Fast Verification of Strong Database Isolation

Zhiheng Cai
Tsinghua University
cai-zh24@mails.tsinghua.edu.cn

Si Liu
ETH Zurich
si.liu@inf.ethz.ch

Hengfeng Wei*
Hunan University
hfwei@hnu.edu.cn

Yuxing Chen
Tencent Inc.
axingguchen@tencent.com

Anqun Pan
Tencent Inc.
aaronpan@tencent.com

ABSTRACT

Strong isolation guarantees, such as serializability and snapshot isolation, are essential for maintaining data consistency and integrity in modern databases. Verifying whether a database upholds its claimed guarantees is increasingly critical, as these guarantees form a contract between the vendor and its users. However, this task is challenging, particularly in black-box settings, where only observable system behavior is available and often involves uncertain dependencies between transactions.

In this paper, we present VeriStrong, a fast verifier for strong database isolation. At its core is a novel formalism called hyperpolygraphs, which compactly captures both certain and uncertain transactional dependencies in database executions. Leveraging this formalism, we develop sound and complete encodings for verifying both serializability and snapshot isolation. To achieve high efficiency, VeriStrong tailors SMT solving to the characteristics of database workloads, in contrast to prior general-purpose approaches. Our extensive evaluation across diverse benchmarks shows that VeriStrong not only significantly outperforms state-of-the-art verifiers on the workloads they support, but also scales to large, general workloads beyond their reach, while maintaining high accuracy in detecting isolation anomalies.

PVLDB Reference Format:
Zhiheng Cai, Si Liu, Hengfeng Wei, Yuxing Chen, and Anqun Pan. Fast Verification of Strong Database Isolation . PVLDB, 19(4): 563 - 575, 2025.
doi:10.14778/3785297.3785300

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CzxingcHen/VeriStrong.

1 INTRODUCTION

Strong isolation levels or guarantees—the I in ACID [36]—such as Serializability and Snapshot Isolation, are essential for ensuring data consistency and integrity in modern databases. These guarantees prevent subtle concurrency anomalies, such as lost updates, that can lead to incorrect application behavior. Once a database is deployed, its promised isolation guarantees become a binding contract between the vendor and its users. This raises a critical question: *How can we be sure that the database upholds its promise?*

Answering this question is nontrivial. Modern databases have large codebases that are often inaccessible or too complex to reason about, even when available. This makes full verification of isolation guarantees practically infeasible. Black-box verification [4, 33] offers an effective alternative: a verifier collects execution histories of database transactions as an *external observer* and checks whether these histories satisfy the isolation level in question.

However, verifying database histories remains challenging. The verification problem for strong isolation levels has been proven to be NP-hard, even for verifying a single history [4]. This complexity arises from the fact that, as an external observer of a black-box database, a verifier must *infer* the internal execution order of transactions, e.g., determining which transaction wrote an earlier version and which one wrote a later one. This challenge is further compounded by practical constraints: given limited time and memory, only a finite number of "guesses" can be made by the verifier, and thus only a finite number of histories can be verified. Yet, examining more histories is highly desirable, either to increase the likelihood of uncovering anomalies or to strengthen confidence in their absence.

Recent years have seen significant progress in accelerating the verification of strong isolation guarantees [4, 13, 17, 33, 35, 39]. Representative verifiers include Cobra [33] for Serializability, PolySI [13] and Viper [39] for Snapshot Isolation, and Elle [17] for both (a detailed discussion is provided in Section 8). Despite these advances, verification efficiency remains a key limitation. Many verifiers have attempted to reduce verification overhead by leveraging advanced SMT (Satisfiability Modulo Theories) techniques [3]. However, general-purpose SMT solvers are prone to missing optimization opportunities specific to database verification.

Beyond this, two additional challenges make efficient verification even harder. First, all existing verifiers assume that transaction histories contain *unique write values*, i.e., each read can be deterministically matched to a single write. This greatly simplifies the task of inferring the internal execution order and serves as a key enabler of Elle's design in particular. Yet this assumption does not hold in many real-world applications, where different transactions often write the same value. For instance, in an e-commerce platform, multiple users may submit identical orders for the same quantity of an item; in social media, thousands of users may "like" the same post. Likewise, standard database benchmarks that closely reflect real-world workloads, such as Twitter [16] and RUBiS [32], include *duplicate write values* by default. In fact, in our black-box setting, a (cloud) database is unaware that it is under test; we act purely as an

*The corresponding author, Hengfeng Wei, is with College of Computer Science and Electronic Engineering, Hunan University. The initial draft of this work was prepared while Hengfeng Wei and Zhiheng Cai were with Software Institute, Nanjing University.

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785300

external monitor, collecting the execution results of real workloads processed by the system. Hence, supporting such general workloads is highly desirable and essential for practical applicability.

Moreover, achieving both soundness (no false alarms) and completeness (no missed bugs) without compromising efficiency remains challenging in verifying strong isolation. In particular, identifying all anomalies in a history is computationally expensive, yet critical: overlooking even a single anomaly can require significant effort to rediscover. This challenge is further exacerbated by recent findings [22, 25], which reveal that certain anomalies arise *only* when duplicate write values are present—precisely the scenarios exemplified in real-world workloads above. Hence, relying on existing verifiers with unique write values risks both false negatives and false positives, undermining the reliability of verification results.

Our Solution. We present a novel black-box verifier, VeriStrong, that checks database histories against the two most widely used strong isolation guarantees, i.e., Serializability and Snapshot Isolation. VeriStrong tackles the aforementioned challenges through an end-to-end verification pipeline. This includes (i) representing and encoding general database histories, (ii) establishing a sound and complete correspondence between the encoding and the verification algorithm, and (iii) optimizing checking efficiency using SMT techniques tailored for strong isolation verification.

At the core of our solution is a new formalism called *hyper-polygraphs*, which captures both certain transactional dependencies (e.g., session order) and uncertain ones (e.g., version order). In black-box settings, uncertainties can also arise from ambiguous read-from relations caused by duplicate write values—an aspect that existing formalisms, such as polygraphs [27, 36] and their variants [13, 33, 39], cannot adequately express. Hyper-polygraphs offer two key advantages. First, they provide a compact and expressive means of encoding transactional dependencies, making them particularly amenable to SMT-based verification. Second, they are generic, capable of representing a wide range of dependency-based isolation theories [1, 4, 6]. In this paper, we focus on Adya's theory [1], with another application discussed in Section 9.

Building on this formalism, we develop sound and complete encodings for verifying both strong isolation guarantees, providing a rigorous theoretical foundation for reliable SMT-based verification. This enables our verifier to accurately (re)discover isolation anomalies in production databases (e.g., MariaDB), particularly those arising from duplicate write values that prior tools fail to handle.

To make verification fast, our key insight is that leveraging workload-specific knowledge can greatly boost SMT solving efficiency. To this end, we propose two domain-specific optimizations. First, we offload part of the solver's effort to a lightweight preprocessing phase by proactively resolving small conflict patterns. This reduces expensive backtracking during solving by eliminating many infeasible search paths early and strikes a practical balance between preprocessing and solving costs, improving overall performance.

Second, we design a polarity picking strategy—i.e., deciding whether to explore a dependency—guided by a dynamically maintained pseudo-topological order during SMT solving. Unlike general-purpose solvers that make such decisions "blindly", we leverage known partial orders, such as session order and certain read-from edges, to align these decisions with the likely transaction schedule

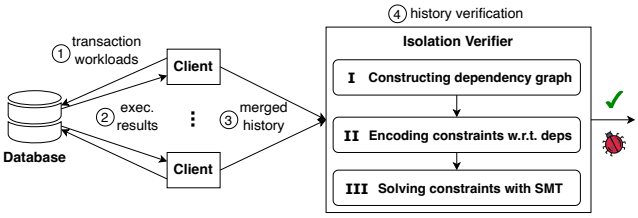


Figure 1: Workflow for verifying isolation guarantees in deployed databases. Steps II and III are specific to SMT-based approaches (see also Sections 4 and 6).

enforced by the database. This alignment steers the solver away from conflict-prone search paths, thereby enhancing efficiency.

Contributions. Overall, we make the following contributions.

(1) We introduce hyper-polygraphs, a new formalism that compactly characterizes isolation levels, including Serializability and Snapshot Isolation, over general database histories.
(2) We formally establish the soundness and completeness of our new characterizations, grounded in Adya's theory, providing a theoretical foundation for reliable SMT-based verification.
(3) We develop novel isolation verification algorithms by tailoring SMT solving to the characteristics of database workloads, incorporating optimizations that reduce solver overhead.
(4) We implement our approach in VeriStrong and assess it extensively on a variety of benchmarks. Results show that VeriStrong not only substantially outperforms state-of-the-art verifiers on the workloads they support, but also scales to large, general workloads beyond their reach, while maintaining high accuracy in detecting isolation anomalies.

This work complements recent advances in isolation verification along two dimensions. First, it complements prior efforts [19, 24] on black-box verification of weaker levels like Read Committed, which are known to be less complex to verify [4]. Second, it targets deployed databases, complementing recent work on verifying isolation in their designs [10, 21] and implementations [23].

We focus on Serializability as the running example throughout the paper, unless noted otherwise. We defer to [5, Appendix A] how our approach extends to verifying Snapshot Isolation.

2 BACKGROUND

Black-box verification of database isolation guarantees involves four key steps, as illustrated in Figure 1. In Step ①, clients issue transactional requests to the database. In Step ②, each client records the corresponding execution results, including the values read or written and the status of each transaction (either committed or aborted). Next, in Step ③, the logs from all clients are merged into a single history, which is then passed to an isolation verifier. Finally, in Step ④, the verifier analyzes the history to determine whether it satisfies the specified isolation guarantee.

How does the verifier make this decision? It constructs a dependency graph based on, e.g., Adya's theory [1] that is the *de facto* formalization of isolation guarantees. The verifier then searches the graph for cycles, which indicate violations of guarantees such as

Serializability. Next, we recall the formal definitions underlying the history verification steps I–III from [13].

Relations. A binary relation R over a given set A is a subset of $A \times A$, i.e., $R \subseteq A \times A$. For $a, b \in A$, we use $(a, b) \in R$ and $a \xrightarrow{R} b$ interchangeably. We use $R?$ and R^+ to denote the reflexive closure and the transitive closure of R, respectively. A relation $R \subseteq A \times A$ is *acyclic* if $R^+ \cap I_A = \emptyset$, where $I_A \triangleq \{(a, a) \mid a \in A\}$ is the identity relation on A. Given two binary relations R and S over the set A, we define their composition as $R\ ;\ S = \{(a, c) \mid \exists b \in A. a \xrightarrow{R} b \xrightarrow{S} c\}$. 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.

Transactions. We consider a distributed key-value store that manages a set of keys $\mathsf{Key} = \{x, y, z, \dots\}$; each key is associated with a value from a set Val. The set of operations, denoted by Op, consists of both read and write operations: $\mathsf{Op} = \{\mathsf{R}_\iota(x, v), \mathsf{W}_\iota(x, v) \mid \iota \in \mathsf{OpId}, x \in \mathsf{Key}, v \in \mathsf{Val}\}$, where OpId is the set of operation identifiers. For simplicity, operation identifiers may be omitted. We use _ to represent an irrelevant value in an operation, e.g., $\mathsf{R}(x, \_)$, which is implicitly existentially quantified.

Clients interact with the key-value store by issuing transactions.

Definition 2.1. A *transaction* is a pair (O, po), where $O \subseteq \mathsf{Op}$ is a finite, non-empty set of operations; $\mathrm{po} \subseteq O \times O$ is a strict total order, referred to as the *program order*, which indicates the execution order of operations within the transaction.

For a transaction T, we write $T \vdash \mathsf{W}(x, v)$ if T writes to the key x and v is the last written value, 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 define $\mathsf{WriteTx}_x = \{T \mid T \vdash \mathsf{W}(x, \_)\}$ to denote the set of transactions that write to x.

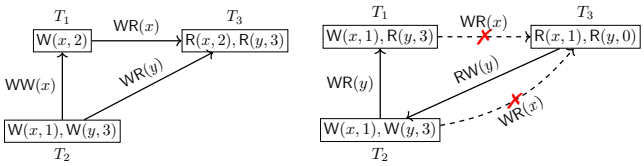
Histories. Transactions are grouped into client *sessions*, where each session consists of a sequence of transactions. We use *histories* to record the client-visible outcomes of these transactions.

Definition 2.2. A *history* is a pair $\mathcal{H} = (\mathcal{T}, \mathrm{SO})$, where $\mathcal{T}$ is a set of transactions with disjoint sets of operations; $\mathrm{SO} \subseteq \mathcal{T} \times \mathcal{T}$ is the *session order*, which is a union of strict total orders over disjoint sets of $\mathcal{T}$, each corresponding to a distinct client session.

In line with existing formal models [1, 4, 6], we consider only committed transactions in a history; aborted transactions are handled separately. We also assume that every history contains a special transaction $T_\perp$ that writes the initial values of all keys.[1] This transaction precedes all the other transactions in SO across sessions. Note that existing work commonly assumes UniqueValue histories, where each write to the same key assigns a distinct value, whereas real-world scenarios often involve DuplicateValue writes.

Dependency Graphs. A dependency graph extends a history by introducing three types of relations (or edges), WR, WW, and RW, each capturing a different kind of dependency between transactions. These relations underlies the formalization of isolation guarantees in the style of Adya [1, 6]. The WR relation connects a transaction that reads a value to the transaction that wrote it. The WW relation defines a strict total order, also referred to as the *version order* [1], among transactions that write to the same key. The RW relation is derived from WR and WW, associating a transaction that reads a value to the one that overwrites it based on the version order.

[1] In practice, we use multiple short, write-only transactions to initially populate the database. These transactions can be viewed as a single, logical write-only transaction.



(a) A serializable UniqueValue history.

(b) An unserializable DuplicateValue history.

Figure 2: Capturing Serializability via dependency graphs. Solid arrows denote known dependencies, whereas dashed arrows indicate uncertain ones.

Definition 2.3. A *dependency graph* is a tuple $\mathcal{G} = (\mathcal{T}, \mathrm{SO}, \mathrm{WR}, \mathrm{WW}, \mathrm{RW})$, where $(\mathcal{T}, \mathrm{SO})$ is a history and

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

We use $\exists!$ to denote unique existence. For any component of a dependency graph $\mathcal{G}$, such as WW, we write it as $\mathrm{WW}_\mathcal{G}$. When the key x in $T \xrightarrow{R(x)} S$ is irrelevant or clear from context, we write $T \xrightarrow{R} S$, where $R \in \{\mathrm{WR}, \mathrm{WW}, \mathrm{RW}\}$.

Characterizing Serializability. Serializability is characterized by the *existence* of an acyclic dependency graph, along with the internal consistency of each transaction, as axiomatized by Int [6]. The emphasis on existence is crucial: as external observers of a black-box database, verifiers cannot directly observe the internal execution order, e.g., WW between writes. Instead, they must infer a possible internal schedule. The existence of an acyclic dependency graph provides a witness that explains how the database could have scheduled the transactions in a way that satisfies Serializability.

In addition, the Int axiom ensures that a read on a key returns the same value as the most recent proceeding access—either a write or a read—to that key within the same transaction.

Theorem 2.4. *For a history* $\mathcal{H} = (\mathcal{T}, \mathrm{SO})$,

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

Example 2.5. Figure 2 illustrates the dependency-graph-based characterization of Serializability using two example histories. For simplicity, we omit $T_\perp$ in both cases. Figure 2a shows (the existence of) a serializable dependency graph with edges like $T_2 \xrightarrow{\mathrm{WW}(x)} T_1$, which are constructed from a UniqueValue history.

In contrast, it is impossible to build an acyclic dependency graph from the DuplicateValue history in Figure 2b. Specifically, for key y, we obtain the dependencies $T_2 \xrightarrow{\mathrm{WR}(y)} T_1$ and $T_3 \xrightarrow{\mathrm{RW}(y)} T_2$ (due to $T_\perp \xrightarrow{\mathrm{WR}(y)} T_3$ and $T_\perp \xrightarrow{\mathrm{WW}(y)} T_2$). Now consider the transaction from which $\mathrm{R}(x, 1)$ in T_3 reads its value. This leads to two possible WR edges (shown as dashed arrows): $T_1 \xrightarrow{\mathrm{WR}(x)} T_3$ and $T_2 \xrightarrow{\mathrm{WR}(x)} T_3$. However, in both cases, the resulting dependency graph contains a cycle.

Polygraphs. A dependency graph extending a history represents *one* possible interpretation of the dependencies among the involved transactions, which may or may not satisfy Serializability. To capture *all* possible dependency scenarios—allowing a verifier to check whether any of them satisfies Serializability—state-of-the-art tools [13, 33, 39] utilize *polygraphs* [27, 36]. Intuitively, a polygraph can be seen as a compact representation of a family of dependency graphs, each corresponding to a different possible internal execution consistent with the observed history.

Definition 2.6. A polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ associated with a history $\mathcal{H} = (\mathcal{T}, \mathrm{SO})$ is a directed labeled graph $(\mathcal{V}, \mathcal{E})$ called the *known graph*, together with a set C of *constraints*, such that

- $\mathcal{V}$ corresponds to all the transactions in $\mathcal{H}$;
- $\mathcal{E} = \{(T, S, \mathrm{SO}) \mid T \xrightarrow{\mathrm{SO}} S\} \cup \{(T, S, \mathrm{WR}) \mid T \xrightarrow{\mathrm{WR}} S\}$, where SO and WR, when used as the third component in a tuple, serve as edge labels (i.e., types of dependencies); and
- $C = \{(T_k, T_i, \mathrm{WW}), (T_j, T_k, \mathrm{RW}) \mid (T_i \xrightarrow{\mathrm{WR}(x)} T_j) \wedge T_k \in \mathrm{WriteTx}_x \wedge T_k \neq T_i \wedge T_k \neq T_j\}$.

Example 2.7. According to the above definition, the polygraph associated with the history in Figure 2a consists of a known graph with nodes $\{T_1, T_2, T_3\}$ and edges $\{T_1 \xrightarrow{\mathrm{WR}} T_3, T_2 \xrightarrow{\mathrm{WR}} T_3\}$, along with a set of two constraints $\{(T_2, T_1, \mathrm{WW}), (T_3, T_2, \mathrm{RW})\}$ capturing the uncertainty in version order between T_1 and T_2.

These constraints are then encoded as a SAT formula and solved using an SMT solver that supports theories like graph acyclicity, as we will see in Section 4. The acyclic dependency graph that satisfy Serializability in Figure 2a can be viewed as a solution produced by this solving process.

3 HYPER-POLYGRAPHS

A dependency graph represents one possible execution of database transactions. A polygraph captures a family of such graphs by fixing the WR relation—an assumption that holds under UniqueValue, where each written value is unique and thus unambiguously matched by reads—while allowing uncertainty in the WW relation. However, this assumption, relied upon by all existing verifiers, does not always hold in practice, particularly under general workloads where the same value may be written multiple times, making read-from mappings ambiguous.

To address this representation problem, we introduce *hyper-polygraphs* as a generalization of polygraphs that additionally account for uncertainty in WR. Intuitively, a hyper-polygraph is a family of polygraphs, each corresponding to a distinct resolution of the WR relations, i.e., one polygraph per read-from mapping.

Definition 3.1. A hyper-polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ for a history $\mathcal{H} = (\mathcal{T}, \mathrm{SO})$ is a directed labeled graph $(\mathcal{V}, \mathcal{E})$, referred to as the known graph, together with *a pair of constraint sets* $C = (C^{\mathrm{WW}}, C^{\mathrm{WR}})$, where

- $\mathcal{V}$ is the set of nodes, corresponding to the transactions in $\mathcal{H}$;
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathrm{Type} \times \mathrm{Key}$ is a set of edges, where each edge is labeled with a dependency type from $\mathrm{Type} = \{\mathrm{SO}, \mathrm{WR}, \mathrm{WW}, \mathrm{RW}\}$ and a key from Key;[2]
- C^{WW} is a constraint set over uncertain version orders, defined as $C^{\mathrm{WW}} = \left\{\left\{T \xrightarrow{\mathrm{WW}(x)} S, S \xrightarrow{\mathrm{WW}(x)} T\right\} \mid T \in \mathrm{WriteTx}_x \wedge S \in \mathrm{WriteTx}_x \wedge T \neq S\right\}$; and
- C^{WR} is a constraint set over uncertain read-from mappings, defined as $C^{\mathrm{WR}} = \left\{ \bigcup_{T_i \vdash \mathrm{W}(x,v)} \{T_i \xrightarrow{\mathrm{WR}(x)} S\} \mid S \vdash \mathrm{R}(x, v)\right\}$.

Given two transactions T and S, a type $\mathsf{T} \in \mathrm{Type}$, and a key $x \in \mathrm{Key}$, we also write the edge (T, S, T, x) as $T \xrightarrow{\mathsf{T}(x)} S$.

Example 3.2. Consider the general history $\mathcal{H}$ shown in Figure 3a. It consists of three transactions T_1, T_2, and T_3, with a session order edge $T_2 \xrightarrow{\mathrm{SO}} T_3$. Both T_1 and T_2 write the same value 1 to the same key x. The hyper-polygraph $\mathcal{G}$ constructed from $\mathcal{H}$ includes the following two constraint sets:

- $C^{\mathrm{WW}} = \left\{\left\{T_1 \xrightarrow{\mathrm{WW}(x)} T_2, T_2 \xrightarrow{\mathrm{WW}(x)} T_1\right\}\right\}$, representing the uncertainty in version order between T_1 and T_2 (shown as dashed edges in Figure 3b);
- $C^{\mathrm{WR}} = \left\{(T_1 \xrightarrow{\mathrm{WR}(x)} T_3,\ T_2 \xrightarrow{\mathrm{WR}(x)} T_3)\right\}$, representing the uncertainty in read-from mappings for T_3 (also shown as dashed edges in Figure 3b).

Characterizing Serializability using Hyper-Polygraphs. A hyper-polygraph for a DuplicateValue history can be viewed as a family of dependency graphs that are *compatible with* it, where each graph resolves the constraints by selecting exactly one WW or WR edge from the C^{WW} and C^{WR} constraint sets, respectively.

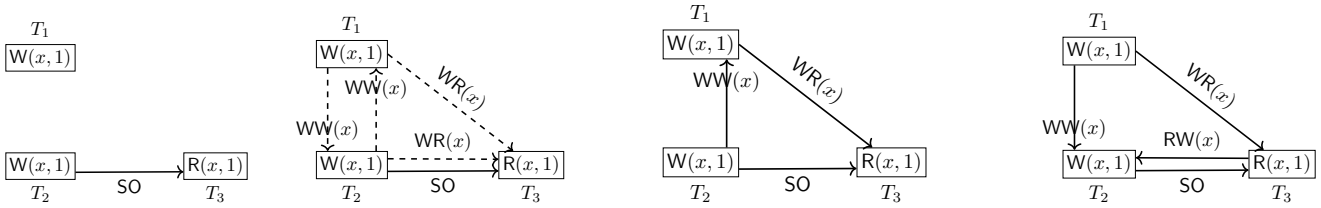
Definition 3.3. A directed labeled graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ is compatible with a hyper-polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ if

- $\mathcal{V}' = \mathcal{V}$;
- $\mathcal{E}' \supseteq \mathcal{E}$ such that $\forall x \in \mathrm{Key}.\ \forall T, T', S \in \mathcal{V}'.\ (T', T, \mathrm{WR}, x) \in \mathcal{E}' \wedge (T', S, \mathrm{WW}, x) \in \mathcal{E}' \implies (T, S, \mathrm{RW}, x) \in \mathcal{E}'$;
- $\forall C \in C^{\mathrm{WW}}.\ |\mathcal{E}' \cap C| = 1$; and
- $\forall C \in C^{\mathrm{WR}}.\ |\mathcal{E}' \cap C| = 1$.

Example 3.4. Figures 3c and 3d depict two graphs compatible with the hyper-polygraph $\mathcal{G}$ of the history $\mathcal{H}$. In Figure 3c, T_3 reads x from T_1, which overwrites the value written by T_2. In contrast, in Figure 3d, T_3 reads x from T_1, but the value is overwritten by T_2, resulting in $T_3 \xrightarrow{\mathrm{RW}(x)} T_2$.

Based on Theorem 2.4, we obtain the following hyper-polygraph-based characterization of Serializability; the proof is provided in [5, Appendix B].

[2] For edges of type SO, the key component is irrelevant.



(a) General history $\mathcal{H}$. (b) Hyper-polygraph $\mathcal{G}$ of $\mathcal{H}$. (c) An acyclic graph compatible with $\mathcal{G}$. (d) A cyclic graph compatible with $\mathcal{G}$.

Figure 3: A general history $\mathcal{H}$, along with its hyper-polygraph $\mathcal{G}$ and two compatible graphs.

Theorem 3.5. *A history $\mathcal{H}$ satisfies* Serializability *if and only if $\mathcal{H} \models$ Int and there exists an acyclic graph compatible with the hyper-polygraph of $\mathcal{H}$.*

Example 3.6. The graph compatible with $\mathcal{G}$ in Figure 3d contains a cycle: $T_3 \xrightarrow{\text{RW}(x)} T_2 \xrightarrow{\text{SO}} T_3$. In contrast, the compatible graph in Figure 3c is acyclic. According to Theorem 3.5, we conclude that $\mathcal{H}_3$ satisfies Serializability.

4 OFF-THE-SHELF SMT SOLVING: A BASELINE

To begin, we present a strong baseline approach for verifying Serializability using MonoSAT [3], an off-the-shelf SMT solver optimized for checking graph properties such as acyclicity. MonoSAT serves as the core engine for all state-of-the-art SMT-based isolation verifiers, including Cobra, PolySI, and Viper.

Given a history $\mathcal{H}$ and following the workflow in Figure 1, we first construct its hyper-polygraph. This involves extracting the transaction set $\mathcal{T}$, the session order SO, and unique WR dependencies for the known graph, along with possible WW and WR dependencies as constraints. We then focus on the two key steps: *encoding* the hyper-polygraph into SAT formulas (Section 4.1) and *solving* them with MonoSAT (Section 4.2). We illustrate each step using the example history $\mathcal{H}$ shown in Figure 3a.

Preliminaries. In propositional logic, a boolean variable v can take the value true or false. A *literal* l refers to a variable v or its negation $\neg v$. A *clause* C is a disjunction of one or more literals, e.g., $C = l_1 \vee l_2 \vee \cdots \vee l_n$. A *formula* $\mathcal{F}$ is constructed using literals and the logical connectives $\wedge$, $\vee$, $\neg$, or $\implies$. Typically, we represent formulas in conjunctive normal form (CNF), where $\mathcal{F}$ is a conjunction of multiple clauses: $\mathcal{F} = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. An *assignment* of a variable v is either true or false, represented as v or $\neg v$, respectively. A *model* $\mathcal{M}$ for a formula $\mathcal{F}$ is a set of assignments to the boolean vairables appearing in $\mathcal{F}$ that makes $\mathcal{F}$ true. If $\mathcal{M}$ does not specify assignments for all variables in $\mathcal{F}$, it is referred to as a *partial model*.

4.1 Encoding

We represent the existence of each edge in the hyper-polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ using a boolean variable. First, the known graph of $\mathcal{G}$ is encoded by the formula

$$\Phi_{KG} = \text{SO}_{2,3}$$

where the variable $\text{SO}_{2,3}$, set to true, indicates the presence of the SO edge $T_2 \xrightarrow{\text{SO}} T_3$.

Second, the C^{WW} constraints of $\mathcal{G}$ are encoded by the formula

$$\Phi_{C^{\text{WW}}} = (\text{WW}^x_{1,2} \vee \text{WW}^x_{2,1}) \wedge (\neg\text{WW}^x_{1,2} \vee \neg\text{WW}^x_{2,1})$$

where the variables $\text{WW}^x_{1,2}$ and $\text{WW}^x_{2,1}$ represent the existence of the edges $T_1 \xrightarrow{\text{WW}(x)} T_2$ and $T_2 \xrightarrow{\text{WW}(x)} T_1$, respectively. This formula enforces that exactly one of the two variables is assigned true, ensuring a total order between the two conflicting writes.

Moreover, the C^{WR} constraints of $\mathcal{G}$ are encoded by the formula

$$\Phi_{C^{\text{WR}}} = (\text{WR}^x_{1,3} \vee \text{WR}^x_{2,3}) \wedge (\neg\text{WR}^x_{1,3} \vee \neg\text{WR}^x_{2,3})$$

where the variables $\text{WR}^x_{1,3}$ and $\text{WR}^x_{2,3}$ denote the presence of the edges $T_1 \xrightarrow{\text{WR}(x)} T_3$ and $T_2 \xrightarrow{\text{WR}(x)} T_3$, respectively. This formula enforces that exactly one of these edges holds, ensuring a unique read-from relation for T_3.

Finally, the encoding of the derivation rule for RW edges in $\mathcal{G}$ is captured by the formula

$$\begin{aligned}\Phi_{\text{RW}} = (\text{WW}^x_{1,2} \wedge \text{WR}^x_{1,3} \implies \text{RW}^x_{3,2}) \wedge (\text{WW}^x_{2,1} \wedge \text{WR}^x_{2,3} \implies \text{RW}^x_{3,1}) \\ \Leftrightarrow (\neg\text{WW}^x_{1,2} \vee \neg\text{WR}^x_{1,3} \vee \text{RW}^x_{3,2}) \wedge (\neg\text{WW}^x_{2,1} \vee \neg\text{WR}^x_{2,3} \vee \text{RW}^x_{3,1})\end{aligned}$$

where $\text{RW}^x_{3,1}$ and $\text{RW}^x_{3,2}$ represent the existence of the $\text{RW}(x)$ edges $T_3 \xrightarrow{\text{RW}(x)} T_1$ and $T_3 \xrightarrow{\text{RW}(x)} T_2$, respectively. This encoding ensures that an RW edge is derived only when both the corresponding WW and WR edges are present.

Overall, the complete encoding for $\mathcal{G}$ is given by

$$\Phi_{\mathcal{G}} = \Phi_{KG} \wedge \Phi_{C^{\text{WW}}} \wedge \Phi_{C^{\text{WR}}} \wedge \Phi_{\text{RW}}.$$

4.2 Solving

Each satisfying assignment of $\Phi_{\mathcal{G}}$ corresponds to a graph $\mathcal{G}'$ that is compatible with $\mathcal{G}$. To ensure that $\mathcal{G}'$ is acyclic, we assert the predicate $\text{Acyclic}(\mathcal{G}')$ in MonoSAT. Hence, the history $\mathcal{H}$ is serializable if and only if the formula $\Phi_{\mathcal{G}} \wedge \text{Acyclic}(\mathcal{G}')$ is satisfiable.

MonoSAT implements the *Conflict-Driven Clause Learning with Theory* (CDCL(T)) framework [26] to decide the satisfiability of SAT formulas augmented with theory predicates. It combines a *SAT solver*, which searches for a satisfying assignment to the boolean formula (e.g., $\Phi_{\mathcal{G}}$), with a *theory solver*, which ensures that the asserted predicates (e.g., $\text{Acyclic}(\mathcal{G}')$) hold under the current assignment. As the SAT solver assigns variables, the theory solver incrementally updates a graph G^* by adding edges for variables set to true. If a predicate is violated, a conflict clause is generated and learned by the SAT solver to prevent revisiting the same conflict.

The solving process may vary depending on the order in which variables are assigned. Figure 4 illustrates one such procedure for

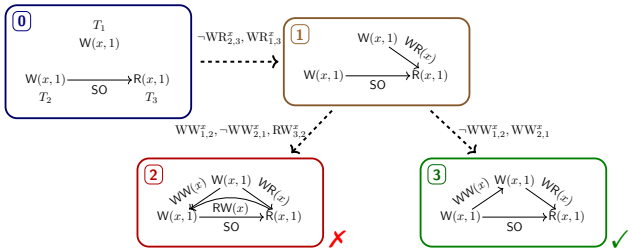


Figure 4: An example solving process for $\mathcal{H}$ in Figure 3.

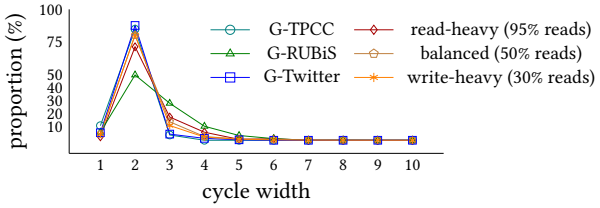


Figure 5: Distribution of minimal cycle widths in conflicts.

$\mathcal{G}$, with the graph G^* progressing through four stages.[3] Initially, only $\mathrm{SO}_{2,3}$ is assigned true, yielding 0. Suppose the SAT solver next assigns $\neg\mathrm{WR}^x_{2,3}$. Then $\mathrm{WR}^x_{1,3}$ must be true to satisfy the clause $\mathrm{WR}^x_{1,3} \vee \mathrm{WR}^x_{2,3}$. This is also known as *unit propagation* [7] that occurs when a clause becomes unit, i.e., only one literal remains unassigned ($\mathrm{WR}^x_{1,3}$ in this case), forcing the literal to be assigned true to satisfy the clause. The theory solver adds the edge $T_1 \xrightarrow{\mathrm{WR}(x)} T_3$ to G^*, leading to 1. No cycles are detected at this point.

Next, assigning $\mathrm{WW}^x_{1,2}$ triggers $\neg\mathrm{WW}^x_{2,1}$ due to mutual exclusion, and then $\mathrm{RW}^x_{3,2}$ via the clause $\neg\mathrm{WW}^x_{1,2} \vee \neg\mathrm{WR}^x_{1,3} \vee \mathrm{RW}^x_{3,2}$, which leads to the updated graph G^* in 2. At this point, the theory solver detects a cycle: $T_2 \xrightarrow{\mathrm{SO}} T_3 \xrightarrow{\mathrm{RW}(x)} T_2$ (see also Figure 3d). It then returns a conflict clause $\neg\mathrm{RW}^x_{3,2} \vee \neg\mathrm{SO}_{2,3}$ to the SAT solver, indicating that removing either edge would break the cycle. The SAT solver then backtracks from $\mathrm{WW}^x_{1,2}$, assigns its negation $\neg\mathrm{WW}^x_{1,2}$, and propagates $\mathrm{WW}^x_{2,1}$. This results in 3, where all variables are assigned and no cycles are present. Hence, an acyclic compatible graph is found (see also Figure 3c), and the history $\mathcal{H}$ is serializable.

5 WORKLOAD CHARACTERISTICS FOR SMT SOLVING

General-purpose SMT solvers like MonoSAT are agnostic to the characteristics of database workloads, thereby missing opportunities for targeted optimization. In contrast, we hypothesize that leveraging workload-specific characteristics can significantly improve solving efficiency for isolation verification. This section presents two key observations that motivate our tailored SMT strategies.

5.1 Prevalence of 2-width Cycles

A key bottleneck arises in the solving stage from "bad" guesses that introduce cycles in the dependency graph. When this occurs, the solver must backtrack and explore alternative choices, which can be costly. To mitigate this, prior work [13, 33, 39] introduces *pruning*, a preprocessing technique that identifies and precludes infeasible WW constraints by analyzing reachability *before* the actual solving process starts. For example, consider a WW constraint $\{T_1 \xrightarrow{\mathrm{WW}(x)} T_2, T_2 \xrightarrow{\mathrm{WW}(x)} T_1\}$. If T_2 is already reachable from T_1, then choosing $T_2 \xrightarrow{\mathrm{WW}(x)} T_1$ would result in a cycle. Hence, this choice can be safely pruned, eliminating the need for the solver to explore it further. In the context of SAT solving, pruning corresponds to adding a conflict clause with a single WW variable, e.g., $\neg\mathrm{WW}^x_{2,1}$ in this case, to the encoding for $\Phi_{\mathcal{G}}$ (Section 4.1).

Essentially, pruning shifts the cost of resolving search-time conflicts to a pre-solving analysis phase. This raises a natural question: can more of the solving overhead be offloaded to preprocessing? In other words, is there a sweet spot that balances the cost of precomputation with the benefit of reduced solving effort, thereby improving overall performance?

To explore this, we adapt the notion of *cycle width* [9, 12], which we define as the number of *unique* WW and WR variables appearing in a conflict clause. While conventional pruning relies on unit clauses (i.e., 1-width cycles), broader pruning can be achieved by encoding larger cycles using multi-variable clauses. To avoid redundancy, we count each variable only once and disregard SO variables, which are always true, and RW variables, which can be derived from WW and WR.

We empirically measure the widths of conflict cycles encountered during solving across six representative benchmarks, including TPC-C and YCSB-like transaction workloads with varying read proportions (see Section 7.1 for details). Each benchmark consists of 1000 transactions. For each conflict, we record the *minimal* cycle width, as it may correspond to multiple cycles depending on the search path. As shown in Figure 5, 2-width cycles dominate across all benchmarks. This suggests that many conflicts could be prevented by pre-encoding 2-variable clauses. We leverage this insight in Section 6.1, incorporating 2-width cycle encodings into the SAT formula $\Phi_{\mathcal{G}}$ to strike a balance between pre-solving effort and solving efficiency (further validated in Section 7.3.3).

5.2 Polarity as Schedule Reconstruction

Another major source of inefficiency in SMT solving stems from poor polarity choices when assigning decision variables. A key insight we introduce is that each polarity decision—whether to include or exclude a dependency edge—implicitly defines a transaction schedule. Suboptimal choices can conflict with the actual, though hidden (due to the black-box setting), schedule enforced by the database, leading to dependency cycles and costly backtracking.

General-purpose solvers such as MonoSAT treat all variables uniformly and assign polarity without regard to the execution context. In contrast, we observe that many dependencies, including session order SO and uniquely determined read-from relation WR, are already known from the observable history and can be leveraged to guide more informed polarity decisions. Ideally, if the internal transaction schedule were accessible, one could align polarity assignments perfectly to avoid conflicts. While this is infeasible in

[3] We also illustrate in [5, Appendix C] how the SAT solver and the theory solver collaborate in CDCL(T) for this example.

black-box settings, the visible dependencies still capture a significant portion of the execution order. We hypothesize that exploiting this partial order can substantially reduce conflict rates and backtracking overhead. The following example illustrates this intuition.

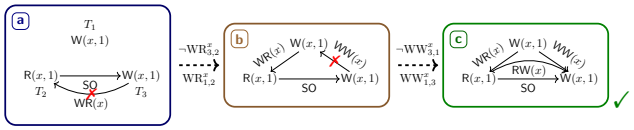


Figure 6: Guiding polarity picking via known dependencies.

Example 5.1. In Figure 6, the solver must resolve two constraints: $\{T_1 \xrightarrow{\mathrm{WR}(x)} T_2, T_3 \xrightarrow{\mathrm{WR}(x)} T_2\}$ and $\{T_1 \xrightarrow{\mathrm{WW}(x)} T_3, T_3 \xrightarrow{\mathrm{WW}(x)} T_1\}$. Initially, in a, the solver must decide between two possible read-from candidates for T_2. By recognizing the known session order $T_2 \xrightarrow{\mathrm{SO}} T_3$, the solver correctly assigns $\mathrm{WR}^x_{3,2}$ = false, avoiding a potential cycle between T_2 and T_3. The solver then proceeds to b, where it must resolve the WW constraint. Again, guided by the inferred dependency $T_1 \xrightarrow{\mathrm{WR}(x)} T_3$, it avoids introducing a cycle, e.g., $T_1 \xrightarrow{\mathrm{WR}(x)} T_2 \xrightarrow{\mathrm{SO}} T_3 \xrightarrow{\mathrm{WW}(x)} T_1$, by choosing $\mathrm{WW}^x_{3,1}$ = false.

This example underscores the importance of polarity decisions: in large histories, a single incorrect polarity choice may introduce an edge that contradicts the actual schedule. Consequently, the solver is misled into exploring an exponentially larger search space, incurring substantial overhead due to cascading conflicts and backtracking. Motivated by this insight, the next section introduces a polarity picking strategy—integrated into our dedicated SMT solver—that exploits the observable partial order in the execution history. We evaluate its impact in Section 7.3.4.

6 VeriStrong with Tailored SMT Solving

VeriStrong builds upon the baseline algorithm described in Section 4. Motivated by our observations in Section 5, it incorporates two key optimizations: efficient handling of 2-width cycles (Section 6.1) and polarity picking for decision variables (Section 6.2).

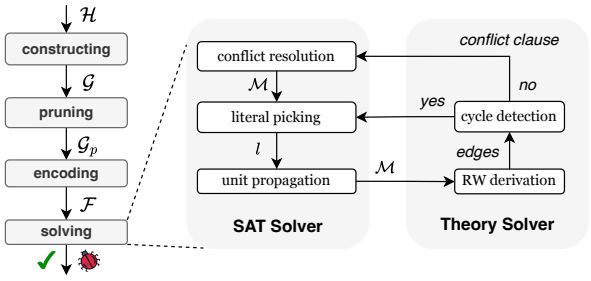


Figure 7: Workflow of VeriStrong.

Figure 7 shows the high-level workflow of VeriStrong, comprising four main stages: (i) *constructing* the hyper-polygraph $\mathcal{G}$ from the input history $\mathcal{H}$, (ii) *pruning* infeasible constraints in $\mathcal{G}$, (iii) *encoding* the pruned graph $\mathcal{G}_p$ into a SAT formula $\mathcal{F}$, and (iv) *solving* $\mathcal{F}$. The corresponding pseudo-code is given in Algorithm 1, which we refer to alongside the figure to explain the procedure.

Algorithm 1 The VeriStrong algorithm

$\mathcal{H}$: the input history
$G^* = (\mathcal{V}, E^*)$: the current graph maintained in the theory solver; initialized as the hyper-polygraph $\mathcal{G}$ constructed from $\mathcal{H}$
Vars: a set of boolean variables managed by the SAT solver; initially $\emptyset$
Clauses: a set of clauses managed by the SAT solver; initially $\emptyset$
Both *Vars* and *Clauses* are constructed in Encode (Line 5).

```
1:  function VerifySER(H)
2:      G ← Construct(H)                          ▷ Alg. 4 in [5, Appendix D]
3:      if ¬Prune(G)                              ▷ Alg. 5 in [5, Appendix D]; see also Section 6.1.1
4:          return false
5:      F ← Encode(G_p)                           ▷ Alg. 7 in [5, Appendix D]; see also Section 6.1.2
6:      return Solve(F, G_p)

7:  function Solve(F, G)
8:      for T --T(x)--> S ∈ E
9:          E* ← E* ∪ {T → S}
10:     return SatSolve(F, G)

11: function SatSolve(F, G)
12:     M_total ← ∅
13:     while |M_total| ≠ |Vars|
14:         v ← ChooseDecisionVar(M_total, Vars)
15:         l ← PickPolarity(v)                   ▷ Alg. 3 in Section 6.2
16:         M ← UnitPropagate(l, F)
17:         (ret, conflict_clause) ← TheorySolve(M)
18:         if ¬ret                               ▷ a conflict is detected
19:             if ¬ResolveConflict(M_total, M, conflict_clause)
20:                 return UNSAT
21:             else
22:                 M_total ← Backtrack(M_total, M, conflict_clause)
23:                 continue, goto Line 13
24:         M_total ← M_total ∪ M
25:     return SAT

26: function TheorySolve(M)
27:     edges ←DeriveRWEdges(M)
28:     if (E* ∪ edges) is cyclic                 ▷ use the PK algorithm [28]
29:         conflict_clause ← GenConflictClause(E*, edges)
30:         return (false, conflict_clause)
31:     else
32:         E* ← E* ∪ edges
33:         return (true, null)
```

The reminder of Algorithm 1 details the solving phase, specifically the interaction between the SAT frontend (SatSolve, Line 11) and the theory backend (TheorySolve, Line 26). Let G^* denote the current known graph maintained in the theory solver. Initially, G^* is set to the hyper-polygraph $\mathcal{G}$ constructed from $\mathcal{H}$. In each iteration, the SAT solver selects a decision variable v (Line 14) and its polarity, yielding a decision literal l (Line 15). This literal is then unit propagated, producing a partial assignment $\mathcal{M}$ (Line 16) that is passed to the theory solver for conflict checking (Line 17).

Specifically, the theory solver first derives RW edges from the current WW and WR edges (Line 27) and checks whether the updated graph G^* contains any cycles (Line 28). If a cycle is found, it generates a conflict clause and returns it to the SAT solver (Line 30). The SAT solver then resolves the conflict, either terminating with UNSAT (Line 20) or backtracking and updating $\mathcal{M}$ (Line 22). If no cycle is detected, the solver continues by selecting the next decision variable. Once all variables have been assigned without encountering conflicts (Line 13), the solver reports SAT (or serializable).

Next, we discuss two key optimizations in detail, while deferring the formal proofs of their correctness to [5, Appendix E]. We conclude this section with a complexity analysis of Algorithm 1.

6.1 Small-Width Cycle Preprocessing

This optimization aims to shift the burden of handling conflicts within the solver to a lightweight pre-solving analysis that focuses on small-width cycles. Intuitively, this preprocessing step acts as a "sanity check" before solving: it rules out choices that are guaranteed to fail (i.e., those that create cycles as exemplified in Section 5.1), preventing the solver from wasting effort on them later. It consists of two components: (i) an aggressive pruning phase that eliminates all 1-width cycles through reachability analysis, and (ii) a proactive encoding of 2-width cycles into the SAT formula, enabling early conflict detection via unit propagation during solving.

6.1.1 Eliminating 1-Width Cycles. We extend the pruning techniques from prior work [13, 33, 39], which focus on WW constraints, by also pruning WR constraints and their derived RW counterparts. The following examples illustrate the core ideas of our approach.

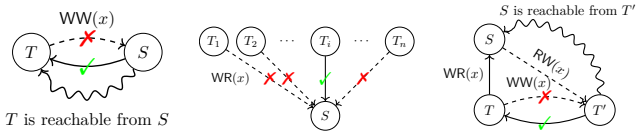
Case I: Pruning WW *Constraints.* Consider the two transactions T and S in Figure 8a, both writing to the same key. If T is already reachable from S in the current known graph, then adding $T \xrightarrow{\mathrm{WW}(x)} S$ would introduce a 1-width cycle. This infeasible option can thus be pruned; the valid edge $S \xrightarrow{\mathrm{WW}(x)} T$ is added to the known graph.

Case II: Pruning WR *Constraints.* We can similarly prune a WR edge if its inclusion would introduce a 1-width cycle in the known graph. Consider the scenario in Figure 8b, where a transaction S reads from key x and has n candidate writers $T_1, \ldots, T_n$. If $n-1$ of these candidates have been eliminated due to cycle formation, the remaining writer must be the only feasible source. In this case, we can safely add the edge $T_i \xrightarrow{\mathrm{WR}(x)} S$ to the known graph.

Case III: Pruning via Derived RW *Edges.* Derived edges may also lead to cycles. In Figure 8c, assume that adding $T \xrightarrow{\mathrm{WW}(x)} T'$ induces $S \xrightarrow{\mathrm{RW}(x)} T'$ (via $T \xrightarrow{\mathrm{WR}(x)} S$). If S is already reachable from T', this creates a cycle, so $T \xrightarrow{\mathrm{WW}(x)} T'$ is pruned.

VeriStrong iteratively applies this pruning process until the known graph stabilizes, i.e., all 1-width cycles have been eliminated. The complete pseudo-code is provided in [5, Appendix D].

6.1.2 Encoding 2-width Cycles. To prevent conflicts caused by 2-width cycles during solving, VeriStrong encodes them into SAT formulas. The corresponding pseudo-code is provided in Algorithm 2. We consider two scenarios, as also illustrated in Figure 9.

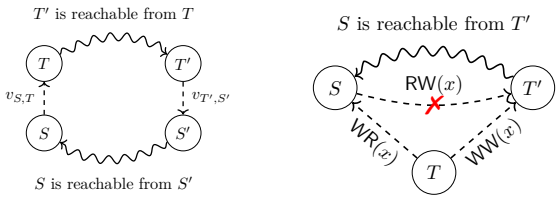


(a) Pruning WW **(b) Pruning WR** **(c) Pruning via RW**

Figure 8: Cases for pruning 1-width cycles.

Algorithm 2 Encoding 2-width cycles

```
1: function EncodingCyclesOfWidth2(V, E)
2:     R ← Reachability(V, E)
3:     for v1, v2 ≠ v1 ∈ Vars
4:         if CanonicalCycle(v1, v2, R) ∨ RWCycle(v1, v2, R)
5:             Clauses ← Clauses ∪ {¬v1 ∨ ¬v2}
6: function CanonicalCycle(v1, v2, R)
7:     let (S, T) ← v1, (T', S') ← v2
8:     return (T, T') ∈ R ∧ (S, S') ∈ R
9: function RWCycle(v1, v2, R)
10:    return ∃x. ∃T, S, T'. T' ≠ T ∧ (T', S) ∈ R ∧ ((v1 ≜ WW^x_{T,T'} ∧
       v2 ≜ WR^x_{T,S}) ∨ (v2 ≜ WW^x_{T,T'} ∧ v1 ≜ WR^x_{T,S}))
```



(a) Canonical cycle **(b) Derived RW cycle**

Figure 9: Cases for encoding 2-width cycles.

Case I: Canonical 2-Width Cycles. This case illustrates a canonical 2-width cycle formed by two known paths, as shown in Figure 9a. Let $T \rightsquigarrow T'$ and $S' \rightsquigarrow S$ be known dependencies in the graph, where $\rightsquigarrow$ denotes reachability between transactions. Adding both candidate edges $S \rightarrow T$ and $T' \rightarrow S'$ would complete a cycle. We prevent this by adding the clause $\neg v_{S,T} \vee \neg v_{T',S'}$ to the formula, where $v_{S,T}$ and $v_{T',S'}$ are the corresponding boolean variables.

Case II: Derived RW *Cycles.* In Figure 9b, assume $T' \rightsquigarrow S$ is known. Adding both $T \xrightarrow{\mathrm{WR}(x)} S$ and $T \xrightarrow{\mathrm{WW}(x)} T'$ would derive $S \xrightarrow{\mathrm{RW}(x)} T'$, forming a cycle. To avoid this, we encode the clause: $\neg \mathrm{WW}^x_{T,T'} \vee \neg \mathrm{WR}^x_{T,S}$.

While longer cycles could, in theory, also be encoded, enumerating clauses over k boolean variables incurs a time complexity of $O(|Vars|^k)$, making it impractical for large k. Our analysis shows that 2-width clauses capture the majority of conflicts effectively (Figure 15). Hence, we focus on encoding only 2-width cycles, striking a balance between preprocessing overhead and solving efficiency.

6.2 Order-Guided Polarity Picking

VeriStrong leverages the current partial order of the dependency graph to guide polarity picking, as motivated in Section 5.2. To

achieve this, we incorporate the *pseudo-topological order* maintained by the PK algorithm [28], which is originally developed for dynamic cycle detection and also adopted by MonoSAT. The full polarity picking procedure is given in Algorithm 3.

Algorithm 3 Pseudo-topological-order-guided polarity picking

```
1: function PickPolarity(v)
2:     let (from, to) ← v
3:     if level(from) < level(to)
4:         return v                          ▷ positive polarity
5:     else
6:         return ¬v                         ▷ negative polarity
```

The PK algorithm dynamically maintains a level $level(\cdot)$ for each vertex, which approximates its position in a topological ordering. When a candidate edge $(from, to)$ is considered, the algorithm first checks whether $level(from) < level(to)$. If so, the edge is guaranteed not to introduce a cycle and is immediately accepted. If not, the algorithm performs a reachability check to ensure that no cycle would be formed, and if the edge is added, it updates the levels of affected vertices to maintain consistency.

This pseudo-topological order reflects the known partial order in the current graph: every edge included satisfies $level(from) < level(to)$. We exploit this order to guide polarity selection. For a decision variable v representing a candidate edge $(from, to)$, we assign it a *positive polarity* (i.e., include the edge) when $level(from) < level(to)$, and a *negative polarity* (i.e., exclude the edge) otherwise.

6.3 Complexity Analysis

Constructing. Let $\|C\| = \sum_{cons \in C^{\mathrm{WW}} \cup C^{\mathrm{WR}}} |cons|$. The construction phase requires enumerating all edges in C, yielding a time complexity of $O(\|C\|)$. Let k be the number of keys and n the number of transactions. In the worst case, $O(\|C\|) = O(kn^2)$.

Pruning. Pruning proceeds in iterative passes. Each pass first computes reachability and then enumerates constraints in C to test the three pruning cases shown in Figure 8. Computing reachability costs $O\big(|\mathcal{V}| \cdot |\mathcal{E}|\big)$, where $(\mathcal{V}, \mathcal{E})$ is the known graph. With reachability cached, testing each pruning case costs only $O(1)$. The main bottleneck is enumerating possible RW edges, which can be characterized by the number of tuples (T, T', S, x) with $T, T', S \in \mathcal{T}, x \in \mathsf{Key}, T, T' \vdash \mathsf{W}(x, \_)$ and $S \vdash \mathsf{R}(x, \_)$. This costs $O\big(\sum_{x \in \mathsf{Key}} |\mathsf{WriteTx}_x|^2 \cdot |\mathsf{ReadTx}_x|\big)$, where $\mathsf{ReadTx}_x = \{T \mid T \vdash \mathsf{R}(x, \_)\}$. Let $N_{\mathcal{P}}$ be the number of pruning passes. The overall complexity of pruning is then $O\big(N_{\mathcal{P}} \cdot \big(|\mathcal{V}| \cdot |\mathcal{E}| + \sum_{x \in \mathsf{Key}} |\mathsf{WriteTx}_x|^2 \cdot |\mathsf{ReadTx}_x|\big)\big)$. Experiments show that $N_{\mathcal{P}}$ is typically small (less than 10), and can thus be treated as $O(1)$. In the worst case, pruning has complexity $O(kn^3)$.

Encoding. Let C_p denote the constraints in the pruned hyperpolygraph $\mathcal{G}_p$. The encoded formula consists of two parts: the constraints in C_p and the clauses for 2-width cycles. The bottleneck lies in enumerating edge pairs to detect 2-width cycles (see Algorithm 2), which costs $O(\|C_p\|^2)$. Hence, the total encoding complexity is $O(\|C_p\|^2)$, or $O(k^2n^4)$ in the worst case.

Solving. The final stage, SMT solving, is NP-hard in general. Although prior research has analyzed SMT solvers and established lower bounds in certain settings [31], the precise complexity of our setting remains open [18]. Nonetheless, experiments show that our approach achieves high efficiency on practical workloads, where worst-case scenarios rarely arise (Section 7).

7 EXPERIMENTS

We have implemented our algorithm in a tool called VeriStrong, built on top of MiniSat [8], a minimalistic and open-source SAT solver. We selected MiniSat due to its simplicity and extensibility, which allowed us to integrate our optimization strategies effectively.

To support the optimization of small-width cycles (Section 6.1), we compute reachability in the known graph using dynamic programming. Specifically, VeriStrong maintains a 0-1 matrix of size $|\mathcal{T}| \times |\mathcal{T}|$, where each entry records whether one transaction is reachable from another. By traversing nodes in reverse topological order, it incrementally updates each node's reachable set: for an edge $(from, to)$, it unions the reachability set of *to* into that of *from*. We implement the matrix using bit vectors and bitwise operations, providing a lightweight alternative to GPU-based solutions [33]. Overall, VeriStrong has approximately 5k lines of C++ code.

We conduct an extensive evaluation of VeriStrong, focusing on its checking efficiency, while also assessing its bug-finding effectiveness. Specifically, we aim to answer the following questions:

- **Efficiency:** How efficiently does VeriStrong perform across various general workloads (Section 7.2.1)? Can it outperform existing tools (Section 7.2.2)? Does it scale to large histories (Section 7.2.3)? How do its individual components contribute to the overall performance (Section 7.3)?
- **Effectiveness:** How effective is VeriStrong at uncovering isolation bugs in production database systems, particularly those related to DuplicateValue (Section 7.4)?

7.1 Benchmarks and Experimental Setup

We evaluate VeriStrong and competing verifiers using two categories of benchmarks. The first category comprises YCSB-like transactional workloads, produced by a parametric workload generator built on top of PolySI [13]. This generator supports several configurable parameters, with default values shown in parentheses: the number of client sessions (20), the number of transactions per session (100), the number of read/write operations per transaction (20), the overall read proportion (50%), the total number of keys (5k), and the proportion of keys with duplicate write values (50%).

We characterize duplicate write values using a Zipfian distribution, where the parameter θ (0.5) controls the degree of duplication, and N (100) defines the size of value space. We then generate a set of representative workloads, including RH (read-heavy, 95% reads), WH (write-heavy, 70% writes), and BL (balanced, 50% reads). These workloads align with UniqueValue ones commonly used in prior work. In addition, we include HD, a variant of BL with a high degree of duplication ($\theta = 1.5$). Each workload is executed on a PostgreSQL v15 instance to produce transactional histories, each containing at least 10k transactions and 80k operations.

The second category comprises application-level benchmarks: TPC-C [34], a standard OLTP benchmark configured with 1 warehouse, 10 districts, 30k customers, and 5 transaction types; RUBiS [32], an auction-based system with 20k users and 200k items;

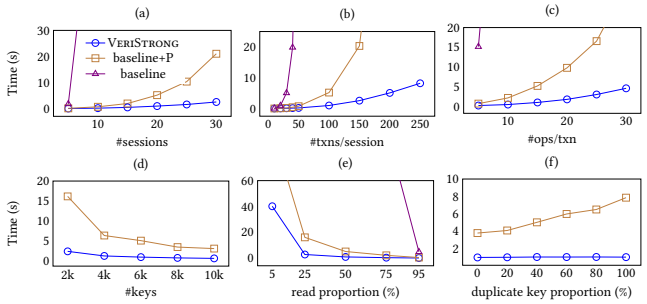


Figure 10: Verification time comparison across a range of general workloads. Timeout (60s) data points are not plotted.

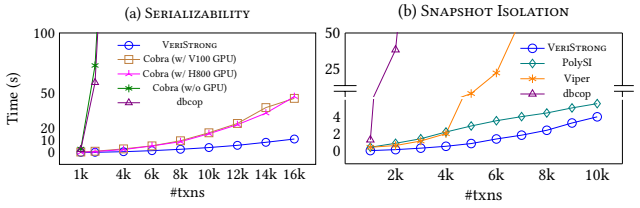


Figure 11: Comparison with existing verifiers under UniqueValue histories using the BlindW-RW datasets (with 10k keys in (a) and 2k keys in (b)).

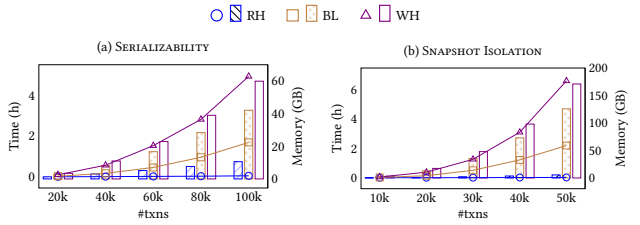


Figure 12: VeriStrong's performance under large workloads. Time and memory usage are plotted in lines and bars.

and TwitterClone [16], a blogging application with Zipfian key access patterns. We use a PostgreSQL instance to generate general transactional histories from these applications, which we refer to as G-TPCC, G-RUBiS, and G-Twitter, respectively. Notably, both RUBiS and TwitterClone naturally generate duplicate write values. However, prior work [13, 33, 39] disabled this behavior to enforce the UniqueValue assumption. In contrast, we preserve these duplicate-value semantics to more accurately reflect real-world behavior.

Unless otherwise stated, all experiments are performed on a local machine equipped with an Intel 13th Gen i5 CPU and 32GB RAM.

7.2 Performance Evaluation

7.2.1 Comparison with Baselines. Our first set of experiments compares VeriStrong with the baseline algorithm (Section 4) across various general workloads. We also include a stronger variant, baseline+P, which incorporates the pruning technique for WW constraints (Section 5.1).

The experimental results are shown in Figure 10, omitting data points that exceed the 60s timeout. VeriStrong consistently outperforms both the baseline and its stronger variant. Specifically, under increased concurrency, such as more sessions (a), more transactions per session (b), more operations per transaction (c), and smaller key space (d), the two baselines suffer from exponential verification time, while VeriStrong incurs only moderate overhead. Furthermore, in scenarios with increased uncertainty in WW and WR dependencies, such as write-heavy workloads (e) and higher proportions of duplicate keys (f), VeriStrong remains the fastest while maintaining fairly stable verification efficiency.

In addition, we measure the memory usage of all three tools under the same settings as in Figure 10. The trends are similar: VeriStrong uses less memory than both baselines across a range of workloads. Memory plots are provided in [5, Appendix F].

7.2.2 Comparison with State-of-the-Art. Our second set of experiments evaluate VeriStrong against four state-of-the-art verifiers: Cobra [33] for Serializability, PolySI [13] and Viper [39] for Snapshot Isolation, and dbcop [4] for both. Since Cobra supports GPU acceleration for pruning, we also include its GPU-enabled versions in our evaluation. Note that dbcop is included as a representative of non-SMT-based verifiers, which rely on graph traversal algorithms, such as depth-first search, to detect cycles.

For a fair comparison, we use the BlindW-RW dataset adopted in prior work [33, 39]. This dataset exclusively consists of UniqueValue histories, as all the above verifiers are restricted to such input. Each history contains an even mix of read-only and write-only transactions, with each transaction comprising eight operations.

As shown in Figure 11, VeriStrong outperforms all competitors in verifying both isolation levels. In particular, with only 16k transactions, it achieves up to a 4.4x speedup, even compared to Cobra with GPU acceleration. Interestingly, even with a stronger H800 GPU (compared to the V100 used in the original experiments [33]), VeriStrong still maintains the performance gap. This suggests that the primary bottleneck in verification lies not in GPU capability but in the solving stage, which is consistent with the fact that GPUs are mainly used to accelerate reachability analysis before solving. For Snapshot Isolation, VeriStrong delivers up to a 4x speedup over PolySI and substantially outperforms Viper and dbcop. All these results highlight the benefits of our dedicated SMT solving.

7.2.3 Scalability. Verifying large histories is highly desirable for increasing confidence in a database's fulfillment of promised isolation guarantee. Following the previous experiments, we evaluate VeriStrong's scalability on large histories, each containing up to 100k transactions and 2 million operations with a total of 50k keys.[4]

As shown in Figure 12, verifying large histories is manageable for VeriStrong on modern hardware, e.g., Serializability verification completes in under 5 hours using less than 64 GB of memory. In addition, two expected trends are observed. First, both runtime and memory usage increase as the proportion of read operations decreases (from RH to BL to WH workloads). This trend is primarily due to the increased uncertainty in WW dependencies, which

[4]We used a server with an AMD EPYC 7H12 64-Core processor and 1TB of memory.

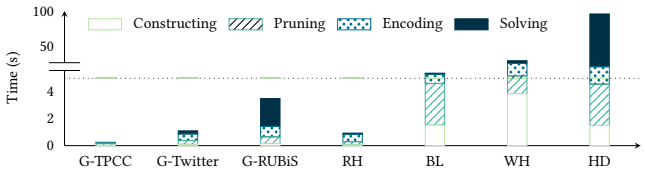


Figure 13: Breakdown of checking time across stages.

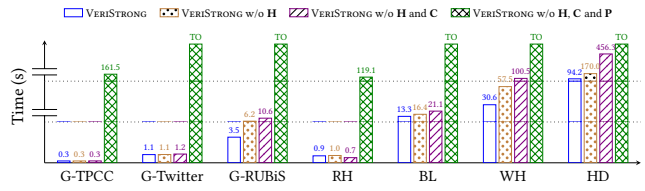


Figure 14: Ablation analysis. Timeout is set to 600s.

expands the set of potential read-from candidates and ultimately amplifies the overall uncertainty.

Second, verifying Snapshot Isolation incurs higher overhead than Serializability. This is expected: while Serializability requires only an acyclic dependency graph, Snapshot Isolation permits certain cycles, making the analysis inherently more complex. In practice, VeriStrong maintains an additional induced graph for Snapshot Isolation, which is more expensive to construct than deriving RW edges alone as for Serializability. This gap is also evident in prior evaluations on UniqueValue histories: Cobra verifies a history with 10k transactions against Serializability in 15s [33], whereas Viper requires over 400s to verify Snapshot Isolation for the same amount of transactions [39]. This gap becomes even more pronounced in the presence of duplicate values.

7.3 Dissecting VeriStrong

In this section, we take a closer look at VeriStrong, examining how its individual components and optimization strategies contribute to the overall checking efficiency.

7.3.1 Decomposition Analysis. We decompose VeriStrong's checking time into four stages: constructing, pruning, encoding, and solving. Figure 13 presents the breakdown across seven benchmarks (see Section 7.1). Overall, the constructing and pruning stages are relatively inexpensive. The G-TPCC workload is a special case where all constraints are pruned prior to encoding and solving, resulting in negligible cost for the latter stages. In most benchmarks—except for G-RUBiS and HD—the solving time remains relatively low, whereas the encoding stage dominates. This behavior is expected: our small-width cycles optimization shifts some of the burden from solving to pruning and encoding, while keeping these two stages efficient. As a result, the overall checking time is reduced.

7.3.2 Ablation Analysis. To assess the contributions of the major optimizations in VeriStrong, we consider three variants: (i) VeriStrong without the heuristic for polarity picking (**H**); (ii) VeriStrong without **H** and without the encoding of small-width cycles (**C**); and (iii) VeriStrong without **H**, **C**, and pruning (**P**).

Figure 14 presents our ablation analysis results. Overall, all optimizations proves effective, though their impact varies across different workloads. Optimization **H** is particularly beneficial for G-RUBiS, BL, WH, and HD, but shows little effect on G-Twitter and RH. This is because the latter two are read-heavy workloads, where cycles are rarely encountered during solving. Consequently, the potential for **H** to resolve conflicts through polarity picking is limited.

We also observe that the workloads where Optimization **C** is most effective largely overlap with those where **H** is effective. Interestingly, for RH, Optimization **C** slightly increases the overall checking time. This is expected, as the overhead stems from the enumeration of small-width cycles, which might be unnecessary in cycle-sparse workloads like RH.

Finally, we find that Optimization **P** provides significant gains, as a large number of 1-width cycles are eliminated, including WR constraints and the associated RW ones unique to DuplicateValue histories. For instance, as noted earlier, all constraints in G-TPCC are pruned, allowing the solving phase to be bypassed entirely.

7.3.3 Impact of Cycle Width. Recall that in Optimization **C**, we encode 2-width cycles (see Section 6.1.2). A natural question arises: what is the performance impact of encoding cycles of larger widths? To answer this, we conduct a comparative experiment using variants of Variant (i), varying the width of encoded cycles. We plot the results in Figure 15, where $w = 1$ corresponds to a baseline with pruning only (i.e., no cycle encoding), while $w \geqslant 2$ indicates that cycles of width w are explicitly encoded. Across all six benchmarks (excluding G-TPCC, where solving is bypassed), encoding 2-width cycles ($w = 2$) consistently delivers the best performance, striking a balance between pruning effectiveness and solving overhead. The only exception is RH, where solving time is already low, suggesting that in cycle-sparse workloads, the modest overhead of encoding small-width cycles has limited impact on overall performance.

7.3.4 Impact of Polarity Picking. To better understand how Optimization **H** improves VeriStrong's performance, we examine its effect on reducing the number of encountered conflicts (i.e., the number of backtracks) during solving. We conduct a comparative experiment between VeriStrong and Variant (i) across seven benchmarks. As shown in Table 1, **H** significantly reduces the number of conflicts. This effect becomes more pronounced as the write proportion and the duplication rate of write values increase, which is consistent with the performance differences observed in Figure 14. The only two exceptions are G-TPCC and RH, both of which exhibit no conflicts. In G-TPCC, all constraints are pruned prior to solving. In RH, although not all constraints are pruned, its read-heavy nature results in conflict-free solving even without **H**.

7.4 Effectiveness

Beyond checking performance, an essential criterion for a verifier is its ability to accurately detect isolation anomalies. To this end, we validate VeriStrong against an extensive set of known anomalies.

Reproducing UniqueValue Anomalies. VeriStrong reproduces all known anomalies from a substantial set of 2507 UniqueValue histories: 2073 for Serializability and 434 for Snapshot Isolation. These histories were originally collected by prior work [4, 13, 33]

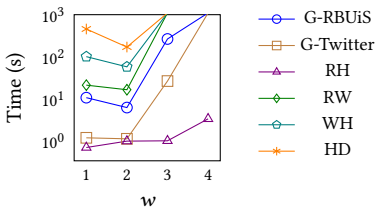


Figure 15: Runtime of Variant (i) under varying encoded cycle widths (w). Timeout (600s) data points are not plotted.

Table 1: The number of conflicts encountered during solving with or without Optimization H.

Benchmark	VeriStrong	w/o **H**
G-TPCC	0	0
G-Twitter	79	99
G-RUBiS	89	246
RH	0	0
BL	50	497
WH	86	1427
HD	936	2047

Table 2: Average checking time on anomalous histories (excluding those that timed out beyond 900s).

Verifier	SER	SI
Cobra	1293ms	–
PolySI	–	359ms
Viper	–	6559ms
dbcop	551ms	108ms
VeriStrong	143ms	18ms
#histories	2073	434

from earlier versions of widely-used databases, including CockroachDB and YugabyteDB. In addition, we report the runtime of each tool: as shown in Table 2, VeriStrong achieves significantly lower average checking times than the state-of-the-art verifiers.

(Re)discovering DuplicateValue Anomalies. Motivated by recent bug reports [22, 25], we apply VeriStrong to test both MySQL and MariaDB. VeriStrong replicates the reported anomalies in earlier versions of both databases and, notably, rediscovers the same bug in the latest stable version of MariaDB (v 11.5.2). We reported this issue to the developers, who confirmed it as a valid bug. Note that all these anomalies lie beyond the capabilities of existing verifiers, which are limited to verifying UniqueValue histories.

8 RELATED WORK

Recent years have seen significant progress in verifying database isolation guarantees. We focus on the black-box approaches.

A recent line of work [13, 33, 39], including Cobra and PolySI, applies SMT solving to verify strong isolation. All these tools encode histories as polygraphs or their variants and rely on the off-the-shelf solver MonoSAT to detect cycles that represent anomalies.

VeriStrong follows this approach but differs in three key ways. First, rather than relying solely on general-purpose SMT solvers, it incorporates workload-specific optimizations to enhance performance. Second, existing polygraph-based approaches are limited to UniqueValue histories. VeriStrong introduces an expressive hyper-polygraph representation that also supports DuplicateValue, thereby broadening applicability. Third, VeriStrong provides both soundness and completeness, ensuring reliable isolation verification. In contrast, existing verifiers risk false positives and false negatives, particularly in the presence of duplicate write values.

Non-solver tools [4, 17, 19, 24, 35] employ graph traversal algorithms, such as DFS or optimized variants, to verify isolation guarantees. The dbcop tool [4] applies a polynomial-time algorithm to verify Serializability over UniqueValue histories (with a bounded number of sessions), along with a polynomial-time reduction from verifying Snapshot Isolation to verifying Serializability. However, dbcop has been reported to be less efficient than Cobra [33] and PolySI [13], which in turn are outperformed by VeriStrong.

Elle [17], the isolation checker used in the Jepsen framework [14], infers version orders (i.e., WW dependencies) from list-append workloads. For example, reading the list `[2,1]` on key x indicates that $A(x, 2)$ precedes $A(x, 1)$ in the version order. However, such inference relies on the UniqueValue assumption, which limits its effectiveness when applied to general workloads containing duplicate write values, e.g., when reading a list `[1,1]`. MTC [35] exploits a class of simplified database workloads, called mini-transactions (e.g., read-modify-write transactions) to achieve quadratic-time verification under the same UniqueValue assumption. While MTC strikes a practical balance between performance and bug-finding effectiveness, its applicability is limited, as real-world workloads often diverge from such simplified transaction patterns.

Other tools, such as Plume [19] and AWDIT [24], focus on verifying weaker isolation guarantees [2, 20]. These properties are computationally less complex than the stronger ones we target [4].

9 DISCUSSION AND CONCLUSION

Supporting Other Formalisms. Our hyper-polygraphs are not restricted to Adya's formalism; rather, they are expressive enough to support other dependency-based characterizations of isolation guarantees, such as the axiomatic framework proposed by Biswas and Enea [4]. See [5, Appendix G] for a demonstration.

Supporting Weak Isolation Levels. While our dedicated SMT solving is highly efficient for verifying strong isolation levels, it may be less suitable for weaker ones, where SMT may be heavyweight [19]. We have realized an integration of VeriStrong and the Plume weak isolation checker in the IsoVista system [11].

Limitations. Our approach currently targets key-value databases. Extending it to relational models is a promising direction, e.g., by building on techniques from [15, 37]. Another direction is to unify correctness reasoning for both isolation guarantees and query semantics [30, 38], drawing inspiration from recent advances like [29].

Conclusion. We have presented a novel approach for verifying strong database isolation, centered around hyper-polygraphs—a general formalism for modeling dependency-based isolation semantics. We have focused on Adya's theory and established sound and complete characterizations for both Serializability and Snapshot Isolation. Our verifier achieves high efficiency by tailoring SMT solving to database workload characteristics, in contrast to the state-of-the-art that relies solely on general-purpose solvers.

ACKNOWLEDGMENTS

We appreciate the anonymous reviewers for their valuable feedback. Hengfeng Wei was supported by the NSFC (62472214). Si Liu was supported by an ETH Zurich Career Seed Award.

REFERENCES

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph. D. Dissertation. USA.

[2] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (nov 2013), 181–192.

[3] Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. 2015. SAT Modulo Monotonic Theories. In *AAAI 2015*. AAAI Press, 3702–3709.

[4] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 165 (2019).

[5] Zhiheng Cai, Si Liu, Hengfeng Wei, Yuxing Chen, and Anqun Pan. 2025. *Fast Verification of Strong Database Isolation (Extended Version).* Technical Report. http://arxiv.org/abs/2511.14067.

[6] Andrea Cerone and Alexey Gotsman. 2018. Analysing Snapshot Isolation. *J. ACM* 65, 2, Article 11 (Jan. 2018).

[7] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5, 7 (July 1962), 394–397.

[8] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003 (LNCS, Vol. 2919)*. Springer, 502–518.

[9] Hongyu Fan, Zhihang Sun, and Fei He. 2023. Satisfiability Modulo Ordering Consistency Theory for SC, TSO, and PSO Memory Models. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 6 (March 2023), 37 pages.

[10] Shabnam Ghasemirad, Si Liu, Christoph Sprenger, Luca Multazzu, and David Basin. 2025. VerIso: Verifiable Isolation Guarantees for Database Transactions. *Proc. VLDB Endow.* 18, 5 (2025), 1362–1375.

[11] Long Gu, Si Liu, Tiancheng Xing, Hengfeng Wei, Yuxing Chen, and David Basin. 2024. IsoVista: Black-Box Checking Database Isolation Guarantees. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4325–4328.

[12] Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability Modulo Ordering Consistency Theory for Multi-threaded Program Verification. In *PLDI '21*. ACM, 1264–1279.

[13] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-box Checking of Snapshot Isolation in Databases. *Proc. VLDB Endow.* 16, 6 (2023), 1264–1276.

[14] Jepsen. Accessed in May, 2025. https://jepsen.io.

[15] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *OSDI 23*. USENIX Association, 397–417.

[16] Nick Kallen. Accessed in May, 2025. Big Data in Real Time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter/.

[17] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280.

[18] Donald E. Knuth. 2015. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability* (1st ed.). Addison-Wesley Professional.

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

[20] Si Liu, Luca Multazzu, Hengfeng Wei, and David A. Basin. 2024. NOC-NOC: Towards Performance-optimal Distributed Transactions. *Proc. ACM Manag. Data* 2, 1, Article 9 (March 2024), 25 pages.

[21] Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. 2019. Automatic Analysis of Consistency Properties of Distributed Transaction Systems in Maude. In *TACAS 2019 (LNCS, Vol. 11428)*. Springer, 40–57.

[22] MariaDB-#26642. 2022. Weird SELECT view when a record is modified to the same value by two transactions. https://jira.mariadb.org/browse/MDEV-26642.

[23] Anders Alnor Mathiasen, Léon Gondelman, Léon Ducruet, Amin Timany, and Lars Birkedal. 2025. Reasoning about Weak Isolation Levels in Separation Logic. *Proc. ACM Program. Lang.* ICFP (2025).

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

[25] MySQL-#100328. 2020. Inconsistent behavior with isolation levels when binlog enabled. https://bugs.mysql.com/bug.php?id=100328.

[26] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM* 53, 6 (Nov. 2006), 937–977. doi:10.1145/1217856.1217859

[27] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (oct 1979), 631–653. doi:10.1145/322154.322158

[28] David J. Pearce and Paul H. J. Kelly. 2006. A Dynamic Topological Sort Algorithm for Directed Acyclic Graphs. *ACM J. Exp. Algorithmics* 11 (2006).

[29] Lauren Pick, Amanda Xu, Ankush Desai, Sanjit A. Seshia, and Aws Albarghouthi. 2025. Checking Observational Correctness of Database Systems. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 139 (April 2025), 28 pages.

[30] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *OSDI '20*. USENIX Association, 667–682.

[31] Robert Robere, Antonina Kolokolova, and Vijay Ganesh. 2018. The Proof Complexity of SMT Solvers. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 275–293.

[32] RUBiS. Accessed in May, 2025. Auction Site for e-Commerce Technologies Benchmarking. https://projects.ow2.org/view/rubis/.

[33] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI 2020*. USENIX Association, 63–80.

[34] TPC. Accessed in October, 2023. TPC-C: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

[35] Hengfeng Wei, Jiang Xiao, Na Yang, Si Liu, Zijing Yin, Yuxing Chen, and Anqun Pan. 2025. Boosting End-to-End Database Isolation Checking via Mini-Transactions. In *ICDE 2025*. IEEE Computer Society, 3998–4010.

[36] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann.

[37] Rui Yang, Ziyu Cui, Wensheng Dou, Yu Gao, Jiansen Song, Xudong Xie, and Jun Wei. 2025. Detecting Isolation Anomalies in Relational DBMSs. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA076 (June 2025), 23 pages.

[38] Zijing Yin, Si Liu, and David Basin. 2025. Testing Graph Databases with Synthesized Queries. *Proc. ACM Manag. Data* 3, 4, Article 268 (Sept. 2025), 26 pages.

[39] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *EuroSys 2023*. ACM, 654–671.