text{SSER}$, if and only if it satisfies INT and one of its dependency graphs is acyclic.

Definition 4 (Strict Serializability [12], [28])**.** *For a history* $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{RT})$*,*

$$\mathcal{H} \models \text{SSER} \iff \mathcal{H} \models \text{INT} \wedge \exists \mathsf{WR}, \mathsf{WW}, \mathsf{RW}.\ \mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \wedge (\mathsf{RT}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G} \cup \mathsf{RW}_\mathcal{G} \ \textit{is acyclic}).$$

Being weaker than SSER, SER does not require the preservation of the real-time order RT but only the session order SO.

Definition 5 (Serializability [29])**.** *For a history* $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{RT})$*,*

$$\mathcal{H} \models \text{SER} \iff \mathcal{H} \models \text{INT} \wedge \exists \mathsf{WR}, \mathsf{WW}, \mathsf{RW}.\ \mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \wedge (\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G} \cup \mathsf{RW}_\mathcal{G} \ \textit{is acyclic}).$$

SI is even weaker than SER: a history satisifes SI if and only if it satisfies INT and one of its dependency graph (without the RT dependency edges) is acyclic or contains only cycles with at least two adjacent RW edges. Formally,

Definition 6 (Snapshot Isolation [13])**.** *For a history* $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{RT})$*,*

$$\mathcal{H} \models \text{SI} \iff \mathcal{H} \models \text{INT} \wedge \exists \mathsf{WR}, \mathsf{WW}, \mathsf{RW}.\ \mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \wedge (((\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G})\ ;\ \mathsf{RW}_\mathcal{G}?) \ \textit{is acyclic}).$$

Example 1. *Consider the history* $\mathcal{H}$ *of Figure 3, where* T_2 *and* T_3 *read the same value of* x *from* T_1 *and then write different values to* x*. The* WR *and* WW *dependency edges from* T_1 *to* T_2 *and* T_3 *respectively must be in any of the dependency graphs of* $\mathcal{H}$*. There are two cases regarding the* WW *dependency edge between* T_2 *and* T_3*. Suppose that* $T_2 \xrightarrow{\mathsf{WW}(x)} T_3$*. Since* $T_1 \xrightarrow{\mathsf{WW}(x)} T_2$ *and* $T_1 \xrightarrow{\mathsf{WR}(x)} T_3$*, we have* $T_3 \xrightarrow{\mathsf{RW}(x)} T_2$*. Therefore, the corresponding dependency graph contains a cycle* $T_2 \xrightarrow{\mathsf{WW}(x)} T_3 \xrightarrow{\mathsf{RW}(x)} T_2$*. By Definition 6,* $\mathcal{H}$ *does not satisfy* SI*. The case for* $T_3 \xrightarrow{\mathsf{WW}(x)} T_2$ *is similar. Actually, the history* $\mathcal{H}$ *contains a* LOSTUPDATE *anomaly.*

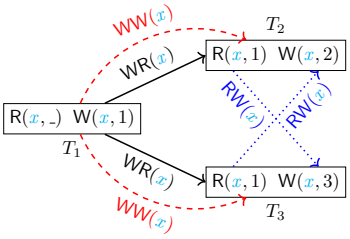


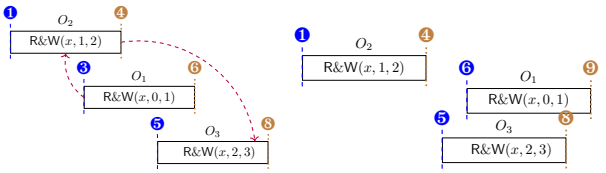
Fig. 3: The DIVERGENCE pattern: T_2 and T_3 read the same value of x from T_1 and write different values.

F. Linearizability

When each transaction comprises only one operation, the SSER isolation level is the same as the well-known LINEARIZABILITY (LIN) condition [18] for concurrent objects in shared-memory multiprocessors and distributed systems [30], [14]. We use this connection when studying the verification algorithm for lightweight-transaction histories in Section IV-E.

In the context of concurrent objects, in addition to read and write operations, we also consider the *read&write* operations [31]. A read&write operation, denoted $\mathsf{R\&W}(x, v, v')$, reads the value v from object x and then writes v' to x. From the perspective of executions, a read&write operation can be regarded as a transaction that contains a read operation followed by a write operation on the same object. From the perspective of clients, a read&write operation invocation may correspond to a lightweight transaction (also known as Compare-And-Set (CAS) operation [32]) that reads the current value of an object x and, if the value is equal to a given expected value v, writes a new value v' to the object. If the value is not equal to v, the lightweight transaction is equivalent to a simple read operation. Another special case of lightweight transaction is the *insert-if-not-exists* operation [33], which inserts an object with a value only if the object does not exist. From the perspective of executions, a successful insert-if-not-exists operation is equivalent to a simple write operation. Lightweight transactions have been widely adopted in DBMSs like Apache Cassandra [33], ScyllaDB [34], PNUTS [35], Azure Cosmos DB [36], and etcd [37].

When the start time s and finish time f of an operation is concerned, we denote by $\mathsf{R}(s, f, x, v)$, $\mathsf{W}(s, f, x, v)$, and $\mathsf{R\&W}(s, f, x, v, v')$, correspondingly. We use $o.s$ and $o.f$ to denote the start and finish time of an operation o, respectively. If an operation o_1 finishes before another operation o_2 starts, then we say that o_1 precedes o_2 in the real-time order. LIN requires that all operations appear to be executed in some sequential order that is consistent with the real-time order.



(a) A linearizable history. (b) A non-linearizable history.

Fig. 4: Illustration of linearizability on histories of R&W operations. The initial value of the object x is 0.

Definition 7 (Linearizability [18])**.** *A history $\mathcal{H}$ is linearizable if and only if there exists a permutation Π of all the operations in $\mathcal{H}$ such that Π preserves the real-time order of operations and follows the sequential semantics of each object.*

Example 2. *Figure 4a presents both a linearizable and a non-linearizable history of R&W operations. The history of Figure 4a is linearizable as witnessed by the operation sequence consisting of O_1, O_2, and O_3 in this order. In contrast, the history of Figure 4b is non-linearizable since $O_1 : \text{R\&W}(x, 0, 1)$ starts after $O_2 : \text{R\&W}(x, 1, 2)$ finishes.*

III. Mini-Transactions

In this section, we provide a precise definition of mini-transactions and demonstrate that, despite their conceptual simplicity, they can capture common data anomalies.

A. Defining Mini-Transactions

A mini-transaction is compact and adheres to the RMW pattern.

Definition 8 (Mini-transaction; MT)**.** *A mini-transaction is a transaction meeting the following two criteria:*

- *It contains one or two read operations and at most two write operations.*
- *Each write operation, if present, is (not necessarily immediately) preceded by a read operation on the same object.*

R1-M1

Definition 9 (Mini-Transaction History)**.** *A mini-transaction history is a history comprising solely mini-transactions (except the initial transaction $\perp_T$) such that each write operation on the same object assigns a unique value.*

The first criterion of mini-transactions helps expedite database execution during history generation, while the second criterion, along with the unique values written, guarantees the existence of efficient verification algorithms for mini-transaction histories. Note that according to this definition, a read&write operation can be regarded as a mini-transaction.

B. Capturing Isolation Anomalies

We demonstrate that mini-transactions are expressive enough to characterize all common isolation anomalies. As there is no standard for formally defining all kinds of isolation anomalies, we refer to the state-of-the-art specification frameworks for transactional isolation levels [12], [19], [13], [5], [6], which, to some extent, have been shown to be equivalent.

Figure 5 illustrates 14 common isolation anomalies specified in the frameworks above, where each anomaly is represented by a mini-transaction history. They cover both the intra-transactional anomalies (i.e., Figure 5c-5g) specified by [12], [6] and the inter-transactional anomalies (i.e., Figure 5h-5n) emphasized by [19], [13], [5]. As demonstrated in Figure 5, a maximum of four operations per transaction is *necessary and sufficient* for mini-transaction histories to capture these 14 isolation anomalies. Increasing the number of operations per transaction may incur a performance penalty during both history generation and verification. Table I lists these anomalies and their descriptions, as well as the references to the frameworks that define them. We believe this list is exhaustive with respect to the aforementioned specification frameworks.

R1-O4

IV. Efficient History Verification Algorithms

Despite the NP-hardness of verifying strong isolation levels for general histories [17], [5], we show that verifying whether an MT history of n MTs satisfies SSER, SER, or SI can be remarkably efficient in $O(n^2)$, $O(n)$, and $O(n)$ time, respectively, thanks to the RMW patterns in MTs and the unique value conditions in MT histories. Our verification algorithms are both sound and complete for a given MT history, i.e., producing no false positives or negatives. Furthermore, in [38, Appendix C], we establish that verifying strong isolation levels for MT histories *without* unique values is NP-hard.

R3-D4

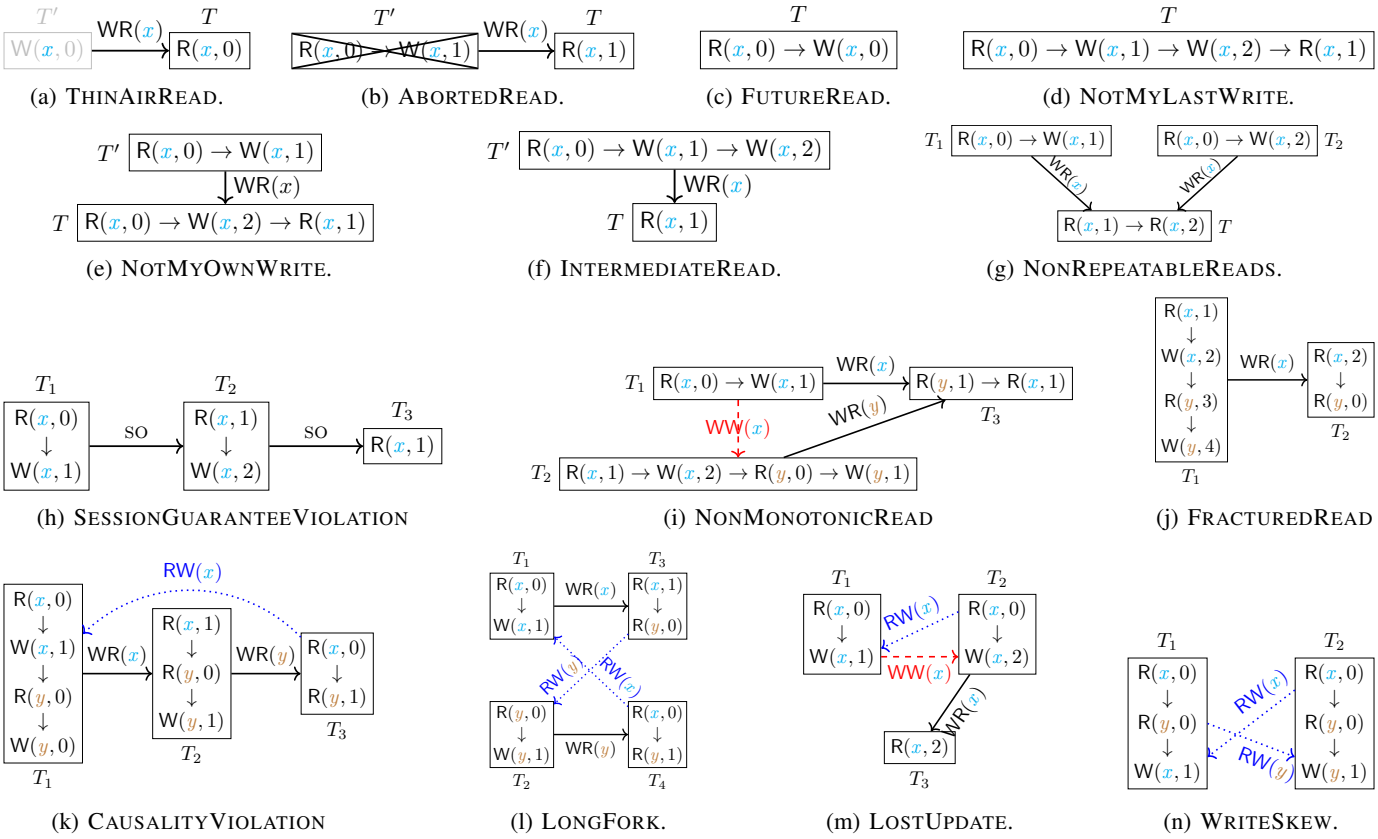
A. Our Insight

The efficiency of our verification algorithms stems from a key insight: *the dependency graph of a mini-transaction history is (nearly) unique and can be constructed efficiently*. First, due to the unique values written, the WR dependency in a mini-transaction history is entirely determined. Furthermore, since each write operation is preceded by a read operation on the same object in mini-transactions, the WW dependency is *almost* determined. *Exceptions* only arise when multiple transactions read the same value of an object and subsequently write different values to it. Finally, by definition, the RW dependency is entirely determined by the WR and WW dependencies. Hence, the challenge for the verification algorithms lies in correctly handling the *exceptions* in the WW dependency for SSER, SER, and SI, respectively.

We define the exceptions in the WW dependency as the following Divergence pattern, which, as explained in Example 1, violates SI.

Definition 10 (Divergence Pattern)**.** *A history $\mathcal{H}$ contains an instance of the* Divergence *pattern if there exist three transactions T_1, T_2, and T_3 in $\mathcal{H}$ such that T_2 and T_3 read the same value of an object x from T_1 and then write different values to x.*

Lemma 1. *If a history contains an instance of the* Divergence *pattern, then it does not satisfy* SI.



(a) THINAIRREAD. (b) ABORTEDREAD. (c) FUTUREREAD. (d) NOTMYLASTWRITE.

(e) NOTMYOWNWRITE. (f) INTERMEDIATEREAD. (g) NONREPEATABLEREADS.

(h) SESSIONGUARANTEEVIOLATION (i) NONMONOTONICREAD (j) FRACTUREDREAD

(k) CAUSALITYVIOLATION (l) LONGFORK. (m) LOSTUPDATE. (n) WRITESKEW.

Fig. 5: R1-M2 Capturing 14 isolation anomalies in contemporary specification frameworks [12], [19], [13], [5], [6] by MTs.

TABLE I: Common isolation anomalies captured by MTs (corresponding to Figure 5).

	Anomaly	References	Description
(a)	THINAIRREAD	[6]	A transaction reads a value out of thin air.
(b)	ABORTEDREAD	[12, Phenomenon G1a] [6]	A transaction reads a value from an aborted transaction.
(c)	FUTUREREAD	[6]	A transaction reads from a write that occurs later in the same transaction.
(d)	NOTMYLASTWRITE	[6]	A transaction reads from its own but not the last write on the same object.
(e)	NOTMYOWNWRITE	[6]	A transaction does not read from its own write on the same object.
(f)	INTERMEDIATEREAD	[12, Phenomenon G1b] [6]	A transaction reads a value that was later overwritten by the transaction that wrote it.
(g)	NONREPEATABLEREADS	[5, Figure 3b]	A transaction reads multiple times from the same object but receives different values.
(h)	SESSIONGUARANTEEVIOLATION	[13, Figure 2a] [5, Figure 3c]	Transaction T_3 misses the effect of the preceding transaction T_2 in the same session.
(i)	NONMONOTONICREAD	[5, Figure 3a]	Transaction T_3 reads y from T_2 and *then* reads x from T_1, but T_2 has overwritten T_1 on x.
(j)	FRACTUREDREAD	[13, Figure 2c] [5, Figure 3d]	Transaction T_1 updates both x and y, but T_2 observes only the update to x.
(k)	CAUSALITYVIOLATION	[13, Figure 2d] [5, Figure 3e]	Transaction T_3 sees the effect of T_2 on y, but misses the effect of T_1, which is seen by T_2, on x.
(l)	LONGFORK	[13, Figure 2e] [5, Figure 3f]	Concurrent transactions T_1 and T_2 write to x and y, respectively. Transaction T_3 observes the write of T_1 to x but misses the write of T_2 to y, while T_4 observes the write of T_2 to y but misses the write of T_1 to x.
(m)	LOSTUPDATE	[13, Figure 2b] [5, Figure 3g]	Concurrent transactions T_1 and T_2 write to the same object, resulting in one of the writes getting lost.
(n)	WRITESKEW	[13, Figure 2f] [5, Figure 3h]	Concurrent transactions T_1 and T_2 read from both x and y, and then write to x and y, respectively.

B. Algorithms

The three verification algorithms for SSER, SER, and SI respectively follow the same structure, as outlined in Algorithm 1.[1] They all construct the (partial) dependency graph $\mathcal{G}$ of the input history $\mathcal{H}$ based on the dependency relations RT, SO, WR, WW, and RW by calling BUILDDEPENDENCY. In BUILDDEPENDENCY, the RT dependency edges are added for SSER only (line 5 of BUILDDEPENDENCY and line 2 of CHECKSSER). Due to unique values written, the WR dependency edges are entirely determined (line 9 of BUILDDEPENDENCY). Note that for mini-transaction histories, the WW dependency edges are inferred from the WR dependency (lines 10-11 of BUILDDEPENDENCY). To make $\mathsf{WW}(x)$ transitive for each x, BUILDDEPENDENCY then computes the transitive closure of $\mathsf{WW}(x)$ edges. This step is convenient for the correctness proof and can be optimized away (refer to Section IV-C). Finally, the RW dependency edges are added based on the WR and WW dependencies (lines 14-15 of BUILDDEPENDENCY).

Subsequently, we check whether the partial dependency graph $\mathcal{G}$ is acyclic for SSER (line 3 of CHECKSSER) and SER (line 3 of CHECKSER). For the SI verification algorithm, we check whether the induced graph $\mathcal{G}' \leftarrow (V_\mathcal{G}, (\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_G)\ ;\ \mathsf{RW}_\mathcal{G}?)$ of $\mathcal{G}$ is acyclic (line 6 of CHECKSI). Crucially, before checking the acyclicity of $\mathcal{G}'$, we examine whether there exists a triple of transactions T, S, and S' such that S and S' read the same value of some object from T and then write different values to this object (line 2 of CHECKSI). If this is the case, CHECKSI immediately return false.

Example 3. *Consider the history $\mathcal{H}$ depicted in Figure 3, which, according to Lemma 1, violates* SI *and* SER. *Figure 3 also illustrates the graph $\mathcal{G}$ generated by* CHECKSER *and* CHECKSI. *Since $\mathcal{G}$ forms a cycle $T_2 \xrightarrow{\mathsf{RW}(x)} T_3 \xrightarrow{\mathsf{RW}(x)} T_2$,* CHECKSER *returns* false *at line 3, as expected. However, such a cycle is not forbidden by* SI. *Put differently, if we examine the induced graph $\mathcal{G}'$ of $\mathcal{G}$ (line 5 of* CHECKSI*), we would find it to be acyclic.* CHECKSI *effectively pinpoints the* SI *violation by detecting the* DIVERGENCE *pattern early at line 3 in* CHECKSI.

The correctness proofs of these three verification algorithms are surprisingly intricate and are provided in [38, Appendix A].

C. Optimizations

In this section, we show that the step of computing the transitive closure of the WW edges at lines 12–13 of BUILDDEPENDENCY can be optimized away (the proof is given in [38, Appendix B]). Let $\widehat{\mathcal{G}}$ be the graph constructed by BUILDDEPENDENCY *without* computing the transitive closure of the WW edges. As illustrated in Figure 6, $\widehat{\mathcal{G}}$ may lack some $\widehat{\mathsf{WW}}$ edges that can be derived from a sequence of WW edges by transitivity and $\widehat{\mathsf{RW}}$ edges that are derived from $\widehat{\mathsf{WW}}$ edges and WR edges.

[1]We assume that the input history $\mathcal{H}$ satisfies the INT axiom (see Section II-D). In practice, we first check whether $\mathcal{H}$ contains THINAIRREAD, ABORTEDREAD, or any intra-transactional anomalies (see Section III-B).

Algorithm 1 The algorithms for verifying SSER, SER, and SI on mini-transaction histories.

```
1: procedure BUILDDEPENDENCY(H, rt)
2:     G ← (T, ∅)                          ▷ initialize the dependency graph G
3:     if rt then                          ▷ add RT edges for CHECKSSER only
4:         for all T, S ∈ T such that T --RT--> S do
5:             E_G ← E_G ∪ {(T, S, RT)}
6:     for all T, S ∈ T such that T --SO--> S do
7:         E_G ← E_G ∪ {(T, S, SO)}
8:     for all x ∈ X. T, S ∈ T such that T --WR(x)--> S do
9:         E_G ← E_G ∪ {(T, S, WR(x))}
10:        if W(x, _) ∈ S then             ▷ determine WW based on WR
11:            E_G ← E_G ∪ {(T, S, WW(x))}
12:    for all x ∈ X do ▷ Convenient for the correctness proof and can be optimized away (see Section IV-C).
13:        compute the transitive closure of WW(x) edges in E_G
14:    for all x ∈ X. T', T, S ∈ T such that (T', T, WR(x)) ∈ E_G ∧ (T', S, WW(x)) ∈ E_G do
15:        E_G ← E_G ∪ {(T, S, RW(x))}
16:    return G

1: procedure CHECKSSER(H)
2:     G ← BUILDDEPENDENCY(H, ⊤)
3:     return ACYCLIC(G)

1: procedure CHECKSER(H)
2:     G ← BUILDDEPENDENCY(H, ⊥)
3:     return ACYCLIC(G)

1: procedure CHECKSI(H)
2:     if ∃x ∈ X. ∃v, v' ≠ v ∈ V. ∃T, S, S' ≠ S ∈ T. (T --WR(x)--> S ∧ T --WR(x)--> S') ∧ (S ⊢ W(x, v) ∧ S' ⊢ W(x, v')) then
3:         return false                    ▷ the DIVERGENCE pattern (Figure 3)
4:     G ← BUILDDEPENDENCY(H, ⊥)
5:     G' ← (V_G, (SO_G ∪ WR_G ∪ WW_G) ; RW_G?)
6:     return ACYCLIC(G')
```

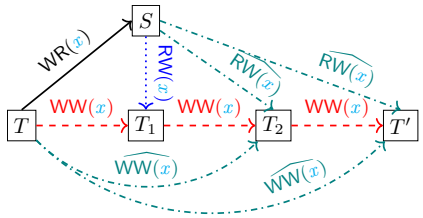


Fig. 6: Illustrating the missing $\widehat{\mathsf{WW}}$ and $\widehat{\mathsf{RW}}$ edges in $\widehat{\mathcal{G}}$.

Lemma 2. *For each $S \xrightarrow{\widehat{\mathsf{RW}}} T'$ edge in $\mathcal{G}$, there exist a set of transactions $T_1, T_2, \ldots, T_n$ such that $S \xrightarrow{\mathsf{RW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T_n \xrightarrow{\mathsf{WW}} T'$ in $\mathcal{G}$.*

The key observation is that any cycle in $\mathcal{G}$ can be transformed into a cycle in $\widehat{\mathcal{G}}$ by replacing the $\widehat{\mathsf{WW}}$ and $\widehat{\mathsf{RW}}$ edges in the cycle with WW and RW edges in $\widehat{\mathcal{G}}$, respectively.

Therefore, we have the following theorem, from which the correctness of the optimized versions of CHECKSSER and CHECKSER follows directly.

Theorem 1. *$\mathcal{G}$ is acyclic if and only if $\widehat{\mathcal{G}}$ is acyclic.*

Let $\widehat{\mathcal{G}}'$ be the induced graph of $\widehat{\mathcal{G}}$, i.e., $\widehat{\mathcal{G}}' \leftarrow (V_{\widehat{\mathcal{G}}}, (\mathsf{SO}_{\widehat{\mathcal{G}}} \cup \mathsf{WR}_{\widehat{\mathcal{G}}} \cup \mathsf{WW}_{\widehat{\mathcal{G}}})\ ;\ \mathsf{RW}_{\widehat{\mathcal{G}}}?)$. The correctness of the optimized version of CHECKSI follows from the following theorem.

Theorem 2. *$\mathcal{G}'$ is acyclic if and only if $\widehat{\mathcal{G}}'$ is acyclic.*

D. Time Complexity

Suppose that the input history comprises n transactions, and the dependency graph $\mathcal{G}$ returned by the (optimized) BUILDDEPENDENCY procedure contains m edges. Given that a mini-transaction may have at most n incoming RT edges, one incoming SO edges (note that, similar to the optimization in Section IV-C, the SO edges derived by transitivity can be optimized away), two incoming WR edges, two incoming WW edges, and two incoming RW edges, we can deduce that $m = O(n)$. Therefore, the time complexity of both the (optimized) CHECKSER and CHECKSI procedures is $\Theta(n)$. Moreover, since establishing the RT edges requires $\Theta(n^2)$ time, the time complexity of CHECKSSER is $\Theta(n^2)$.

E. On SSER and Lightweight-transaction Histories

In this section, we show that for lightweight-transaction histories with unique values, we can check SSER more efficiently in (expected) linear time. In such histories, each transaction is either a *read&write* operation or an *insert-if-not-exist* operation. In lightweight-transaction histories, we do *not* assume the existence of an initial transaction $\bot_T$. Instead, insert-if-not-exists operations are utilized to insert objects with initial values. It is important to note that in this context, SSER degenerates to linearizability.

1) Algorithm: Since linearizability is a local property [18], meaning that a history $\mathcal{H}$ is linearizable if and only if every sub-history restricted to individual object is linearizable [18], Algorithm 2 takes as an input a history $\mathcal{H}_x$ of lightweight transactions on a single object x. The history $\mathcal{H}_x$ is valid only if it contains exactly one insert-if-not-exists operation on x (line 2) and the transactions in it can be organized into a chain in which each transaction (i.e., an R&W operation) reads the value written by the previous transaction in the chain. In the first step (❶), Algorithm 2 constructs this chain if possible (lines 4–10). Then, in the second step (❷), Algorithm 2 checks the real-time requirement: for each transaction, it *cannot* start after any of the following transactions in the chain commits.

2) Efficiency: Consider a history $\mathcal{H}_x$ of n lightweight transactions. The next transaction t in the chain (line 7) can be found in (expected) $O(1)$ time by using a hash table. Thus, step ❶ takes $O(n)$ time. It is easy to see that step ❷ takes $O(n)$ time. Therefore, Algorithm 2 takes $O(n)$ time in total.

Algorithm 2 The algorithm for verifying linearizability on lightweight-transaction (LWT) histories.

```
1:  procedure VL-LWT(H_x)                          ▷ H_x is non-empty
2:      if |WriteTx_x| ≠ 1 then
3:          return false                  ▷ exactly one insert-if-not-exists
        ❶ construct the transaction chain if possible
4:      v ← the value of the only write operation W(x, v)
5:      chain ← ⟨⟩                        ▷ initialize the chain to be empty
6:      for T ≠ ∅ do
7:          if ∃v' ∈ V. ∃! t ∈ T. t = R&W(_, _, x, v, v') then
8:              T ← T \ {t}
9:              chain ← chain ∘ t          ▷ append t to the chain
10:             v ← v'
11:         else
12:             return false
        ❷ check the real-time requirement in linear time
13:     min_f ← ∞
14:     for T ∈ chain in reverse order do
15:         if T.s > min_f then
16:             return false
17:         min_f ← min{min_f, T.f}
18:     return true
```

V. EXPERIMENTS

We have implemented our verification algorithms for MT histories in a tool called MTC, which incorporates three verification components for SSER, SER, and SI, as well as a MT workload generator. The implementation consists of a total of 800 lines of code in Java and 300 lines of code in Go.

This section presents a comprehensive evaluation of MTC and state-of-the-art black-box isolation checkers that test databases under highly concurrent GT workloads. We aim to address the following questions:

- Q1: How efficient are MTC's verification components, i.e., MTC-SSER, MTC-SER, and MTC-SI, compared to other checkers across different concurrency levels?
- Q2: How does MTC perform in terms of time and memory in end-to-end checking when history generation is included? R4-O3
- Q3: Is MTC's MT workload generator more effective, resulting in a higher success rate for committing transactions?
- Q4: Can MTC effectively detect isolation violations in production databases?

A. Workloads, Histories, and Setup

1) Workloads: We generate two types of workloads: MT workloads and GT workloads. MT workloads are used to compare MTC with existing checkers, focusing on the efficiency of their verification components (Q1). GT workloads, commonly employed by state-of-the-art checkers, are used to evaluate end-to-end checking overhead, highlighting the impact of MTs on the overall checking process (Q2). Moreover, we assess the effectiveness of workload generators by comparing their success rates in achieving committed transactions (Q3). Finally, we evaluate the effectiveness of MTC in detecting isolation violations in PostgreSQL and MongoDB (Q4).

Both workload generators are parametric and designed to assess checker performance at various concurrency levels. The

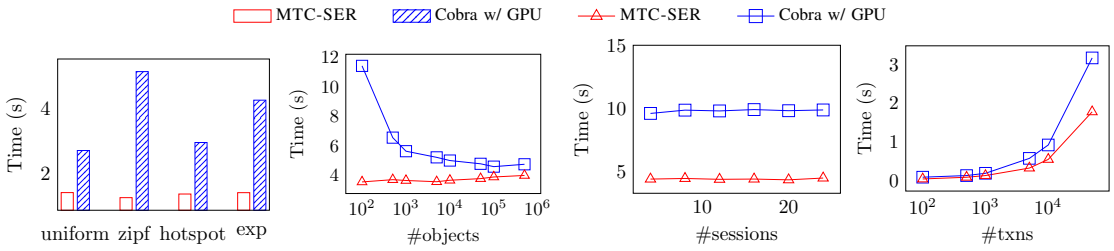


Fig. 7: Performance comparison for verifying SER histories.

MT workload generator parameters include the number of sessions (#sessions), transactions (#txns), objects (#objects), and the object-access distribution (i.e., uniform, zipfian, hotspot, and exponential), which defines workload skewness. For the GT workload generator, we use Cobra's implementation [9], which allows configuration of #objects, #txns, and #ops/txn (operations per transaction). Each GT workload consists of 20% read-only transactions, 40% write-only transactions, and 40% RMW transactions. During workload generation, transactions are uniformly distributed across sessions, and unique values are assigned to each object using counters.

2) Histories: We use a PostgreSQL (v14.7) instance to generate *valid* histories without isolation violations. Specifically, for SI, we set the isolation level to REPEATABLE READ (implemented as SI in PostgreSQL); for SER, we set the level to SERIALIZABILITY to generate serializable histories. To facilitate the comparison of SSER checkers at varying concurrency levels, we implement a parametric, synthetic history generator tailored for lightweight transactions.[2] The generator's parameters include #sessions, #txns/session (number of transactions per session), and concurrent sessions (the percentage of sessions issuing transactions concurrently).

3) Experimental Setup: We co-locate client/sessions and PostgreSQL on a local machine. Each session issues a stream of transactions generated by the workload generator to the database and logs its execution history. The histories from all sessions are then combined and saved to a file, which is used to benchmark the performance of the checkers. All experiments are conducted on a system with a 12-core CPU, 64GB of memory, and an NVIDIA P4 GPU.

B. State-of-the-Art Isolation Checkers

We consider the following checkers for comparison.

- **SER**: Cobra [9] is an efficient SER checker that utilizes the advanced MonoSAT solver [39]. We also evaluate its GPU-accelerated version, serving as a strong baseline.
- **SI**: PolySI [4] is an efficient SI checker that also relies on the MonoSAT solver.
- **SSER**: Porcupine [11] is a state-of-the-art LIN checker that does not rely on off-the-shelf solvers. It utilizes P-compositionality [40] to further improve performance.

R3-O1

Both Cobra and PolySI transform the verification problem into a constraint-solving task, by constructing a polygraph from the history, where uncertainties in transaction execution order are represented as constraints. They prune constraints through domain-specific optimizations, encode the polygraph into SAT formulas, and use the MonoSAT solver to detect SER/SI-specific cycles. While effective for general workloads, we will shortly show that this approach is less efficient than our tailored verification algorithms on MT histories.

C. Q1: History Verification Performance

1) Serializability and Snapshot Isolation: The first two sets of experiments compare MTC with state-of-the-art SER and SI checkers on MT histories. The experimental results are presented in Figures 7 and 8.

MTC-SER consistently outperforms Cobra's GPU-accelerated version across various concurrency levels. For MT histories, Cobra exhibits similar overhead without GPU acceleration. Under highly skewed object access patterns (implying high concurrency), such as the zipfian distribution, MTC-SER achieves approximately 5x better performance than Cobra (Figure 7a). Additionally, MTC-SER maintains stable performance with respect to skewness, while Cobra's verification time increases exponentially with fewer objects (implying more skewed access patterns); see Figure 7b. Both checkers maintain stable performance with respect to the number of sessions, yet MTC-SER consistently achieves around 2x better performance than Cobra (Figure 7c). Furthermore, as the number of transactions increases, Cobra's verification time escalates significantly faster (Figure 7d).

Compared to PolySI in verifying SI, MTC-SI demonstrates significantly greater performance gains across various workloads. For example, under the zipfian object-access distribution, MTC-SI achieves approximately 1600x faster verification (Figure 8a). With an increasing number of transactions, MTC-SI reduces verification time by up to 93x (Figure 8d).

2) Strict Serializability: To benchmark the efficiency of SSER checkers, we generate valid, synthetic SSER histories. As shown in Figure 9, MTC-SSER outperforms Porcupine across various concurrency levels. Under extreme concurrency

[2]We observe that, for databases supporting lightweight transactions, e.g., Apache Cassandra, adjusting the workload parameters, as in our black-box setting, cannot predictably control the concurrency level in generated histories.

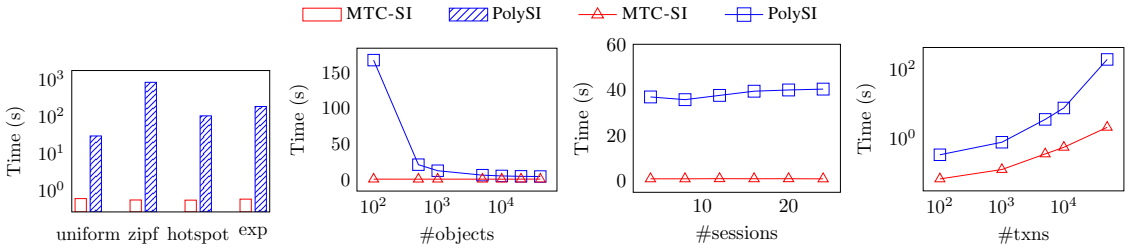


Fig. 8: Performance comparison for verifying SI histories.

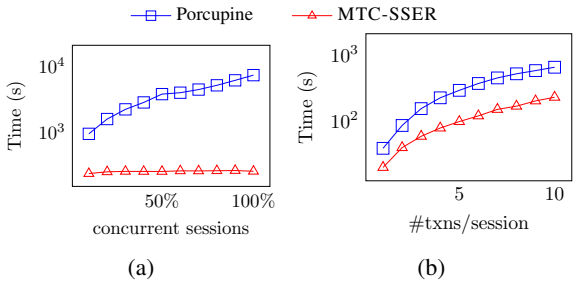


Fig. 9: Performance comparison for verifying SSER histories.

where all clients execute transactions simultaneously, MTC-SSER demonstrates a substantial 28x improvement in verification. Furthermore, MTC-SSER maintains stable performance as the number of concurrent sessions increases (Figure 9a).

D. Q2: End-to-End Checking Performance

To assess the performance improvement that MTs bring to the end-to-end isolation checking process, we compare baseline checkers using their standard GT workload generators.

As shown in Figures 10a-c, MTC-SER with MT workloads substantially and consistently outperforms Cobra with GT workloads in both history generation and verification. achieving up to two and three orders of magnitude improvement in history verification compared to Cobra with and without GPU, respectively. Figures 10a-c also show that Cobra's non-solver components, including polygraph construction, constraint pruning, and encoding, are significantly more time-consuming than the solver itself. This observation aligns with the findings in the original works on Cobra [9] and PolySI [4]. Additionally, as concurrency levels increase, characterized by more transactions, more operations per transaction, and less number of objects, MT workload generation becomes increasingly more efficient compared to Cobra. Figures 10d-f show that MTC-SER consistently consumes significantly less memory than Cobra, achieving up to 30x and 14x improvement compared to Cobra with and without GPU, respectively. We observe similar trends in both time and memory in the end-to-end isolation checking comparison between MTC-SI and PolySI; see [38, Appendix D].

R1-O3
R4-O3
R4-O3

E. Q3: Effectiveness of Workload Generation

The effectiveness of stress-testing DBMSs relies significantly on histories containing numerous committed transactions. Figure 11 presents the abort rates for executing MT and GT workloads in PostgreSQL under SER and SI. For GT workloads, we use a moderate transaction size of 20 operations (as we observed, larger transaction sizes lead to higher abort rates).

GT workloads result in substantially more aborted transactions, leading to less effective histories. As shown in Figure 11a, even with only a few client sessions, nearly half of the transactions are aborted. The abort rate increases as more sessions are involved, indicating greater stress on PostgreSQL. Moreover, the GT workload generator is sensitive to access skewness. In particular, when roughly 20 transactions access the same object, over 60% of transactions are aborted, as shown in Figure 11b. In contrast, our MT workload generator demonstrates robustness against variations in both cases.

F. Q4: Detecting Isolation Bugs

1) Rediscovering Bugs: MTC can successfully (re)discover real-world bugs across different isolation levels with MTs, as summarized in Table II and illustrated in Figure 12. These bugs appear in six releases of five databases and represent various data anomalies, e.g., the LostUpdate anomaly found in MariaDB Galera (Figure 12a) which violates the claimed SI. MTC demonstrates high efficiency in detecting these bugs, particularly with instant history verification. Figure 12b depicts the WriteSkew anomaly found in PostgreSQL, which violates the claimed SER. MTC-SER reports this cycle containing two consecutive RW edges (see also Figure 5n), where each involved MT has only two or three operations. We defer other bugs found by MTC to [38, Appendix E]. Notably, the counterexamples returned by MTC are relatively concise and easy to interpret with MTs.

2) Effectiveness: To evaluate the effectiveness of MTC in detecting isolation bugs, we compare it against randomized testing of PostgreSQL (v12.3) and MongoDB (v4.2.6) using Elle [3] under GT workloads, including list append and read-write registers, with varying transaction sizes. Every database execution is required to successfully commit 3,000 transactions. To increase concurrency, we set the number of objects

R1-O1

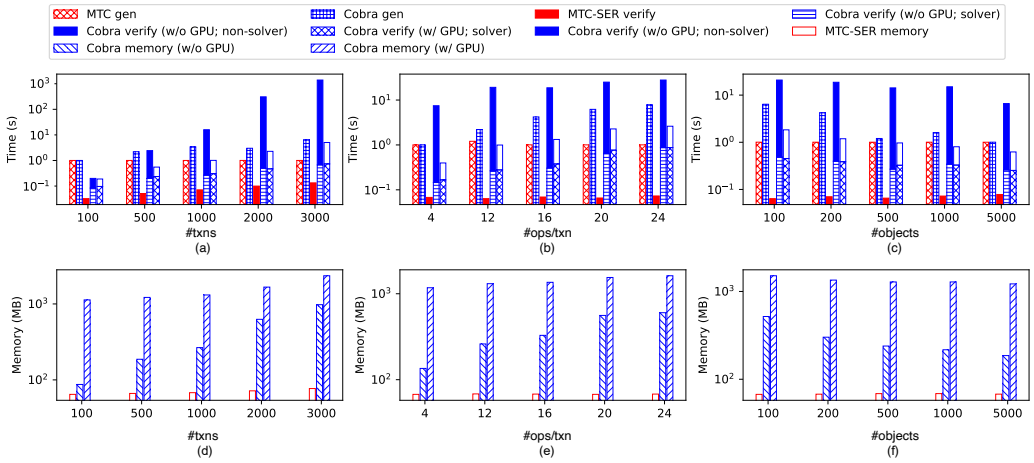


Fig. 10: End-to-end checking performance, with time decomposed into history generation and verification.

TABLE II: R3-D1 R4-O1 Summary of rediscovered isolation bugs. Counterexample (CE) position refers to the position of the first MT of the counterexample in the history. Time includes both history generation (Hist. Gen.) and verification (Hist. Verify).

Isolation Level	Anomaly	Database	Bug Report	Status	CE Position	Hist. Gen.	Hist. Verify
SI	LostUpdate	MariaDB Galera 10.7.3	[41]	Fixed	20	< 1s	< 1s
SI	AbortedRead	MongoDB 4.2.6	[42]	Fixed	0.3k	123s	< 1s
SI	CausalityViolation	Dgraph 1.1.1	[43]	Confirmed	2.5k	615s	< 1s
SER	WriteSkew	PostgreSQL 12.3	[44]	Fixed	3.5k	120s	< 1s
SER	LongFork	PostgreSQL 11.8	[8]	Fixed	24k	240s	< 2s
SSER	AbortedRead	Apache Cassandra 2.0.1	[45]	Fixed	30	60s	< 1s

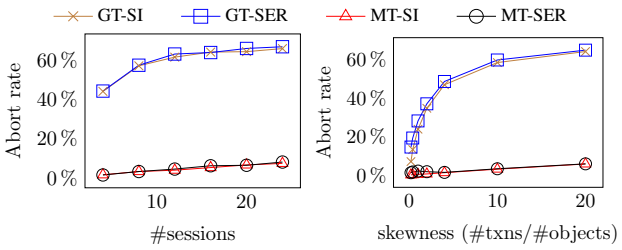
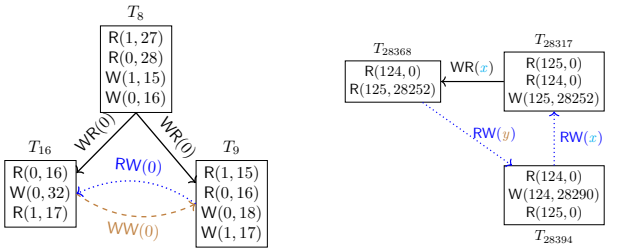
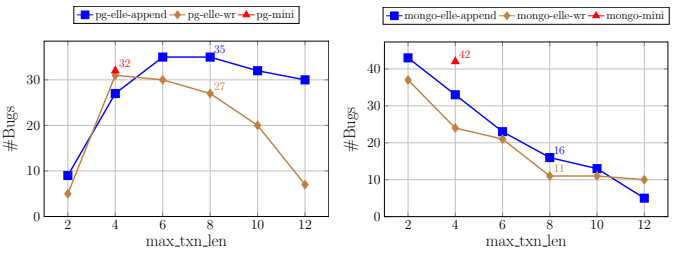


Fig. 11: Abort rates for GT and MT workloads.



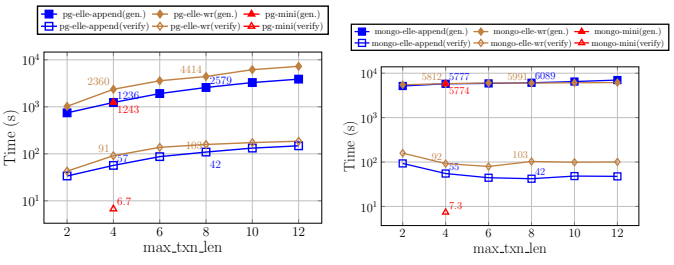
(a) MariaDB Galera: LostUpdate that violates SI (b) PostgreSQL: WriteSkew that violates SER

Fig. 12: Rediscovered isolation bugs by MTC.



(a) On PostgreSQL and SER. (b) On MongoDB and SI

Fig. 13: Effectiveness of MTC in detecting isolation bugs.



(a) On PostgreSQL and SER. (b) On MongoDB and SI

Fig. 14: End-to-end checking time for MTC and Elle.

to 10 and use an "exponential" object-access distribution. For each configuration, the experiment lasts 30 minutes. R1-O5

In Figure 13, the x-axis labeled "max_txn_len" represents the *maximum* number of operations per transaction. For MTC, it is 4. We use "pg" for PostgreSQL and "mongo" for MongoDB, and "append", "wr", and "mini" for the list append, read-write registers, and MT workloads, respectively. As shown in Figure 13a, within 30 minutes, MTC detects bugs in 32 successful trials on PostgreSQL, while Elle achieves R4-O2 R1-O1

the highest bug detection rate with 35 successful trials at a maximum transaction length of 8 operations under the list append workload. On the other hand, MTC consistently detects bugs in more successful trials than Elle under the read-write register workload. On MongoDB (Figure 13b), MTC is also highly competitive in bug detection compared to Elle.

Note that Elle's effectiveness in bug detection is highly sensitive to transaction sizes across databases, whereas a transaction size of 4 serves as a baseline. R3-O2

In this experiment, we also report the average time for history generation and verification for MTC and Elle. Figure 14 shows that MTC consistently outperforms Elle in both tasks. For example, on PostgreSQL with "max_txn_len = 8" under a read-write register workload, MTC achieves up to 3.5x and 15x speedups in history generation and verification, respectively.

VI. Related Work

We focus on the state-of-the-art black-box checkers for database isolation levels, summarizing their features, including the isolation levels they support, how they decide if a hisotry satisfies the desired isolation level, how they optimize the verification performance, and how they report bugs.

A recent advance leverages SMT solvers to check strong database isolation levels. Representative checkers include Viper [10] and PolySI [4] for SI and Cobra [9] for SER. Each checker encodes a database execution history as a *polygraph* [17] or its variants (e.g., generalized polygraphs [4] or begin and commit polygraphs [10]) and prunes constraints through domain-specific optimizations. They all invoke the off-the-shelf MonoSAT [39] solver tailored for checking graph properties like acyclicity. Our MTC also tests DBMSs with randomly generated, highly concurrent workloads. However, it substantially improves efficiency in both stages of history generation and verification via MTs. Upon detecting a bug, Cobra and Viper return unsatisfied clauses as the counterexample.

Isolation checkers not based on SMT solving employ graph traversal algorithms to identify isolation bugs in a dependency graph. The dbcop tool [5] builds on a polynomial-time algorithm for verifying SER (with a fixed number of sessions) and a polynomial-time algorithm for reducing verifying SI to verifying SER. However, it is less efficient than Cobra and PolySI [9], [4]. Moreover, when identifying a bug, dbcop only returns "false", without providing any details.

Elle [3] is an integrated isolation checker of the Jepsen testing framework [46]. It leverages Jepsen's *list-append* workloads to efficiently infer write-write dependencies. Specifically, reading a list of n values infers a potentially sequential version order among the corresponding n append operations. Our utilization of the RMW pattern shares the similar spirit: we can build a chain of MTs with consecutive writes on the same object. Elle also incorporates a linearizability checker called Knossos [47], which operates without using an off-the-shelf solver. However, it is less efficient than Porcupine [11].

Porcupine is a fast LIN checker for verifying histories on concurrent data types. It implements *P-compositionality* [40], a generalization of the locality principle [18] for LIN. Our MTC-SSER checker concentrates on histories of read&write operations on a single read-write register (owing to the locality principle). By initially establishing a chain of these read&write operations (owing to the RMW pattern) and then verifying the real-time requirement, it achieves an $O(n)$ time complexity for histories comprising n operations.

Plume [6] is tailored for checking weak isolation levels, including TRANSACTIONAL CAUSAL CONSISTENCY and READ COMMITTED. It builds on modular transactional anomalous patterns that faithfully characterize these isolation levels. By utilizing vectors and tree clocks, Plume achieves superior performance in checking weak isolation levels. This work complements Plume by optimizing checking performance for stronger isolation levels.

TxCheck [7] uses complex SQL queries to detect logic bugs in database transactions, which are not necessarily isolation bugs. This approach enables the testing of advanced database features, such as range queries and indexes, complementing existing checkers that are primarily limited to simpler data models, like read-write registers. However, TxCheck relies on metamorphic testing [48] to construct its test oracles, which cannot establish the ground truth. Consequently, the observed "incorrect" query results may include false positives. Furthermore, as TxCheck is not specifically designed for discovering isolation bugs, it may generate false positives due to spurious dependencies created between SQL statements.

VII. Conclusion and Future Work

Utilizing MTs, we have tackled the inefficiency issues throughout the end-to-end black-box checking process for three strong isolation levels in a unified manner. Moreover, we have demonstrated the effectiveness of our MTC tool in detecting isolation bugs with MT workloads. R1-O1

MTC's effectiveness may be sensitive to workload parameters such as object-access distribution and the number of objects, which we plan to explore in future work. R1-O2 We also plan to further conduct experiments on FaunaDB and VoltDB, evaluating MTC's performance in verifying SSER on arbitrary MT histories (in addition to lightweight transaction histories in Section V-C2) and comparing it to Elle [49], [50]. A key challenge is to accurately collect transaction start and finish wall-clock timestamps while handling potential clock skew to minimize false positives and negatives [28]. Furthermore, R3-D3 extending MTC to incorporate Plume [6] for verifying weaker isolation levels would broaden its applicability.

Our MTC tool includes a workload generator that creates randomized test cases with MT histories. R3-D2 We believe that guiding the generator to cover the isolation anomalies depicted in Figure 5, as shown in [6], [51], would further benefit developers. Moreover, the mechanisms for executing MTs, such as R3-O3 simulating stalls, altering state visibility, and injecting faults, are orthogonal to our primary contributions. An interesting avenue for future work is to explore how to execute MTs in a controlled manner to trigger specific anomalies.

REFERENCES

[1] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *SIGMOD '95*. ACM, 1995, pp. 1–10.

[3] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies from experimental observations," *Proc. VLDB Endow.*, vol. 14, no. 3, pp. 268–280, Nov. 2020.

[4] K. Huang, S. Liu, Z. Chen, H. Wei, D. A. Basin, H. Li, and A. Pan, "Efficient black-box checking of snapshot isolation in databases," *Proc. VLDB Endow.*, vol. 16, no. 6, pp. 1264–1276, 2023.

[5] R. Biswas and C. Enea, "On the complexity of checking transactional consistency," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 2019.

[6] S. Liu, L. Gu, H. Wei, and D. Basin, "Plume: Efficient and complete black-box checking of weak isolation levels," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, 2024.

[7] Z.-M. Jiang, S. Liu, M. Rigger, and Z. Su, "Detecting transactional bugs in database engines via graph-based oracle construction," in *OSDI 23*. USENIX Association, 2023, pp. 397–417.

[8] Kyle Kingsbury, "Jepsen testing of PostgreSQL 12.3," https://jepsen.io/analyses/postgresql-12.3, Accessed in February, 2024.

[9] C. Tan, C. Zhao, S. Mu, and M. Walfish, "COBRA: Making transactional key-value stores verifiably serializable," in *OSDI'20*, 2020.

[10] J. Zhang, Y. Ji, S. Mu, and C. Tan, "Viper: A fast snapshot isolation checker," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. ACM, 2023, p. 654–671.

[11] A. Athalye, "Porcupine: A fast linearizability checker in Go," https://github.com/anishathalye/porcupine, Accessed in December, 2023.

[12] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, USA, 1999.

[13] A. Cerone and A. Gotsman, "Analysing snapshot isolation," *J. ACM*, vol. 65, no. 2, 2018.

[14] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, nov 2013.

[15] S. Liu, L. Multazzu, H. Wei, and D. A. Basin, "NOC-NOC: Towards performance-optimal distributed transactions," *Proc. ACM Manag. Data*, vol. 2, no. 1, mar 2024.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[17] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[18] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[19] A. Cerone, G. Bernardi, and A. Gotsman, "A framework for transactional consistency models with atomic visibility," in *CONCUR'15*, vol. 42, 2015, pp. 58–71.

[20] M. F. Daniel Abadi, "Serializability vs "strict" serializability: The dirty secret of database isolation levels," https://fauna.com/blog/serializability-vs-strict-serializability-the-dirty-secret-of-database-isolation-levels, Accessed in April, 2024.

[21] MongoDB, https://www.mongodb.com/docs/manual/core/transactions/, Accessed in April, 2024.

[22] PostgreSQL, https://www.postgresql.org/docs/current/transaction-iso.html, Accessed in April, 2024.

[23] MariaDB, Accessed in April, 2024, https://mariadb.com/kb/en/mariadb-transactions-and-isolation-levels-for-sql-server-users/.

[24] YugabyteDB, https://docs.yugabyte.com/preview/explore/transactions/isolation-levels/, Accessed in April, 2024.

[25] TiDB, https://docs.pingcap.com/tidb/stable/transaction-isolation-levels, Accessed in April, 2024.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 2013.

[27] R. Biswas, D. Kakwani, J. Vedurada, C. Enea, and A. Lal, "MonkeyDB: Effectively testing correctness under weak isolation levels," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, 2021.

[28] H. Lu, S. Mu, S. Sen, and W. Lloyd, "NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall," in *OSDI '23*. USENIX Association, 2023, pp. 305–323.

[29] A. Cerone, A. Gotsman, and H. Yang, "Algebraic laws for weak consistency," in *CONCUR '17*, vol. 85, 2017, pp. 26:1–26:18.

[30] R. C. Steinke and G. J. Nutt, "A unified theory of shared memory consistency," *J. ACM*, vol. 51, no. 5, p. 800–849, 2004.

[31] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Comput.*, vol. 26, no. 4, pp. 1208–1244, aug 1997.

[32] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[33] Apache Cassandra, "Using lightweight transactions," https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useInsertLWT.html, Accessed in April, 2024.

[34] ScyllaDB, "Lightweight transactions," https://opensource.docs.scylladb.com/stable/using-scylla/lwt.html, Accessed in April, 2024.

[35] B. F. Cooper, P. Narayan, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS to sherpa: lessons from Yahoo!'s cloud database," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2300–2307, 2019.

[36] Azure Cosmos DB, "Azure Cosmos DB for Apache Cassandra lightweight transactions with conditions," https://learn.microsoft.com/en-us/azure/cosmos-db/cassandra/lightweight-transactions, Accessed in April, 2024.

[37] etcd, "etcd3 API: transaction," https://etcd.io/docs/v3.4/learning/api/\#transaction, Accessed in April, 2024.

[38] Anonymous Authors, "Boosting Database Isolation Checking via Mini-Transactions (Technical Report)," https://github.com/anonymous-mts/mtc, Tech. Rep., 2024, supplementary material.

[39] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu, "SAT modulo monotonic theories," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15, 2015, pp. 3702–3709.

[40] A. Horn and D. Kroening, "Faster linearizability checking via p-compositionality," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, 2015, pp. 50–65.

[41] Transactions on MariaDB-Galera violated Causal Consistency, "Issue #609," https://github.com/codership/galera/issues/609#issuecomment-1046763283, Accessed in February, 2025.

[42] K. Kingsbury, "Jepsen testing of MongoDB 4.2.6," http://jepsen.io/analyses/mongodb-4.2.6, Accessed in February, 2025.

[43] ——, "Jepsen testing of Dgraph 1.1.1," http://jepsen.io/analyses/dgraph-1.1.1, Accessed in February, 2025.

[44] ——, "Avoid update conflict out serialization anomalies," https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=5940ffb221316ab73e6fdc780dfe9a07d4221ebb, Accessed in February, 2025.

[45] ——, "Jepsen: Cassandra," https://aphyr.com/posts/294-call-me-maybe-cassandra, Accessed in April, 2025.

[46] ——, "Jepsen," https://jepsen.io, Accessed in February, 2024.

[47] ——, "Knossos," https://github.com/jepsen-io/knossos, Accessed in December, 2023.

[48] T. Y. Chen, S. C. Cheung, and S. Yiu, "Metamorphic testing: A new approach for generating next test cases," *CoRR*, vol. abs/2002.12543, 2020. [Online]. Available: https://arxiv.org/abs/2002.12543

[49] K. Kingsbury, "Jepsen testing of FaunaDB 2.5.4," https://jepsen.io/analyses/faunadb-2.5.4, Accessed in February, 2025.

[50] ——, "Jepsen testing of VoltDB 6.3," https://jepsen.io/analyses/voltdb-6-3, Accessed in February, 2025.

[51] K. Li, S. Weng, L. Ni, C. Yang, R. Zhang, X. Zhou, and A. Zhou, "Dbstorm: Generating various effective workloads for testing isolation levels," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. ACM, 2024, p. 755–767. [Online]. Available: https://doi.org/10.1145/3650212.3680318

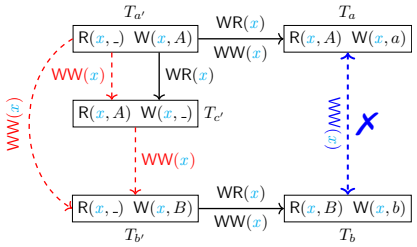


Fig. 15: Transactions T_a and $T_{c'}$ read the same value of the object x from the transaction $T_{a'}$ and then write different values.

APPENDIX A
CORRECTNESS PROOFS OF THE VERIFICATION ALGORITHMS

Lemma 3. *If the graph $\mathcal{G}$ returned by* BUILDDEPENDENCY *does not contain any* WW *loops or any instances of the* DIVERGENCE *pattern, then it is a dependency graph of $\mathcal{H}$.*

Proof. Suppose that $\mathcal{G}$ does not contain any WW loops or any instances of the DIVERGENCE pattern. We need to show that for each object $k \in \mathsf{X}$, $\mathsf{WW}(k)$ is a strict total order on the set $\mathsf{WriteTx}_x$. Suppose by contradiction that there exists an object $x \in \mathsf{X}$ such that $\mathsf{WW}(x)$ is not a strict total order on $\mathsf{WriteTx}_x$. In the following, we identify an instance of the DIVERGENCE pattern in $\mathcal{G}$.

Since $\mathcal{G}$ does not contain any WW loops, $\mathsf{WW}(x)$ is a strict order. Therefore, $\mathsf{WW}(x)$ is not total. Consider two transactions T_a and T_b such that they both write to x with different values, but there are no $\mathsf{WW}(x)$ dependency edges between them in $\mathcal{G}$; see Figure 15. If T_a and T_b read the value of x from the same transaction, then we are done. Otherwise, consider the transactions $T_{a'}$ and $T_{b'}$ ($T_{a'} \neq T_{b'}$) from which T_a and T_b read the value of x, respectively. That is, $T_{a'} \xrightarrow{\mathsf{WR}(x)/\mathsf{WW}(x)} T_a$ and $T_{b'} \xrightarrow{\mathsf{WR}(x)/\mathsf{WW}(x)} T_b$ (see line 11 of BUILDDEPENDENCY). It is clear that $T_{a'} \neq T_b$ and $T_{b'} \neq T_a$, since otherwise there would be a $\mathsf{WW}(x)$ dependency edge between T_a and T_b. There are two cases regarding whether there is a $\mathsf{WW}(x)$ dependency edge between $T_{a'}$ and $T_{b'}$ in $\mathcal{G}$:

- Suppose there is a $\mathsf{WW}(x)$ dependency edge between $T_{a'}$ and $T_{b'}$ in $\mathcal{G}$, say $T_{a'} \xrightarrow{\mathsf{WW}(x)} T_{b'}$. Note that this $\mathsf{WW}(x)$ edge may be derived by transitivity (see line 13 of BUILDDEPENDENCY). Let $T_{c'}$ be a transaction *directly following* $T_{a'}$ in the $\mathsf{WW}(x)$ dependency chain from $T_{a'}$ to $T_{b'}$. That is, the edge $T_{a'} \xrightarrow{\mathsf{WW}(x)} T_{c'}$ coincides with $T_{a'} \xrightarrow{\mathsf{WR}(x)} T_{b'}$ (see line 11 of BUILDDEPENDENCY). As illustrated in Figure 15, we have $T_{a'} \xrightarrow{\mathsf{WR}(x)/\mathsf{WW}(x)} T_{c'} \xrightarrow{\mathsf{WW}(x)} T_{b'}$. It is possible that $T_{c'} = T_{b'}$, but we argue that
  - $T_{c'} \neq T_a$. Otherwise, we have $T_a = T_{c'} \xrightarrow{\mathsf{WW}(x)} T_{b'} \xrightarrow{\mathsf{WW}(x)} T_b$. Then by transitivity, $T_a \xrightarrow{\mathsf{WW}(x)} T_b$, contradicting the assumption that there is no $\mathsf{WW}(x)$ dependency edge between T_a and T_b.
  - $T_{c'} \neq T_b$. Otherwise, we have $T_b = T_{c'} \xrightarrow{\mathsf{WW}(x)} T_{b'} \xrightarrow{\mathsf{WW}(x)} T_b$. Then by transitivity, $T_b \xrightarrow{\mathsf{WW}(x)} T_b$, contradicting the assumption that $\mathcal{G}$ contains no $\mathsf{WW}(x)$ loops.

  Therefore, $T_{a'}$, $T_{c'}$, and T_a forms an instance of the DIVERGENCE pattern.
- If there is no a $\mathsf{WW}(x)$ dependency edge between $T_{a'}$ and $T_{b'}$ in $\mathcal{G}$, we apply the same argument above to $T_{a'}$ and $T_{b'}$.

Since the number of transactions in $\mathcal{H}$ is finite and $\mathcal{H}$ contains an initial transaction $\bot_T$ which install initial values to all objects, we will eventually identify three transactions that form an instance of the DIVERGENCE pattern. □

Lemma 4. *If the graph $\mathcal{G}$ returned by* BUILDDEPENDENCY *is acyclic, then it is a dependency graph of $\mathcal{H}$.*

Proof. Suppose by contradiction that $\mathcal{G}$ is not a dependency graph of $\mathcal{H}$. By Lemma 3, $\mathcal{G}$ contains WW loops or instances of the DIVERGENCE pattern. Since $\mathcal{G}$ is acyclic, there are no WW loops in $\mathcal{G}$. Thus, $\mathcal{G}$ must contain instances of the DIVERGENCE pattern, as illustrated in Figure 3. However, in this case, there would be a cycle $T_2 \xrightarrow{\mathsf{RW}(x)} T_3 \xrightarrow{\mathsf{RW}(x)} T_2$ in $\mathcal{G}$. □

Theorem 3 (Correctness of CHECKSSER)**.** CHECKSSER *returns* true *if and only if $\mathcal{H}$ satisfies* SSER.

Proof. The proof proceeds in two directions.

- (“ $\Longrightarrow$ ”) Suppose that CHECKSSER returns true. Then the graph $\mathcal{G}$ at line 2 of CHECKSSER is acyclic. By Lemma 4, $\mathcal{G}$ is a dependency graph of $\mathcal{H}$. By Definition 4, $\mathcal{H}$ satisfies SSER.
- (“ $\Longleftarrow$ ”) Suppose that CHECKSSER returns false. Then the graph $\mathcal{G}$ at line 2 of CHECKSSER is cyclic. Since $\mathcal{G}$ is a subgraph of any dependency graph of $\mathcal{H}$, no dependency graph of $\mathcal{H}$ is acyclic. By Definition 4, $\mathcal{H}$ does not satisfy SSER.

□

Similarly, by Lemma 4 and Definition 5, we can establish the correctness of CHECKSER. The proof is omitted here.

Theorem 4 (Correctness of CHECKSER)**.** CHECKSER *returns* true *if and only if $\mathcal{H}$ satisfies* SER.

Finally, we show the correctness of CHECKSI.

Theorem 5 (Correctness of CHECKSI)**.** CHECKSI *returns* true *if and only if $\mathcal{H}$ satisfies* SI.

Proof. The proof proceeds in two directions.

- (“ $\Longleftarrow$ ”) We show that if CHECKSI returns false, then $\mathcal{H}$ does not satisfy SI. There are two cases:
  - Suppose that CHECKSI returns false at line 3 of CHECKSI. Then $\mathcal{G}$ contains an instance of the DIVERGENCE pattern. By Lemma 1, $\mathcal{H}$ does not satisfy SI.

– Suppose that CHECKSI returns false at line 6 of CHECKSI. There are two cases regarding whether $\mathcal{G}$ contains a WW loop:
  * If $\mathcal{G}'$ contains a WW loop, then, by the construction of $\mathcal{G}'$ (line 5 of CHECKSI), the graph $\mathcal{G}$ at line 5 also contains a WW loop. Since $\mathcal{G}$ is a subgraph of any dependency graph of $\mathcal{H}$, $\mathcal{H}$ does not admit any legal dependency graphs. By Definition 6, $\mathcal{H}$ does not satisfy SI.
  * If $\mathcal{G}'$ contains no WW loops, then, by the construction of $\mathcal{G}'$ (line 5 of CHECKSI), $\mathcal{G}$ does not contain any WW loop either. On the other hand, the graph $\mathcal{G}$ at line 5 does not contain any instances of the DIVERGENCE pattern. By Lemma 3, the graph $\mathcal{G}$ at line 5 is a dependency graph of $\mathcal{H}$. Since $\mathcal{G}$ is a subgraph of any dependency graph of $\mathcal{H}$, $\mathcal{G}$ is the unique dependency graph of $\mathcal{H}$. Then by Definition 6, $\mathcal{H}$ does not satisfy SI.
- (" $\Longrightarrow$ ") If CHECKSI returns true at line 6 of CHECKSI, then $\mathcal{G}'$ at line 5 of CHECKSI is acyclic. Therefore, $\mathcal{G}'$ contains no WW loops. By the line of reasoning above, $\mathcal{G}$ at line 5 is a dependency graph of $\mathcal{H}$. Therefore, by Definition 6, $\mathcal{H}$ satisfies SI.

□

APPENDIX B
CORRECTNESS PROOFS OF THE OPTIMIZED VERIFICATION ALGORITHMS

Proof of Lemma 2. For the $S \xrightarrow{\widehat{\mathsf{RW}}} T'$ edge in $\mathcal{G}$, there exists a transaction T such that $T \xrightarrow{\mathsf{WR}} S$ and $T \xrightarrow{\widehat{\mathsf{WW}}} T'$. Therefore, there exists a sequence of WW edges from T to T', denoted $T \xrightarrow{\mathsf{WW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T'$ such that $S \xrightarrow{\mathsf{RW}} T_1$. □

The correctness of the optimized versions of CHECKSSER and CHECKSER directly follows from the following theorem.

Proof of Theorem 1. The proof proceeds in two directions.
- " $\Longrightarrow$ ": Suppose that $\mathcal{G}$ is acyclic. Since $E_{\widehat{\mathcal{G}}} \subseteq E_{\mathcal{G}}$, $\widehat{\mathcal{G}}$ is also acyclic.
- " $\Longleftarrow$ ": Suppose by contradiction that $\mathcal{G}$ contains a cycle, denoted $\mathcal{C}$. We need to show that $\widehat{\mathcal{G}}$ also contains a cycle. If $E_{\mathcal{C}} \subseteq E_{\widehat{\mathcal{G}}'}$, then we are done. Otherwise, any edge in $E_{\mathcal{C}} \setminus E_{\widehat{\mathcal{G}}}$ must be either a $\widehat{\mathsf{WW}}$ edge or a $\widehat{\mathsf{RW}}$ edge. We can transform $\mathcal{C}$ into a cycle within $\widehat{\mathcal{G}}$ as follows (see Figure 6):
  – For each $\widehat{\mathsf{WW}}$ edge in $\mathcal{C}$, replace it with a sequence of WW edges from which the $\widehat{\mathsf{WW}}$ edge can be derived by transitivity.
  – For each $S \xrightarrow{\widehat{\mathsf{RW}}} T'$ edge in $\mathcal{C}$, replace it with a path $S \xrightarrow{\mathsf{RW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T'$, as indicated by Lemma 2.

□

Let $\widehat{\mathcal{G}}'$ be the induced graph of $\widehat{\mathcal{G}}$, i.e., $\widehat{\mathcal{G}}' \leftarrow (V_{\widehat{\mathcal{G}}}, (\mathsf{SO}_{\widehat{\mathcal{G}}} \cup \mathsf{WR}_{\widehat{\mathcal{G}}} \cup \mathsf{WW}_{\widehat{\mathcal{G}}}) \ ; \ \mathsf{RW}_{\widehat{\mathcal{G}}}?)$. The correctness of the optimized versions of CHECKSI directly follows from the following theorem.

Proof of Theorem 2. The proof proceeds in two directions.
- " $\Longrightarrow$ ": Suppose that $\mathcal{G}'$ is acyclic. Since $E_{\widehat{\mathcal{G}}'} \subseteq E_{\mathcal{G}'}$, $\widehat{\mathcal{G}}'$ is also acyclic.
- " $\Longleftarrow$ ": Suppose by contradiction that $\mathcal{G}'$ contains a cycle, denoted $\mathcal{C}'$. We need to show that $\widehat{\mathcal{G}}'$ also contains a cycle.
  If $E_{\mathcal{C}'} \subseteq E_{\widehat{\mathcal{G}}'}$, then we are done. Otherwise, we can transform $\mathcal{C}'$ into a cycle within $\widehat{\mathcal{G}}'$ by replacing each edge in $E_{\mathcal{C}'} \setminus E_{\widehat{\mathcal{G}}'}$ with edge(s) in $\widehat{\mathcal{G}}'$ as follows:
  1) Consider a $\widehat{\mathsf{WW}}$ edge in $\mathcal{C}'$. Replace it with a sequence of WW edges from which the $\widehat{\mathsf{WW}}$ edge can be derived by transitivity.
  2) Consider an $S' \xrightarrow{\mathsf{SO} \ ; \ \widehat{\mathsf{RW}}} T'$ edge in $\mathcal{C}'$. Suppose that it is derived from $S' \xrightarrow{\mathsf{SO}} S \xrightarrow{\widehat{\mathsf{RW}}} T'$. As indicated by Lemma 2, $S \xrightarrow{\widehat{\mathsf{RW}}} T'$ can be replaced by a path $S \xrightarrow{\mathsf{RW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T'$, Therefore, $S' \xrightarrow{\mathsf{SO} \ ; \ \widehat{\mathsf{RW}}} T'$ can be replaced by the path $S' \xrightarrow{\mathsf{SO} \ ; \ \mathsf{RW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T'$ in $\widehat{\mathcal{G}}'$.
  3) Consider a WR ; $\widehat{\mathsf{RW}}$ edge in $\mathcal{C}'$. This is similar to case (2).
  4) Consider a WW ; $\widehat{\mathsf{RW}}$ edge in $\mathcal{C}'$. This is similar to case (2).
  5) Consider a $S' \xrightarrow{\widehat{\mathsf{WW}} \ ; \ \mathsf{RW}} T'$ edge in $\mathcal{C}'$. Suppose that it is derived from $S' \xrightarrow{\widehat{\mathsf{WW}}} S \xrightarrow{\mathsf{RW}} T'$. The edge $S' \xrightarrow{\widehat{\mathsf{WW}}} S$ can be replaced by a path $S' \xrightarrow{\mathsf{WW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T_n \xrightarrow{\mathsf{WW}} S$. Therefore, $S' \xrightarrow{\widehat{\mathsf{WW}} \ ; \ \mathsf{RW}} T'$ can be replaced by a path $S' \xrightarrow{\mathsf{WW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T_n \xrightarrow{\mathsf{WW} \ ; \ \mathsf{RW}} T'$ in $\widehat{\mathcal{G}}'$.
  6) Consider a $S' \xrightarrow{\widehat{\mathsf{WW}} \ ; \ \widehat{\mathsf{RW}}} T'$ edge in $\mathcal{C}'$. Suppose that it is derived from $S' \xrightarrow{\mathsf{WW}} S \xrightarrow{\mathsf{RW}} T'$. Replace $S' \xrightarrow{\mathsf{WW}} S$ with a path $S' \xrightarrow{\mathsf{WW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T_n \xrightarrow{\mathsf{WW}} S$. Replace $S \xrightarrow{\mathsf{RW}} T'$ with a path $S \xrightarrow{\mathsf{RW}} T_{n+1} \xrightarrow{\mathsf{WW}} T_{n+2} \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T_{n+m} \xrightarrow{\mathsf{WW}} T'$, as indicated by Lemma 2. Therefore, $S' \xrightarrow{\widehat{\mathsf{WW}} \ ; \ \widehat{\mathsf{RW}}} T'$ can be replaced by a path $S' \xrightarrow{\mathsf{WW}} T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T_n \xrightarrow{\mathsf{WW} \ ; \ \mathsf{RW}} T_{n+1} \xrightarrow{\mathsf{WW}} T_{n+2} \xrightarrow{\mathsf{WW}} \cdots \xrightarrow{\mathsf{WW}} T_{n+m} \xrightarrow{\mathsf{WW}} T'$ in $\widehat{\mathcal{G}}'$.

□

APPENDIX C
COMPLEXITY ISSUES

It is known that verifying whether a given general history satisfies SSER, SER or SI is NP-complete [17], [5]. In this section, we show that verifying whether a given mini-transaction history *without unique values* satisfies SSER, SER, or SI is also NP-complete.

Theorem 6. *The problem of verifying whether a given mini-transaction history without unique values satisfies* SSER *is* NP-complete.

Proof. Directly from [31, Theorem 4.11] which shows that verifying whether an execution with read&write operations only satisfies LIN is NP-complete. □

Our proof of the NP-hardness of verifying SER for mini-transaction histories without unique values relies on the NP-hardness of verifying SEQUENTIAL CONSISTENCY (SC). SC requires that all operations appear to be executed in some sequential order that is consistent with the session order.

Definition 11 (Sequential Consistency)**.** *A history $\mathcal{H}$ is sequentially consistent if and only if there exists a permutation* Π *of all the operations in $\mathcal{H}$ such that* Π *preserves the program order of operations and follows the sequential semantics of each object.*

Note that when each transaction comprises only one operation, SER is the same as SC.

Theorem 7. *The problem of verifying whether a given mini-transaction history without unique values satisfies* SER *is* NP-complete.

Proof. Directly from [31, Theorem 4.9] which shows that verifying whether an execution with read&write operations only satisfies SC is NP-complete. □

The proof of the NP-hardness of verifying SI for mini-transaction histories without unique values is much more involved. It is heavily inspired by that of [5, Theorem 3.2].[3] In the axiomatic framework of [5], SI is characterized by two axioms: PREFIX and CONFLICT. The crucial difference is that in histories without unique values, the write-read relation is not given as input. In our context, a mini-transaction history (without unique values) satisfies SI if there exist a write-read relation, denoted by wr to keep the notation consistent with [5], and a commit order co (which is a strict total order on the set of transactions in $\mathcal{H}$) such that the PREFIX and CONFLICT axioms are satisfied. In the following, we omit the RT relation in a history since it is not relevant for SI. An *abstract execution* $\mathcal{A} = (\mathcal{T}, so, wr, co)$ is a history $(\mathcal{T}, so)$ associated with a write-read relation wr and a commit order co. We use R^* to denote the reflexive and transitive closure of the relation R.

Definition 12 (PREFIX axiom [5])**.** *An abstract execution* $\mathcal{A} = (\mathcal{T}, \mathsf{SO}, wr, co)$ *satisfies the* PREFIX *axiom if and only if*

$$\forall x \in \mathsf{X}.\ \forall t_1, t_2 \neq t_1, t_3 \in \mathcal{T}.\\ (\langle t_1, t_3\rangle \in wr(x) \wedge t_2 \vdash \mathsf{W}(x, \_) \wedge \langle t_2, t_3\rangle \in co^*; (wr \cup \mathsf{SO})) \\ \implies \langle t_2, t_1\rangle \in co.$$

[3]We largely follow the account of [5] and adapt it to our context when necessary.

Definition 13 (CONFLICT axiom [5])**.** *An abstract execution* $\mathcal{A} = (\mathcal{T}, \mathsf{SO}, wr, co)$ *satisfies the* CONFLICT *axiom if and only if*

$$\forall x, y \in \mathsf{X}.\ \forall t_1, t_2, t_3, t_4 \in \mathcal{T}.\\ t_1 \neq t_2 \wedge \langle t_1, t_3\rangle \in \mathsf{WR}_x \wedge t_2 \vdash \mathsf{W}(x, \_) \wedge t_3 \vdash \mathsf{W}(y, \_) \wedge \\ t_4 \vdash \mathsf{W}(y, \_) \wedge \langle t_2, t_4\rangle \in co^* \wedge \langle t_4, t_3\rangle \in co \\ \Rightarrow \langle t_2, t_1\rangle \in co$$

Theorem 8. *The problem of verifying whether a given mini-transaction history without unique values satisfies* SI *is* NP-complete.

Proof of Theorem 8. It is easy to see that the problem is in NP. To show NP-hardness, we establish a reduction from boolean satisfiability. Let $\phi = D_1 \wedge \cdots \wedge D_m$ be a CNF formula over boolean variables $x_1, \cdots, x_n$, where each D_i is a disjunctive clause with m_i literals. We use λ_{ij} denote the j-th literal of D_i. We construct a mini-transaction history h_ϕ for ϕ such that ϕ is satisfiable if and only if h_ϕ satisfies SI.

The insight is to represent truth values of each variable and literal in ϕ with the polarity of the commit order between corresponding transaction pairs. Specifically, for each variable x_k, h_ϕ contains a pair of mini-transaction a_k and b_k such that x_k is false if and only if $\langle a_k, b_k\rangle \in co$. For each literal λ_{ij}, h_ϕ contains a triple of mini-transactions w_{ij}, y_{ij}, and z_{ij} such that λ_{ij} is false if and only if $\langle y_{ij}, z_{ij}\rangle \in co$.

The history h_ϕ should ensure that the co ordering corresponding to an assignment that makes the formula false form a cycle. To this end, we add all pairs $\langle z_{ij}, y_{i,(j+1)\%m_i}\rangle$ in the session order so. Consequently, an unsatisfied clause D_i leads to a cycle of the form $y_{i1} \xrightarrow{co} z_{i1} \xrightarrow{so} y_{i2} \xrightarrow{co} z_{i2} \cdots z_{im_i} \xrightarrow{so} y_{i1}$.

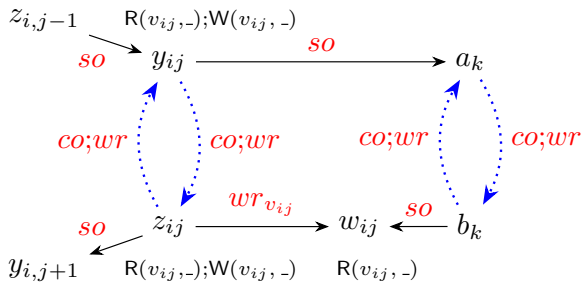
We use special sub-histories to ensure the consistency between the truth value of literals and variables. That is, $\lambda_{ij} = x_k$ is false if and only if x_k is false. Figure 16a shows the sub-history associated to a positive literal $\lambda_{ij} = x_k$, while Figure 16b shows the case of a negative literal $\lambda_{ij} = \neg x_k$.

For a positive literal $\lambda_{ij} = x_k$ (Figure 16a),

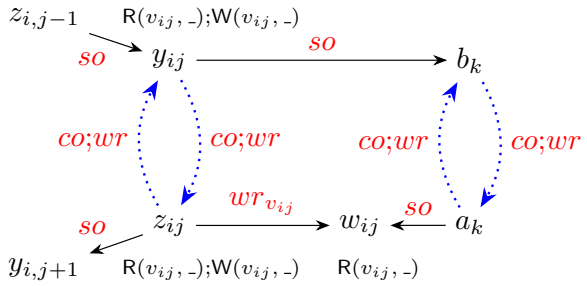
1) we enrich the session order with the pairs $\langle y_{ij}, a_k\rangle$ and $\langle b_k, w_{ij}\rangle$;
2) we include reads and writes to a variable v_{ij} in the transaction y_{ij} and z_{ij};
3) we make w_{ij} read v_{ij}; and
4) we make all the read and written value the same.

The steps (2)–(4) are specially designed for our reduction, different from that in the proof of [5]: *it utilizes the fact that unique values are not required in such a history and ensures that every transaction in the sub-history is a mini-transaction.*

We only need to determine the write-read relation between y_{ij}, z_{ij}, and w_{ij}, since other transactions do not read or write the same variable. Specifically, we construct a write-read relation from z_{ij} to w_{ij}. For transactions y_{ij} and z_{ij}, which read and write the same variable with the same value, we require the direction of wr between y_{ij} and z_{ij} be the same as that of co between them. Otherwise, the history h_ϕ associated



(a) $\lambda_{ij} = x_k$



(b) $\lambda_{ij} = \neg x_k$

Fig. 16: Sub-histories in h_ϕ for literal λ_{ij} and variable x_k

with these wr *and* co *relations cannot satisfy* PREFIX *and* CONFLICT.

Lemma 5. *The special sub-histories enforce that if history* h_ϕ *satisfies SI, then there exist a write-read relation* wr *and a commit order* co *such that* $\langle h_\phi, wr, co\rangle$ *satisfies* PREFIX *and* CONFLICT *and*

$$\langle a_k, b_k\rangle \in co \text{ iff } \langle y_{ij}, z_{ij}\rangle \in co \text{ when } \lambda_{ij} = x_k, \text{and}$$
$$\langle a_k, b_k\rangle \in co \text{ iff } \langle z_{ij}, y_{ij}\rangle \in co \text{ when } \lambda_{ij} = \neg x_k.$$

The proof of Lemma 5 and the remaining correctness proof of the reduction can be conducted similarly as those in [5]. We refer the interested reader to [5] for details.

□

APPENDIX D
END-TO-END CHECKING PERFORMANCE FOR SI

MTC-SI with the MT workload generation substantially outperforms PolySI with the GT workload generation in terms of both time and memory under varying concurrency levels, as shown in Figure 17.

APPENDIX E
ISOLATION BUGS REDISCOVERED BY MTC

Figure 18 depicts the isolation bugs detected by MTC when checking six releases of five DBMSs (see also Table II).

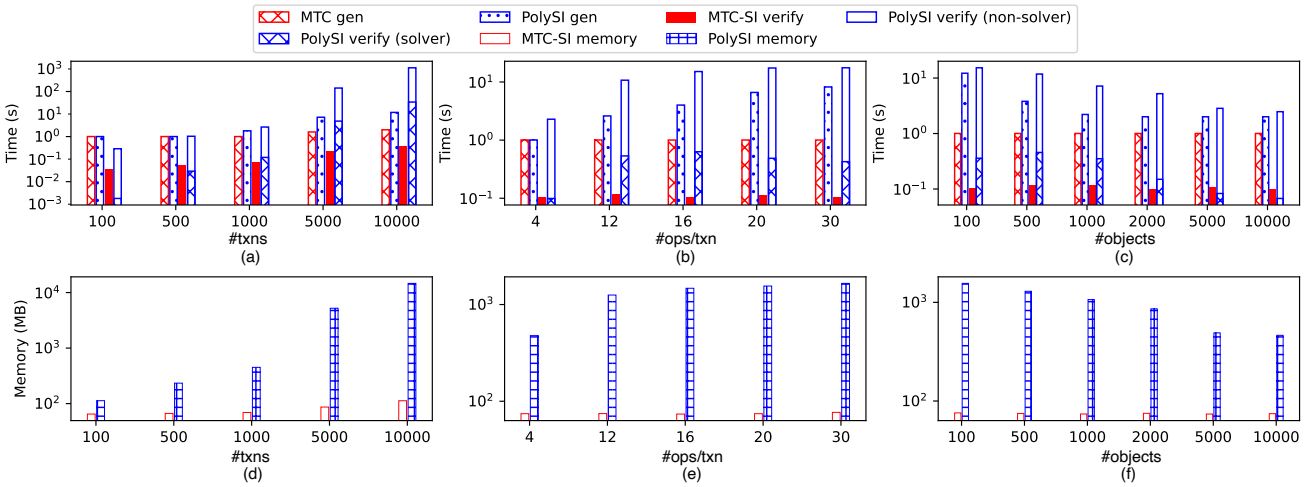
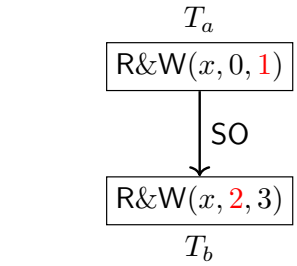
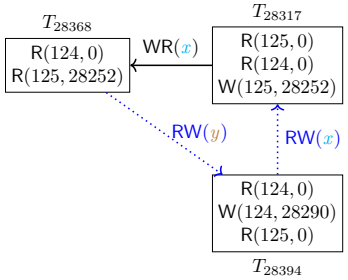


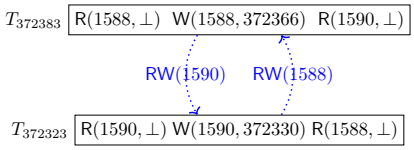
Fig. 17: End-to-end checking performance, with time decomposed into history generation and verification (on SI).



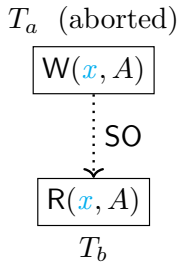
(a) Cassandra: READ UNCOMMITTED that violates SSER



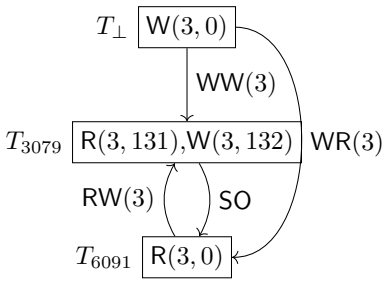
(b) PostgreSQL: WRITE SKEW that violates SER



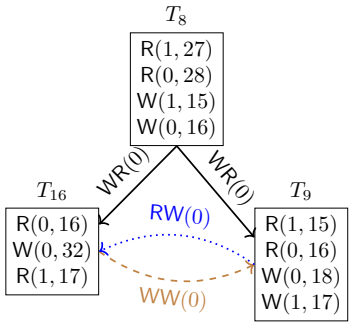
(c) PostgreSQL: LONG FORK that violates SER



(d) MongoDB: READ UNCOMMITTED that violates SI



(e) Dgraph: CAUSALITY VIOLATION that violates SI



(f) MariaDB Galera: LOST UPDATE that violates SI

Fig. 18: Rediscovered isolation bugs by MTC.