



# AWDIT: An Optimal Weak Database Isolation Tester

LASSE MØLDRUP, Aarhus University, Denmark

ANDREAS PAVLOGIANNIS, Aarhus University, Denmark

*Database isolation* is a formal contract concerning the level of data consistency that a database provides to its clients. In order to achieve low latency, high throughput, and partition tolerance, modern databases forgo strong transaction isolation for *weak isolation* guarantees. However, several production databases have been found to suffer from *isolation bugs*, breaking their data-consistency contract. *Black-box testing* is a prominent technique for detecting isolation bugs, by checking whether histories of database transactions adhere to a prescribed isolation level.

In order to test databases on realistic workloads of large size, isolation testers must be as efficient as possible, a requirement that has initiated a study of the complexity of isolation testing. Although testing strong isolation has been known to be NP-complete, weak isolation levels were recently shown to be testable in polynomial time, which has propelled the scalability of testing tools. However, existing testers have a large polynomial complexity, restricting testing to workloads of only moderate size, which is not typical of large-scale databases. *How efficiently can we provably test weak database isolation?*

In this work, we develop AWDIT, *a highly-efficient and provably optimal tester for weak database isolation*. Given a history  $H$  of size  $n$  and  $k$  sessions, AWDIT tests whether  $H$  satisfies the most common weak isolation levels of Read Committed (RC), Read Atomic (RA), and Causal Consistency (CC) in time  $O(n^{3/2})$ ,  $O(n^{3/2})$ , and  $O(n \cdot k)$ , respectively, improving significantly over the state of the art. Moreover, we prove that AWDIT is essentially *optimal*, in the sense that there is a lower bound of  $n^{3/2}$ , based on the combinatorial BMM hypothesis, for *any* weak isolation level between RC and CC. Our experiments show that AWDIT is significantly faster than existing, highly optimized testers; e.g., for the  $\sim 20\%$  largest histories, AWDIT obtains an average speedup of 245 $\times$ , 193 $\times$ , and 62 $\times$  for RC, RA, and CC, respectively, over the best baseline.

CCS Concepts: • **Information systems**  $\rightarrow$  **Database management system engines**; • **Software and its engineering**  $\rightarrow$  **Consistency**; *Dynamic analysis*; • **Theory of computation**  $\rightarrow$  *Parameterized complexity and exact algorithms*.

Additional Key Words and Phrases: database testing, consistency, highly-available transactions (HATs)

## ACM Reference Format:

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

## 1 Introduction

Modern databases must handle enormous amounts of data, provide low latency, and be robust to network anomalies such as delays and partition faults. To respond to such demands, databases

Authors' Contact Information: Lasse Møldrup, Aarhus University, Aarhus, Denmark, [moeldrup@cs.au.dk](mailto:moeldrup@cs.au.dk); Andreas Pavlogiannis, Aarhus University, Aarhus, Denmark, [pavlogiannis@cs.au.dk](mailto:pavlogiannis@cs.au.dk).



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/6-ART209

<https://doi.org/10.1145/3742465>

typically forgo strong data consistency guarantees, such as any collection of concurrent database transactions admitting a serial view (i.e., being *serializable*). Instead, modern NewSQL and NoSQL databases support transactions that are weakly isolated, but ensure that the system remains available and efficient at all times (aka highly available transactions (HATs)) [Akkoorath et al. 2016; Bailis et al. 2016, 2013; Didona et al. 2018]. The precise database guarantees of data integrity are specified as *isolation levels* and have been subject to extensive formalization using various techniques such as axiomatic/graph-based [Adya et al. 2000; Bailis et al. 2016; Berenson et al. 1995; Terry et al. 1994], operational [Crooks et al. 2017], and most recently, using an atomic visibility relation [Biswas and Enea 2019; Burckhardt et al. 2014; Cerone et al. 2015]. Common examples of weak isolation include Read Committed (RC, the default level for most database transactions) [Bailis et al. 2013; Pavlo 2017], Read Atomic (RA) [Bailis et al. 2016; Cheng et al. 2021], and (Transactional) Causal Consistency [Akkoorath et al. 2016; Didona et al. 2018; Mehdi et al. 2017] (CC, available in, e.g., MongoDB [Mon 2024], Azure Cosmos [Azu 2024] and Neo4j [Neo 2024]).

Unfortunately, isolation levels can be tricky to understand and their implementation is error prone, with isolation bugs being continuously discovered in production databases [Jep 2024; Kingsbury and Alvaro 2020]. The prevalence of isolation bugs has been targeted by database-testing techniques. In particular, black-box testing is a popular approach, operating in two steps. First, a client interacts with the database and records (logs) its history of interaction as a collection of transactions, each sending and receiving data to and from the database. Second, an isolation tester analyzes the history and checks whether it adheres to the prescribed isolation level. This process involves large histories spanning thousands to millions of transactions, in order to create a realistic load that is likely to expose an isolation anomaly. As such, it has spawned a research interest in isolation testers that are as efficient as possible, both in theory, by analyzing the computational complexity of database isolation testing, and in practice, by utilizing clever optimizations.

Testing strong isolation levels, such as Serializability and Snapshot Isolation, is known to be NP-complete (in the number of operations performed) [Biswas and Enea 2019; Papadimitriou 1979], leading most isolation testers to utilize SAT/SMT solvers [Geng et al. 2024; Huang et al. 2023; Tan et al. 2020; Zhang et al. 2023]. On the other hand, weak isolation levels, such as Read Committed, Read Atomic, and Causal Consistency, have been shown to be checkable in polynomial time, allowing black-box testing algorithms of higher scalability, both in theory and in practice [Biswas and Enea 2019]. Elle is another popular database tester that runs in polynomial time and supports weak isolation levels [Kingsbury and Alvaro 2020], although its soundness is only guaranteed for certain types of transactions following “list-append” semantics. The most recent development in this progression has been Plume [Liu et al. 2024], which appears to be the only algorithm stating an explicit polynomial complexity, which is of degree 6 (in particular,  $O(n^3 \cdot \ell^2 \cdot k)$  for a history of  $n$  transactions,  $\ell$  keys and  $k$  sessions). Although other testers may possibly have a better complexity, Plume targets efficiency by utilizing efficient data structures including Vector Clocks [Friedemann 1989] and Tree Clocks [Mathur et al. 2022], and was shown to clearly outperform existing testers.

All these recent advances in database isolation testing highlight a demand for more performant testers, both in theory and in practice. *What is the precise complexity of testing weak database isolation? Are there provably optimal testing algorithms?* We address this challenge in this work by developing AWDIT (A Weak Database Isolation Tester): *a highly-efficient tester for weak database isolation that is provably optimal* under standard assumptions.

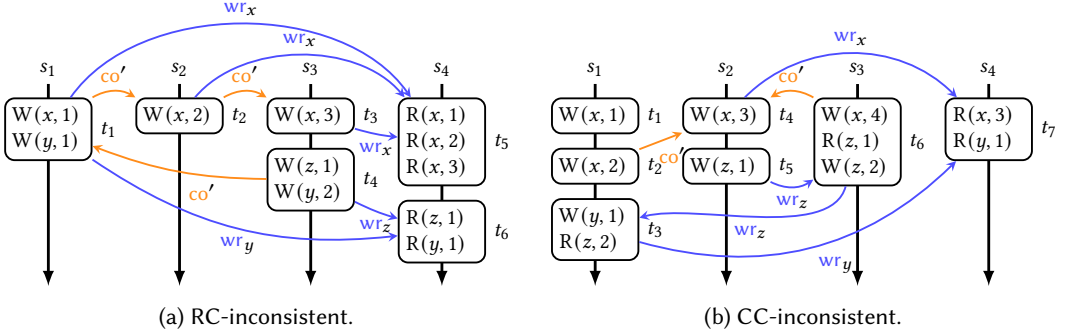


Fig. 1. An RC-inconsistent history (a) and a CC-inconsistent history (b). AWDIT infers a small set of partial commit edges  $\text{co}'$  that are sufficient to witness the inconsistency in each case and identify small witnesses by means of simple cycles. Inferred  $\text{co}'$  edges that go along so  $\cup$   $\text{wr}$  are not shown explicitly.

### 1.1 Motivating Example

We illustrate violations of the Read Committed (RC) and Causal Consistency (CC) isolation levels on two small histories in Fig. 1. At a high level, the task of an isolation tester is to determine a *commit order*  $\text{co}$  that is a total order on all transactions and satisfies certain properties, specific to the prescribed isolation level. This  $\text{co}$  must also agree with the *session order*, written so, which totally orders the transactions of each session (shown in vertical black arrows), and the *write-read order*  $\text{wr}$ , which pairs transactions that common data is written by and read from (shown in blue).

Similarly to other testers, AWDIT infers a *partial relation*  $\text{co}'$  based on isolation-level-dependent inference rules. Its key advantage lies in  $\text{co}'$  being small enough to be efficiently computable and yet sound and complete, in the sense that the history adheres to the isolation level iff  $\text{co}'$  is acyclic: if not, a cycle in  $\text{co}'$  witnesses an isolation anomaly, whereas if yes, any total extension of  $\text{co}'$  serves as the commit order  $\text{co}$  that witnesses conformance to the isolation level (see Definition 3.1).

Let us see why each of the two histories in Fig. 1 violate their respective isolation levels and how AWDIT determines this fact.

**Read Committed.** RC states that (i) only committed transactions can be read, and (ii) a transaction  $t$  cannot read a key  $x$  from another transaction  $t'$ , if  $t$  has previously observed (i.e., read a value from) a transaction that writes to  $x$  and is  $\text{co}$ -after  $t'$ . Exploiting that  $\text{co}$  must be a total order, we can view this requirement as an inference rule: if a transaction  $t_3$  first observes some transaction  $t_2$  that writes to  $x$ , and then  $t_3$  reads  $x$  from  $t_1$ , then (ii) implies (we infer)  $t_2 \xrightarrow{\text{co}} t_1$  for any RC-consistent commit order  $\text{co}$  (see Fig. 3a for a visual depiction).

Let us apply the above inference process to the history in Fig. 1a, inferring the edges labeled  $\text{co}'$ . The fact that these edges form a cycle, when including that  $t_3 \xrightarrow{\text{so}} t_4$ , then proves that *no* total commit order exists, demonstrating that the history does not satisfy RC. Since  $t_1 \xrightarrow{\text{wr}_x} t_5$  (via  $R(x, 1)$ ) and later  $t_2 \xrightarrow{\text{wr}_x} t_5$  (via  $R(x, 2)$ ), we infer  $t_1 \xrightarrow{\text{co}} t_2$ . Similarly, since  $t_2 \xrightarrow{\text{wr}_x} t_5$  and later  $t_3 \xrightarrow{\text{wr}_x} t_5$ , we infer  $t_2 \xrightarrow{\text{co}} t_3$ . Finally, since  $t_4$  writes to  $y$ ,  $t_4 \xrightarrow{\text{wr}_y} t_6$ , and later  $t_1 \xrightarrow{\text{wr}_y} t_6$ , we infer  $t_4 \xrightarrow{\text{co}} t_1$ , completing the cycle. AWDIT constructs a  $\text{co}'$  that contains exactly these three edges (as well as so and  $\text{wr}$  edges). Notably, AWDIT does not directly create some inferrable  $\text{co}'$  orderings, as long as they are present transitively (in  $(\text{co}')^+$ ), such as  $t_1 \xrightarrow{\text{co}'} t_4$  and  $t_2 \xrightarrow{\text{co}'} t_1$  in this example. More importantly, it does not even need to check whether such transitive orderings are present. Overall, AWDIT spends only  $O(\sqrt{n})$  time per transaction on average.

**Causal Consistency.** Intuitively, CC states that transactions must obey causality: if one transaction could have caused another, an observer should not observe the effect without also observing the cause. Formally, a transaction  $t_1$  is causally dependent on a transaction  $t_2$ , if there is a sequence of so and wr edges connecting  $t_2$  to  $t_1$ , written succinctly as  $t_2 \xrightarrow{\text{so} \cup \text{wr}}^+ t_1$ . CC specifies that if a transaction  $t$  reads a key  $x$  from another transaction  $t'$ , then  $t'$  must be the **co**-latest among all transactions writing to  $x$  that  $t$  is causally dependent on. We can also phrase this as an inference rule: if a transaction  $t_3$  reads  $x$  from another transaction  $t_1$  and causally depends on a transaction  $t_2$  that writes to  $x$ , we can infer  $t_2 \xrightarrow{\text{co}} t_1$  (see Fig. 3c for a visual depiction).

Let us see how AWDIT infers the **co**' in Fig. 1b via the above inference rule. Since  $t_7$  reads  $x$  from  $t_4$ , while  $t_2$  and  $t_6$  write to  $x$ , and  $t_2 \xrightarrow{\text{so} \cup \text{wr}}^+ t_7$  and  $t_6 \xrightarrow{\text{so} \cup \text{wr}}^+ t_7$ , we have  $t_2 \xrightarrow{\text{co}'} t_4$  and  $t_6 \xrightarrow{\text{co}'} t_4$ . The latter **co**' edge completes a cycle witnessing non-conformance to CC. Again, no further **co**' edges need to be inferred, with the guarantee that the existing ones represent all inferrable paths in the graph. Finally, the inferred **co**' edges are computed in an efficient way that requires  $O(k)$  time per transaction on average, where  $k$  is the number of sessions.

## 1.2 Our Contributions

Here we state the main results of this paper, while we refer to the following sections for relevant definitions, algorithms, and lemmas. All proofs are in the technical report [Møldrup and Pavlogiannis 2025b].

**Upper bounds.** First, we address the problem of testing the weak isolation levels Read Committed (RC) and Read Atomic (RA). We consider histories of size  $n$ , measured as the number of read/write operations they contain. We show that testing for RC and RA can be achieved in sub-quadratic time, which is much faster than existing isolation testers, as stated in the following theorem.

**THEOREM 1.1.** *Given a history  $H$  of size  $n$ , checking whether  $H$  satisfies RC or RA can be decided in  $O(n^{3/2})$  time.*

We also remark that, when the size of each transaction is  $O(1)$ , the algorithms behind Theorem 1.1 yield  $O(n)$  running time. Next, we turn our attention to the third common isolation level of Causal Consistency (CC) and prove that it can be tested in quadratic time in general and in sub-quadratic time when the number of sessions is small, again, improving significantly over existing testers.

**THEOREM 1.2.** *Given a history  $H$  of size  $n$  and  $k$  sessions, checking whether  $H$  satisfies CC can be decided in  $O(n \cdot k)$  time.*

Normally, the number of sessions  $k$  is significantly smaller than the number of operations  $n$  of the history. This stems from practical limitations of database deployment and is also prevalent in database-testing benchmarks [Biswas and Enea 2019; Kingsbury and Alvaro 2020; Liu et al. 2024]. In such cases, the bound of Theorem 1.2 becomes sub-quadratic and takes a linear form, when  $k = O(1)$ .

**Lower bounds.** The above complexity improvements make it natural to ask: *Are further improvements possible? Is a linear bound possible for testing weak isolation?* We now turn our attention to lower bounds, showing that Theorem 1.1 and Theorem 1.2 are essentially (conditionally) optimal.

It is well known that Boolean Matrix Multiplication (BMM) can be computed in cubic time by the standard textbook algorithm. The corresponding combinatorial BMM hypothesis states that there is no combinatorial algorithm achieving a truly sub-cubic bound for matrix multiplication [Williams

2019]. Although the term “combinatorial algorithm” does not have a rigorous definition, it generally means an algorithm that does not rely on algebraic, fast matrix multiplication (FMM) techniques and must work irrespective of the structure over which the product is defined.

To state our first lower bound in its full generality, given two isolation levels  $\mathcal{I}_1, \mathcal{I}_2$ , we write  $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$  to denote that  $\mathcal{I}_1$  is stronger than  $\mathcal{I}_2$ , meaning that any history satisfying  $\mathcal{I}_1$  also satisfies  $\mathcal{I}_2$ .

**THEOREM 1.3.** *Consider any isolation level  $\mathcal{I}$  with  $\text{CC} \sqsubseteq \mathcal{I} \sqsubseteq \text{RC}$  and the problem of testing whether a history  $H$  of size  $n$  satisfies  $\mathcal{I}$ . For any fixed  $\epsilon > 0$ , there is*

- (1) *no combinatorial algorithm that runs in  $O(n^{3/2-\epsilon})$  time, under the combinatorial BMM hypothesis, and*
- (2) *no algorithm that runs in  $O(n^{\omega/2-\epsilon})$  time, where  $\omega$  is the matrix multiplication exponent.*

Theorem 1.3 is, perhaps, surprisingly general: it states that the  $n^{3/2}$  lower bound holds, not only for RA, RC, and CC, but also for *any* isolation level between them. It further implies that, among combinatorial algorithms, our algorithms for RA and RC are *optimal*, while our algorithm for CC may only be improved by a sub-linear factor  $\sqrt{n}$ . Although this does not exclude faster isolation testers that use FMM, it is relevant for two reasons. First, it has, thus far, been unclear whether FMM is useful in database testing. Hence, our lower bound can be interpreted as “unless we find a way to use FMM in isolation testing, the  $O(n^{3/2})$  bound is likely tight”. Second, although faster in theory, FMM is generally slow in practice because of large leading constants, and thus considered impractical. Finally, Item 2 of Theorem 1.3 implies that a nearly linear-time tester (possibly relying on FMM) would be a major breakthrough, while a truly linear-time (i.e.,  $O(n)$ ) tester is *impossible* [Coppersmith and Winograd 1982].

Next, note that the lower bound of Theorem 1.3 holds when the number of sessions  $k$  is unbounded. Zooming into each isolation level separately, we show that, in fact, RA retains its  $n^{3/2}$  lower bound already with two sessions.

**THEOREM 1.4.** *Consider the problem of testing whether a history  $H$  of size  $n$  and 2 sessions satisfies RA. For any fixed  $\epsilon > 0$ , there is*

- (1) *no combinatorial algorithm that runs in  $O(n^{3/2-\epsilon})$  time, under the combinatorial BMM hypothesis, and*
- (2) *no algorithm that runs in  $O(n^{\omega/2-\epsilon})$  time, where  $\omega$  is the matrix multiplication exponent.*

Going one step further, we show that RC retains its  $n^{3/2}$  lower bound even with just *one* session.

**THEOREM 1.5.** *Consider the problem of testing whether a history  $H$  of size  $n$  and 1 session satisfies RC. For any fixed  $\epsilon > 0$ , there is*

- (1) *no combinatorial algorithm that runs in  $O(n^{3/2-\epsilon})$  time, under the combinatorial BMM hypothesis, and*
- (2) *no algorithm that runs in  $O(n^{\omega/2-\epsilon})$  time, where  $\omega$  is the matrix multiplication exponent.*

Theorem 1.5 might be surprising, in the sense that analogous consistency problems for concurrent programs with a single thread are trivial (i.e., in linear time). Finally, it is natural to ask how efficiently we can test RA with only  $k = 1$  session. Does it suffer, like RC, the lower bound of  $n^{3/2}$ ? As the following theorem states, one-session histories are testable in linear time for RA.

**THEOREM 1.6.** *Given a history  $H$  of  $n$  operations and  $k = 1$  session, checking whether  $H$  satisfies RA can be decided in  $O(n)$  time.*

**In summary.** Our results draw a fairly complete picture of the (fine-grained) complexity of weak database isolation testing. In summary, for  $k = 1$  session, RA is testable in  $O(n)$  time and is easier than RC. For  $k \geq 2$ , both RA and RC are testable in  $O(n^{3/2})$  time. Moreover, for *any number of sessions*, our algorithms for testing RA and RC are (conditionally) *optimal*. Testing CC takes  $O(n \cdot k)$  time and becomes super-linear only in the presence of many sessions, whereas as the number of sessions grows, *any* isolation level between RC and CC is unlikely to scale better than  $n^{3/2}$ .

**Implementation and experiments.** We develop AWDIT, a prototype tool that implements our algorithms for testing weak isolation levels. We evaluate the efficiency of AWDIT on standard benchmarks and compare its performance against all weak isolation testers from recent literature. Our experiments reveal a clear advantage for AWDIT, which is always significantly faster and achieves speedups that exceed 1000 $\times$  in extreme cases, over all existing weak isolation testers.

## 2 Preliminaries

We start with relevant definitions and notation regarding database transaction histories and weak isolation levels. Our exposition mostly follows recent works [Biswas and Enea 2019; Liu et al. 2024].

### 2.1 Definitions

**Notation on relations.** A (binary) relation  $R$  over a set  $X$  is a subset of  $X \times X$ . We write  $x \xrightarrow{R} y$  to mean  $\langle x, y \rangle \in R$ . The identity relation over  $X$  is denoted by  $[X] = \{\langle x, x \rangle \mid x \in X\}$ . The inverse of  $R$  is  $R^{-1}$ . The reflexive closure and transitive closure of  $R$  are  $R^?$  and  $R^+$ , respectively, also written as  $x \xrightarrow{R^?} y$  and  $x \xrightarrow{R^+} y$ . A relation  $R$  over  $X$  is *irreflexive* if  $\langle x, x \rangle \notin R$  for all  $x \in X$ , and  $R$  is *acyclic* if  $R^+$  is irreflexive. For two relations  $R_1, R_2$  over a common domain, we say that  $R_1$  *respects*  $R_2$  (equivalently,  $R_2$  respects  $R_1$ ), if  $R_1 \cup R_2$  is irreflexive.

**Databases.** We consider transactional key-value databases over a set of keys  $\text{Key} = \{x, y, \dots\}$  and a set of values  $\text{Val}$ . Clients send operations to the database in the form of reads and writes. The set of possible operations for a set of keys  $\text{Key}$  and a set of values  $\text{Val}$  is denoted  $\text{Op} = \{R_i(x, v), W_i(x, v) \mid i \in \text{OpId}, x \in \text{Key}, v \in \text{Val}\}$ , where  $\text{OpId}$  is a set of operation identifiers. When not relevant, we omit the operation identifier and simply write  $R(x, v)$  or  $W(x, v)$ . For brevity, we sometimes also refer to operations simply as  $r$ ,  $w$ , or  $o$ , depending on if they are reads, writes, or arbitrary. In such cases, the key of an operation  $o$  is denoted by  $o.\text{key}$ , and its value by  $o.\text{val}$ .

**Transactions.** Client interactions with a database are grouped in transactions.

*Definition 2.1.* A transaction  $t = \langle O, \text{po} \rangle$  is a set of operations  $O \subseteq \text{Op}$  and a *program order*  $\text{po}$ , which is a strict total order over  $O$ .

For a transaction  $t = \langle O, \text{po} \rangle$ , the set of all read (resp. write) operations in  $t$  is  $t|_R = \{R(x, v) \in O\}$  (resp.  $t|_W = \{W(x, v) \in O\}$ ). This is naturally extended to sets of transactions  $T$ , i.e.,  $T|_R = \bigcup_{t \in T} t|_R$  and  $T|_W = \bigcup_{t \in T} t|_W$ . The set of operations in  $t$  acting on a key  $x \in \text{Key}$  is denoted by  $t|_x = \{o \in O \mid o.\text{key} = x\}$ . The set of reads in  $t$  reading a key  $x \in \text{Key}$  is denoted by  $t|_{R(x)} = t|_R \cap t|_x$ , and the set of writes in  $t$  writing to  $x$  is  $t|_{W(x)} = t|_W \cap t|_x$ . We also extend  $T|_x$ ,  $T|_{R(x)}$ , and  $T|_{W(x)}$  to sets of transactions  $T$  in the natural way. For  $o \in O$ , we let  $o.\text{txn} = t$ . The set of keys read (resp. written) by  $t$  is denoted by  $\text{KeysRd}(t)$  (resp.  $\text{KeysWt}(t)$ ). If  $t$  contains a write to  $x$ , we say that  $t$  writes  $x$ .

**Histories.** At a high level, the collection of transactions between a database and its clients constitutes a *history*. The *session order*, written so, over the transactions captures the total order of transactions executed in a single session and is thus a union of disjoint total orders.



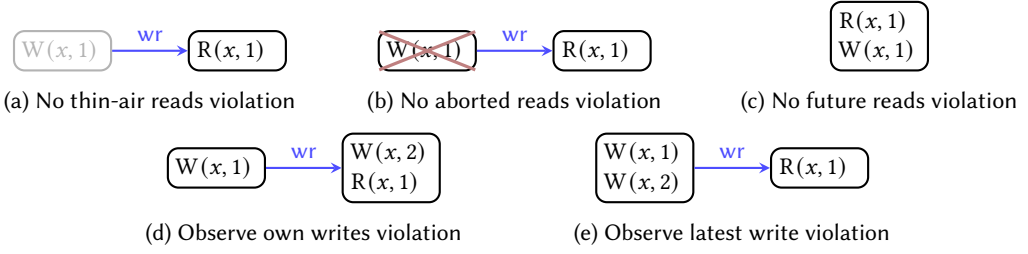


Fig. 2. Examples of violations of the five axioms of Read Consistency.

In the setting of black-box database testing, the writes sent to the database are controlled by the tester. Since database implementations are normally data-independent [Wolper 1986], i.e., their behavior is independent of the concrete values written/read by the transactions, database testers use unique values on each write, because all isolation anomalies are preserved under this interaction scheme. This implies that each read  $R(x, v)$  observes the unique write  $W(x, v)$  sent to the database in some (possibly remote) transaction. Formally, the two events are related by the *write-read* relation  $wr \subseteq T|_W \times T|_R$ , where  $T$  is the set of all transactions. We occasionally view  $wr$  as a relation on distinct transactions, i.e.  $t_1 \xrightarrow{wr} t_2$  iff (i)  $t_1 \neq t_2$ , and (ii)  $w \xrightarrow{wr} r$ , where  $w \in t_1|_W$  and  $r \in t_2|_R$ . We also write  $t \xrightarrow{wr} r$  to denote that  $w \xrightarrow{wr} r$  for some  $w \in t|_W$  and read  $r \notin t$ . Finally, we project  $wr$  onto a specific key by writing  $wr_x = wr \cap [T|_x]$ . Transactions can either commit or abort; intuitively, an aborted transaction should not be visible to other transactions.

**Definition 2.2.** A history  $H = \langle T, so, wr \rangle$  is a set of transactions  $T = T_c \uplus T_a$ , a (strict partial) *session order*  $so \subseteq T \times T$ , and a *write-read order*  $wr \subseteq T|_W \times T|_R$ , where  $T_c$  is a set of committed transactions and  $T_a$  is a set of aborted transactions. We require that  $wr^{-1}$  is a (partial) function.

We let the set of sessions of  $H$  be  $sessions(H) = \{s_1, s_2, \dots, s_k\}$ , and for a session  $s \in sessions(H)$ , we let  $H|_s$  be the *committed* transactions of  $H$  belonging to  $s$ . If  $t \in H|_s$ , we let  $t.sess = s$ . The *size* of  $H$  is the total number of operations it contains.

## 2.2 Weak Isolation Levels

We follow the standard axiomatic approach of isolation specification using a commit order [Biswas and Enea 2019]. However, we are also interested in capturing more fine-grained transaction anomalies, which are assumed away in those axiomatic definitions. For this purpose, we adapt some of the Transactional Anomalous Patterns (TAPs) proposed recently in [Liu et al. 2024].

**Read Consistency.** Read Consistency intuitively states that each read on  $x$  observes either an earlier write on  $x$  in its own transaction, or, if no such write exists, the last write on  $x$  of a committed transaction<sup>1</sup>. Formally, this is stated as five basic axioms (illustrated in Fig. 2 as TAPs).

**Definition 2.3 (Read Consistency).** A history  $H = \langle T, so, wr \rangle$  satisfies *Read Consistency* if the following conditions hold.

- (a) No thin-air reads:  $\forall r \in T_c|_R, \exists w \in T|_W : w \xrightarrow{wr} r$ .
- (b) No aborted reads:  $\forall r \in T_c|_R, \forall w \in T|_W : w \xrightarrow{wr} r \implies w \notin T_a|_W$ .
- (c) No future reads:  $\forall r \in T_c|_R, \forall w \in T|_W : w \xrightarrow{wr} r \implies \neg(r \xrightarrow{po} w)$ .
- (d) Observe own writes:  $\forall r \in T_c|_R, \forall w \in T|_W : w \xrightarrow{wr} r \wedge w.txn \neq r.txn \implies \nexists w' \in T|_{W(r.key)} : w' \xrightarrow{po} r$ .

<sup>1</sup>This is equivalent to disallowing G1a and G1b in [Adya et al. 2000], in addition to their basic assumptions on histories.

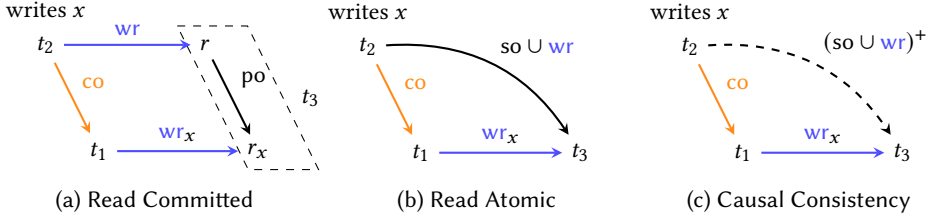


Fig. 3. The axioms of Read Committed (a), Read Atomic (b), and Causal Consistency (c). In each case, the  $co$  ordering is required when the other orderings hold.

(e) Observe latest write:  $\forall r \in T_c|_R, \forall w, w' \in T|_{W(r.key)} : w \xrightarrow{wr} r \wedge w \xrightarrow{po} w' \implies r \xrightarrow{po} w'$ .

We are now ready to define the three main weak isolation levels of Read Committed, Read Atomic, and Causal Consistency. Each level requires Read Consistency, as well as an additional axiom involving a *commit order*  $co$ , which is a strict total order over all committed transactions that respects  $so \cup wr$  and also satisfies a predicate specific to the isolation level at hand.

**Read Committed (RC).** The Read Committed<sup>2</sup> isolation level formalizes the intuition that the database can only read from committed transactions, and also adheres to a monotonicity requirement: a transaction  $t$  is not allowed to read a key  $x$  from another transaction  $t'$ , if it has previously observed (i.e., read a value from) a transaction that writes to  $x$  and is  $co$ -later than  $t'$ .

*Definition 2.4 (Read Committed).* A history  $H = \langle T, so, wr \rangle$  satisfies *Read Committed* (RC), if it is Read Consistent, and there is a strict total *commit order*  $co$  over  $T_c$  respecting  $so \cup wr$ , such that the following holds (see Fig. 3a for a pictorial depiction).

$$\forall x \in \text{Key}, \forall t_1, t_2 \in T_c, \forall r, r_x \in T_c|_R :$$

$$t_1 \neq t_2 \wedge t_1 \xrightarrow{wr_x} r_x \wedge t_2 \text{ writes } x \wedge t_2 \xrightarrow{wr} r \xrightarrow{po} r_x \implies t_2 \xrightarrow{co} t_1 .$$

*Example 2.5.* The history in Fig. 4a does not satisfy RC. In particular,  $t_1 \xrightarrow{so} t_2$  forces that  $t_1 \xrightarrow{co} t_2$ . Hence, the second read of  $x$  in  $t_3$  should read  $t_2$  instead of  $t_1$ . The history in Fig. 4b, on the other hand, satisfies RC. Even though  $t_3$  only observes the latter of the writes in  $t_2$ ,  $t_1$  is observed first, so there is no violation.

**Read Atomic (RA).** The Read Atomic isolation level formalizes the intuition that transactions should be atomic, in the sense that either all or none of the effects of a transaction can be observed.

*Definition 2.6 (Read Atomic).* A history  $H = \langle T, so, wr \rangle$  satisfies *Read Atomic* (RA), if it is Read Consistent, and there is a strict total *commit order*  $co$  over  $T_c$  respecting  $so \cup wr$ , such that the following holds (see Fig. 3b for a pictorial depiction).

$$\forall x \in \text{Key}, \forall t_1, t_2, t_3 \in T_c : t_1 \neq t_2 \wedge t_1 \xrightarrow{wr_x} t_3 \wedge t_2 \text{ writes } x \wedge t_2 \xrightarrow{so \cup wr} t_3 \implies t_2 \xrightarrow{co} t_1 .$$

*Example 2.7.* Consider again the history in Fig. 4b. Transaction  $t_3$  reads  $y$  from  $t_2$ , but does not read its write to  $x$ , instead reading the older version written by  $t_1$ . Hence,  $t_3$  observes some, but not all, effects of  $t_2$ , violating RA. The history in Fig. 4c, on the other hand, satisfies RA. Even though  $t_4$  displays weak behavior by reading from  $t_1$  instead of  $t_2$ , it observes all effects of the transactions that it directly reads from.

<sup>2</sup>Some literature [Crooks et al. 2017] interprets RC as proscribing G1 from [Adya et al. 2000], which is the weaker requirement of Read Consistency plus acyclicity of  $so \cup wr$ . This is easily checkable in  $O(n)$  time, for a history of size  $n$ .



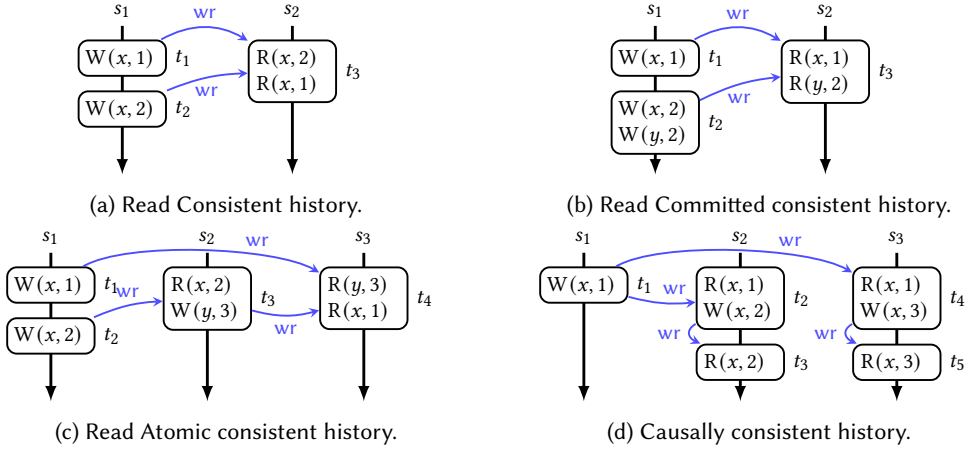


Fig. 4. Examples of consistent histories that violate consistency of stronger isolation levels.

**Causal Consistency (CC).** Causal Consistency<sup>3</sup> specifies that reads must respect causal relationships between transactions: intuitively, if a transaction  $t$  reads a key  $x$  from another transaction  $t'$ , then  $t'$  must be the **co**-latest among all transactions that  $t$  is causally dependent on, and write to  $x$ . The notion of causality is formalized via the *happens-before* relation, dictating that transaction  $t_1$  happens before transaction  $t_2$ , if  $t_1 \xrightarrow{\text{so} \cup \text{wr}}^+ t_2$ .

*Definition 2.8 (Causal Consistency).* A history  $H = \langle T, \text{so}, \text{wr} \rangle$  satisfies *Causal Consistency* (CC), if it is Read Consistent, and there is a strict total *commit order* **co** over  $T_c$  respecting  $\text{so} \cup \text{wr}$ , such that the following holds (see Fig. 3c for a pictorial depiction).

$$\forall x \in \text{Key}, \forall t_1, t_2, t_3 \in T_c: \quad t_1 \neq t_2 \wedge t_1 \xrightarrow{\text{wr}_x} t_3 \wedge t_2 \text{ writes } x \wedge t_2 \xrightarrow{\text{so} \cup \text{wr}}^+ t_3 \implies t_2 \xrightarrow{\text{co}} t_1.$$

*Example 2.9.* Consider again the history in Fig. 4c, which does not satisfy CC. Transaction  $t_4$  observes  $t_2$  through its read on  $y$ , and it should therefore not observe  $t_1$ , which happens before  $t_2$ . The history in Fig. 4d, on the other hand, satisfies CC. Note that there is still weak behavior, however, as both  $t_2$  and  $t_4$  read a value of 1 on  $x$  and then overwrite it, making the history non-serializable.

**Comparison of isolation levels.** Given two isolation levels  $I_1, I_2$ , we say that  $I_1$  is *stronger than*  $I_2$ , denoted by  $I_1 \sqsubseteq I_2$ , if any history that satisfies  $I_1$  also satisfies  $I_2$ .

**The consistency problem.** The primary task of a back-box isolation tester is consistency checking: given an isolation level  $I \in \{\text{RC}, \text{RA}, \text{CC}\}$  and history  $H$ , decide whether  $H$  satisfies  $I$ .

### 3 Weak Isolation Algorithms

We now present algorithms for checking consistency under RC, RA, and CC, towards Theorem 1.1 and Theorem 1.2. Each algorithm starts by checking the axioms of Read Consistency (Fig. 2). Given a history of size  $n$ , this check can easily be carried out in  $O(n)$  time. The precise algorithm for this task is delegated to the technical report [Møldrup and Pavlogiannis 2025b].

Each axiom of RC, RA, and CC requires the existence of a commit order **co** satisfying certain properties (Fig. 3). For an input history  $H$ , the respective algorithm builds a *partial* commit relation **co'** that holds *necessary* orderings, in the sense that *any* **co** witnessing the consistency of  $H$  satisfies

<sup>3</sup>Sometimes also called Transactional Causal Consistency [Akkoorath et al. 2016; Liu et al. 2024].

$\text{co}' \subseteq \text{co}$ . This implies that, if  $\text{co}'$  is cyclic, then  $H$  is inconsistent. Moreover, at the end of the algorithm's execution, the orderings in  $\text{co}'$  are also *sufficient*, in the sense that if  $\text{co}'$  is acyclic, any linearization of  $\text{co}'$  serves as the total commit order  $\text{co}$  witnessing the consistency of  $H$ . The key property of  $\text{co}'$  is that it is *saturated* and *minimal*, as defined below.

**Definition 3.1 (Saturated and minimal commit relations).** Given an isolation level  $I \in \{\text{RC}, \text{RA}, \text{CC}\}$  and a history  $H = \langle T, \text{so}, \text{wr} \rangle$ , a (partial) commit relation  $\text{co}'$  is *saturated* for  $I$  if (i)  $\text{so} \cup \text{wr} \subseteq \text{co}'$ , and (ii) if the premise in Fig. 3 holds for  $I$ , for transactions  $t_1, t_2$ , and  $t_3$  (i.e., the respective figure without the  $\text{co}$  edge), then  $t_2 \xrightarrow{\text{co}'}^+ t_1$ . Moreover,  $\text{co}'$  is *minimal* for  $I$  if, for any transactions  $t_1$  and  $t_2$  with  $t_2 \xrightarrow{\text{co}'} t_1$ , either  $t_2 \xrightarrow{\text{so} \cup \text{wr}} t_1$  or Fig. 3 requires  $t_2 \xrightarrow{\text{co}} t_1$  for  $I$  (possibly both).

We note that saturated relations for consistency exist in the literature (e.g., [Biswas and Enea 2019]), but our  $\text{co}'$  has an advantage due to its minimality, which allows for more efficient algorithms. The correctness of the presented algorithms is based on the fact that saturated and minimal commit relations exactly characterize the consistency of  $H$ , as stated in the following lemma.

**LEMMA 3.2.** *Given an isolation level  $I \in \{\text{RC}, \text{RA}, \text{CC}\}$ , a history  $H$ , and a minimal saturated commit relation  $\text{co}'$ ,  $H$  satisfies  $I$  iff  $H$  satisfies Read Consistency and  $\text{co}'$  is acyclic.*

### 3.1 Read Committed

In this section, we present the algorithm for checking consistency for RC (Algorithm 1).

**Description of algorithm.** The algorithm starts by checking the history for Read Consistency (Line 2). Then, it initializes  $\text{co}'$  as  $\text{so} \cup \text{wr}$  (Line 3), which must hold for  $\text{co}'$  to be saturated. The main part of the algorithm saturates  $\text{co}'$  according to the RC axiom (Fig. 3a), by looping over all committed transactions  $t_3$  (Line 4). The loop on Line 6 iterates over each transaction  $t_2$  that  $t_3$  reads from, and stores in the set  $\text{firstTxnReads}$  the first read operation of  $t_3$  reading from  $t_2$ . The algorithm then loops over all reads  $R(y, v)$  in  $t_3$  in reverse order (Line 12), while maintaining the set of keys that have been read below the current read in the  $\text{readKeys}$  variable (Line 21). This is because  $R(y, v)$  plays the role of  $r$  in the RC axiom (Fig. 3a), hence the intersection on Line 15 contains all  $x$  such that  $t_2 \xrightarrow{\text{wr}} R(y, v) \xrightarrow{\text{po}} r_x$ , where  $t_2$  is some transaction writing  $x$ , and  $r_x$  reads  $x$ . To achieve the stated complexity, it is crucial to only compute this intersection once for each  $t_2$ , hence the check on Line 14. By inspecting Fig. 3a, it is apparent that no  $\text{co}$  edges are missed this way, since  $R(y, v)$  is the po-first read of  $t_2$  by  $t_3$ . Any reads  $r_x$  po-below

#### Algorithm 1: Read Committed

---

```

1 Def CheckRC( $H = \langle T, \text{so}, \text{wr} \rangle$ ):
  // From [Møldrup and Pavlogiannis 2025b]
2 CheckReadConsistency( $H$ )
3  $\text{co}' \leftarrow \text{so} \cup \text{wr}$ 
4 for  $t_3 = \langle O, \text{po} \rangle \in T_c$  do
5    $\text{readTxns} \leftarrow \emptyset$ ;  $\text{firstTxnReads} \leftarrow \emptyset$ 
6   for  $r \in t_3|_R$  in po order do
7     Let  $t_2$  be such that  $t_2 \xrightarrow{\text{wr}} r$ 
8     if  $t_2 \notin \text{readTxns}$  then
9        $\text{readTxns} \leftarrow \text{readTxns} \cup \{t_2\}$ 
10       $\text{firstTxnReads} \leftarrow \text{firstTxnReads} \cup \{r\}$ 
11    $\text{earliestWts} \leftarrow \lambda x. (\perp, \perp)$ ;  $\text{readKeys} \leftarrow \emptyset$ 
12   for  $R(y, v) \in t_3|_R$  in reverse po order do
13     Let  $t_2$  be such that  $t_2 \xrightarrow{\text{wr}} R(y, v)$ 
14     if  $R(y, v) \in \text{firstTxnReads}$  then
15       // Loop over the smaller set
16       for  $x \in \text{KeysWt}(t_2) \cap \text{readKeys}$  do
17          $t_1 \leftarrow \text{earliestWts}[x][1]$ 
18         if  $t_1 = t_2$  then  $t_1 \leftarrow \text{earliestWts}[x][0]$ 
19         if  $t_1 \neq \perp$  then  $\text{co}' \leftarrow \text{co}' \cup \{(t_2, t_1)\}$ 
20     if  $\text{earliestWts}[y][1] \neq t_2$  then
21        $\text{earliestWts}[y] \leftarrow \langle \text{earliestWts}[y][1], t_2 \rangle$ 
22        $\text{readKeys} \leftarrow \text{readKeys} \cup \{y\}$ 
23 if  $\text{co}'$  has a cycle then report cycle

```

---

$R(y, v)$  reading a key  $x$  from this intersection could then create a  $\text{co}$ -inference: if there is  $t_1 \neq t_2$  such that  $t_1 \xrightarrow{\text{wr}} r_x$ , we have  $t_2 \xrightarrow{\text{co}} t_1$ . However, recall that a saturated  $\text{co}'$  only needs to contain this ordering *transitively* (cf. Definition 3.1):  $t_2 \xrightarrow{\text{co}'}^+ t_1$ . Hence, it suffices to infer  $\text{co}'$  for the earliest such read (in po). In particular, consider two reads  $r_x$  and  $r'_x$  reading  $x$  from  $t_1$  and  $t'_1$ , respectively, with  $R(y, v) \xrightarrow{\text{po}} r_x \xrightarrow{\text{po}} r'_x$ . When the algorithm processes the first read of  $t_1$  on Line 12, it infers  $t_1 \xrightarrow{\text{co}'} t'_1$ , thus it only remains to infer  $t_2 \xrightarrow{\text{co}'} t_1$ . The algorithm efficiently identifies  $t_1$  as follows.

The *earliestWts* map (Line 11) maintains, for each key  $x$ , the two po-earliest *unique* transactions from which  $t_3$  reads  $x$  in the future. It is essentially a stack of two elements for each key, where a new transaction ejects the oldest writer (Line 19 and Line 20). When finding the transaction  $t_1$  writing the value read for a key  $x$ , the top element of the stack is chosen (Line 16), except if the top is equal to  $t_2$ , in which case the second element is used (Line 17). Finally,  $t_2 \xrightarrow{\text{co}'} t_1$  is added on Line 18.

To understand the need for this two-element stack, suppose the algorithm instead always used the most recent transaction that  $t_3$  read  $x$  from. One could have  $r \xrightarrow{\text{po}} r_x \xrightarrow{\text{po}} r'_x$ , where  $r$  and  $r_x$  read from the *same* transaction  $t_2$ . In such a case, there should still be a  $\text{co}'$  ordering between  $t_2$  and the writer of  $r'_x$ , which would be missed.

The correctness of Algorithm 1 follows by arguing that  $\text{co}'$  is saturated and minimal, thereby applying Lemma 3.2.

LEMMA 3.3. *Given a history  $H$ , Algorithm 1 reports a violation iff  $H$  does not satisfy RC.*

**Running time.** Read Consistency can be checked in linear time, so the running time is dominated by the loop on Line 15. The intersection in this loop is performed by iterating over the smaller of the two sets, which leads to amortized  $O(\sqrt{n})$  time. We sketch the argument here. Call a transaction *large*, if it has more than  $\sqrt{n}$  reads, and call it *small* otherwise. We count separately the total running time for small and large transactions encountered on Line 4. Note that there are  $\leq \sqrt{n}$  large transactions, hence we argue that each large transaction  $t_3$  takes  $O(n)$  time. This is true because  $t_2$  is unique each time we enter Line 15, and  $\sum_{t_2 \xrightarrow{\text{wr}} t_3} |\text{KeysWt}(t_2)| = O(n)$ . We now turn our attention to small transactions. For each small transaction  $t_3$ , we have  $|\text{readKeys}| \leq |t_3|_{\text{R}}$ , hence the inner loop runs  $O(|t_3|_{\text{R}}^2)$  times. The sum of these is maximized, when each  $|t_3|_{\text{R}} = \theta(\sqrt{n})$ . In this case, there are  $O(\sqrt{n})$  small transactions, yielding  $O(n^{3/2})$  total time. Finally, note that when each transaction has constant size  $O(1)$ , the above argument yields  $O(n)$  running time. We thus arrive at the following lemma, which concludes Theorem 1.1 for RC.

LEMMA 3.4. *Given a history  $H$  of size  $n$ , Algorithm 1 runs in  $O(n^{3/2})$  time.*

### 3.2 Read Atomic

In this section, we present the algorithm for checking consistency for RA (Algorithm 2).

**Description.** The algorithm for RA is similar to Algorithm 1 in its overall approach. It starts by checking Read Consistency (Line 2), followed by checking the *repeatable reads* property (Line 3). In short, repeatable reads states that committed transactions don't read the same key from different transactions, and is implied by the RA axiom (Fig. 3b). The algorithm proceeds by initializing  $\text{co}'$  (Line 4) and then looping over all sessions  $s$  (Line 5) and all committed transactions  $t_3$  in  $s$  (Line 7). The algorithm maintains *lastWrite*, which holds, for each key  $x$ , the latest transaction in  $s$

**Algorithm 2:** Read Atomic

---

```

1  Def CheckRA( $H = \langle T, \text{so}, \text{wr} \rangle$ ):
   // From [Møldrup and Pavlogiannis 2025b]
2  CheckReadConsistency( $H$ )
3  CheckRepeatableReads( $H$ )
4   $\text{co}' \leftarrow \text{so} \cup \text{wr}$ 
5  for  $s \in \text{sessions}(H)$  do
6     $\text{lastWrite} \leftarrow \lambda x. \perp$ 
7    for  $t_3 \in H|_s$  in so order do
8      for  $t_1 \xrightarrow{\text{wr}_x} t_3$  do
9         $t_2 \leftarrow \text{lastWrite}[x]$ 
10       if  $t_1 \neq t_2 \neq \perp$  then
11          $\text{co}' \leftarrow \text{co}' \cup \{\langle t_2, t_1 \rangle\}$ 
12     for  $t_2 \xrightarrow{\text{wr}} t_3$  do
13       // Loop over the smaller set
14       for  $x \in \text{KeysWt}(t_2) \cap \text{KeysRd}(t_3)$  do
15         //  $t_1$  is unique due to repeatable reads
16         Let  $t_1$  be such that  $t_1 \xrightarrow{\text{wr}_x} t_3$ 
17         if  $t_1 \neq t_2$  then
18            $\text{co}' \leftarrow \text{co}' \cup \{\langle t_2, t_1 \rangle\}$ 
19     for  $x \in \text{KeysWt}(t_3)$  do
20        $\text{lastWrite}[x] \leftarrow t_3$ 
21 if  $\text{co}'$  has a cycle then
22   report cycle

```

---

```

21 Def CheckRepeatableReads( $H = \langle T, \text{so}, \text{wr} \rangle$ ):
22 for  $t \in T_c$  do
23    $\text{lastWriter} \leftarrow \lambda x. \perp$ 
24   for  $R(x, v) \in t|_R$  in po order do
25     if  $t \neq W(x, v). \text{txn} \neq \text{lastWriter}[x] \neq \perp$  then
26       // Found a cycle between the writer of  $v$ 
27       and  $\text{lastWriter}[x]$ 
28       report non-repeatable read
29     else
30        $\text{lastWriter}[x] \leftarrow W(x, v). \text{txn}$ 

```

---

so far that writes  $x$ . The RA axiom includes the condition  $t_2 \xrightarrow{\text{so} \cup \text{wr}} t_3$ , which is handled as two separate cases. The so case is handled by the loop on Line 8, whereas the wr case is handled by the loop on Line 12. For the so case, the algorithm exploits that saturation only requires *transitive*  $\text{co}'$  orderings, when the RA axiom applies; in a scenario  $t'_2 \xrightarrow{\text{so}} t_2 \xrightarrow{\text{so}} t_3$ , where  $t_2$  and  $t'_2$  write  $x$  and  $t_1 \xrightarrow{\text{wr}_x} t_3$ , it is only necessary to infer  $t_2 \xrightarrow{\text{co}'} t_1$ , because  $t'_2 \xrightarrow{\text{so}} t_2 \xrightarrow{\text{co}'} t_1$ . In the wr case, the algorithm iterates all possible  $t_2$  (Line 12) and finds exactly those keys  $x$  for which the RA axiom could apply by computing an intersection (Line 13). As with RC, it is crucial for the complexity that this intersection is performed by iterating over the smaller set. The inferred  $\text{co}'$  edges are added (Line 11 and Line 16) and  $\text{co}'$  is finally checked for cycles (Line 19).

The complexity of Algorithm 2 follows a similar line of reasoning to that of RC: the running time is dominated by the loop on Line 13, which can be shown to run in amortized  $O(\sqrt{n})$  time, by again reasoning about small and large transactions separately. Formally, the correctness and complexity of Algorithm 2 is captured in the following lemmas, which conclude Theorem 1.1 for RA.

LEMMA 3.5. *Given a history  $H$ , Algorithm 2 reports a violation iff  $H$  does not satisfy RA.*

LEMMA 3.6. *Given a history  $H$  of size  $n$ , Algorithm 2 runs in  $O(n^{3/2})$  time.*

### 3.3 Causal Consistency

In this section, we present the algorithm for checking consistency for CC (Algorithm 3).

---

**Algorithm 3:** Causal Consistency
 

---

```

1 Def CheckCC( $H = \langle T, \text{so}, \text{wr} \rangle$ ):
   // From [Moldrup and Pavlogiannis 2025b]
2 CheckReadConsistency( $H$ )
3  $HB \leftarrow \text{ComputeHB}(H)$ 
4  $\text{co}' \leftarrow \text{so} \cup \text{wr}$ 
5 for  $s \in \text{sessions}(H)$  do
6    $\text{lastWrite}_{s'} \leftarrow \lambda x. \perp$  for each  $s' \in \text{sessions}(H)$ 
7   for  $t_3 \in H|_s$  in so order do
8     for  $t_1 \xrightarrow{\text{wr}_x} t_3$  do
9       for  $s' \in \text{sessions}(H)$  do
10        for  $t_2 \in \text{Writes}_{s'}[x]$  so2-after  $\text{lastWrite}_{s'}[x]$  do
11          if  $t_2 \xrightarrow{\text{so}^?} HB_{t_3}[s']$  then
12             $\text{lastWrite}_{s'}[x] \leftarrow t_2$ 
13          else break
14        if  $t_1 \neq \text{lastWrite}_{s'}[x] \neq \perp$  then
15           $\text{co}' \leftarrow \text{co}' \cup \langle \text{lastWrite}_{s'}[x], t_1 \rangle$ 
16 if  $\text{co}'$  has a cycle then report cycle

17 Def ComputeHB( $H = \langle T, \text{so}, \text{wr} \rangle$ ):
18 if  $\text{so} \cup \text{wr}$  has a cycle then report cycle
19 for  $s \in \text{sessions}(H)$  do
20    $HB^s \leftarrow [\perp, \dots, \perp]$ 
21 Let  $\sigma$  be a topological sort of  $\text{so} \cup \text{wr}$ 
22 for  $t \in \sigma$  do
23    $s \leftarrow t.\text{sess}$ 
24    $HB_t \leftarrow HB^s \sqcup \sqcup_{t' \xrightarrow{\text{wr}} t} HB_{t'}$ 
25    $HB^s \leftarrow HB_t[s \mapsto t]$ 
26 return  $HB$ 

```

---

**Description.** The algorithm starts by checking Read Consistency (Line 2) and computing the happens before relation (Line 3) by calling  $\text{ComputeHB}(H)$ . In turn, this function verifies that  $\text{so} \cup \text{wr}$  is acyclic (Line 18) and computes the happens before relation as a set of Vector Clocks  $HB_t$ , one for each transaction  $t$ . Vector Clocks are indexed by sessions, so that for each  $s \in \text{sessions}(H)$ ,  $HB_t[s]$  holds the so-latest transaction  $t'$  of  $s$  such that  $t' \xrightarrow{\text{so} \cup \text{wr}}^+ t$ . The join operation between two Vector Clocks  $A$  and  $B$  (used on Line 24) is defined as a point-wise maximum wrt so, i.e.,

$$A \sqcup B = \lambda s. \left( A[s] \xrightarrow{\text{so}} B[s] \quad ? \quad B[s] \quad : \quad A[s] \right).$$

The algorithm then initializes  $\text{co}'$  to  $\text{so} \cup \text{wr}$  (Line 4) and enters its main computation in the loop of Line 5, so as to saturate  $\text{co}'$  based on selective applications of the CC axiom in Fig. 3c. This is achieved by iterating over all transactions  $t_3$  of each session  $s$ , in so-order. To make the computation efficient, the algorithm relies on two simple data structures. The last-writer data structure  $\text{lastWrite}_{s'}[x]$  points to the so-latest transaction  $t'_2$  writing  $x$  of session  $s'$  such that  $t'_2 \xrightarrow{\text{so} \cup \text{wr}}^+ t_3$ . In accordance with the CC axiom, for  $t_2 = \text{lastWrite}_{s'}[x]$  and  $t_1$ , the transaction for which  $t_1 \xrightarrow{\text{wr}_x} t_3$ , if  $t_1 \neq t_2$ , we have  $t_2 \xrightarrow{\text{co}'} t_1$  (Line 15). Importantly, the algorithm avoids inserting orderings  $t'_2 \xrightarrow{\text{co}'} t_1$  from transactions  $t'_2$  that are so-predecessors of  $t_2$ , since these will be ordered before  $t_1$  transitively via  $t_2$ . Finally, the last-writer data structure  $\text{lastWrite}_{s'}[x]$  is updated by traversing  $\text{Writes}_{s'}[x]$ , which is an array storing the transactions of  $s'$  that write on  $x$ , in so order. The key insight is that last

writers grow monotonically with so: for  $t_3$  and  $t'_3$  of  $s$  with  $t_3 \xrightarrow{\text{so}} t'_3$ , we have

$$\{t_2 \in T \mid t_2 \xrightarrow{\text{so} \cup \text{wr}}^+ t_3\} \subseteq \{t_2 \in T \mid t_2 \xrightarrow{\text{so} \cup \text{wr}}^+ t'_3\}.$$

This implies that, after processing  $t_3$  and proceeding to  $t'_3$ ,  $\text{lastWrite}_{s'}[x]$  does not have to scan the array  $\text{Writes}_{s'}[x]$  from the beginning, but rather proceed from where it left off on  $t_3$  (Line 10).

**Running time.** We now sketch the running time of Algorithm 3.  $\text{ComputeHB}(H)$  clearly takes  $O(n \cdot k)$ , by spending  $O(k)$  time per event for each join operation. For every  $t_1 \xrightarrow{\text{wr}_x} t_3$  in Line 8, the algorithm spends  $O(k)$  time, if we exclude the inner loop of Line 10. Finally, the total time spent per session in the inner loop of Line 10 is bounded by  $O(n)$ , since, as argued above, each array  $\text{Writes}_{s'}[x]$  is scanned once for each transaction  $s$ . We thus arrive at a total running time of  $O(n \cdot k)$ .

Formally, the correctness and complexity of Algorithm 3 is captured in the following lemmas, which conclude Theorem 1.2.

LEMMA 3.7. *Given a history  $H$ , Algorithm 3 reports a violation iff  $H$  does not satisfy CC.*

LEMMA 3.8. *Given a history  $H$  of size  $n$  and  $k$  sessions, Algorithm 3 runs in  $O(n \cdot k)$  time.*

### 3.4 Witnesses of Reported Violations

Besides merely reporting whether a history  $H$  satisfies a given isolation level, it is informative to extract witnesses of isolation anomalies. It is also desirable, that we extract several independent anomalies that are possibly present in  $H$ . Although our algorithms so far make coarse-grained reports (e.g., in terms of the existence of a  $\text{co}'$  cycle) for reasons of brevity, here we present some fine-grained witness-reporting strategies that are easily obtainable with our algorithms.

**Read Consistency.** Each algorithm starts by checking Read Consistency (Fig. 2), required by all isolation levels. Each read is checked independently, thus the algorithms can report *all* reads failing one of the five basic axioms (e.g., future read or thin-air read). Even in the presence of violations, consistency checking can still proceed, by discarding the reads that suffer an anomaly at this level.

**Causality cycles.** The next weakest anomaly is the presence of causality ( $\text{so} \cup \text{wr}$ ) cycles, which violates all isolation levels. Though the number of cycles can be exponential, one can obtain meaningful witnesses by reporting one cycle per strongly connected component (SCC) of  $\text{so} \cup \text{wr}$ . At this point, consistency checks for RC and RA (the axioms that do not involve  $(\text{so} \cup \text{wr})^+$ ) may continue, while consistency checks for CC is likely to produce too many violation reports.

**Commit-order violations.** Next, we proceed to isolation-level-specific anomalies. The repeatable read property of RA is checked (and reported) independently for each transaction. All other anomalies (for all isolation levels) involve the presence of a  $\text{co}'$  cycle. Again, we find it meaningful to report one cycle per SCC, but it is also insightful to consider the edges constituting each cycle. One approach is to prioritize cycles that contain the fewest non- $(\text{so} \cup \text{wr})$  edges, which is likely to report weaker (and thus more serious) anomalies.

**Reporting all violations.** Finally, we remark that more exhaustive witness reporting is possible. Although it may come at a higher complexity cost, it only has to execute after *the first* violation is reported, which can be done optimally using the algorithms of this section. Since the vast majority of tested histories do not have violations, this approach still benefits from our faster algorithms.



## 4 Complexity Lower Bounds

In this section, we turn our attention to the lower bounds of Theorem 1.3, Theorem 1.4, and Theorem 1.5, which broadly state that testing weak database isolation on histories of size  $n$  essentially requires  $n^{3/2}$  time, in the sense that polynomial improvements over this scaling are unlikely.

**Triangle freeness and boolean matrix multiplication.** Triangle freeness is a simple graph-theoretic problem: given an undirected graph  $G = \langle V, E \rangle$ , does it contain a triangle, i.e., three nodes  $a, b, c$  with  $\langle a, b \rangle, \langle b, c \rangle, \langle a, c \rangle \in E$ ? Triangle freeness has been studied extensively. It is solvable in  $O(n^3)$  time, on a graph of  $n$  nodes, and, although faster algorithms exist, it is also BMM-hard [Williams and Williams 2018]. This means that any combinatorial algorithm computing triangle freeness in  $O(n^{3-\epsilon})$  time would imply the existence of a combinatorial algorithm for multiplying two  $n \times n$  matrices in  $O(n^{3-\epsilon'})$  time, for fixed  $\epsilon, \epsilon' > 0$ . The latter is considered unlikely (or at least notoriously difficult). It also implies that triangle freeness cannot be solved in  $O(n^{\omega-\epsilon})$  time, where  $\omega$  is the matrix multiplication exponent.

Our lower bounds are based on fine-grained reductions from triangle freeness.

### 4.1 A General Lower Bound for Weak Isolation Testing

We begin with Theorem 1.3. Given an undirected graph  $G = \langle V, E \rangle$ , we construct a history  $H = \langle T, \text{so}, \text{wr} \rangle$  such that, for any isolation level  $I$  with  $\text{CC} \subseteq I \subseteq \text{RC}$ , we have that  $H$  satisfies  $I$  iff  $G$  is triangle-free. We achieve this by means of a *range reduction*, which has the property that (i) if  $G$  is triangle-free, then  $H$  satisfies  $\text{CC}$  (and thus also  $I$ ), and (ii) if  $H$  satisfies  $\text{RC}$  (and thus also  $I$ ), then  $G$  is triangle-free.

**Construction.** For each node  $a \in V$ ,  $H$  has two (committed) transactions  $t_a^R$  and  $t_a^W$ . We call the former the *read transaction* and the latter the *write transaction* of  $a$ .

- The read transaction  $t_a^R$  begins with a sequence of reads  $R(x_b^a, b)$ , one for each edge  $\langle b, a \rangle \in E$ . Next (in po),  $t_a^R$  executes a sequence of reads  $R(x_b, b)$ , one for each edge  $\langle b, a \rangle \in E$ .
- The write transaction  $t_a^W$  contains a sequence of writes  $W(x_b, a)$  and  $W(x_a^b, a)$ , one for each edge  $\langle a, b \rangle \in E$ , as well as a write  $W(x_a, a)$ . The po in  $t_a^W$  is irrelevant.

Note that, for a given key, every read observes a unique value. In particular, the  $\text{wr}$  relation is fully characterized by the following orderings. For every edge  $\langle a, b \rangle \in E$ , we have (i)  $W(x_b^a, a) \xrightarrow{\text{wr}} R(x_b^a, a)$  and (ii)  $W(x_a, a) \xrightarrow{\text{wr}} R(x_a, a)$ , where each write appears in  $t_a^W$  and each read appears in  $t_b^R$ . Finally, each transaction appears in its own session (i.e.,  $\text{so} = \emptyset$ ). See Fig. 5 for an illustration.

**Correctness.** We now sketch the correctness of the construction. Consider an edge  $\langle a, b \rangle \in E$ . The ordering  $W(x_b^a, a) \xrightarrow{\text{wr}} R(x_b^a, a)$  ensures that  $t_a^W \xrightarrow{\text{wr}} R(x_b^a, a) \xrightarrow{\text{po}} R(x_c, c)$ , for any  $R(x_c, c)$  of  $t_b^R$  and edge  $\langle b, c \rangle \in E$ . At this point, a triangle will be formed iff  $\langle a, c \rangle \in E$ . If so, then  $t_a^W$  also writes to  $x_c$ , forcing the commit order  $t_a^W \xrightarrow{\text{co}} t_c^W$  according to the semantics of  $\text{RC}$  (Fig. 3a). Repeating the argument symmetrically implies  $t_c^W \xrightarrow{\text{co}} t_a^W$ , making  $H$  inconsistent. On the other hand, if there is no triangle, no  $\text{co}$  orderings are forced between write transactions, meaning that they can be committed in any order (followed by the read transactions).

*Example 4.1.* Let us illustrate the above argument on the example in Fig. 5. The edge  $\langle 3, 1 \rangle$  implies  $t_3^W \xrightarrow{\text{wr}} R(x_3^2, 3) \xrightarrow{\text{po}} R(x_2, 2)$ , where the reads belongs to  $t_1^R$  and  $R(x_2, 2)$  reads from  $W(x_2, 2)$  of  $t_3^W$ . Further, the edge  $\langle 3, 2 \rangle$  implies that  $t_3^W$  also writes on  $x_2$ , namely via  $W(x_2, 3)$ , implying a

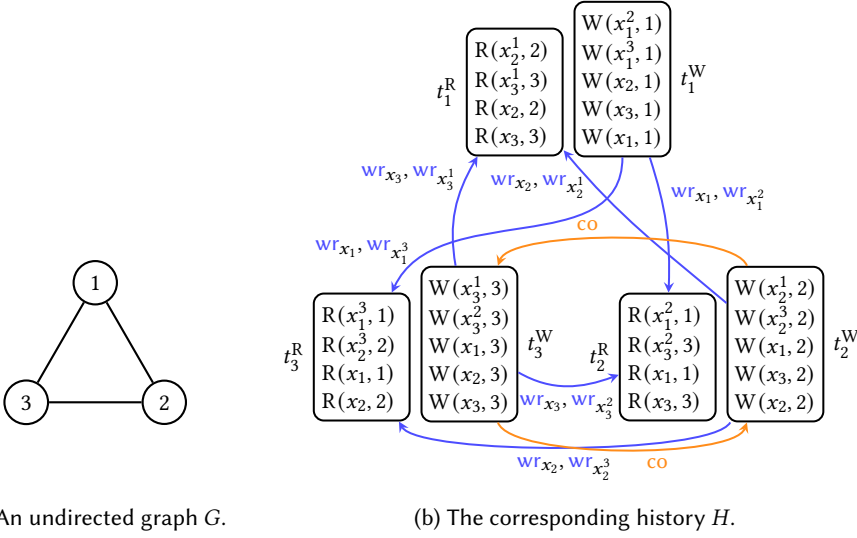


Fig. 5. The history  $H$  given an undirected graph  $G$ . Using the semantics of RC (Fig. 3c), we derive  $t_3^W \xrightarrow{\text{co}} t_2^W$  and  $t_2^W \xrightarrow{\text{co}} t_3^W$ , implying that  $H$  does not satisfy RC, indicating that  $G$  has a triangle.

commit order  $t_3^W \xrightarrow{\text{co}} t_2^W$ . Exchanging nodes 3 and 2 and repeating this argument yields  $t_2^W \xrightarrow{\text{co}} t_3^W$ , producing a cycle that witnesses the inconsistency of  $H$  under RC.

Formally, we have the following lemma.

LEMMA 4.2. *The following assertions hold.*

- (1) *If  $G$  is triangle-free, then  $H$  satisfies the CC isolation level.*
- (2) *If  $H$  satisfies the RC isolation level, then  $G$  is triangle free.*

Finally, observe that if  $G$  has  $n$  nodes and  $m$  edges,  $H$  has size  $O(m)$ , which is bounded by  $O(n^2)$ . Thus, if there is a combinatorial algorithm for the consistency of  $H$  in time  $O(m^{3/2-\epsilon}) = O(n^{3-\epsilon'})$ , for some fixed  $\epsilon, \epsilon' > 0$ , then triangle freeness in  $G$  would be determined in time  $O(n^{3-\epsilon'})$ , contradicting the combinatorial BMM hypothesis. Similarly, if there is an algorithm for the consistency of  $H$  in time  $O(m^{\omega/2-\epsilon}) = O(n^{\omega-\epsilon'})$ , then triangle freeness would be determined in  $O(n^{\omega-\epsilon'})$ . This concludes the proof of Theorem 1.3.

#### 4.2 Lower Bounds with One and Two Sessions

Our algorithms for RA and RC have time complexity of  $O(n^{3/2})$ , even if the number of sessions is small. It is thus natural to ask whether faster testing algorithms exist for histories with a small number of sessions. This section proves Theorem 1.4 and Theorem 1.5, which state that it is unlikely to break below  $n^{3/2}$  for RA even with only two sessions and for RC with just one session. The proofs are by a modification of the reduction of Section 4.1.

**Reduction for RA.** The transactions of  $H$  are the same as in Section 4.1, except that operations on keys  $x_a^b$  are removed. Formally, for every node  $a \in V$ ,  $H$  has two transactions  $t_a^R$  and  $t_a^W$ .

- For each edge  $\langle b, a \rangle \in E$ , the read transaction  $t_a^R$  executes a read  $R(x_b, b)$ . The po order of these is irrelevant.

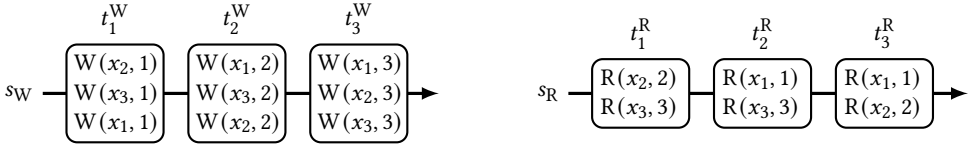


Fig. 6. The RA-inconsistent history  $H$  for the undirected graph  $G$  of Fig. 5a, consisting of two sessions.

- For each edge  $\langle a, b \rangle \in E$ , the write transaction  $t_a^W$  executes a write  $W(x_b, a)$ . Finally, it executes a write  $W(x_a, a)$ . The po order is, again, irrelevant.

Note that  $\text{wr}$  is fully specified by the orderings  $W(x_a, a) \xrightarrow{\text{wr}} R(x_a, a)$  for each edge  $\langle a, b \rangle \in E$ , where  $W(x_a, a)$  is an operation of  $t_a^W$  and  $R(x_a, a)$  is an operation of  $t_b^R$ . The session order so consists of two sessions  $s_W$  and  $s_R$ , executing all write and read transactions, respectively, in some arbitrary order. It can be easily verified, that  $\text{so} \cup \text{wr}$  is acyclic. See Fig. 6 for an illustration.

**Correctness for RA.** The correctness of the construction can be intuitively stated as follows. First, the existence of a triangle  $\langle a, b, c \rangle$  implies the commit orderings  $t_a^W \xrightarrow{\text{co}} t_c^W$  and  $t_c^W \xrightarrow{\text{co}} t_a^W$ , witnessing the inconsistency of  $H$ . This holds, because the existence of edges  $\langle a, b \rangle, \langle c, b \rangle \in E$  implies that  $t_a^W \xrightarrow{\text{wr}} t_b^R$  and  $t_c^W \xrightarrow{\text{wr}} t_b^R$ , while the existence of the edge  $\langle a, c \rangle \in E$  implies that both  $t_a^W$  and  $t_c^W$  write the key of the other, i.e.,  $x_c$  and  $x_a$ , respectively. On the other hand, if there is no triangle, no such  $\text{co}$  orderings are imposed. The fact that all write and read transactions can be grouped into two sessions  $s_W$  and  $s_R$  follows by inspecting the RA axiom (Fig. 3b): so is irrelevant, since any transaction that reads ( $t_3$  in Fig. 3b) is so-unordered with any transaction that writes ( $t_1$  in Fig. 3b). Thus,  $H$  is consistent by first committing all transactions in  $s_W$  and then all transactions in  $s_R$ . Formally, we have the following lemma, which concludes Theorem 1.4.

LEMMA 4.3.  $H$  satisfies the RA isolation level iff  $G$  is triangle-free.

**Read Atomic with one session.** A natural question is whether a  $n^{3/2}$  lower bound holds for RA with a single session. The answer is no, as consistency in this case is checkable optimally in linear time, by scanning the single session once and keeping track of the most recent transaction writing to each location. This concludes Theorem 1.6.

**Reduction for RC.** Finally, we turn our attention to the reduction for RC. The construction is the same as in Section 4.1, except that we place all transactions in one session (first write transactions, followed by the read transactions). The existence of a triangle  $\langle a, b, c \rangle$ , again, implies two conflicting commit orderings  $t_a^W \xrightarrow{\text{co}} t_c^W$  and  $t_c^W \xrightarrow{\text{co}} t_a^W$ , by following exactly the same argument as in Theorem 1.3. On the other hand, the absence of a triangle implies that  $H$  is consistent, since no additional  $\text{co}$  orderings are imposed, and the axiom of RC (Fig. 3a) does not involve so (thus, trivially  $\text{co} = \text{so}$  in this case). Formally, we have the following lemma, which concludes Theorem 1.5.

LEMMA 4.4.  $H$  satisfies the RC isolation level iff  $G$  is triangle-free.

## 5 Implementation and Experiments

In this section, we report on an implementation of our algorithms in Section 3 as a tool, and on an experimental evaluation of its performance against existing weak isolation testers.

**The AWDIT weak isolation tester.** AWDIT (A W e a k D a t a b a s e i s e t a t e r) is a Rust implementation of our algorithms for testing weak isolation levels from Section 3. For CC, the implementation differs from Algorithm 3 by computing  $HB$  on the fly and replacing *lastWrite*

with binary search, which we found performed better. It parses database transaction histories in various formats also used by other isolation testers such as Plume [Liu et al. 2024], PolySI [Huang et al. 2023], DBCop [Biswas and Enea 2019], and Cobra [Tan et al. 2020]. Finally, AWDIT follows witness-reporting strategies close to those described in Section 3.4.

### 5.1 Experimental Setup

The goal of our experiments is to shed light on the efficiency of existing database isolation testers, and how AWDIT performs in comparison. For this reason, we have followed the experimental setup of recent literature on database isolation testing [Biswas and Enea 2019; Huang et al. 2023; Liu et al. 2024; Tan et al. 2020], relying on databases and benchmarks utilized in those works.

**Databases.** We make use of the following databases.

- PostgreSQL 17.0, a popular relational database [Pos 2024].
- CockroachDB 24.2.4, a relational database that achieves high availability [Coc 2024]. We ran this database as a cluster of three local replicas.
- RocksDB 5.15.10, a fast key-value database [Roc 2024].

**Benchmarks.** In order to simulate realistic client interaction with the aforementioned databases, we use the following benchmarks.

- TPC-C, an online transaction processing (OLTP) benchmark [TPC 2024].
- C-Twitter, a benchmark from the Cobra framework [Tan et al. 2020] simulating the handling of real-time big data at Twitter [Twi 2011].
- RUBiS, an auction site benchmark modeled after eBay [Amza et al. 2002].

**History generation.** A concrete history is generated by specifying a database and a benchmark, as well as some benchmark-specific parameters such as the number of sessions and the number of transactions. We rely on the framework of Cobra [Tan et al. 2020] for this task, which configures each databases to provide strong transaction isolation. Each history is then given as input to an isolation tester in its respective format, together with an isolation level.

**Weak isolation testers.** We use the following database isolation testers, covering recent literature.

- AWDIT, developed in this work.
- Plume, the most recent and optimized weak isolation tester [Liu et al. 2024] that supports RC, RA, and CC. Implemented in Java, Plume utilizes Vector Clocks [Friedemann 1989] (like AWDIT) and Tree Clocks [Mathur et al. 2022] to efficiently compute a valid commit order, and was shown to significantly outperform all existing testers.
- DBCop, a polynomial-time tester [Biswas and Enea 2019] that supports CC, implemented in Rust.
- CausalC+, a Datalog-based tester for CC [Liu et al. 2024; Zennou et al. 2022].
- TCC-Mono, a MonoSAT-based tester for CC based on monotonic SMTs [Bayless et al. 2015; Liu et al. 2024].
- PolySI, a MonoSAT-based tester for Snapshot Isolation (SI) [Huang et al. 2023].

For CausalC+ and TCC-Mono, we used implementations from the experimental setup of [Liu et al. 2024]. Note that, since  $SI \subseteq RC, RA, CC$ , PolySI can be used to make complete (but possibly unsound) reports of weak-isolation anomalies. Finally, we have excluded Elle [Kingsbury and Alvaro 2020] from our experiments because it is generally unsound (its sound “list-append” mode is not

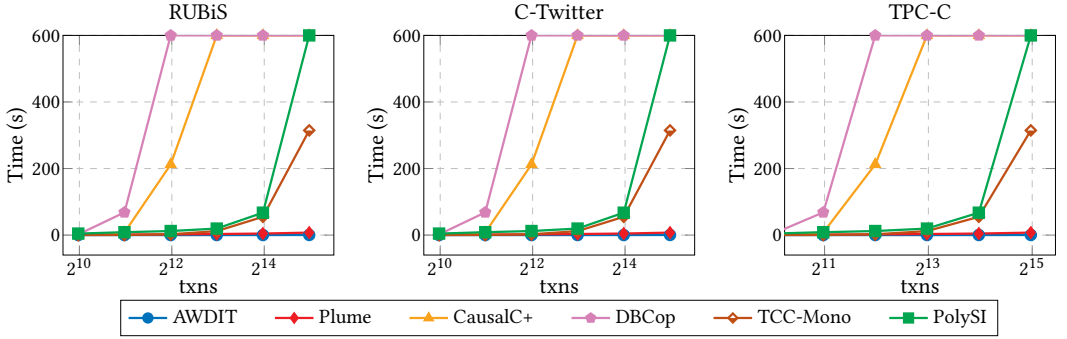


Fig. 7. Running times of all isolation testers for checking Causal Consistency on histories collected from CockroachDB, on three benchmarks (RUBiS, C-Twitter, and TPC-C), using 50 sessions. The timeout is set to 10m.

applicable here). Our experiments are run on an Ubuntu 22.04 machine with a second-generation 2.3 GHz AMD Epyc CPU and 64 GB of memory.

## 5.2 Small-Scale Experiments

Since different isolation testers have different complexity guarantees (from polynomial to exponential), we perform a preliminary set of small-scale experiments to obtain an indication for the scalability of each tester.

**Setup.** We generate histories using all three benchmarks and by scaling the number of transactions within a small range  $[2^{10}, 2^{15}]$ , while keeping the number of sessions fixed at 50. We execute Plume, DBCop, and AWDIT at the CC isolation level (recall that Causal+ and TCC-Mono run at CC by default, while PolySI runs at SI). We set a timeout of 10 minutes for processing each history.

**Results.** Fig. 7 shows the results for the three benchmarks running on CockroachDB, using 50 sessions and varying the number of transactions. DBCop, PolySI, CausalC+ and PolySI scale poorly, while AWDIT and Plume run almost instantaneously. This is in alignment with the experimental observations in [Liu et al. 2024], which identified scalability as one of the main challenges in weak isolation testing. Similar observations hold with other databases and input parameters. Given this clear difference, we only compare AWDIT and Plume on large-scale experiments.

## 5.3 Large-Scale Experiments

We now focus on the scalability of AWDIT and Plume in more detail, by performing large-scale experiments across various parameters.

**Setup.** We gather histories by running all three benchmarks on all three databases. Each history consists of either 50 or 100 sessions, while we scale the number of transactions in the range  $[2^{10}, 2^{20}]$ . This results in 198 histories in total. We set a time out of two hours for processing each history

**Results.** Fig. 8 shows the aggregate performance of AWDIT and Plume across all histories gathered in the above setup. We see that AWDIT has a clear performance advantage, with average speedups on big histories of 245×, 193×, and 62× for RC, RA, and CC, respectively, exceeding 1000× in the most extreme cases. These averages are calculated by taking the geometric mean of the ~20% largest histories (by transaction count). The average speedups across all histories are 80×, 70×, and 36× for RC, RA, and CC, respectively. Plume starts with a construction phase that builds a certain dependency graph for a given history, which dominates its running time for non-demanding inputs

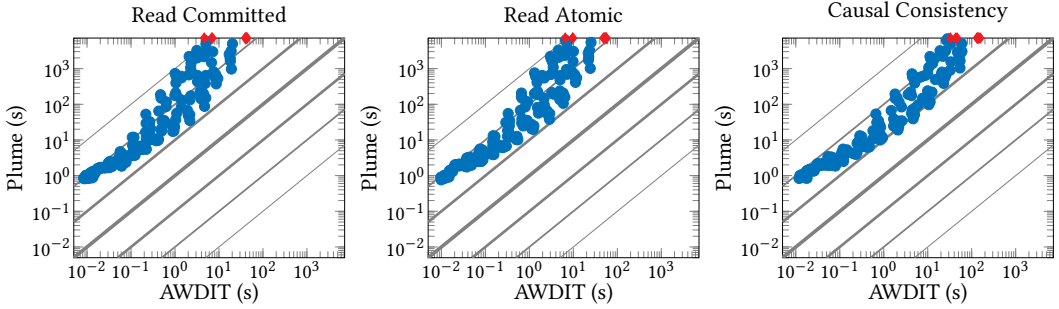


Fig. 8. An aggregate performance comparison of Plume vs AWDIT across all histories, for each weak isolation level. A red diamond indicates a timeout (2h) for Plume. Gray lines mark regions of the plot indicating speedup/slowdown of  $10^i \times$ , for  $i \in \{0, 1, 2, 3\}$ .

(i.e., towards the left end of the plot in each figure). This is consistent with prior measurements [Liu et al. 2024], which showed that Plume’s running time is often determined by the construction phase, and explains why the speedup appears to decrease initially in each figure. However, as we move towards more demanding histories (towards the right end of the plot in each figure), Plume’s solving time becomes dominant, and the speedup of AWDIT increases. Finally, Plume times out after 2 hours while analyzing a few histories, while the maximum time of AWDIT is in the order of a few minutes (in particular,  $\leq 2$  minutes for RC and RA, and  $\leq 6$  minutes for CC).

#### 5.4 Scalability Experiments

To understand the parameters that affect the running time of AWDIT, we turn our attention to scalability experiments.

**Setup.** We consider the following three scaling settings.

- Increasing the number of transactions, while keeping the number of sessions fixed at 100 and the size of each transaction bounded. This also increases the size of each history (i.e.,  $n$ ).
- Increasing the number of sessions (i.e.,  $k$ ), while keeping the number of transactions fixed at  $10^5$  and the size of each transaction bounded. This keeps the size of each history constant.
- Increasing the size of each transaction, while keeping the size of each history and the number of sessions fixed at  $10^6$  and 100, respectively.

We use CockroachDB in all three settings, running the C-Twitter benchmark for the first two, which average 7.6 operations per transaction. The last setting is not possible within the C-Twitter benchmark (it does not allow scaling the size of transactions). Instead, we rely on a custom benchmark from the Cobra framework [Tan et al. 2020].

**Results.** Fig. 9 shows the scalability of AWDIT. We see a linear effect of the number of transactions on running time (left), for each isolation level. This aligns with our theoretical analysis, which shows a linear dependency on  $n$  when the size of each transaction is bounded (for RC and RA), and when the number of sessions is fixed (for CC). The slope of each curve (for each isolation level) depends on the number of sessions and the size of each transaction. Normally, we expect the latter to be smaller, which is also the case in our experiments, resulting in a smaller slope for RA and RC.

Next, we turn our attention to scaling the number of sessions  $k$  (middle). We observe an increase in the running time of AWDIT for CC, again in alignment with our theoretical analysis, which predicts a cost of  $k$  on average for each operation in the history. On the other hand, increasing the



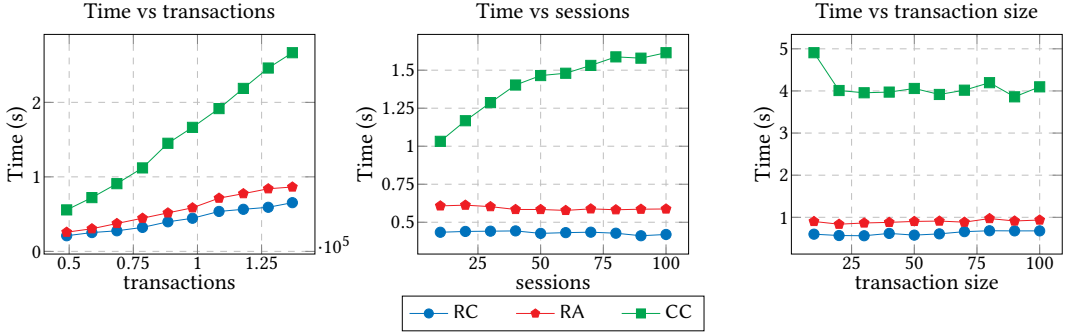


Fig. 9. Scalability experiments on AWDIT as a function of the number of transactions (left), number of sessions (middle), and number of operations per transaction (right), for each isolation level.

Table 1. Isolation anomalies reported by AWDIT and Plume.

History	Parameters				Violation(s)	Reported?	
	Size	Sessions	Database	Benchmark		AWDIT	Plume
$H_1$	32768	100	CockroachDB	TPC-C	Future Read	✓	✓
$H_2$	50000	30	CockroachDB	TPC-C	Future Read Causality Cycle	✓	✓ / ✗ (only in RC)
$H_3$	2048	50	PostgreSQL	TPC-C	Future Read	✓	✓
$H_4$	16384	50	PostgreSQL	TPC-C	Future Read Causality Cycle	✓	✓ / ✗ (only in RC)
$H_5$	32768	100	PostgreSQL	TPC-C	Future Read	✓	✓
$H_6$	50000	30	PostgreSQL	TPC-C	Future Read	✓	✓
$H_7$	50000	40	PostgreSQL	TPC-C	Future Read	✓	✓
$H_8$	1048576	100	PostgreSQL	TPC-C	Causality Cycle	✓	✗

number of sessions has no effect on the running time of AWDIT for RC and RA, which are only affected by  $n$  and the size of each transaction (both of which are bounded in this case).

Finally, we turn our attention to scaling the size of each transaction (right). Here, we observe no discernible scaling for any of the isolation levels. This is as predicted for CC, whereas our worst-case analysis predicts scaling for RC and RA, as transaction size approaches  $\sqrt{n}$ . This indicates that the RC and RA algorithms exhibit near-linear scaling on a variety of inputs.

## 5.5 Isolation Anomalies Detected

We have verified that AWDIT and Plume agree on their reports of inconsistent histories. Naturally, this only holds for histories that Plume does not time out. In total, AWDIT finds isolation anomalies on 8 histories across all our experiments, summarized in Table 1. Plume misses the anomaly in  $H_8$  due to a timeout (after two hours) and also misses the anomalies in  $H_2$  and  $H_4$  when run on the RA and CC isolation levels, due to a timeout (after 10 minutes) and a crash, respectively.

## 6 Related Work

The formalization of database isolation has been a subject of continuous work following various approaches, such as axiomatically via conflict graphs and variants thereof [Adya et al. 2000; Berenson

et al. 1995; Terry et al. 1994] and operational semantics [Crooks et al. 2017]. AWDIT follows an axiomatic style using a visibility relation, initially developed in [Burckhardt et al. 2014; Cerone et al. 2015], and used by many current weak-isolation testers [Biswas and Enea 2019; Liu et al. 2024].

The polynomial complexity of weak isolation levels admits a unifying view, as shown in [Biswas and Enea 2019]. Intuitively, this stems from the fact that `co` appears only in one of the edges for each isolation level in Fig. 3. This can serve as a first criterion for estimating whether a new isolation level admits polynomial-time testing. Plume [Liu et al. 2024] splits the problem of checking consistency into showing the absence of a number of Transactional Anomalous Patterns (TAPs), each catching a certain kind of a consistency violation that (typically) involves 3 transactions and relations between them. The fine-grained complexity of each weak isolation level is subject to further insights specific to that level. AWDIT achieves a significant improvement in theoretical complexity and practical performance by avoiding an exhaustive search over all TAPs.

Black-box testing techniques have also been developed for strong isolation levels, most notably for Serializability [Geng et al. 2024; Tan et al. 2020] and Snapshot Isolation [Huang et al. 2023; Zhang et al. 2023]. Since testing for strong isolation is NP-complete [Biswas and Enea 2019; Papadimitriou 1979], these testers mostly rely on SAT/SMT solving, though more efficient algorithms exist when parameterized by the number of sessions or the communication topology [Biswas and Enea 2019].

Analogous consistency testing problems arise frequently in the context of shared-memory concurrent programs, where isolation levels give their place for memory models [Furbach et al. 2015]. The landmark work of [Gibbons and Korach 1997] shows that the problem is NP-complete for Sequential Consistency, via a reduction from the Serializability isolation level [Papadimitriou 1979]. Similar results are known for weaker memory models, such as x86-TSO, which are still relatively strong [Furbach et al. 2015]. Nevertheless, parameterization by the number of threads and the communication topology is also known to yield polynomial-time algorithms [Abdulla et al. 2019; Bui et al. 2021; Chalupa et al. 2018; Gibbons and Korach 1994; Mathur et al. 2020].

Causally-consistent memory models have also been manifested in shared memory, perhaps most prominently in the C/C++ memory model [Batty et al. 2011]. Their weak semantics were shown to allow for efficient, polynomial time consistency checks [Lahav and Vafeiadis 2015], though the problem is known to become NP-complete [Bouajjani et al. 2017], and even notoriously difficult to parameterize [Chakraborty et al. 2024], when store operations do not have unique values. On the technical level, our upper bound for CC extends a recent result for efficient consistency checks for the Strong Release-Acquire (SRA) memory model [Tunç et al. 2023] to the transactional setting.

## 7 Conclusion

We have presented AWDIT, a highly efficient database tester for weak isolation levels. AWDIT is supported by strong theory, guaranteeing a running time of  $O(n^{3/2})$ ,  $O(n^{3/2})$ , and  $O(n \cdot k)$  when testing transaction histories of size  $n$  and  $k$  sessions, against the isolation levels Read Committed, Read Atomic, and Causal Consistency, respectively. Moreover, we have proven that, under standard complexity-theoretic hypotheses, all weak isolation levels between Read Committed and Causal Consistency basically require at least  $n^{3/2}$  time, implying that AWDIT is essentially optimal. Interesting future directions include tackling other isolation levels, possibly using saturation techniques from shared-memory concurrency [Pavlogiannis 2020; Tunç et al. 2024], as well as incorporating weak-isolation testing in a predictive analysis scheme, e.g., in the spirit of [Geng et al. 2024].

## Data-Availability Statement

Data for our experiments and the experimental setup can be found in the accompanying artifact [Møldrup and Pavlogiannis 2025a]. The newest version of AWDIT is available at <https://github.com/lassemoldrup/AWDIT>. All proofs and an additional algorithms can be found in the technical report [Møldrup and Pavlogiannis 2025b].

## Acknowledgments

This work was partially supported by a research grant (VIL42117) from VILLUM FONDEN, and by a research grant from STIBOFONDEN.

## References

2011. Big Data in Real Time at Twitter. <https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter/>.
2024. Causal Consistency and Read and Write Concerns. <https://www.mongodb.com/docs/manual/core/causal-consistency-read-write-concerns/>.
2024. CockroachDB. <https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer>.
2024. Consistency levels in Azure Cosmos DB. <https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels>.
2024. Jepsen: Distributed Systems Safety Research. <https://jepsen.io/analyses>.
2024. Neo4j. <https://neo4j.com/docs/operations-manual/current/clustering/introduction/>.
2024. PostgreSQL. <https://www.postgresql.org/docs/current/transaction-iso.html>.
2024. RocksDB. <https://github.com/facebook/rocksdb/wiki/Transactions>.
2024. TPC-C: An On-Line Transaction Processing Benchmark. <https://www.tpc.org/tpcc/default5.asp>.
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 150:1–150:29. <https://doi.org/10.1145/3360576>
- A. Adya, B. Liskov, and P. O’Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*. IEEE Comput. Soc, San Diego, CA, USA, 67–78. <https://doi.org/10.1109/ICDE.2000.839388>
- Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguica, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Nara, Japan, 405–414. <https://doi.org/10.1109/ICDCS.2016.98>
- Christiana Amza, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Wily Zwaenepoel, Emmanuel Cecchet, and Julie Marguerite. 2002. Specification and implementation of dynamic Web site benchmarks. In *2002 IEEE International Workshop on Workload Characterization*. 3–13. <https://doi.org/10.1109/WWC.2002.1226489>
- Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (July 2016), 15:1–15:45. <https://doi.org/10.1145/2909870>
- Peter Bailis, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. HAT, Not CAP: Towards Highly Available Transactions. In *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*, Petros Maniatis (Ed.). USENIX Association. <https://www.usenix.org/conference/hotos13/session/bailis>
- Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Austin, Texas, USA) (POPL ’11)*. Association for Computing Machinery, New York, NY, USA, 55–66. <https://doi.org/10.1145/1926385.1926394>
- Sam Bayless, Noah Bayless, Holger Hoos, and Alan Hu. 2015. SAT Modulo Monotonic Theories. *Proceedings of the AAAI Conference on Artificial Intelligence* 29, 1 (March 2015). <https://doi.org/10.1609/aaai.v29i1.9755>
- 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>
- Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 165:1–165:28. <https://doi.org/10.1145/3360591>
- Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On Verifying Causal Consistency. *SIGPLAN Not.* 52, 1 (Jan. 2017), 626–638. <https://doi.org/10.1145/3093333.3009888>
- Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. 2021. The Reads-from Equivalence for the TSO and PSO Memory Models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 1–30. <https://doi.org/10.1145/3485541>
- Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

- Languages (San Diego, California, USA) (POPL '14). Association for Computing Machinery, New York, NY, USA, 271–284. <https://doi.org/10.1145/2535838.2535848>
- Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.CONCUR.2015.58*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. <https://doi.org/10.4230/LIPIcs.CONCUR.2015.58>
- Soham Chakraborty, Shankara Narayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2024. How Hard Is Weak-Memory Testing? *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 66:1978–66:2009. <https://doi.org/10.1145/3632908>
- Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2018. Data-Centric Dynamic Partial Order Reduction. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–30. <https://doi.org/10.1145/3158119>
- Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: layering atomic transactions on Facebook’s online TAO data store. *Proc. VLDB Endow.* 14, 12 (July 2021), 3014–3027. <https://doi.org/10.14778/3476311.3476379>
- Don Coppersmith and Shmuel Winograd. 1982. On the Asymptotic Complexity of Matrix Multiplication. *SIAM J. Comput.* 11, 3 (1982), 472–492. <https://doi.org/10.1137/0211038>
- Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing Is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. Association for Computing Machinery, New York, NY, USA, 73–82. <https://doi.org/10.1145/3087801.3087802>
- Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal consistency and latency optimality: friend or foe? *Proc. VLDB Endow.* 11, 11 (July 2018), 1618–1632. <https://doi.org/10.14778/3236187.3236210>
- Mattern Friedemann. 1989. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel & Distributed Algorithms*. Elsevier Science Publishers B. V., 215–226.
- Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. 2015. Memory-Model-Aware Testing: A Unified Complexity Analysis. *ACM Trans. Embed. Comput. Syst.* 14, 4, Article 63 (Sept. 2015), 25 pages. <https://doi.org/10.1145/2753761>
- Chujun Geng, Spyros Blanas, Michael D. Bond, and Yang Wang. 2024. IsoPredict: Dynamic Predictive Analysis for Detecting Unserializable Behaviors in Weakly Isolated Data Store Applications. *Reproduction Package for 'IsoPredict: Dynamic Predictive Analysis for Detecting Unserializable Behaviors in Weakly Isolated Data Store Applications'* 8, PLDI (June 2024), 161:343–161:367. <https://doi.org/10.1145/3656391>
- Phillip B Gibbons and Ephraim Korach. 1994. On testing cache-coherent shared memories. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. 177–188.
- Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (Aug. 1997), 1208–1244. <https://doi.org/10.1137/S0097539794279614>
- Kaile Huang, Si Liu, Zheng 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>
- Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proceedings of the VLDB Endowment* 14, 3 (Nov. 2020), 268–280. <https://doi.org/10.14778/3430915.3430918>
- Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Vol. 9135. Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323. [https://doi.org/10.1007/978-3-662-47666-6\\_25](https://doi.org/10.1007/978-3-662-47666-6_25)
- Si Liu, Long Gu, Hengfeng Wei, and David A. Basin. 2024. Plume: Efficient and Complete Black-Box Checking of Weak Isolation Levels. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 876–904. <https://doi.org/10.1145/3689742>
- Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 710–725. <https://doi.org/10.1145/3503222.3507734>
- Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Saarbrücken Germany, 713–727. <https://doi.org/10.1145/3373718.3394783>
- Syed Akbar Mehdi, Cody Little, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I can’t believe it’s not causal! scalable causal consistency with no slowdown cascades. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) (NSDI’17). USENIX Association, USA, 453–468.
- Lasse Møldrup and Andreas Pavlogiannis. 2025a. Artifact for "AWDIT: An Optimal Weak Database Isolation Tester". <https://doi.org/10.5281/zenodo.15170736>

- Lasse Møldrup and Andreas Pavlogiannis. 2025b. AWDIT: An Optimal Weak Database Isolation Tester. arXiv:2504.06975 [cs.PL] <https://arxiv.org/abs/2504.06975>
- Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. <https://doi.org/10.1145/322154.322158>
- Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 3. <https://doi.org/10.1145/3035918.3056096>
- Andreas Pavlogiannis. 2020. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–29. <https://doi.org/10.1145/3371085>
- Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4–6, 2020*. USENIX Association, 63–80. <https://www.usenix.org/conference/osdi20/presentation/tan>
- D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. 140–149. <https://doi.org/10.1109/PDIS.1994.331722>
- Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2023. Optimal Reads-From Consistency Checking for C11-Style Memory Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 137:761–137:785. <https://doi.org/10.1145/3591251>
- Hünkar Can Tunç, Ameya Prashant Deshmukh, Berk Cirisci, Constantin Enea, and Andreas Pavlogiannis. 2024. CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis (ASPLOS '24, Vol. 3). Association for Computing Machinery, New York, NY, USA, 223–238. <https://doi.org/10.1145/3620666.3651358>
- Virginia Vassilevska Williams. 2019. *On Some Fine-Grained Questions in Algorithms and Complexity*. 3447–3487. [https://doi.org/10.1142/9789813272880\\_0188](https://doi.org/10.1142/9789813272880_0188)
- Virginia Vassilevska Williams and R. Ryan Williams. 2018. Subcubic Equivalences Between Path, Matrix, and Triangle Problems. *J. ACM* 65, 5 (Aug. 2018), 27:1–27:38. <https://doi.org/10.1145/3186893>
- Pierre Wolper. 1986. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (St. Petersburg Beach, Florida) (POPL '86)*. Association for Computing Machinery, New York, NY, USA, 184–193. <https://doi.org/10.1145/512644.512661>
- Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2022. Checking Causal Consistency of Distributed Databases. *Computing* 104, 10 (Oct. 2022), 2181–2201. <https://doi.org/10.1007/s00607-021-00911-3>
- Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 654–671. <https://doi.org/10.1145/3552326.3567492>

Received 2024-11-15; accepted 2025-03-06