IsoVista: Black-box Checking Database Isolation Guarantees

Long Gu
State Key Laboratory for Novel Software Technology
Nanjing University
502023320005@smail.nju.edu.cn

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

Tiancheng Xing
State Key Laboratory for Novel Software Technology
Nanjing University
xtc1207445468@outlook.com

Hengfeng Wei*
State Key Laboratory for Novel Software Technology
Nanjing University
hfwei@nju.edu.cn

Yuxing Chen
Tencent Inc.
axingguchen@tencent.com

David Basin
ETH Zurich
basin@inf.ethz.ch

ABSTRACT

Transactional isolation is critical to the functional correctness of database management systems (DBMSs). Much effort has recently been devoted to finding isolation bugs and validating isolation fulfilment in production DBMSs. However, there are still challenges that existing isolation checkers have not yet fully addressed. For instance, they may overlook bugs, incur high checking overhead, and return hard-to-understand counterexamples.

We present IsoVista, the first black-box isolation checking system that encompasses all the following features. It builds on faithful characterizations of a range of isolation levels, ensuring the absence of both false positives and missed bugs in collected DBMS execution histories. IsoVista exhibits superior checking efficiency, compared to the state-of-the-art, and visualizes violation scenarios, facilitating the understanding of bugs found. It also supports profiling and benchmarking the performance of isolation checkers under various workloads, assisting developers of both DBMSs and checkers. We showcase all these features through user-friendly interfaces.

PVLDB Reference Format:
Long Gu, Si Liu, Tiancheng Xing, Hengfeng Wei, Yuxing Chen, and David Basin. IsoVista: Black-box Checking Database Isolation Guarantees. PVLDB, 17(12): 4325 - 4328, 2024. doi:10.14778/3685800.3685866

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

1 INTRODUCTION

Database management systems (DBMSs) are the backbone of numerous software systems and applications. *Transactional isolation* is one of the most crucial functional correctness properties. To balance data consistency and performance, DBMSs provide a spectrum of isolation levels (or guarantees), including weak levels like READ COMMITTED, the "sweet spot" TRANSACTIONAL CAUSAL CONSISTENCY that is the highest level achievable in always-available systems, and the gold standard SERIALIZABILITY.

Despite being mature and extensively tested, numerous isolation bugs have been found in many production DBMSs, including PostgreSQL and MariaDB, during a recent testing campaign [2, 4–6]. This raises the concern of whether existing DBMSs effectively uphold the promised isolation guarantees in practice.

Recent years have witnessed a torrent of isolation checkers [2, 4–6, 8, 9] for testing database isolation guarantees. The *de facto* approach adopted by these checkers is *randomized black-box testing*. This approach stresses DBMSs with large, concurrent workloads, serving a dual purpose: increasing the likelihood of triggering isolation bugs and gaining confidence in their absence.

There are still challenges that existing isolation checkers have not yet fully addressed. First, many checkers fail to detect some critical bugs due to their incomplete characterizations of the isolation levels in question. Second, existing checkers often incur significant checking overhead when searching for cycles (or anomalies) in a transactional dependency graph. When utilizing SMT (Satisfiability Modulo Theories) solvers to check isolation guarantees, encoding and solving transactional dependency constraints may still be expensive. These overheads become more pronounced under workloads of higher concurrency. Third, most checkers return hard-to-understand counterexamples, such as unsatisfied clauses. This makes understanding and debugging the violations found hard. Fourth, an ideal checking system would encompass a range of isolation levels, catering for various checking requirements.

We present IsoVista, the *first* black-box isolation checking system that has addressed all these challenges. IsoVista substantially expands our prior work PolySI [4], which is an efficient and complete checker for SNAPSHOT ISOLATION (SI), and incorporates our recent advance in checking weak isolation levels [6]. Overall, it exhibits the following key features.

Many Levels Supported. In addition to SI, IsoVista supports checking a wide spectrum of isolation levels. These include the gold standard SERIALIZABILITY (SER), the default level READ COMMITTED (RC) for most SQL databases, REPEATABLE READ (RR) favorable for read-only transactions, and more recent levels, such as READ ATOMICITY (RA) and TRANSACTIONAL CAUSAL CONSISTENCY (TCC), catering for various modern database applications such as social media [7].

Complete and Efficient Checking. IsoVista builds on our sound and complete formal characterizations of all six isolation levels. This guarantees the absence of both false positives and missed bugs in collected DBMS execution histories. IsoVista also demonstrates superior checking performance by leveraging various design

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685866

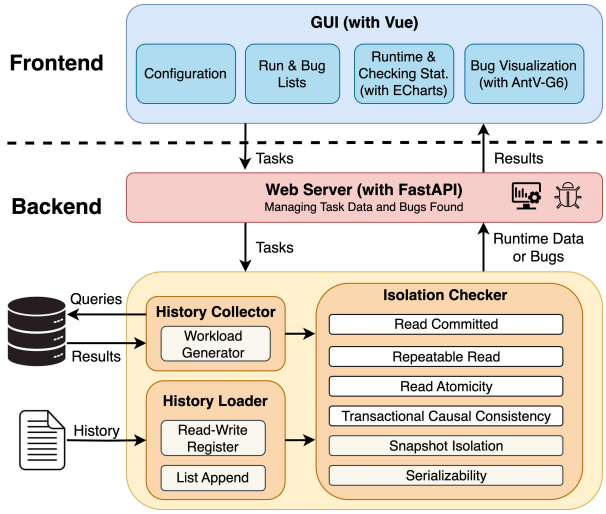

Figure 1: The architecture of IsoVista.

choices and optimization techniques. These include compact and efficient SMT encoding for stronger isolation levels such as SER and SI, as well as efficient data structures for storing and traversing transactional dependency graphs in weaker levels like TCC.

Informative Bug Visualization. Upon detecting an isolation bug, IsoVista reconstructs the full violating scenario by identifying the core participating transactions and their dependencies, rather than reporting a plain cycle or unsatisfied clauses (if an SMT solver is in use). Such detailed counterexamples could assist developers in comprehending the bugs and pinpointing their causes.

Benchmarking Isolation Checkers. IsoVista facilitates monitoring a checker's runtime information, such as CPU and memory utilization. It also supports profiling checking statistics across parameterized workloads. This provides comprehensive insights into tools' performance, such as detecting sudden increases in checking time with a high number of clients, and allows exploration of the tool design space with new optimizations.

2 BACKGROUND

Black-box checking of isolation guarantees proceeds as follows. Clients initiate transactional requests, produced by a randomized workload generator, to a DBMS. Each client session records the requests it sends, along with the corresponding results provided by the DBMS. These records from all clients are combined into a unified history, which is then passed on to the isolation checker. Finally, the checker checks whether the history meets the specified isolation level. Specifically, the checker constructs a certain kind of transactional dependency graph [1–3] (also see below) from the history, and employs graph traversal algorithms or SMT solvers to search for specific cycles, which represent different isolation bugs. Upon finding a bug, some checkers also provide a counterexample.

We base our formal specification of isolation levels on the axiomatic frameworks in [2, 3]. Compared to Adya's formalization [1], these frameworks allow us to unify the characterizations of both SQL-92 standardized isolation levels and more recent ones such as RA and TCC. Moreover, they are suitable for black-box checking over histories, as transactional dependencies can be straightforwardly derived or effectively inferred. In addition, Adya's formalization relies on transactions' start and commit timestamps for defining isolation levels like SI. Databases, such as MongoDB and TiDB, may not expose such information as in our black-box setting.

A transactional dependency graph typically captures four types of dependencies (or relations) between the transactions in a history. The SO relation enforces a strict total order between the transactions within the same session. The WR relation associates a transaction that reads a value with the one that writes this value. The WW relation enforces a strict total order (also known as the version order [1]) between transactions that update the same key. The RW relation is derived from WR and WW relations, relating a transaction that reads a value to the one that overwrites this value. Isolation levels can be characterized by dependency graphs with specific cycles. For example, SER is characterized by acyclic dependency graphs [1], while SI is characterized by dependency graphs that contain only cycles with at least two adjacent RW edges [3].

3 ISOVISTA DESIGN AND IMPLEMENTATION

We present an overview of IsoVista. Figure 1 depicts its architecture: a backend that extends PolySI by integrating all key features, and a frontend facilitating developers' interaction with these features.

3.1 IsoVista Backend

3.1.1 Isolation Checker. At the core of IsoVista's backend lies an isolation checker implemented in Java, which comprises six checking components, one for each isolation level. For stronger isolation levels (i.e., SI and SER) with substantial checking complexity, IsoVista employs *generalized polygraphs* [4] to compactly encode transactional dependency constraints. It integrates the advanced SMT solver MonoSAT, specialized in verifying graph properties like acyclicity. Moreover, IsoVista leverages efficient constraint pruning optimizations to accelerate SMT solving.

For the remaining polynomial-time checkable weaker isolation levels, IsoVista employs graph traversals, instead of SMT solving that specializes in resolving uncertain transactional dependencies such as the order of concurrent writes. Moreover, IsoVista leverages a novel combination of *vectors* and *tree clocks* to efficiently capture transitive dependencies and accelerate graph traversals for both reachability checking and cycle detection [6]. This substantially extends our original PolySI design. Consequently, IsoVista can efficiently validate millions of transactions under weaker isolation levels like TCC in a few minutes. Additionally, IsoVista offers developers an interface to integrate their own checkers.

3.1.2 History Loader and Collector. Developers can upload pre-collected DBMS execution histories through IsoVista's *history loader*. IsoVista currently supports histories of both read-write registers, compatible with all existing checkers, and list-append operations, which enable efficient inference of version orders [5].

Moreover, IsoVista incorporates a parameterized workload generator within its *history collector*. When developers check isolation levels of a DBMS using IsoVista, the workload generator produces transaction workloads (currently limited to read-write registers) in memory based on specified parameters (Section 3.2). The history collector then simulates client interactions with the tested

Figure 2: IsoVista's GUI.

DBMS using the generated workloads and collects the transactional requests along with their corresponding outcomes.

3.1.3 Web Server. IsoVista incorporates a Python-based web server, bridging the above components and the frontend. For example, it orchestrates the history collector and isolation checker based on the frontend configuration, while caching the runtime data, e.g., checking time and detected bugs, for delivery to the frontend.

3.2 IsoVista Frontend GUI

Developers interact with IsoVista through its GUI, illustrated in Figure 2, which is implemented as a web service.

3.2.1 Configuration. IsoVista offers developers a modular interface to configure their experiments, shown in Figure 2A. This interface contains three modules. First, Database Setting includes information about the tested DBMS, its isolation levels, and the JDBC connection. Second, Workload Setting includes parameters for the workload generator, such as the number of sessions and the read/write ratio. IsoVista also supports batched workloads. For example, #sess=[5,10,15,20] will generate four workloads, each with a progressively larger number of sessions. This is useful for benchmarking checker performance. Alternatively, developers can upload their pre-collected histories in the format of either read-write registers or list-append operations by enabling Skip Generator. Third, Checker Setting includes the checking components for different isolation levels. In addition to the built-in six components, custom checkers that have been integrated into IsoVista are also displayed.

3.2.2 Task Tracing. After configuration, the developer can submit a task, potentially amounting to multiple DBMS runs. The web server queues the task for execution. IsoVista displays all history runs in a table, one row for each run, as shown in Figure 2B. Each run has one of four statuses: Healthy (successful validation without isolation bugs), Buggy (isolation bug detected), Running (displaying progress bar), or Pending (waiting for execution). Developers can click the View button to access the runtime information of a task and its checking statistics (Section 3.2.3).

Upon detecting an isolation bug, IsoVista adds the corresponding task to the bug summary table, as shown in Figure 2C. It allows developers to categorize each bug with a label (Open or Fixed) to track its status. Additionally, developers can click the View button to investigate the details of a bug (Section 3.2.4).

3.2.3 Runtime and Checking Statistics. IsoVista monitors a task's CPU and memory usage in *real-time* throughout its execution. This allows developers to timely adjust their tasks like termination (Stop shown in Figure 2B). Moreover, upon (sub-)task completion, additional details such as the average checking time and the maximum memory usage for a batch of histories are also shown. For example,

① and ② in Figure 2D present such statistics for varying numbers of sessions, where data are plotted gradually as the number of sessions increases.

IsoVista additionally performs a *decomposition analysis* that breaks down the checking time into stages. For solver-based checking components like SI, the stages involve constructing the dependency graph, encoding and pruning constraints, and the solving process, as shown in ③. For checking components not based on solving, the stages involve both constructing and traversing the graph. For example, IsoVista incurs less traversal overhead than Elle for checking TCC, as depicted in ④. These statistics are displayed side-by-side, offering developers a comprehensive and detailed perspective on the performance of the compared checkers.

Furthermore, developers can hover over a chart to view the specific value at a particular point. They can also click the Download button to save these charts for further analysis or record-keeping.

3.2.4 Bug Visualization. IsoVista visualizes detected bugs to aid developers in diagnosing their causes. As shown in Figure 2E, a bug is displayed as a transactional dependency graph. IsoVista distinguishes different types of edges by colors. Note that, for solver-based checking components, IsoVista constructs such a graph from the unsatisfied clauses returned by MonoSAT. We develop this feature based on PolySI's *interpretation algorithm* [4].

During the checking process, IsoVista also identifies additional transactions and the associated dependencies that contribute to the reported bug. Developers can expand the highlighted cycle in the initially returned graph, such as ⑥, to reconstruct a comprehensive violating scenario, as shown in ⑦. These additional details elucidate how the core dependencies in the cycle are inferred. Moreover, IsoVista enriches the graph with explanatory details. For instance, hovering over a node reveals the transaction's operations, such as the read operation id1 retrieving value 0 for key 2.

IsoVista enables developers to manipulate the graph by dragging nodes, marking/removing nodes and edges, resizing, and performing undo/redo actions. Additionally, the graph can be downloaded in formats like PNG, TikZ, and DOT.

4 DEMONSTRATION

We showcase IsoVista's key features in three scenarios, highlighting the comprehension of identified bugs, benchmarking checkers, and efficient checking of weak isolation levels.

4.1 Unserializable Transactions in PostgreSQL

Configuration. PostgreSQL (v12.3) is tested under SER. The workload is generated by IsoVista: #sess=20, #txn/sess=1, #ops/txn=2, read proportion=50%, #key=10, and distribution=uniform.

Counterexample. IsoVista rediscovers this bug using its SER checker with a workload of read-write registers, as shown in ⑥. This bug was previously found by Elle with list-append operations. IsoVista presents the counterexample as a transactional dependency graph, highlighting a cycle of two RW edges.

The remaining part of the graph restored by IsoVista explains how these two RW edges are inferred, as shown in ⑦. For instance, the RW(k2) edge from Txn(id17) to Txn(id11) is derived from the WW(k2) edge from Txn(id0) to Txn(id11) and the WR(k2) edge from Txn(id0) to Txn(id17); the other possibility of the WW dependency between Txn(id0) and Txn(id11), depicted as a yellow dashed arrow from Txn(id11) to Txn(id0) is pruned due to the presence of the WR(k10) edge from Txn(id0) to Txn(id11). The rationale for the RW(k10) edge from Txn(id11) to Txn(id17) is similar.

4.2 Comparing SI Checkers

Configuration. Histories are obtained from MySQL (v8.3) under SER using IsoVista (#sess=[5,10,15,20], #txns/sess=100, #ops/txn=5, #key=1000, read proportion=50%, distribution=uniform).

Performance Comparison. We have developed PolySI+ that enhances our previous PolySI checker with heuristic pruning. We have incorporated it into IsoVista, which allows us to compare the performance of these two checkers, along with Viper [9], a state-of-the-art SI checker released concurrently with PolySI. The plots ① and ② in Figure 2D show their performance with varying number of sessions. Overall, PolySI+ outperforms the other two checkers with less checking time and lower memory overhead. Moreover, IsoVista's decomposition analysis reveals that the performance improvement of PolySI+ primarily originates from the solving stage, as shown in ③. This validates our optimization which prunes a significant number of constraints before solving.

4.3 Profiling IsoVista's Weak Isolation Checkers

Configuration. Histories are obtained from PostgreSQL (v15.2) under SER using IsoVista (#sess=100, #txns/sess=10k, #ops/txn=50, #key=100 million, read proportion=50%, distribution=uniform).

Performance Results. To evaluate IsoVista's scalability in checking weaker isolation levels, including RC, RA, and TCC, we generate large workloads comprising one million transactions and 50 million operations. These experiments involve varying read proportions. IsoVista completes all checking tasks within three minutes (⑤). Together with the decomposition analysis result shown in ④, this demonstrates the effectiveness of IsoVista's optimized graph traversals via vectors and tree clocks for checking weak isolation levels.

ACKNOWLEDGMENTS

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. Massachusetts Institute of Technology.
[2] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 165 (Oct 2019), 28 pages.
[3] Andrea Cerone and Alexey Gotsman. 2018. Analysing Snapshot Isolation. *J. ACM* 65, 2, Article 11 (Jan 2018), 41 pages.
[4] 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 (Feb 2023), 1264–1276.
[5] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (Nov 2020), 268–280.
[6] Si Liu, Long Gu, Hengfeng Wei, and David Basin. 2024. *Plume: Efficient and Complete Black-box Checking of Weak Isolation Levels.* Technical Report. https://github.com/dracooooooo/Plume.
[7] 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 (Mar 2024), 25 pages.
[8] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI '20.* 63–80.
[9] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *EuroSys '23.* 654–671.