Efficient Black-Box Serializability Checking in the Presence of Range Predicates

Qikang Liu
Simon Fraser University
qla116@sfu.ca

Si Liu
Texas A&M University
si.liu@tamu.edu

Yuepeng Wang
Simon Fraser University
yuepeng@sfu.ca

Abstract

Black-box database isolation checking determines whether an observed transaction execution satisfies a target isolation level. Yet, the problem remains largely underexplored for workloads containing range predicates. We present a state-based approach for checking Serializability, the gold standard of transaction isolation. The approach searches for a serial ordering of transactions that transforms the initial database state into the observed final state while reproducing all observed query results. We formalize this search as an automaton-based exploration procedure that reduces the checking problem to finding an accepting run. The automaton construction inherently merges equivalent states encountered during exploration. To further accelerate search, we introduce observation-aware pruning, which eliminates runs inconsistent with the observed query results. Experimental results show that our approach effectively identifies Serializability violations and outperforms the state-of-the-art.

1 Introduction

Database isolation levels specify which effects transactions may observe during concurrent execution. Checking whether a database engine correctly implements a target isolation level is challenging. White-box techniques [1, 7] require reasoning about large, complex codebases that are often unavailable in practice. Black-box approaches [4, 17] offer an effective, scalable alternative by checking whether observed transaction execution histories satisfy the target level.

Most existing black-box checkers [4, 6, 11–15, 17–19] target *key-value* histories, where reads and writes on the same key expose explicit read-from dependencies. Isolation checking can then be reduced to detecting forbidden dependency-graph patterns, such as cycles. Histories with range predicates are substantially more challenging. A range update may affect an unknown set of rows, while a range query returns results determined by a predicate rather than a fixed set of keys. As a result, the read-from dependencies required by graph-based approaches are not explicitly available in the observed history and must instead be inferred.

Despite this complexity (e.g., NP-complete for checking Serializability [5]), efficient isolation checking remains important in practice: analyzing more histories within a time budget increases confidence in a database's correctness and improves the likelihood of exposing isolation bugs.

Prior work checks isolation with range predicates in different ways. For example, checkConsistency [5] recovers read-from relations and constructs dependency graphs by instrumenting database tables with an additional write-id column and augmenting updates and deletes with `RETURNING` clauses. Troubaduor [16], assumes timestamps reveal transaction order and adopts a state-based verification approach [8] based on symbolic execution and constraint solving.

However, checking transaction histories with range predicates remains largely underexplored when neither instrumentation nor transaction ordering information is available. In this work, we focus on Serializability, the gold standard of transaction isolation, under which every execution must be equivalent to some serial execution of the same transactions. This guarantee rules out anomalies such as dirty reads, phantom reads, and write skew.

Our Solution. We present a state-based approach for checking Serializability over transaction histories with range predicates. To avoid expensive enumerative search of serial orders (up to $n!$ for a history with n transactions), our key idea is to merge equivalent states and encode the search as a *Deterministic Finite Automaton (DFA)*. Specifically, given an initial database state, a final database state, and an observed history, we construct a DFA whose runs represent candidate serial executions. Checking Serializability then reduces to deciding whether this DFA has an accepting run.

Our *state merging* approach exploits the fact that different transaction prefixes may execute transactions in different orders but reach the same database state after executing the same set of transactions. Such prefixes are equivalent with respect to all remaining transactions, so the checker only needs to continue the search once from the equivalent state. The search also leverages *observation-aware pruning* to eliminate infeasible prefixes early. For any transaction in the candidate prefix, the checker evaluates its operations on the current state. Under Serializability, if any query result differs from the corresponding observed result in the history, then no serial order extending this prefix can satisfy Serializability. Hence, the prefix can be safely pruned.

Contributions. We make the following contributions:

- We introduce DFA as a new formalism for checking Serializability over transaction execution histories.
- We develop a novel checking algorithm that supports broader SQL features, including range predicates, without requiring instrumentation or transaction orders.

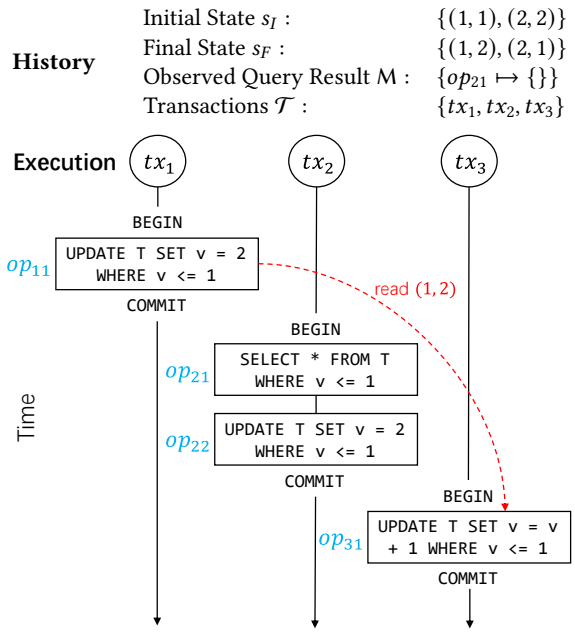

Figure 1. An example history of three transactions and one possible execution that produces it.

- We implement our approach in a tool called SerCheck and evaluate it on histories with range predicates. Results show that it significantly outperforms the state-of-the-art tool, checkConsistency, in checking Serializability.

2 Approach Overview

2.1 Running Example

Consider a database with a single table T containing two columns, $(k : \text{Int}, v : \text{Int})$, where k is the primary key. Initially, T contains two rows, $(1, 1)$ and $(2, 2)$, and the initial database state is thus $s_I = \{(1, 1), (2, 2)\}$. Suppose three concurrent transactions, tx_1, tx_2, and tx_3, execute and produce the history shown in Figure 1.

Each transaction consists of a sequence of SQL operations, and each read (or SELECT) operation has an observable query result. For example, op_{21} issues the query `SELECT * FROM T WHERE v <= 1` and returns $\{\}$, indicating that no row satisfies the range predicate `v <= 1`. After all three transactions commit, the final database state becomes $s_F = \{(1, 2), (2, 1)\}$.

Our goal is to determine whether this history satisfies Serializability. Specifically, we must find a serial order of the three transactions such that executing them in that order from s_I produces s_F, while reproducing the observed result of every SELECT operation ($\{\}$ in this case). Otherwise, the history violates Serializability.

2.2 Our Approach

Exploring State Transitions. We search for such a serial order by exploring all possible state transitions. Figure 2 illustrates the resulting *state transition graph* for the example in Figure 1. Each edge represents the execution of a transaction, transforming one database state into another. The graph contains six paths, corresponding to the six possible serial orders of the three transactions. However, none of them reaches the observed final state s_F while simultaneously reproducing the observed query results. Thus, the history is not serializable.

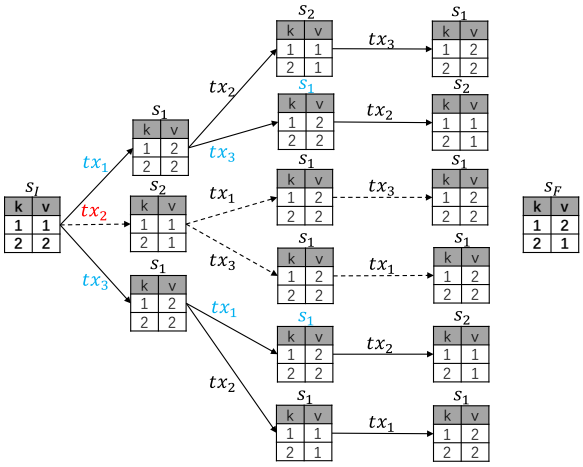

Figure 2. State transitions for the example in Figure 1. Nodes represent database states. An edge labeled tx from s_A to s_B indicates that executing tx transforms s_A into s_B.

State	Value
q_0	$(s_I, \{\})$
q_1	$(s_1\ \{tx_1\})$
q_2	$(s_1, \{tx_3\})$
q_3	$(s_2, \{tx_1, tx_2\})$
q_4	$(s_1, \{tx_1, tx_3\})$
q_5	$(s_1, \{tx_2, tx_3\})$
q_6	$(s_1, \{tx_1, tx_2, tx_3\})$
q_7	$(s_2, \{tx_1, tx_2, tx_3\})$
q_f	$(s_F, \{tx_1, tx_2, tx_3\})$

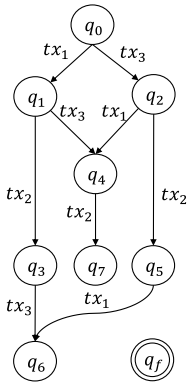

Figure 3. DFA for the example in Figure 1.

The violation originates from op_{31}. It observes an inconsistent snapshot: for key 1, it reads the latest version written by op_{22}, whereas for key 2, it reads the older version written by op_{11}, as indicated by the dashed arrow in Figure 1. All other operations observe the latest version of every key. As a result, op_{31} reads row $(1, 1)$ written by op_{22} and updates it to $(1, 2)$. However, it observes the older version $(2, 2)$ of key 2, and therefore does not update that row. Consequently, key 2 retains its latest committed value $(2, 1)$, while key 1 becomes $(1, 2)$. The resulting final state is thus $\{(1, 2), (2, 1)\}$.

DFA-based Serializability Checking. Since an enumerative search can be expensive due to the potentially large number of candidate serial orders (up to $n!$ for a history with n transactions), we merge equivalent states in the search and

model state transitions as a *Deterministic Finite Automaton* (DFA). The Serializability checking problem then reduces to determining whether the DFA admits an accepting run. If such a run exists, the history is serializable.

Each DFA state consists of a database state together with the set of transactions that have already been executed. Each transition corresponds to executing a single transaction. Tracking the executed transaction set ensures that every transaction appears at most once in a candidate serial order.

For example, the state transition graph in Figure 2 is represented by the DFA in Figure 3. The initial database state corresponds to the initial DFA state $q_0 = (s_I, \{\})$, while the final database state corresponds to the accepting state $q_f = (s_F, \{tx_1, tx_2, tx_3\})$. Because the DFA admits no accepting run from q_0 to q_f, there exists no serial order consistent with the history. Therefore, it violates Serializability.

State Merging. The key observation underlying our DFA approach and state merging is that different serial order prefixes can reach the same state. If two prefixes contain the same set of transactions and produce the same database state from s_I, then they are indistinguishable to all subsequent transactions. Consequently, any continuation from one prefix can be applied to the other as well, and the checker only needs to explore one of them.

For example, in Figure 2, both the prefixes $tx_1 \to tx_3$ and $tx_3 \to tx_1$ reach the state $s_1 = \{(1, 2), (2, 2)\}$ after executing the same set of transactions tx_1 and tx_3. Therefore, they are equivalent and can be merged into a single DFA state q_4.

Observation-Aware Pruning. Observed query results provide another opportunity to reduce the search space. Given a prefix, we execute its operations and compare the resulting query outputs with those observed in the history. If any output differs, then no serial order extending that prefix can reproduce the observed history, and the entire subtree rooted at that prefix can be discarded.

For example, consider the prefix containing only tx_2 in Figure 2. Executing op_{21} on s_I returns $\{(1, 1)\}$, which differs from the observed empty result. Therefore, every serial order beginning with tx_2 is infeasible and can be pruned.

3 The Checking Algorithm

Problem statement. Let $\mathcal{H} = (\mathcal{T}, \mathrm{SO}, \mathsf{M}, s_I, s_F)$ be a history, where $\mathcal{T}$ is a set of transactions, SO is the session order, M records the observed result of each read operation, and s_I and s_F denote the initial and final database states, respectively. The goal of Serializability checking is to determine whether there is a strict total order, also referred to as the *commit order* (CO) [4], such that CO preserves the session order SO; each operation observes results consistent with the database state visible to it under CO; and executing the transactions according to CO from s_I produces s_F. If no such order exists, the history is not serializable.

Algorithm 1 Checking Serializability

Input: A history $\mathcal{H} = (\mathcal{T}, \mathrm{SO}, \mathsf{M}, s_I, s_F)$.
Output: A commit order if $\mathcal{H}$ satisfies Serializability, or $\perp$ otherwise.

procedure CheckSer($\mathcal{H}$)
  $\mathcal{A} \leftarrow$ ConstructDFA($\mathcal{H}$) ▷ *see Fig. 4 for infer. rules*
  return FindAcceptingRun($\mathcal{A}$)

$$\frac{}{q_0 = (s_I, \{\}) \in Q}\ (\textsc{Init}) \qquad \frac{}{(s_F, \mathcal{T}) \in Q_f}\ (\textsc{Final}) \qquad \frac{}{\Sigma = \mathcal{T}}\ (\textsc{Alph})$$

$$\frac{\begin{array}{c} q = (s, \mathcal{T}_p) \in Q \quad q' = [\![tx]\!]_q \quad tx \in \mathcal{T} \setminus \mathcal{T}_p \\ \forall tx'.(tx', tx) \in \mathrm{SO}^+ \implies tx' \in \mathcal{T}_p \\ \textsc{ConsistentObservation}(s, tx, \mathsf{M}) \end{array}}{q' \in Q \quad ((q, tx) \to q') \in \Delta}\ (\textsc{Transit})$$

$$\frac{s' = [\![tx]\!]_s \quad q = (s, \mathcal{T}_p) \quad q' = (s', \mathcal{T}_p \cup \{tx\})}{q' = [\![tx]\!]_q}\ (\textsc{Exec})$$

Figure 4. Inference rules for ConstructDFA. SO^+ is the transitive closure of SO.

Definition 3.1 (Serializability)**.** A history $\mathcal{H} = (\mathcal{T}, \mathrm{SO}, \mathsf{M}, s_I, s_F)$ is serializable if and only if there exists a commit order $\mathrm{CO} \subseteq \mathcal{T} \times \mathcal{T}$, such that

1. $\mathrm{SO} \subseteq \mathrm{CO}$, i.e., CO preserves SO;
2. $\forall tx.\ \forall op \in tx.\ \mathrm{IsRead}(op) \Rightarrow [\![op]\!]_s = \mathsf{M}[op]$, where s is the database state op reads from, $[\![op]\!]_s$ denotes the result of executing op on s, and $\mathsf{M}[op]$ is the observed result;
3. executing the transactions sequentially according to CO from the initial database state s_I yields the final state s_F.

The Algorithm. We search for a commit order by incrementally constructing prefixes of the commit order. To efficiently represent and explore the resulting search space, we construct a *deterministic finite automaton* (DFA).

Definition 3.2 (DFA)**.** A DFA is formally defined as a tuple $(Q, \Sigma, \Delta, q_0, Q_f)$, where Q is a set of states, Σ is the alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a set of transitions, $q_0 \in Q$ is the initial state, $Q_f \subseteq Q$ is a set of final states.

Algorithm 1 presents the main Serializability checking procedure. Given a history $\mathcal{H}$, along with the initial database state s_I and the final database state s_F, it first constructs a DFA $\mathcal{A}$ (via ConstructDFA) and then checks for the existence of an accepting run in $\mathcal{A}$ (via FindAcceptingRun) If such a run exists, it corresponds to a commit order over all transactions, which is returned as a witness that $\mathcal{H}$ satisfies Serializability. Otherwise, the algorithm returns $\perp$, indicating that the history is not serializable.

Figure 4 presents the ConstructDFA procedure using inference rules:

- Rule Init initializes the DFA with the state $q_0 = (s_I, \emptyset)$.
- Rule Final designates $(s_F, \mathcal{T})$ as an accepting state.

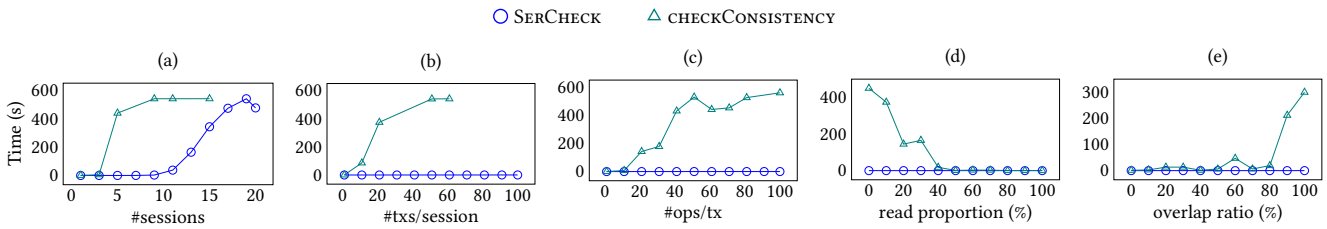

Figure 5. Comparison with checkConsistency across various workloads. Timeout (600s) data points are not plotted. Default configuration: #sessions=3; #txs/session=10; #ops/tx=10; read proportion=50%, update: delete: insert=2: 1: 1; overlap ratio=50%.

- Rule Alph defines the alphabet as all transactions, and each transition corresponds to executing one transaction.
- Rule Transit specifies the construction of states Q and transitions Δ. Given a state $q = (s, \mathcal{T}_p) \in Q$, if there exists a transaction $tx \notin \mathcal{T}_p$ whose execution preserves the session order SO, then a successor state q' is generated and the transition $(q, tx) \rightarrow q'$ is added to Δ. ConsistentObservation ensures that the execution remains consistent with the observed query results; see Appendix A for details.
- Rule Exec defines the successor state q'. The database state s' is obtained by executing tx on s, while the executed transaction set is updated by adding tx to $\mathcal{T}_p$.

The following theorem establishes the soundness and completeness of our Algorithm 1, with the proof in Appendix B.

Theorem 3.3. *A history $\mathcal{H}$ satisfies Serializability if and only if there exists an accepting run in ConstructDFA($\mathcal{H}$).*

4 Evaluation

We implement our approach in a tool, SerCheck, that comprises approximately 3.8K lines of Rust code.

Benchmarks and Setup. We implement a random workload generator for transactional workloads with range predicates. The generator is parameterized by several workload parameters, shown along the x-axis of Figure 5.

In particular, read proportion determines the fraction of `SELECT` queries, while the remaining operations are allocated to `INSERT`, `UPDATE`, and `DELETE` statements according to a configurable write mix. *Overlap ratio* captures the frequency with which SQL operations access overlapping rows, a characteristic specific to workloads with range predicates. Specifically, it is defined as the fraction of operations whose range predicates overlap with those of other operations. For example, if the overlap ratio is 60% in a workload containing 100 operations with range predicates, then 60 operations have pair-wise overlapping predicates, while the remaining 40 operations have predicates disjoint from all others.

We run the generated workloads against a PostgreSQL (v18) instance under Serializability to obtain transaction histories. All experiments are performed on a machine with dual Intel Xeon Gold 6526Y CPU and 512GB of memory.

Performance Comparison. We compare SerCheck with the state-of-the-art checker, checkConsistency [5] (specifically, its Serializability checking component), which also supports range predicates. It augments all write operations with `RETURNING` clauses to retrieve write sets and attaches write identifiers to track explicit write-read dependencies. For a fair comparison, we incorporate these features into SerCheck. In particular, we compare the write sets produced during execution against those returned by `RETURNING` clauses and prune order prefixes violating explicit write-read dependencies.

As shown in Figure 5, SerCheck significantly outperforms checkConsistency across various workloads. In particular, the performance gap widens as the number of sessions (a), transactions per session (b), or operations per transaction (c) increases, and becomes more pronounced as the workload becomes more write-intensive (d).

In addition, checkConsistency performs worse than our tool under high overlap ratios (e), with its runtime increasing rapidly once the overlap ratio exceeds 80%. This is particularly because in such cases write operations are more likely to overlap with subsequent reads. For example, after a delete operation, the removed rows no longer satisfy the corresponding read predicates, resulting in additional dependencies checkConsistency must infer.

Overall, these results demonstrate the efficiency of our DFA-based approach.

Effectiveness in Bug Detection. We evaluate SerCheck on a total of 404 key-value histories collected by prior work [4, 12, 17]. It verified all 106 serializable histories and correctly identified the remaining 298 histories as violating Serializability. In addition, SerCheck successfully reproduced 16 Serializability bugs reported by prior work [9, 10], all of which are triggered by test cases involving SQL queries.

5 Conclusion and Future Work

We have presented a DFA-based approach for checking Serializability over transaction histories with range predicates. We have also established its soundness and completeness. Preliminary results have shown both its effectiveness in finding isolation bugs and its promising checking performance.

Future work includes improving SerCheck's scalability and extending it to support advanced SQL features, e.g., subqueries, as well as weaker isolation levels [2, 3], e.g., repeatable read and read committed.

References

[1] 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.* 9, ICFP (2025). https://doi.org/10.1145/3747515

[2] Peter Bailis, Aaron Davidson, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (2013), 181–192. https://doi.org/10.14778/2732232.2732237

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10. https://doi.org/10.1145/568271.223785

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

[5] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. 2025. On the Complexity of Checking Mixed Isolation Levels for SQL Transactions. In *CAV '25*. 315–337. https://doi.org/10.1007/978-3-031-98685-7_15

[6] Zhiheng Cai, Si Liu, Hengfeng Wei, Yuxing Chen, and Anqun Pan. 2026. Fast Verification of Strong Database Isolation. *Proc. VLDB Endow.* 19, 4 (2026), 563–575. https://doi.org/10.14778/3785297.3785300

[7] Jack Clark, Manuel Rigger, John Wickerson, and Alastair Donaldson. 2026. Bringing Order to Practical Validation of Database Isolation Levels. *ACM Trans. Comput. Syst.* (March 2026). https://doi.org/10.1145/3797871 Just Accepted.

[8] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *PODC '17*. ACM, 73–82. https://doi.org/10.1145/3087801.3087802

[9] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *ICSE '23*. IEEE Press, 1123–1135. https://doi.org/10.1109/ICSE48619.2023.00101

[10] Xiyue Gao, Zhuang Liu, Yiran Shen, Hui Li, Yingfan Liu, Hongjun Xiao, Yanguo Peng, and Jiangtao Cui. 2025. Fucci: Database Transaction Fuzzing via Random Conflict Construction and Multilevel Constraint Solving. *Proc. VLDB Endow.* 18, 6 (Feb. 2025), 1879–1891. https://doi.org/10.14778/3725688.3725713

[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. https://doi.org/10.14778/3685800.3685866

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

[13] Kyle Kingsbury and Peter Alvaro. 2020. Elle: inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280. https://doi.org/10.14778/3430915.3430918

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

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

[16] 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. https://doi.org/10.1145/3720504

[17] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI '20*. USENIX Association, 63–80. https://www.usenix.org/conference/osdi20/presentation/tan

[18] 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 '25*. IEEE Computer Society, 3998–4010. https://doi.org/10.1109/ICDE65448.2025.00298

[19] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 654–671. https://doi.org/10.1145/3552326.3567492

Algorithm 2 Checking Observation Consistency

Input: A database state s, a transaction tx, and the map of observed query result M.
Output: $\top$ if the query results of tx are all consistent with observed results, or $\bot$ otherwise.

```
procedure ConsistentObservation(s, tx, M)
    for each op ∈ tx do
        if op ∈ R then
            if [[op]]_s ≠ M[op] then return ⊥
        else s ← [[op]]_s
    return ⊤
```

A Checking Observation Consistency

Observation-aware pruning. Algorithm 2 shows the procedure of checking whether a transaction tx on state s is consistent with the observed query result M. Given a state s that is the evaluation result of some equivalent order prefixes, we want to check whether the new prefix that appends a new tx is consistent. The procedure takes this state s, the new transaction tx, and the observed query result map as input. It then checks each op in tx in order. If this op is a read operation, it checks whether the query result produced by executing op on its most recent state is consistent with the observed one. If not consistent, we just return $\bot$. Otherwise, we continue to check the next op where the most recent state is not changed. If it is a write operation, we update the most recent state with the result of evaluating this write.

The observation-aware pruning is done by the ObservationConsistent method in the Transit rule. Given a state produced by a consistent order prefix and a new transaction, if the execution of this transaction is not consistent with the observation, then the premises of the Transit rule cannot be satisfied. As a result, the state produced by this transaction, the related transition, and all possible later states and transitions after this inconsistent state will not be added to the DFA. In other words, through pruning an inconsistent order prefix, ObservationConsistent actually prunes many total transaction orders at one time.

B Proof of Theorem 3.3

Soundness. Suppose there exists an accepting run in $\mathcal{A} = \textsc{ConstructDFA}(\mathcal{T}, \mathrm{SO}, \mathsf{M}, s_I, s_F)$, then there exist a sequence of states $q_0 q_1 ... q_n$ where $n = |\mathcal{T}|$ and a sequence $w = l_1 l_2, ... l_n$ over Σ, such that $\forall 0 < i \leq n. q_i = \Delta(q_{i-1}, l_i)$ and $q_n \in Q_f$. Let $\mathrm{CO} = \{(l_{i-1}, l_i) \mid q_i = \Delta(q_{i-1}, l_i) \wedge 0 < i \leq n\}$. By the Alph rule, we know that $\Sigma = \{l_1, l_2, \ldots, l_n\} = \mathcal{T} = \{tx_1, tx_2, \ldots, tx_n\}$.

1. Let $q_i = (s_i, \mathcal{T}_i)$ for all i. By Transit, we have $\mathcal{T}_i = \mathcal{T}_{i-1} \cup \{l_i\}$ and $l_i \in \mathcal{T}_i \setminus \mathcal{T}_{i-1}$. We also have $l_{i-1} \in \mathcal{T}_{i-1}$. By the session order constraints in Transit, we know that if $(l_i, l_{i-1}) \in \mathrm{SO}^+$, i.e., violates session order, then $l_i \notin \mathcal{T}_{i-2}$ where $\mathcal{T}_{i-2} = \mathcal{T}_i \setminus \{l_i, l_{i-1}\}$, therefore $q_{i-1} \neq \Delta(q_{i-2}, l_{i-1})$, which is a contradiction. Therefore CO must preserve SO.
2. By Transit and Exec, we know that a transition always reads the newest state. The ObservationConsistent ensures that $\forall i \in (0, n]. \forall op \in tx. op \in \mathcal{R} \implies [\![op]\!]_s = \mathsf{M}[op]$ where s is the newest state each op reads. Therefore, the executed output of each read operation is consistent with its observed query result.
3. By Init, Final, and the previous assumption, the sequence $w = l_1 l_2, ... l_n$ will make $q_0 = (s_I, \{\})$ transit to $q_n = (s_F, \mathcal{T})$. Since the transition is executing the transaction on both the database state and the used transaction set, executing $w = l_1 l_2, ... l_n$ on s_I will result in s_F.

Therefore, we find an CO that satisfies the definition 3.1, and the soundness is proved. □

Completeness. Suppose $\mathcal{H} = (\mathcal{T}, \mathrm{SO}, \mathsf{M}, s_I, s_F)$ is serializable, then there exists a commit order $\mathrm{CO} = \{(l_{i-1}, l_i) \mid q_i = \Delta(q_{i-1}, l_i) \wedge 0 < i \leq n\}$ such that it satisfies Definition 3.1, where $\Sigma = \{l_1, l_2, \ldots, l_n\} = \mathcal{T} = \{tx_1, tx_2, \ldots, tx_n\}$, and $n = |\mathcal{T}|$. Let a sequence $w = l_1 l_2, ... l_n$. By definition 3.1, we know that the executed output for each read operation is consistent with its observed query result. In addition, CO preserves SO. So we have $\forall i \in (0, n]. \exists q_i. q_i = \Delta(q_{i-1}, l_i)$, as applying l_i on q_i will always satisfy the premises of Transit. By executing w in order, we can get a sequence of states $q_0 q_1 \ldots q_n$. Since executing all transactions on s_I by CO will result s_F, and $q_0 = (s_I, \{\})$, we have $q_n = (s_F, \mathcal{T})$ by applying Transit n times start from q_0. Thus, $q_n \in Q_f$. That means we find an accepting of $\mathcal{A} = \textsc{ConstructDFA}(\mathcal{T}, \mathrm{SO}, \mathsf{M}, s_I, s_F)$. Thus, the completeness is proved. □

C Worst Case Analysis

The worst-case scenario for our approach happens when the workload has no opportunity for state merging or observation-aware pruning. An example workload pattern is that (1) every transaction contains only write operations, and (2) all writes update the same row, with each transaction writing a distinct value. Under this pattern, the database state reached after a prefix is determined by the order of transactions that have been executed. Consequently, two different prefixes cannot be merged into the same DFA state. For this workload pattern, the number of transitions in the DFA can also be exponential, since each reachable prefix may be extended by any transaction that has not yet executed and that preserves the session order. Furthermore, it is impossible to prune inconsistent states and transitions through observations as there is not any read operation. Thus, in the absence of effective state merging and observation-aware pruning, checking whether the DFA has an accepting run has exponential worst-case complexity with respect to the number of transactions.

This bound is pessimistic. Prior work of black-box isolation checking [14, 17] suggests that such maximal write contention is uncommon in practice. Real-world applications

usually distribute writes across many rows, and many transaction prefixes can reach the same database state and can be merged by our approach. Write-only workloads are also rare, as read operations are dominant in practice. Thus, many prefixes can be pruned through observations of reads.