Vbox: Efficient Black-Box Serializability Verification

ANONYMOUS AUTHOR(S)

Verifying the serializability of transaction histories is essential for assessing whether a database management system (DBMS) correctly enforces the claimed serializable isolation level. Black-box serializability verification provides a practical means for such validation without relying on internal system details. Existing approaches often suffer from limitations including incomplete anomaly detection, high verification overhead, excessive memory consumption, or dependence on specific concurrency control protocols. This paper presents Vbox, a black-box serializability verification method that incorporates support for predicate database operations, systematic use of transaction timing information, and a satisfiability (SAT)-based formulation with an efficient solver. Both theoretical analysis and experimental evaluation show that Vbox is correct and efficient, detects a broader range of data anomalies, and does not rely on any particular concurrency control protocol.

Additional Key Words and Phrases: Serializability verification, Transaction histories, Database systems, Concurrency control

ACM Reference Format:
Anonymous Author(s). 2018. Vbox: Efficient Black-Box Serializability Verification. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Transaction processing is fundamental to database management systems (DBMS), which ensures data consistency and persistency [17]. A key aspect of transaction processing is isolation [9, 10] which prevents concurrent transactions from interfering with each other. The highest transaction isolation level serializable [6, 24] guarantees that the concurrent execution of a set of transactions is equivalent to a sequential execution of these transactions. Although many DBMSs claim to implement the serializable isolation level, users occasionally encounter data anomalies [2, 11, 12], which indicates that there are some potential bugs in the implementation of these DBMSs. Thus, it is necessary for users to verify that their transaction histories meet the requirements of the serializable isolation level. Since users are typically restricted to access the internal state of the DBMS, black-box verification methods become practically essential.

However, verifying the serializability of a transaction execution history in a black-box manner is an NP-complete problem [5]. One class of methods constructs a serialization graph (SG) [1] based on the information collected from the clients and checks for cycles in the SG. If the SG is acyclic, the transaction history is serializable. Among these methods, Cobra [28] is notable. It constructs an incomplete SG using the "read-from" information extracted from the clients and leverages an SMT solver MonoSAT [3] to complete the SG.

An alternative approach involves checking if the transaction history conforms to specific concurrency control protocols. A typical method in this category is Leopard [22]. It records the start and the end timestamps of each database operation from the clients and uses these information to

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

infer the execution order of the operations. If the operation order violates the concurrency control protocol (e.g., serializable snapshot isolation (SSI) [14, 26] does not allow concurrent updates to the same data object), Leopard regards the transaction history non-serializable.

The third approach to serializability verification focuses on constructing a valid commit order for the transactions. If such order exists, it ensures serializability. The algorithm proposed by Biswas and Enea [8] is a prominent example. It assumes that a transaction history is organized into sessions, and the transactions in each session follow the session order. The BE algorithm selects transactions from the head of each session and adds them to the commit order, ensuring that each selected transaction does not depend on the transactions already in the commit order.

Although existing methods have shown promising results for black-box serializability verification, they suffer from inherent limitations in both expressiveness and scalability. First, neither Cobra nor BE supports predicate operations, which significantly restricts their applicability. Predicate reads and writes are common in real-world workloads and can introduce subtle anomalies that these methods fail to capture. Second, both approaches exhibit limited scalability. The BE algorithm has time complexity $O(n^{s+3})$, where n is the number of transactions and s is the number of sessions, making it impractical even for moderate workloads; in our experiments, BE exceeded 10 minutes when verifying histories with only 10K transactions. For Cobra, the space complexity is $O(n^2)$, and the time complexity is $O(n^3 + 2^c)$, where c is the number of constraints generated during SG completion. As a result, both verification time and memory usage grow rapidly with the history size. In our experiments, Cobra exceeded 10 minutes and consumed over 6 GB of memory when processing 40K transactions.

Leopard's verification is neither universal nor complete. Although it performs well for certain concurrency control protocols, it cannot infer the order of database operations for protocols such as optimistic concurrency control (OCC) [20], timestamp ordering (TO) [4], and Percolator [25], finally missing some anomalies that occur under these protocols. Additionally, Leopard requires custom configurations to adapt to different concurrency control protocols.

To address the limitations of the existing methods, we propose a novel black-box serializability verification method called Vbox based on Adya's definition of serializability [1]. Vbox constructs a SG and detects cycles in the SG. Our method contributes three main techniques.

(1) We introduce predicate constraints and truth constraints to support SGs with predicate dependency edges, enabling our method to handle transaction histories involving predicate read and write operations.
(2) We incorporate transactions' time information into the verification process. Specifically, the time information is applied to: (a) derive more dependency edges in the SG, (b) add time dependency edges to the SG as a supplement, (c) create a compact transitive closure structure and implement efficient graph algorithms to speed up SG construction, and (d) provide heuristic guidance during SG construction.
(3) We transform serializability verification into a satisfiability (SAT) problem and we design a customized solver to speed up solving the SAT problem.

Our formal study and experimental evaluation verify that Vbox is correct, efficient, and capable of detecting more data anomalies, while not relying on any specific concurrency control protocols.

(1) Vbox finds all the serializability anomalies in the real-world and the synthetic transaction execution histories containing predicate read and write operations, while the existing methods cannot find all these anomalies.
(2) Vbox has high verification efficiency and low memory usage. For an execution history of 10K transactions, Vbox is 60–100X faster than Cobra and has 20–70X lower memory consumption.

(3) Vbox exhibits nearly linear scalability. As the number of transactions in the history increases from 10K to 100K, the verification time of Vbox increases from 0.31s to 4.16s (13.4X), and the memory usage increases from 44MB to 417MB (9.5X).

2 Preliminaries

In this section we present a definition of serializability adapted from Adya's framework [1], and we formalize the black-box serializability verification problem addressed in this paper.

2.1 Database Model

Database and Versions. We model a database $\mathcal{D}$ as a finite set of objects $\mathcal{X}$. Each object $x \in \mathcal{X}$ may have multiple versions created by transactions, which are denoted by $x_1, x_2, x_3, \ldots$. For each object, we introduce two sentinel versions: the *unborn* version x_0, representing the state before x is inserted, and the *dead* version $x_\perp$, representing the state after x is deleted.

For every object x, we assume a total order $\prec_v^x$ over all committed versions of x, together with the sentinel versions x_0 and $x_\perp$. Versions created by aborted transactions are excluded from this order. The sentinel versions bound the order, i.e., $x_0 \prec_v^x x_i \prec_v^x x_\perp$ holds for every committed version x_i.

If a committed transaction t creates several successive versions of the same object, say $x_i, x_{i+1}, \ldots, x_j$, then these versions appear consecutively in $\prec_v^x$. That is, there exists no version x_k created by another transaction $t' \neq t$ such that $x_i \prec_v^x x_k \prec_v^x x_j$.

The global *version order* $\prec_v$ is defined as the union of the per-object version orders, i.e., $\prec_v = \bigcup_{x \in \mathcal{X}} \prec_v^x$.

Operations. Transactions interact with the database through three classes of operations: item operations, predicate operations, and control operations.

Item operations. The basic data actions of a transaction are expressed as *item operations*. An *item read* $r(x, x_i)$ reads a specific version x_i of object x, while an *item write* $w(x, x_j)$ creates a new version x_j of x.

Predicate operations. Predicate operations generalize item operations by guarding reads and writes with Boolean predicates over object versions. A *predicate* θ is a function $\theta : \{x_i \mid x \in \mathcal{X}\} \longrightarrow \{\text{true}, \text{false}\}$. By convention, the sentinel versions x_0 (unborn) and $x_\perp$ (dead) always evaluate to false under any predicate.

Given a predicate θ on an object x, a *predicate read* $r_\theta(x, x_i)$ reads version x_i if $\theta(x_i) = \text{true}$; otherwise, x_i is ignored. A *predicate write* $w_\theta(x, x_i, x_j)$ first evaluates $\theta(x_i)$ and conditionally performs a write. If $\theta(x_i) = \text{true}$, it reads x_i and creates a new version x_j; otherwise, it only observes x_i without producing a new version. Formally,

$$w_\theta(x, x_i, x_j) = \begin{cases} r_\theta(x, x_i);\ w(x, x_j), & \text{if } \theta(x_i) = \text{true}, \\ r_\theta(x, x_i), & \text{if } \theta(x_i) = \text{false}. \end{cases}$$

Control operations. Each transaction is delimited by control operations that mark its execution boundaries. A transaction begins with `begin` operation and terminates with either `commit` or `abort` operation.

Transactions and Histories. A transaction t is defined as a pair $t = (O_t, \prec_t)$, where O_t is a finite set of operations executed by t, and $\prec_t$ is a total order on O_t, referred to as the *event order*, which captures the order in which the operations of t are executed.

Let $\mathcal{T}$ be a finite set of transactions. A *complete history* is a pair $H = (\mathcal{T}, \prec_v)$, where $\prec_v$ is the global version order. A complete history thus captures both the executed transactions and the global ordering of committed versions.

2.2 Dependencies and Serializability of Complete Histories

Transactions in a history interact through shared object versions. Such interactions induce *dependencies* that capture how the execution of one transaction influences another.

Item Dependencies. We first formalize dependencies arising from item operations.

Definition 2.1 (Item Dependencies). Let t and t' be two distinct committed transactions with operation sets O_t and $O_{t'}$, respectively. We define:

(1) t' *item read-depends* on t if there exist operations $w(x, x_i) \in O_t$ and $r(x, x_i) \in O_{t'}$;
(2) t' *item write-depends* on t if there exist operations $w(x, x_i) \in O_t$ and $w(x, x_j) \in O_{t'}$ such that $x_i \prec_v x_j$;
(3) t' *item anti-depends* on t if there exist operations $r(x, x_i) \in O_t$ and $w(x, x_j) \in O_{t'}$ such that $x_i \prec_v x_j$.

An item read $r(x, x_i)$ may be replaced by a predicate read $r_\theta(x, x_i)$ if $\theta(x_i) = \text{true}$.

For convenience, let $\mathbb{R}_x$, $\mathbb{W}_x$, and $\mathbb{A}_x$ denote the sets of item read-, write-, and anti-dependencies, respectively, among transactions accessing object x. We further define $\mathbb{R} = \bigcup_x \mathbb{R}_x$, $\mathbb{W} = \bigcup_{x \in \mathcal{X}} \mathbb{W}_x$, and $\mathbb{A} = \bigcup_{x \in \mathcal{X}} \mathbb{A}_x$.

Predicate Dependencies. Predicate operations introduce dependencies that cannot be fully captured by item dependencies alone. We next formalize predicate dependencies.

Definition 2.2 (Predicate Dependencies). Let t and t' be two distinct committed transactions, and let θ be a predicate.

(1) t' *predicate read-depends* on t if there exist operations $w(x, x_i) \in O_t$ and $r_\theta(x, x_j) \in O_{t'}$ such that
$$x_i = \max_{\prec_v^x}\{x_m \mid x_m \preceq_v^x x_j, \theta(x_m) = \theta(x_j), \theta(x_m) \oplus \theta(x_{m-1}) = \text{true}\}.$$
(2) t' *predicate anti-depends* on t if there exist operations $r_\theta(x, x_j) \in O_t$ and $w(x, x_k) \in O_{t'}$ such that
$$x_k = \min_{\prec_v^x}\{x_m \mid x_j \prec_v^x x_m,\ \theta(x_{m-1}) = \theta(x_j), \theta(x_m) \oplus \theta(x_{m-1}) = \text{true}\}.$$

Here, $x_i \preceq_v^x x_j$ if and only if $x_i \prec_v^x x_j$ or $x_i =_v x_j$, and $\oplus$ denotes the Boolean exclusive-or (XOR) operator.

The behavior of a predicate read $r_\theta(x, x_j)$ is determined by the evaluation of θ on version x_j. In particular, when $\theta(x_j) = \text{false}$, the operation returns no concrete version. Consequently, such reads cannot be modeled using standard item read-dependencies, which are defined only for reads that return a specific object version. By contrast, a write $w(x, x_i)$ induces a predicate read-dependency with $r_\theta(x, x_j)$ if it causes θ to transition from true to false, that is, $\theta(x_{i-1}) = \text{true}$ and $\theta(x_i) = \text{false}$, where x_i is the closest such version preceding the predicate read.

Predicate anti-dependencies capture the dual situation in which the predicate value is changed by a subsequent write after a predicate read. Specifically, the first write that flips the predicate value following the read induces an anti-dependency on the earlier transaction.

Predicate write-dependencies are not defined separately, since a predicate write conditionally decomposes into a predicate read and, when the predicate evaluates to true, an item write, whose dependencies are already covered by the definitions above.

Serialization Graph. Given a complete history $H = (\mathcal{T}, \prec_v)$, the *serialization graph* (SG) is a directed graph $SG(H) = (V, E)$, where V is the set of committed transactions in $\mathcal{T}$, and $E = \{(t, t') \mid t, t' \in V$ and t' depends on $t\}$.

Aborted and Intermediate Reads. Two forms of anomalous reads violate serial semantics. An *aborted read* occurs when a committed transaction reads a version produced by an aborted transaction. An *intermediate read* occurs when a committed transaction reads a non-final version produced by another committed transaction.

Definition 2.3 (Serializable Complete History). A complete history $H = (\mathcal{T}, \prec_v)$ is *serializable* if and only if its SG is acyclic and it contains no aborted reads.

Our definition slightly differs from that of Adya's framework[1] in that we do not explicitly exclude intermediate reads. Such anomalies are already captured by cycles in the SG. Specifically, if a committed transaction t creates multiple successive versions of an object x, say $x_i, x_{i+1}, \ldots, x_j$, and another committed transaction t' reads an intermediate version x_{i+1}, then t and t' induce both an item read-dependency (t, t') and an item anti-dependency (t', t), which together form a cycle in the SG.

2.3 Serializability of Observed Histories

A *complete history* records all internal versioning details of a transaction execution, including the global version order $\prec_v$ and the exact versions accessed or created by each operation. Such information, however, is not observable to external clients.

An *observed history* $H' = (\mathcal{T}', \_)$ captures the externally visible view of execution. For a predicate read $r_\theta(x, x_i)$, if $\theta(x_i) =$ false, the evaluated version x_i is not visible to the observer; the operation is therefore recorded as an *invisible predicate read*, denoted $r_\theta(x, \_)$. Similarly, for a predicate write $w_\theta(x, x_i, x_j)$, the evaluated version x_i is never visible, regardless of the truth value of $\theta(x_i)$; such an operation is recorded as an *invisible predicate write*, denoted $w_\theta(x, \_, x_j)$.

All other operations, including item reads and writes, as well as predicate reads that evaluate to true, are recorded unchanged. As a result, an observed history preserves the structure of operations while abstracting away both the global version order and the evaluated versions of invisible predicate operations.

Because observed histories omit internal versioning details, multiple complete histories may correspond to the same observed history. To formalize this relationship, we introduce the notion of *compatibility*, which characterizes when a complete history could have produced a given observed history.

Definition 2.4 (Compatibility). Let $t = (O_t, \prec_t)$ denote a transaction in a complete history $H = (\mathcal{T}, \prec_v)$, and let $t' = (O_{t'}, \prec_{t'})$ denote a transaction in an observed history $H' = (\mathcal{T}', \_)$. Transaction t is *compatible* with t' if there exists a bijection $f : O_t \rightarrow O_{t'}$ such that, for each operation $o \in O_t$, one of the following holds:

(1) $f(o) = o$;
(2) $o = r_\theta(x, x_i)$ and $f(o) = r_\theta(x, \_)$;
(3) $o = w_\theta(x, x_i, x_j)$ and $f(o) = w_\theta(x, \_, x_j)$.

A complete history H is *compatible* with an observed history H' if there exists a bijection $g : \mathcal{T} \rightarrow \mathcal{T}'$ such that every transaction $t \in \mathcal{T}$ is compatible with $g(t)$.

Definition 2.5 (Serializable Observed History). An observed history $H' = (\mathcal{T}', \_)$ is *serializable* if and only if there exists a complete history $H = (\mathcal{T}, \prec_v)$ such that H is compatible with H' and H is serializable.

Transaction	SQL Statement	Operation	Result
t_1	`INSERT INTO t(k,v) VALUES('x',1);`	$w(x, 1)$	—
	`INSERT INTO t(k,v) VALUES('y',1);`	$w(y, 1)$	
t_2	`UPDATE t SET v=2 WHERE k='x';`	$w(x, 2)$	—
	`UPDATE t SET v=2 WHERE k='y';`	$w(y, 2)$	
t_3	`UPDATE t SET v=3 WHERE v=2;`	$w_{v=2}(x, \_, 3),\ w_{v=2}(y, \_, 3)$	—
t_4	`SELECT k,v FROM t WHERE v=2;`	$r_{v=2}(x, 2),\ r_{v=2}(y, \_)$	$\{(x, 2)\}$

Fig. 1. Observed history of four transactions over relation `t(k,v)`. Transaction t_1 inserts initial versions, t_2 updates them to value 2, t_3 performs a predicate update, and t_4 performs a predicate read observing only $\{(x, 2)\}$. `begin` and `commit` statements are omitted.

Intuitively, serializability of an observed history ensures that, although version orders and evaluated versions of invisible operations are not observable, there exists at least one compatible complete history whose execution is serializable.

2.4 Black-box Serializability Verification

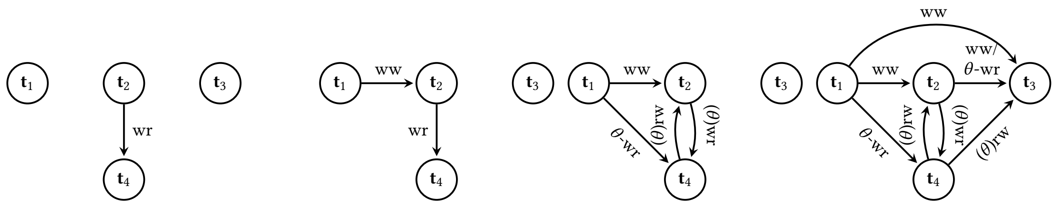
The black-box serializability verification problem asks whether a given observed history is serializable in the sense of Theorem 2.5. We illustrate this problem using the observed history shown in Figure 1.

The history consists of four transactions executed by four clients on a relational database. All statements operate on a single relation $t(k, v)$, where k is the primary key. We use k to denote the object and v to denote its version. Transaction t_1 inserts two tuples, producing item writes $w(x, 1)$ and $w(y, 1)$. Transaction t_2 updates both tuples, producing item writes $w(x, 2)$ and $w(y, 2)$. Transaction t_3 performs a predicate update on $v = 2$. The predicate is evaluated on both x and y, yielding two invisible predicate writes $w_{v=2}(x, \_, 3)$ and $w_{v=2}(y, \_, 3)$. Transaction t_4 performs a predicate read on $v = 2$. It returns x, inducing $r_{v=2}(x, 2)$, while y does not satisfy the predicate and thus induces an invisible predicate read $r_{v=2}(y, \_)$.

Compatible Completions. According to Theorem 2.5, determining whether the observed history in Figure 1 is serializable requires checking whether there exists a *serializable complete history* compatible with it. Constructing such a compatible complete history involves two steps. First, all invisible predicate reads and writes are *completed* by selecting concrete versions for predicate evaluation. Second, a version order $\prec_v$ is selected for each object.

In this example, three invisible operations must be completed: $w_{v=2}(x, \_, 3)$, $w_{v=2}(y, \_, 3)$, and $r_{v=2}(y, \_)$. To complete a predicate write, if the selected version makes the predicate evaluate to true, a new version is created; otherwise, no new version is produced. Completing the predicate read $r_{v=2}(y, \_)$ requires selecting a version on which the predicate evaluates to false, since the read returns no version. The candidate versions are 1 and 3 (if version 3 is created), but not 2, because the predicate must evaluate to false. After completing these operations, a version order is chosen among the materialized versions 1, 2, and 3 (if version 3 is created). Each combination of these choices yields a candidate complete history that is compatible with the observed history.

Checking Serializability. For each candidate complete history $H = (\mathcal{T}, \prec_v)$, we construct its SG, verify that the graph is acyclic, and check that the history contains no aborted reads.



(a) Known graph. (b) SG for item reads and item writes. (c) SG with predicate reads. (d) SG with predicate writes.

Fig. 2. Black-box serializability verification by completing observed histories. Edges labeled `wr`, `ww`, and `rw` denote item read, write, and anti-dependencies; θ-`wr` and θ-`rw` denote predicate dependencies; (θ)`wr` indicates both item and predicate dependencies.

As an example, consider completing $w_{v=2}(x, \_, 3)$ as $w_{v=2}(x, 2, 3)$, $w_{v=2}(y, \_, 3)$ as $w_{v=2}(y, 2, 3)$, and $r_{v=2}(y, \_)$ as $r_{v=2}(y, 1)$, while choosing the version order $1 \prec_v 2 \prec_v 3$ for both x and y. The resulting SG is shown in Figure 2d.

In this graph, transaction t_4 reads version x_2 written by t_2, inducing an item read-dependency from t_2 to t_4. Under the version order $1 \prec_v 2 \prec_v 3$, transaction t_2 item write-depends on t_1, and transaction t_3 item write-depends on both t_2 and t_1. Moreover, t_3 performs a predicate write evaluated using version 2 created by t_2, inducing a predicate read-dependency from t_2 to t_3. Transaction t_4 performs predicate reads on both x and y. For object x, the read $r_{v=2}(x, 2)$ induces a predicate read-dependency from t_2 to t_4 and a predicate anti-dependency from t_4 to t_3. For object y, the read $r_{v=2}(y, 1)$ induces a predicate read-dependency from t_1 to t_4 and a predicate anti-dependency from t_4 to t_2. The resulting SG contains a cycle, and thus the corresponding complete history is not serializable.

Exhaustively examining all candidate complete histories shows that none yields an acyclic SG. Therefore, the observed history in Figure 1 is not serializable.

Problem Statement. The *black-box serializability verification problem* is defined as follows. Given an observed history $H' = (\mathcal{T}, \_)$, determine whether there exists a complete history $H = (\mathcal{T}, \prec_v)$ compatible with H' such that:

(1) H contains no aborted reads;
(2) The SG of H is acyclic.

Detecting aborted reads is straightforward. Accordingly, the remainder of this paper focuses on efficiently identifying compatible histories whose SG are acyclic.

3 Our Method Vbox

Having defined the problem, we now present our verification framework, Vbox. At a high level, Vbox formulates black-box serializability verification as a *constraint assignment problem* (§3.1). To mitigate the large number of constraints induced by version order choices, we leverage client-side transaction timestamps to construct a more informative *known graph* (§3.2), which enables effective constraint filtering and pruning (§3.3).

To further improve scalability, we introduce a compact transitive closure representation and develop optimized graph algorithms (§3.4) that support efficient edge insertion. These techniques substantially reduce both time and space overhead when verifying histories with a large number of transactions. Finally, we encode the verification problem as a satisfiability (SAT) instance (§3.5) and design a customized SAT solver (§3.6) that reuses data structures built during constraint pruning and exploits transaction timestamps to guide the search.

3.1 Constraint Formulation

We formulate black-box serializability verification as a *constraint assignment problem*. Given an observed history H', verification amounts to deciding whether there exists a completion into a complete history whose serialization graph is acyclic. To characterize all admissible completions, we introduce three classes of constraints: *item constraints*, *predicate constraints*, and *truth constraints*.

3.1.1 Item Constraints. We begin with the restricted setting in which the observed history $H' = (\mathcal{T}, \_)$ contains only item operations. In this case, serializability verification reduces to determining whether there exists a version order $\prec_v$ such that the SG of some complete history $H = (\mathcal{T}, \prec_v)$ compatible with H' is acyclic.

We start from a *known graph* $G = (V, E)$, where V is the set of committed transactions and E consists of all item read-dependencies directly observable from H'. Selecting a version order $\prec_v$ induces additional dependency edges according to Theorem 2.1. Extending G with these induced edges yields the SG of a complete history compatible with H'. We refer to any such extension as a *compatible graph*.

Definition 3.1 (Item Constraints C_I). Consider two distinct committed transactions t_i, t_j that create versions x_m and x_n of the same object x. Any complete history must satisfy exactly one of $x_m \prec_v x_n$ or $x_n \prec_v x_m$.

If $x_m \prec_v x_n$, the induced dependency edge set is

$$E_{ij} = \{(t_i, t_j)\} \ \cup \ \{(t_r, t_j) \mid (t_i, t_r) \in \mathbb{R}_x\}, \tag{1}$$

representing the item write-dependency and the corresponding anti-dependencies. Symmetrically, if $x_n \prec_v x_m$, the induced edge set is

$$E_{ji} = \{(t_j, t_i)\} \ \cup \ \{(t_r, t_i) \mid (t_j, t_r) \in \mathbb{R}_x\}. \tag{2}$$

The mutually exclusive pair (E_{ij}, E_{ji}) constitutes an *item constraint*. Let C_I denote the set of all such constraints induced by writers of the same objects in H'.

Given the known graph G and the item constraint set C_I, a *compatible graph* $G' = (V, E')$ corresponds to selecting exactly one branch from each item constraint, such that for every $(E_{ij}, E_{ji}) \in C_I$,

$$\big(E_{ij} \subseteq E' \wedge E_{ji} \cap E' = \emptyset\big) \ \vee \ \big(E_{ji} \subseteq E' \wedge E_{ij} \cap E' = \emptyset\big). \tag{3}$$

A selection of branches for all constraints in C_I is called an *assignment*. If the resulting compatible graph G' is acyclic, then the corresponding observed history is serializable.

Example. Consider the observed history in Figure 1. Its known graph is shown in Figure 2a, and the induced item constraint set is $\{(E_{12}, E_{21})\}$, where $E_{12} = \{(t_1, t_2)\}$ and $E_{21} = \{(t_2, t_1), (t_4, t_1)\}$. Selecting E_{12} yields the compatible graph in Figure 2b, which is acyclic. Thus, when only item operations are considered, the observed history is serializable. However, this conclusion contradicts §2.4, as item constraints alone do not capture the effects of predicate operations. Invisible predicate reads may introduce additional dependencies that render all compatible graphs cyclic.

3.1.2 Predicate Constraints. We now extend the formulation to histories containing predicate reads. Verification must complete each invisible predicate read by assigning a concrete object version such that the resulting compatible graph remains acyclic. To this end, we introduce *predicate constraints*, which capture the dependencies induced by invisible predicate reads.

Definition 3.2 (Predicate Constraints). Let t be a transaction that contains an invisible predicate read $r_\theta(x, \_)$. To characterize its possible completions, we first define the predicate-based version set

$$S^x_\theta(\delta) = \{x_i \mid \theta(x_i) = \delta\}, \qquad \delta \in \{\text{true}, \text{false}\}. \tag{4}$$

Each version $x_i \in S_\theta^x(\delta)$ represents a valid completion of $r_\theta(x, \_)$ with evaluation result δ. For a fixed candidate version $x_i \in S_\theta^x(\delta)$, we define the corresponding predicate-induced dependency edge set as

$$\begin{aligned} E_\theta^{x_i}(\delta) = \big\{\, (t', t) \;\big|\; w(x, x_i) \in O_{t'} \,\big\} \\ \cup \big\{\, (t, t'') \;\big|\; x_i \prec_v x_j, w(x, x_j) \in O_{t''}, \\ \theta(x_j) = \neg\delta \,\big\}. \end{aligned} \tag{5}$$

The edge (t', t) represents the predicate read-dependency induced by completing $r_\theta(x, \_)$ using version x_i. Each edge (t, t'') represents a potential predicate anti-dependency induced by a later version x_j whose predicate evaluation contradicts δ.

Since predicate anti-dependencies depend on the version order $x_i \prec_v x_j$, all edges (t, t'') satisfying $\theta(x_j) = \neg\delta$ are initially included in $E_\theta^{x_i}(\delta)$ and marked as *undetermined*. Such an edge becomes *determined* and is inserted into $E_\theta^{x_i}(\delta)$ only after the assignment for item constraint (E_{ij}, E_{ji}) establishes $x_i \prec_v x_j$. The predicate constraint induced by the invisible predicate read $r_\theta(x, \_)$ is defined as

$$C_P(r_\theta(x, \_), \delta) = \left\{ E_\theta^{x_i}(\delta) \;\middle|\; x_i \in S_\theta^x(\delta) \right\}. \tag{6}$$

Let C_P be the set of predicate constraints from invisible predicate reads in the observed history. During verification, exactly one edge set from each constraint in C_P is chosen and added to the known graph G; this choice is the constraint's *assignment*. After assigning all item constraints in C_I and predicate constraints in C_P, the resulting graph G' is a compatible graph representing the SG of a complete history compatible with the observed history. The observed history is serializable iff at least one such compatible graph G' is acyclic.

Example. In the observed history shown in Figure 1, the only item constraint is $\{(E_{12}, E_{21})\}$. The invisible read $r_{v=2}(y, \_)$ induces the predicate constraint $\{E_{v=2}^{y_1}(\text{false})\}$, where $E_{v=2}^{y_1}(\text{false}) = \{(t_1, t_4), (t_4, t_2)\}$ and (t_4, t_2) is initially undetermined. Assigning the item constraint and choosing E_{12} fixes $1 \prec_v 2$, which determines (t_4, t_2). Choosing $E_{v=2}^{y_1}(\text{false})$ then yields the compatible graph in Figure 2c. This graph has a cycle, so the corresponding complete history is not serializable. Exhaustively enumerating all assignments shows that no acyclic compatible graph exists; thus the observed history is not serializable when predicate reads are considered.

3.1.3 Handling Predicate Writes. We further extend the constraint framework to incorporate *predicate writes*. In an observed history, each predicate write $w_\theta(x, \_, x_j)$ requires completion by selecting a version of x for the evaluation of θ. This choice governs the truth value of θ and determines whether the operation creates a new version of x. To explicitly model this decision, we introduce *truth constraints*, which govern the evaluation outcomes of predicate writes and conditionally activate the corresponding item and predicate constraints.

Definition 3.3 (Truth Constraints). For each predicate write $w_\theta(x, \_, x_j)$, we define a *truth constraint* as a mutually exclusive pair:

$$C_T\big(w_\theta(x, \_, x_j)\big) = \big(\tau_{\text{true}}^\theta(x_j), \tau_{\text{false}}^\theta(x_j)\big), \tag{7}$$

where $\tau_{\text{true}}^\theta(x_j)$ and $\tau_{\text{false}}^\theta(x_j)$ represent the assignment of θ to true and false, respectively. An *assignment* of C_T selects exactly one branch from each pair, thereby fixing the observable semantic effect of the predicate write. Let C_T denote the set of truth constraints induced by the observed history.

Internal Predicate Read Component. Regardless of the selected truth value, a predicate write $w_\theta(x, \_, x_j)$ inherently entails an invisible predicate read $r_\theta(x, \_)$. The evaluation result of this read is constrained by the chosen truth assignment $\tau^\theta_\delta(x_j)$, where $\delta \in \{\text{true}, \text{false}\}$. Accordingly, we construct the corresponding predicate constraint $C_P(r_\theta(x, \_), \delta)$ and add it to C_P.

Materialization of Item Writes. The selection of $\tau^\theta_{\text{true}}(x_j)$ implies that the predicate write $w_\theta(x, \_, x_j)$ materializes a concrete item write $w(x, x_j)$. This item write introduces additional dependencies categorized as follows:

- **Item dependencies:** The item write $w(x, x_j)$ induces standard item constraints between transaction t (where $w_\theta(x, \_, x_j) \in O_t$) and other transactions that write object x. These constraints are *guarded* by $C_T(w_\theta(x, \_, x_j))$: they are ignored unless $\tau^\theta_{\text{true}}(x_j)$ is selected, in which case they are activated and incorporated into the constraint assignment.
- **Predicate dependencies:** The item write $w(x, x_j)$ may affect the predicate constraints of other invisible predicate reads. For an invisible predicate read $r_\theta(x, \_)$ in transaction t' with evaluation result δ, if $\theta(x_j) = \delta$, a new edge set $E^{x_j}_\theta(\delta)$ is appended to the corresponding predicate constraint $C_P(r_\theta(x, \_), \delta)$. If $\theta(x_j) = \neg\delta$, a predicate anti-dependency edge (t', t) is added to all candidate edge sets in $C_P(r_\theta(x, \_), \delta)$. These dependencies are *guarded* by the associated truth constraint: they are ignored unless $\tau^\theta_{\text{true}}(x_j)$ is selected, in which case they are activated and incorporated into the constraint assignment.

Global Consistency Conditions. Truth constraints are subject to two fundamental consistency conditions. First, *read-from consistency*: if a transaction t_r reads a version x_j produced by a predicate write $w_\theta(x, \_, x_j)$, then $\tau^\theta_{\text{true}}(x_j)$ must be assigned. Second, *feasibility*: every predicate constraint must admit at least one valid edge set to ensure each invisible read has a consistent completion.

Example. Consider the history in Figure 1 containing an invisible predicate write $w_{v=2}(x, \_, 3)$. We construct the truth constraint $(\tau^{v=2}_{\text{true}}(x_3), \tau^{v=2}_{\text{false}}(x_3))$. Selecting $\tau^{v=2}_{\text{true}}(x_3)$ activates two item constraints, (E_{13}, E_{31}) and (E_{23}, E_{32}), and introduces the predicate constraint $\{E^{x_2}_{v=2}(\text{true})\}$. Furthermore, this write induces a predicate anti-dependency edge (t_4, t_3) corresponding to $r_{v=2}(x, 2)$. Base on the compatible graph shown in Figure 2c, if we choose E_{13} and E_{23} for item constraint and choose $\{E^{x_2}_{v=2}(\text{true})\}$ for predicate constraint, we obtain the compatible graph shown in Figure 2d.

The following theorem formalizes the reduction of black-box serializability verification to this constraint assignment problem, and we prove this theorem in A.

Theorem 3.4 (Constraint Serializability). *An observed history is serializable if and only if it contains no aborted reads and there exists a joint assignment of its item constraints C_I, predicate constraints C_P, and truth constraints C_T such that the resulting compatible graph is acyclic.*

3.2 Effective Construction of the Known Graph

Our next design goal is to construct a more complete known graph by identifying additional dependencies from client-side observations while avoiding unnecessary constraints. To this end, we record the client-side timestamps of transactions.

For each transaction t, we record its *client start timestamp* $\mathsf{s}(t)$ when the client issues the `begin` command, and its *client end timestamp* $\mathsf{e}(t)$ when the client receives the response to the `commit` or `abort` command. These timestamps are measured using the client's wall-clock time and are assumed to be synchronized across clients. Due to network latency, the client start timestamp $\mathsf{s}(t)$ precedes the actual server-side start time $\mathsf{s}^r(t)$, and the client end timestamp $\mathsf{e}(t)$ follows the actual server-side completion time $\mathsf{e}^r(t)$.

We assume a bounded clock skew between clients. Let Δ be an upper bound on the skew between any two client clocks. To ensure correctness under this assumption, we conservatively adjust the timestamps by setting $\mathrm{s}(t) \leftarrow \mathrm{s}(t) - \Delta$ and $\mathrm{e}(t) \leftarrow \mathrm{e}(t) + \Delta$.

Time Dependencies. Based on the adjusted timestamps, we define a new class of dependencies between transactions. For two committed transactions t_i and t_j, if $\mathrm{e}(t_i) \le \mathrm{s}(t_j)$, then all operations of t_i must have completed on the server before the DBMS received the `begin` of t_j. In this case, we say that t_j *time-depends* on t_i. Let $\mathbb{T}$ denote the set of all such *time dependencies* in the observed history.

Time dependencies allow us to infer additional item-level dependencies. If two committed transactions t_i and t_j write different versions of the same object and t_j time-depends on t_i, then $\mathrm{e}^r(t_i) \le \mathrm{s}^r(t_j)$ must hold. Consequently, t_j item write-depends on t_i, and we add $(t_i, t_j) \in \mathbb{W}$. Let $\mathbb{W}^t \subseteq \mathbb{W}$ denote the set of item write-dependencies inferred from time dependencies.

Similarly, if a committed transaction t_i reads an object x and another committed transaction t_j writes x, and if t_j time-depends on t_i, then t_j item anti-depends on t_i. In this case, we add $(t_i, t_j) \in \mathbb{A}$. Let $\mathbb{A}^t \subseteq \mathbb{A}$ denote the set of item anti-dependencies inferred from time dependencies.

Constructing the Known Graph. Item anti-dependencies can also be inferred by composing existing dependencies. Specifically, if a committed transaction t_i read-depends on a committed transaction t_r with respect to an object x, and another committed transaction t_j item write-depends on t_i with respect to the same object, then t_j item anti-depends on t_r with respect to x, that is, $(t_r, t_j) \in \mathbb{A}$. Let $\mathbb{A}^w \subseteq \mathbb{A}$ denote the set of item anti-dependencies inferred by composing item read-dependencies in $\mathbb{R}$ with item write-dependencies in $\mathbb{W}^t$.

After identifying the item read-dependencies in $\mathbb{R}$, the time dependencies in $\mathbb{T}$, the inferred item write-dependencies in $\mathbb{W}^t$, and the inferred item anti-dependencies in $\mathbb{A}^t \cup \mathbb{A}^w$, we construct the known graph G by adding edges corresponding to all these dependencies.

Unlike item and predicate dependencies, time dependencies are not defined directly by read or write operations.The following theorem establishes that incorporating time dependencies into the known graph is sound under most common concurrency control protocols. Its proof is deferred to B.

THEOREM 3.5. *If the DBMS employs Serializable Snapshot Isolation (SSI) [14], Two-Phase Locking (2PL) [27], Optimistic Concurrency Control (OCC) [20], Timestamp Ordering (TO) [4], or Percolator [25], then adding time-dependency edges to a SG compatible with the observed history does not change reachability between any pair of transactions.*

3.3 Constraint Reduction

The number of constraints directly determines the scale of the resulting SAT instance and, consequently, the efficiency of verification. We therefore propose a set of reduction techniques to eliminate unnecessary constraints and prune infeasible choices as early as possible. These techniques apply to item, predicate, and truth constraints, respectively.

Item Constraint Avoidance. An item constraint (E_{ij}, E_{ji}) is constructed for each pair of transactions t_i and t_j that create different versions of the same object (see Eq. (1) and Eq. (2)). However, such a constraint is unnecessary if t_i and t_j do not overlap in time, i.e., if $\mathrm{e}(t_i) \le \mathrm{s}(t_j)$ or $\mathrm{e}(t_j) \le \mathrm{s}(t_i)$. In this case, the write order between t_i and t_j is already fixed by the time dependency and has been incorporated into the known graph as an item write-dependency in $\mathbb{W}^t$. Moreover, all item anti-dependencies derived from this write-dependency have already been added to $\mathbb{A}^w$. Therefore, constructing an item constraint for such a pair is redundant and can be safely avoided.

Algorithm 1 Constraint Pruning

Input: Graph $G = (V, E)$ and constraint set C
Output: Updated graph G and pruned constraint set C

$Q \leftarrow \emptyset$
for each $(E_{ij}, E_{ji}) \in C$ **do**
$l \leftarrow (E \cup E_{ij}$ contains a cycle$)$
$r \leftarrow (E \cup E_{ji}$ contains a cycle$)$
if $l \wedge r$ **then**
return Non-serializable
else if l **then**
$Q \leftarrow Q \cup E_{ji}; C \leftarrow C \setminus \{(E_{ij}, E_{ji})\}$
else if r **then**
$Q \leftarrow Q \cup E_{ij}; C \leftarrow C \setminus \{(E_{ij}, E_{ji})\}$
end if
end for
while $Q \neq \emptyset$ **do**
Extract e from Q
$\Delta R \leftarrow \textsc{UpdateTransitiveClosure}(G, e)$
if G contains a cycle **then**
return Non-serializable
end if
for each $(t_i, t_j) \in \Delta R$ **do**
if $(t_j, t_i) \in E_{ij}$ **then**
$Q \leftarrow Q \cup E_{ji}; C \leftarrow C \setminus \{(E_{ij}, E_{ji})\}$
else if $(t_j, t_i) \in E_{ji}$ **then**
$Q \leftarrow Q \cup E_{ij}; C \leftarrow C \setminus \{(E_{ij}, E_{ji})\}$
end if
end for
end while
return (G, C)

Item Constraint Consolidation. Consider two item constraints (E_{ij}, E_{ji}) and (E_{mn}, E_{nm}). If $E_{ij} \cap E_{mn} \neq \emptyset$, these two constraints can be merged into a single constraint $(E_{ij} \cup E_{mn},\ E_{ji} \cup E_{nm})$. The correctness of this consolidation follows from the definition of compatible graphs (Eq. (3)). Since $E_{ij} \cap E_{mn} \neq \emptyset$, selecting E_{ij} necessarily implies selecting E_{mn}, and selecting E_{ji} necessarily implies selecting E_{nm}. Hence, the two constraints are logically equivalent to their union and can be replaced without affecting the solution space.

Item Constraint Pruning. Let (E_{ij}, E_{ji}) be an item constraint. If there exists an edge $(u, v) \in E_{ij}$ such that a directed path from v to u already exists in the known graph G, then adding (u, v) would create a cycle in any compatible graph. Consequently, no edge in E_{ij} can be selected, and the constraint forces the selection of E_{ji}. In this case, we remove the constraint (E_{ij}, E_{ji}) and add all edges in E_{ji} directly to G. This process is referred to as *constraint pruning*.

We implement item constraint pruning using a two-stage algorithm shown in 1.

The first stage identifies constraints that can be resolved immediately based on the current known graph. The second stage propagates newly added edges and incrementally updates reachability information to enable further pruning. Item constraint pruning terminates either when no more constraints can be resolved or when a cycle is detected, in which case the history is not serializable.

Predicate Constraint Avoidance. For each predicate read $r_\theta(x, \_)$ evaluated to δ, we construct a predicate constraint $C_P(r_\theta(x, \_), \delta) = \{ E_\theta^{x_i}(\delta) \mid x_i \in S_\theta^x(\delta) \}$. For an edge $(t_i, t_j) \in E_\theta^{x_i}(\delta)$, if $\mathrm{e}(t_i) \le \mathrm{s}(t_j)$, the edge has already been added to the known graph as a time-dependency. If $\mathrm{e}(t_j) \le \mathrm{s}(t_i)$, adding (t_i, t_j) would immediately form a cycle with the existing time-dependency edge (t_j, t_i). Therefore, any edge between non-overlapping transactions is unnecessary and can be removed from $E_\theta^{x_i}(\delta)$. After eliminating such edges, empty edge sets are discarded and duplicate edge sets are merged, yielding a reduced predicate constraint.

Predicate Constraint Pruning. Consider a predicate constraint $C_P(r_\theta(x, \_), \delta) = \{ E_\theta^{x_i}(\delta) \mid x_i \in S_\theta^x(\delta) \}$. Initially, in each edge set $E_\theta^{x_i}(\delta)$, only the predicate read-dependency edge (t', t) is determined, while the predicate anti-dependency edges remain undetermined.

For an undetermined anti-dependency edge (t, t'') derived from a potential version order $x_i \prec_v x_j$, let t_i and t_j be the transactions that write versions x_i and x_j, respectively. If there exists a directed path from t_i to t_j in the known graph G, then $x_i \prec_v x_j$ is enforced and the edge (t, t'') becomes determined. Conversely, if there exists a directed path from t_j to t_i in G, then $x_i \prec_v x_j$ can never be established, and the corresponding anti-dependency (t, t'') is removed from $E_\theta^{x_i}(\delta)$.

For a determined edge $(t_i, t_j) \in E_\theta^{x_i}(\delta)$, if there exists a directed path from t_j to t_i in G, selecting $E_\theta^{x_i}(\delta)$ would introduce a cycle. In this case, the entire edge set $E_\theta^{x_i}(\delta)$ is removed from the predicate constraint. If a predicate constraint is reduced to a single edge set, its determined edges are added to G. The constraint is removed only after all its undetermined edges have been resolved.

The two-stage item constraint pruning process shown in 1 can be naturally extended to prune predicate constraints. Item constraint pruning and predicate constraint pruning are executed alternately until the reachability relation in the known graph G stabilizes.

Truth Constraint Avoidance. For each predicate write $w_\theta(x, \_, x_j)$, we construct a truth constraint $C_T(w_\theta(x, \_, x_j)) = \{\tau_{\text{true}}^\theta(x_j), \tau_{\text{false}}^\theta(x_j)\}$. If version x_j is read by any transaction, then the predicate write must have been evaluated to true, and $\tau_{\text{true}}^\theta(x_j)$ is forced. In this case, the corresponding truth constraint need not be constructed.

3.4 Efficient Reachability Test

Both constraint pruning (§3.3) and the customized SAT solver (§3.6) require frequent reachability queries over the known graph G. To support these queries efficiently, we maintain the transitive closure of G, conceptually represented as a Boolean relation indicating whether one transaction is reachable from another.

Explicitly constructing the full transitive closure is both unnecessary and inefficient. By leveraging transaction timestamps, we introduce a *compact transitive closure* that records only reachability information not directly implied by timestamps. In the following, we describe the design, construction, and updating of this compact representation.

3.4.1 Compact Transitive Closure. For two committed transactions t_i and t_j, if $\mathrm{e}(t_i) \le \mathrm{s}(t_j)$, then (t_i, t_j) forms a time-dependency edge, and t_j is trivially reachable from t_i. More generally, for any pair of non-overlapping transactions ($\mathrm{e}(t_i) \le \mathrm{s}(t_j)$ or $\mathrm{e}(t_j) \le \mathrm{s}(t_i)$), their reachability can be determined solely from timestamps and therefore need not be stored explicitly.

The compact transitive closure records reachability information only between overlapping transactions. Transactions are first sorted by start timestamp, yielding an ordered list T^s. For each transaction t_i, let t_{l_i} and t_{r_i} denote the first and last transactions in T^s that overlap with t_i,

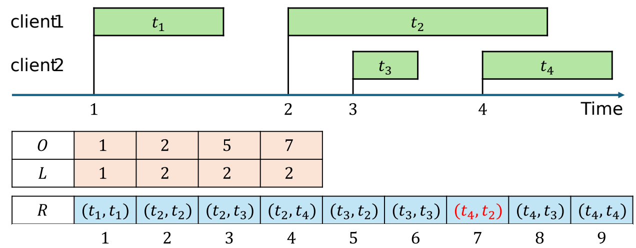


Fig. 3. Compact transitive closure.

respectively. Accordingly, the compact transitive closure maintains reachability information only for transactions t_j satisfying $l_i \leq j \leq r_i$.

To achieve cache-efficient storage, we adopt an array-based representation instead of a hash-based structure. The compact transitive closure consists of a main array R and two auxiliary arrays L and O. For each transaction t_i, $L[i] = l_i$ stores the index of its first overlapping transaction, and $O[i]$ stores the offset in R at which the reachability entries for t_i begin. For any t_j with $l_i \leq j \leq r_i$, the reachability value $r(t_i, t_j)$ is stored at position $R[O[i] + j - L[i]]$.

Figure 3 illustrates the compact transitive closure for four transactions t_1–t_4 sorted by start timestamp. Transaction t_1 does not overlap with any other transaction, hence $l_1 = r_1 = 1$, $L[1] = 1$, and $R[1]$ stores $r(t_1, t_1)$ with $O[1] = 1$. Transaction t_2 overlaps with t_3 and t_4, so $l_2 = 2$ and $r_2 = 4$; its reachability entries $r(t_2, t_2)$, $r(t_2, t_3)$, and $r(t_2, t_4)$ are stored at $R[2]$, $R[3]$, and $R[4]$, respectively, with $O[2] = 2$.

Space Complexity. Compared with a full reachability matrix, the compact transitive closure stores reachability information only for overlapping transaction pairs. Let $d = \sum_{i=1}^{|V|}(r_i - l_i)/|V|$ be the average number of transactions that overlap in time with a given transaction. The resulting space complexity is $O(d|V|)$, instead of $O(|V|^2)$ for a full matrix, where $|V|$ is the number of transactions.

3.4.2 Computing the Compact Transitive Closure. The transitive closure of a directed graph $G = (V, E)$ can be computed in $O(|V|^3)$ time using Warshall's algorithm [30]. When G is a directed acyclic graph (DAG), Purdom's algorithm [19] provides a more efficient alternative with time complexity $O(|E| + |V|^2)$. Purdom's algorithm proceeds in two phases: (1) computing a topological ordering of the vertices, and (2) for each vertex $v \in V$, constructing the descendant set $D(v)$ containing all vertices reachable from v. Initially, $D(v) = \{v\}$ for all v. Vertices are processed in reverse topological order, and for each edge $(v, u) \in E$, the descendant set is updated as $D(v) \leftarrow D(v) \cup D(u)$. After termination, $D(v)$ exactly captures the reachability from v.

In our setting, vertices correspond to transactions with client-side timestamps. By exploiting this information, we further optimize Purdom's algorithm and construct the compact transitive closure in $O(d|V| + |E \setminus E_T|)$ time, where E_T denotes the set of time-dependency edges from $\mathbb{T}$. The optimized construction procedure is summarized in 2.

The algorithm follows the two-phase structure of Purdom's method. In the first phase, temporal information is used to accelerate the topological traversal. When a transaction t_i is visited, all transactions t_j with $j > r_i$ are already visited via time-dependency edges. A global variable o_g records the earliest transaction whose descendants have not yet been fully visited. Consequently, depth-first search only traverses edges (t_i, t_j) with $r_i < j < o_g$ or $(t_i, t_j) \in E \setminus E_T$. Consequently, the time complexity of the first step of Purdom's algorithm is reduced to $O(|V| + |E \setminus E_T|)$.

In the second phase, redundant merges of descendant sets are avoided. For three transactions t_i, t_j, t_k, if $(t_i, t_j) \in E_T$ and $(t_j, t_k) \in E_T$, then $(t_i, t_k) \in E_T$. Therefore, it is unnecessary to update

Algorithm 2 Compact Transitive Closure Construction

Input: Graph $G = (V, E)$
Output: Compact transitive closure (R, L, O)

$o_g \leftarrow |V|$; $Q \leftarrow$ empty queue
for each $t_i \in V$ **do**
    if t_i is not visited **then**
        DFS(t_i)
    end if
end for
while $Q \neq \emptyset$ **do**
    $t_i \leftarrow$ Dequeue(Q)
    $S_i \leftarrow \{t_i\}$; $d_i \leftarrow r_i$
    for all $(t_i, t_j) \in E \setminus E_T$ **with** $j \leq r_{m_i}$ **do**
        $S_i \leftarrow S_i \cup S_j$
        $d_i \leftarrow \min(d_i, d_j)$
    end for
    for $j = r_i + 1$ **to** r_{m_i} **do**
        $S_i \leftarrow S_i \cup S_j$
        $d_i \leftarrow \min(d_i, d_j)$
    end for
    for each $t_j \in S_i \cup \{k \mid d_i \leq k \leq r_i\}$ **do**
        $R[O[i] + j - L[i]] \leftarrow$ **true**
    end for
end while
return (R, L, O)
function DFS(t_i)
    for each $(t_i, t_j) \in E \setminus E_T$ **or** $r_i < j < o_g$ **do**
        if t_j is not visited **then**
            DFS(t_j)
        end if
    end for
    Mark t_i as visited
    Enqueue t_i into Q
    $o_g \leftarrow \min(o_g, r_i)$
end function

the descendant set $D(t_i)$ with $D(t_k)$ because $D(t_k)$ has already been merged into $D(t_j)$ according to the reverse topological sort, and $D(t_j)$ will be subsequently merged into $D(t_i)$. We define the minimum time-successor t_{m_i} of t_i such that $(t_i, t_{m_i}) \in E_T$, and for all $(t_i, t_k) \in E_T$, it holds that $\mathrm{e}(t_{m_i}) \leq \mathrm{e}(t_k)$. When updating $D(t_i)$, edges (t_i, t_j) with $j > r_{m_i}$ can be skipped.

To further optimize, we represent $D(t_i)$ using two parts: an integer d_i such that all t_j with $j > d_i$ are descendants, and a set S_i of remaining descendants with $j \leq d_i$. Merging $D(t_j)$ into $D(t_i)$ updates d_i to $\min(d_i, d_j)$ and S_i to $S_i \cup S_j$. For each S_i, at most $r_i - l_i$ elements are added to it. The second step completes in $O(d|V|)$ time.

3.4.3 Updating Compact Transitive Closure. During constraint pruning and SAT problem solving, edges are added to the graph $G = (V, E)$, potentially changing vertex reachability. Recomputing the transitive closure from scratch is inefficient for a small number of edge additions. Since G is a DAG, Italiano's algorithm[18] can update the transitive closure in $O(|V|)$ amortized time when an edge is added. We improve Italiano's algorithm by leveraging transactions' temporal information.

Italiano's algorithm identifies vertex pairs whose reachability changes due to the added edge (t_i, t_j). If $r(t_i, t_j)$ = true, the edge has no effect. If $r(t_j, t_i)$ = true, it forms a cycle. Otherwise, the affected vertex pairs are:

$$\mathbb{I} = \{(t_u, t_v) \mid r(t_u, t_i) = \text{true}, r(t_j, t_v) = \text{true}, r(t_u, t_v) = \text{false}\}. \tag{8}$$

Italiano's algorithm visits all vertices to find $\mathbb{I}$. We improve it by constructing a candidate set of vertex pairs based on transactions' temporal information, significantly reducing the search space.

Lemma 3.6. *For all* $(t_u, t_v) \in \mathbb{I}$, $l_j \leq u \leq \min(r_i, r_j)$.

Proof. Suppose $u > r_i$. It holds that t_i can reach t_u through a time-dependency edge. In Eq. (8), we have $r(t_u, t_i)$ = true, that is, t_i is reachable from t_u. Thus, there is a cycle in G, which contradicts with the fact that G is acyclic.

Suppose $u > r_j$. It holds that t_j can reach t_u through a time-dependency edge. In Eq. (8), we have $r(t_u, t_i)$ = true, that is, t_i is reachable from t_u. Plus the edge (t_i, t_j), there is a cycle in G, which leads to a contradiction.

Suppose $u < l_j$. We have that t_u reaches t_j. Since t_v is a descendant of t_j, t_u can also reach t_j, which contradicts with the fact $r(t_u, t_v)$ = false in Eq. (8). □

Similarly, we have the following lemma.

Lemma 3.7. *For all* $(t_u, t_v) \in \mathbb{I}$, $\max(l_u, l_i, l_j) \leq v \leq \min(r_u, r_i)$.

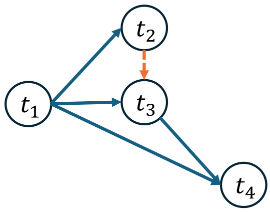
Consequently, after adding an edge (t_i, t_j) to G, we first obtain a set of candidate vertices for t_u in $\mathbb{I}$ according to Lemma 3.6. For each candidate t_u, we obtain a set of candidate vertices for t_v in $\mathbb{I}$ according to Lemma 3.7. If t_u and t_v satisfy the condition given by Eq. (8), $r(t_u, t_v)$ is updated to true.

Figure 4a shows the graph of the history in Figure 3. When we add the edge (t_2, t_3) to the graph, we start by finding the candidate set for t_u, which is $\{t_2, t_3, t_4\}$. For the candidate t_3, we compute the candidate set for t_v and get $\{t_3, t_4\}$. We see that both (t_2, t_3) and (t_2, t_4) are in $\mathbb{I}$, so we update $r(t_2, t_3)$ and $r(t_2, t_4)$ to true. We repeat this process for the other candidates for t_u. Finally, we have the updated transitive closure shown in Figure 4b.

3.4.4 Path Finding. When solving the SAT problem, we need to find a path from transaction t_i to t_j if reachable. We maintain an array P similar to R. When adding an edge (t_p, t_q) makes $r(t_i, t_j)$ true, we update $P[O[i] + j - L[i]]$, i.e., $p(i, j)$, to (t_p, t_q).

To retrieve the path from t_i to t_j, we access $p(t_i, t_j)$ to obtain the edge (t_p, t_q), which decomposes the path into $t_i \rightsquigarrow t_p \rightarrow t_q \rightsquigarrow t_j$. We recursively retrieve $p(t_i, t_p)$ and $p(t_q, t_j)$ to reconstruct the subpaths $t_i \rightsquigarrow t_p$ and $t_q \rightsquigarrow t_j$. This process continues until $p(t_m, t_n)$ directly corresponds to the edge (t_m, t_n).

Figure 4c illustrates the update of P when edge (t_2, t_3) is added to the graph in Figure 4a. This addition sets $r(t_2, t_3)$ and $r(t_2, t_4)$ to true, updating $p(t_2, t_3)$ and $p(t_2, t_4)$ to (t_3, t_4). To find the path from t_2 to t_4, we access $p(t_2, t_4)$, retrieve (t_2, t_3), and decompose the path as $t_2 \rightarrow t_3 \rightsquigarrow t_4$. Since $p(t_3, t_4)$ is absent in P, it implies that t_3 reaches t_4 via the time-dependency edge (t_3, t_4). Thus, we add (t_3, t_4) to complete the path $t_2 \rightarrow t_3 \rightarrow t_4$.



(a) Graph G.

	t_1	t_2	t_3	t_4
t_1	T			
t_2		T	T	T
t_3		F	T	
t_4		F	F	T

(b) Transitive closure.

	t_1	t_2	t_3	t_4
t_1	(1,1)			
t_2		(2,2)	(2,3)	(2,3)
t_3		N	(3,3)	
t_4		N	N	(4,4)

(c) Path matrix.

Fig. 4. Updating transitive closure and path matrix. Empty entries in both matrices are not stored, as we use compact matrices. In the transitive closure R, $R_{ij} = T$ indicates that t_i can reach t_j, while $R_{ij} = F$ indicates that t_i cannot reach t_j. In the path matrix P, $P_{ij} = N$ indicates that there is no path from t_i to t_j, whereas $P_{ij} = (a, b)$ indicates that a path from t_i to t_j can be decomposed into a path from t_i to t_a, the edge (t_a, t_b), and a path from t_b to t_j.

3.5 SAT Problem Formulation

We reduce the constraint assignment problem to a Boolean satisfiability (SAT) instance. We first introduce the Boolean variables used to represent candidate edges and edge sets, and then present the SAT encodings of item constraints, predicate constraints, and truth constraints.

3.5.1 Variables and Literals. To formulate the SAT instance, we introduce the following classes of Boolean variables:

- **Item variables**: For each item constraint $c = (E_{ij}, E_{ji}) \in C_I$, we define two variables b^{item}_{ij} and b^{item}_{ji}. The variable $b^{\text{item}}_{ij} = \text{true}$ if and only if all edges in E_{ij} are included in the compatibility graph G'.
- **Predicate variables**: For each predicate constraint $c \in C_P$ and each candidate edge set $E^{x_i}_{\theta}(\delta) \in c$, we introduce a variable $b^{\text{pred}}_{c,i}$, which evaluates to true if and only if the determined edges in $E^{x_i}_{\theta}(\delta)$ are added to G'.
- **Truth variables**: For each truth constraint $(\tau^{\theta}_{\text{true}}, \tau^{\theta}_{\text{false}}) \in C_T$, we define b^{trut}_{θ}. Here, $b^{\text{trut}}_{\theta} =$ true indicates the selection of the true branch $\tau^{\theta}_{\text{true}}$, while $b^{\text{trut}}_{\theta} =$ false indicates the false branch.
- **Edge variables**: For each undetermined edge e, we define b^{edge}_{e}, where $b^{\text{edge}}_{e} = \text{true}$ if and only if e is included in G'.

To unify the encoding, for each edge e, we define a literal l_e that reflects its existence in G' based on the mapping:

$$l_e = \begin{cases} b^{\text{edge}}_{e} & \text{if } e \text{ is an undetermined edge,} \\ b^{\text{pred}}_{c,i} & \text{if } e \in E^{x_i}_{\theta}(\delta) \text{ for some } c \in C_P, \\ b^{\text{item}}_{ij} & \text{if } e \in E_{ij} \text{ for some } (E_{ij}, E_{ji}) \in C_I, \\ b^{\text{item}}_{ji} & \text{if } e \in E_{ji} \text{ for some } (E_{ij}, E_{ji}) \in C_I. \end{cases} \tag{9}$$

3.5.2 SAT Encoding of Constraints. The requirements of the constraint assignment problem are encoded into four sets of SAT clauses.

Item Constraint Encoding. For an item constraint $c = (E_{ij}, E_{ji}) \in C_I$ that is always active, exactly one of the two directions must be selected:

$$\phi_I^{\text{basic}}(c) = (b_{ij}^{\text{item}} \vee b_{ji}^{\text{item}}) \wedge (\neg b_{ij}^{\text{item}} \vee \neg b_{ji}^{\text{item}}). \tag{10}$$

If the existence of c is guarded by a truth constraint with variable b_θ^{trut}, then the item constraint is enabled only when $b_\theta^{\text{trut}} = \text{true}$:

$$\begin{aligned}\phi_I^{\text{guard}}(c) = (\neg b_\theta^{\text{trut}} \vee b_{ij}^{\text{item}} \vee b_{ji}^{\text{item}}) \wedge (\neg b_\theta^{\text{trut}} \vee \neg b_{ij}^{\text{item}} \vee \neg b_{ji}^{\text{item}}) \\ \wedge (b_\theta^{\text{trut}} \vee \neg b_{ij}^{\text{item}}) \wedge (b_\theta^{\text{trut}} \vee \neg b_{ji}^{\text{item}}).\end{aligned} \tag{11}$$

When $b_\theta^{\text{trut}} = \text{true}$, the constraint reduces to Eq. (10); when $b_\theta^{\text{trut}} = \text{false}$, both b_{ij}^{item} and b_{ji}^{item} are forced to false.

The conjunction of all item constraints is:

$$\Phi_I = \bigwedge_{c \in C_I} \phi_I(c), \quad \phi_I(c) \in \{\phi_I^{\text{basic}}(c), \phi_I^{\text{guard}}(c)\}. \tag{12}$$

Predicate Constraint Encoding. Let $c \in C_P$ be a predicate constraint with candidate edge sets $\{E_\theta^{x_1}, \ldots, E_\theta^{x_n}\}$ and corresponding variables $\{b_{c,1}^{\text{pred}}, \ldots, b_{c,n}^{\text{pred}}\}$. Exactly one candidate must be selected:

$$\phi_P^{\text{choice}}(c) = (b_{c,1}^{\text{pred}} \vee \cdots \vee b_{c,n}^{\text{pred}}), \tag{13}$$

$$\phi_P^{\text{exclusive}}(c) = \bigwedge_{1 \le i < j \le n} (\neg b_{c,i}^{\text{pred}} \vee \neg b_{c,j}^{\text{pred}}). \tag{14}$$

Let $c_{\text{ctrl}} \subseteq c$ denote the subset of candidate edge sets guarded by a truth constraint b_θ^{trut}. For each $E_\theta^{x_i} \in c_{\text{ctrl}}$, we add the guarding clause:

$$\phi_P^{\text{guard}}(c, i) = (\neg b_{c,i}^{\text{pred}} \vee b_\theta^{\text{trut}}). \tag{15}$$

The full encoding of predicate constraint c is:

$$\phi_P(c) = \phi_P^{\text{choice}}(c) \wedge \phi_P^{\text{exclusive}}(c) \wedge \bigwedge_{E_\theta^{x_i} \in c_{\text{ctrl}}} \phi_P^{\text{guard}}(c, i), \tag{16}$$

and the total predicate formula is:

$$\Phi_P = \bigwedge_{c \in C_P} \phi_P(c) \tag{17}$$

Undetermined Edge Clauses. Let e be an undetermined anti-dependency edge arising from a predicate candidate $E_\theta^{x_i} \in c$. The inclusion of e (represented by the Boolean variable b_e^{edge}) is conditioned on the selection of the corresponding predicate candidate $b_{c,i}^{\text{pred}}$, the presence of the write-dependency edge that determines e, represented by the literal $l_{e'}$, and, if applicable, the associated truth constraint b_θ^{trut}:

$$\phi_U(e) = (\neg b_e^{\text{edge}} \vee b_{c,i}^{\text{pred}}) \wedge (\neg b_e^{\text{edge}} \vee l_{e'}) \wedge (\neg b_e^{\text{edge}} \vee b_\theta^{\text{trut}}). \tag{18}$$

All such clauses are conjoined to form:

$$\Phi_U = \bigwedge_{e \in E_{\text{undetermined}}} \phi_U(e). \tag{19}$$

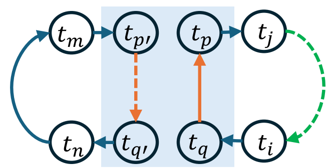


Fig. 5. Example illustrating propagation and analysis.

Acyclicity and Overall Formula. To enforce serializability, the resulting graph G' must be acyclic:

$$\Phi_{\text{acyc}} \equiv \text{acyclic}(G'). \tag{20}$$

The complete SAT formula is defined as:

$$\Phi_{\text{SER}} = \Phi_I \wedge \Phi_P \wedge \Phi_U \wedge \Phi_{\text{acyc}}. \tag{21}$$

Serializability verification reduces to checking the satisfiability of Φ_{SER}. A satisfying assignment corresponds to an acyclic SG.

3.6 Customized SAT Problem Solver

For SAT instances consisting only of pure Boolean formulas, a number of highly optimized solvers are available, including MiniSat[13], CaDiCaL [7], and Kissat [15]. However, our formulation additionally includes the acyclicity constraint $\Phi_{\text{acyc}} \equiv \text{acyclic}(G')$, which cannot be handled directly by standard SAT solvers.

MonoSAT [3] is a state-of-the-art solver for SAT acyclicity. Nevertheless, MonoSAT is not applicable to our formulation. MonoSAT assumes a one-to-one correspondence between Boolean variables and individual graph edges, whereas in our encoding a Boolean variable may represent an entire set of edges. Moreover, MonoSAT does not fully exploit the structural information maintained in the transitive closure. For these reasons, we design a customized solver tailored to our SAT formulation.

Our solver consists of a MiniSat [13] core and a dedicated theory solver for acyclicity. MiniSat is responsible for generating partial assignments that satisfy the Boolean part $\Phi_I \wedge \Phi_P \wedge \Phi_U$. During assignment, we heuristically guide the branching process using timestamp information. For example, given an item constraint (E_{ij}, E_{ji}), if $\text{s}(t_i) < \text{s}(t_j)$, we prioritize exploring the branch E_{ij} first.

Given such an assignment, the theory solver checks whether the acyclicity constraint Φ_{acyc} is satisfied, using a compact representation of the transitive closure. In addition, the theory solver feeds back conflict and propagation information to guide MiniSat's search. The interaction between the two components is realized through two mechanisms: propagation and analysis.

Propagation. During propagation, the theory solver derives additional assignments based on the partial model produced by MiniSat. It applies the pruning procedure described in §3.3 to identify edge sets or constraints that must be selected or eliminated, and assigns the corresponding Boolean variables to true or false accordingly.

As an illustrative example, consider the known graph shown in Figure 5, where blue solid arcs denote edges that have already been added. Let $c = (E_{pq}, E_{qp})$ be an item constraint with $E_{pq} = \{(t_{p'}, t_{q'})\}$ and $E_{qp} = \{(t_q, t_p)\}$. After adding the edge (t_m, t_n), $t_{p'}$ becomes reachable from $t_{q'}$. Consequently, adding $(t_{p'}, t_{q'})$ would create a cycle. The solver therefore selects the alternative edge $(t_q, t_p) \in E_{qp}$ and assigns the corresponding item variable to false.

Analysis. Assignments produced during propagation may still lead to cycles in the known graph. When this occurs, the analysis phase identifies the underlying reasons for the cycle and derives conflict clauses that are added to the Boolean formula $\Phi_I \wedge \Phi_P \wedge \Phi_U$, thereby preventing the same cyclic configuration from reoccurring.

Continuing the example in Figure 5, suppose an assignment sets the literal corresponding to the edge (t_j, t_i) to true. When attempting to add (t_j, t_i) to the current graph G, the theory solver discovers a path $p_{ij} = t_i \rightarrow t_q \rightarrow t_p \rightarrow t_j$, which together with (t_j, t_i) forms a cycle. To eliminate this conflict, the solver generates the clause $p_1 = \neg l_{iq} \vee \neg l_{qp} \vee \neg l_{pj} \vee \neg l_{ji}$, where each literal corresponds to one edge on the cycle. This clause excludes assignments in which all four edges are simultaneously selected. By adding p_1 to the Boolean formula, future assignments that would induce the same cycle are ruled out.

In the implementation, we further strengthen conflict clauses by identifying deeper reasons using the First Unique Implication Point (F-UIP) technique [31], which allows the solver to prune a larger portion of the search space.

4 Evaluation

We implemented our black-box serializability verification method Vbox in C++ and evaluated its performance by experiments. We compared Vbox with three existing verification methods BE [8], Cobra [28] and Leopard [22]. All the experiments were conducted on a Linux server equipped with an Intel Xeon Gold 6130 CPU and 512 GB of RAM. Cobra uses GPU to accelerate transitive closure computation, while other methods can only use CPU.

4.1 Completeness

We first evaluate the completeness of the verification methods in terms of their ability to detect various types of anomalies.

Workloads. To evaluate the completeness of the verifiers, we consider both real-world and synthetic transaction execution histories.

For real-world workloads, we select three histories reported in [28], which together contain seven confirmed serializability anomalies.

For synthetic workloads, we follow the methodology of [21] and reuse transaction examples from its project repository.[1] Each example specifies a concrete interleaving of operations that would exhibit a serializability anomaly in the absence of concurrency control. We replay these operation sequences on MySQL under the READ COMMITTED isolation level and record the resulting execution histories. Since each example involves only a small number of transactions, we exhaustively enumerate all possible serial schedules and retain only those histories for which no equivalent serial schedule exists. This process yields 29 synthetic anomalous histories. Following the classification in [21], the synthetic anomalies are divided into three categories: Read Anomaly Type (RAT), Write Anomaly Type (WAT), and Intersect Anomaly Type (IAT), containing 13, 9, and 7 anomalies, respectively.

Results. Table 1 shows the number of anomalies detected by the verification methods. It is remarkable that our method Vbox successfully detects all the real and the synthetic anomalies, while other methods fail to detect all of them. This is because Vbox detects aborted reads and intermediate reads and can cover all the anomalies by item constraints and predicate constraints.

Cobra and BE exhibit consistent anomaly detection capabilities since determining the commit order in BE is fundamentally equivalent to identifying the acyclic compatible graph in Cobra. Both

[1] https://github.com/Tencent/3TS/tree/coo-consistency-check

Table 1. Fractions of detected anomalies.

History type	Anomaly type	Vbox	Cobra	BE	Leopard
Synthetic	RAT	13/13	6/13	7/13	3/13
Synthetic	WAT	9/9	5/9	5/9	0/9
Synthetic	IAT	7/7	5/7	5/7	0/7
Real-world	Real	7/7	7/7	6/7	1/7

Cobra and BE miss some anomalies because both of them do not check for aborted reads and intermediate reads, assume each transaction writes to each object only once, and lack support for predicates. These limitations are overcome by Vbox.

Leopard demonstrates the weakest anomaly detection capability among the three methods. This is mainly because Leopard relies on the time information of the operations in the transactions, which is absent in the real-world histories. Although the synthetic histories contain time information, Leopard still fails to recognize some anomalies due to its incomplete abstraction of MySQL's concurrency control protocol, which only detects anomalies resulting from concurrent writes and does not address those caused by reads. Vbox also overcomes this limitation of Leopard.

4.2 Efficiency

We next evaluate the efficiency of the verification methods in terms of verification time and memory consumption.

Workloads. To assess efficiency, we generate transaction execution histories using three benchmarks: TPC-C [29], C-Twitter [23], and BlindW [28].

- **TPC-C** is a standard OLTP benchmark comprising five transaction types: new-order, order-status, payment, delivery, and stock-level. We run TPC-C with a single warehouse and default parameter settings.
- **C-Twitter** models transaction patterns in social-network applications and is designed to stress high-concurrency, update-intensive workloads. We follow the implementation described in [28].
- **BlindW** is a synthetic benchmark operating on a randomly generated table with 10,000 rows and schema (k, v_1, v_2). It consists of read-only and write-only transactions and is evaluated under four configurations: BlindW-RH (80% reads, 20% writes), BlindW-WR (50% reads, 50% writes), BlindW-WH (20% reads, 80% writes), and BlindW-Pred (50% predicate reads, 50% predicate writes). In BlindW-Pred, predicates are randomly generated range filters on either column v_1 or v_2, and the number of qualifying objects is uniformly distributed between 1 and 1,000.

All benchmarks are executed under the SERIALIZABLE isolation level of PostgreSQL. For each benchmark configuration, we collect one execution history consisting of 10,000 committed transactions.

Results. Figure 6 reports the verification time and memory consumption of all methods. Across most workloads, Vbox consistently achieves the lowest verification time. This advantage arises from a combination of complementary optimizations, including constraint pruning and consolidation, time-aware graph algorithms, and a customized SAT solver tailored to the serializability verification problem. In addition, Vbox maintains low memory consumption by storing a compact representation of the transitive closure instead of a full reachability matrix.

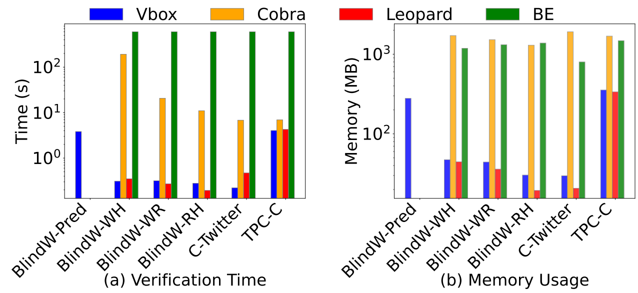


Fig. 6. Verification time and memory usage.

BE times out (exceeding 10 minutes) on all workloads. This behavior is expected, as BE exhaustively enumerates possible commit orders and retains a large number of intermediate states during the search, leading to exponential time complexity and substantial memory overhead.

Leopard exhibits verification time and memory usage comparable to those of Vbox. Rather than constructing a serialization or compatibility graph, Leopard directly checks execution histories against protocol-specific constraints, resulting in linear time and space complexity.

Both Cobra and Vbox verify serializability by constructing a compatible graph. However, Cobra relies on Warshall's algorithm [30] to maintain a full transitive closure and employs a general-purpose SAT solver, MonoSAT [3], which does not exploit the structure of the verification problem. Consequently, Cobra incurs higher verification time and memory consumption than Vbox.

4.3 Scalability

We next evaluate the scalability of the verification methods with respect to the number of transactions in the execution history. We generate 10 BlindW-WR histories containing between 10,000 and 100,000 transactions and verify them using different methods. The results are shown in Figure 7. Note that BE times out (exceeding 10 minutes) on histories with 10,000 transactions and is therefore omitted from this experiment.

As the history size increases from 10K to 100K transactions, the verification time of Vbox grows from 0.31 s to 4.16 s (13.4×), while memory consumption increases from 44 MB to 417 MB (9.5×), demonstrating near-linear scalability. This behavior primarily results from the use of a compact transitive closure and optimized algorithms for its construction and incremental maintenance.

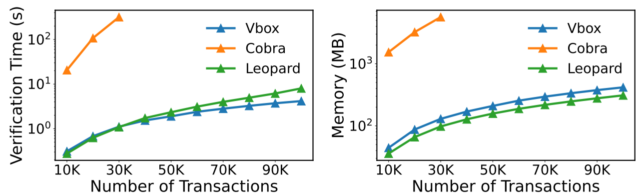
Leopard also exhibits approximately linear growth in both verification time and memory usage. However, once the history size exceeds 30,000 transactions, its verification time surpasses that of Vbox. This is because Leopard repeatedly checks temporal overlap among operations, and the cost of such checks increases with the number of transactions.

In contrast, Cobra shows limited scalability. As the number of transactions increases from 10,000 to 30,000, its verification time rises sharply from 20 s to 318 s, and it times out on histories with more than 40,000 transactions.

4.4 Effectiveness

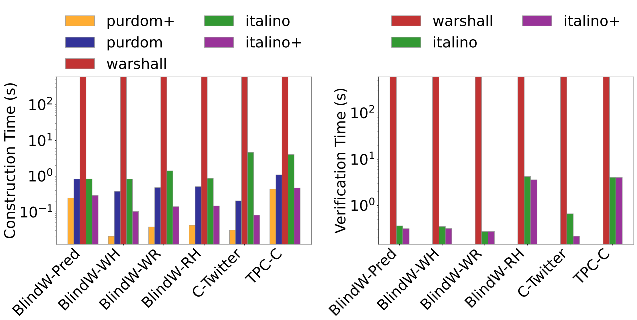
We further evaluate the effectiveness of the techniques specifically designed for Vbox by isolating and comparing their individual contributions.

Transitive Closure Construction. Figure 8a reports the time required to construct the full transitive closure using Warshall's, Purdom's, and Italiano's algorithms, as well as the time to construct the



(a) Execution time. (b) Memory usage.

Fig. 7. Scalability.



(a) Construction. (b) Updating.

Fig. 8. Constructing and updating transitive closure.

compact transitive closure using our enhanced variants, Purdom+ and Italiano+. Note that Italiano and Italiano+ construct the transitive closure incrementally by adding edges one at a time.

Across all histories, Purdom+ achieves the fastest construction time, outperforming the original Purdom's algorithm by a factor of 3-17×. This improvement stems from exploiting transaction time information to accelerate topological sorting and descendant-set merging. Although Italiano's algorithm is primarily designed for incremental updates, Italiano+ also outperforms the original Purdom's algorithm. In contrast, Warshall's algorithm times out (exceeding 10 minutes) on all histories due to its $O(n^3)$ time complexity, where n is the number of transactions.

Transitive Closure Updating. Figure 8b compares different variants of Vbox that employ alternative transitive-closure update strategies, including reconstructing the full closure using Warshall's algorithm, incrementally updating the full closure using Italiano's algorithm, and incrementally updating the compact closure using Italiano+. Among these variants, Vbox with Italiano+ achieves the shortest verification time, as transaction time information enables filtering out unnecessary updates to the transitive closure.

Constraint Reduction. Table 2 compares the number of item constraints remaining after constraint reduction in Vbox and Cobra. The row labeled "Total" reports the number of potential item constraints between every pair of transactions that write the same object. Both methods eliminate a substantial fraction of redundant constraints. However, Vbox achieves significantly stronger reduction than Cobra by incorporating time-dependency edges into the known graph, which yields a more complete partial order and enables more aggressive pruning of redundant constraints.

Table 2. Quantity of item constraints after reduction.

	BlindW-WH	BlindW-WR	BlindW-RH	C-Twitter	TPC-C
Total	211,566	98,451	25,908	530,605	612,565
Cobra	17,061	3,829	437	1,371	0
Vbox	43	17	2	40	0

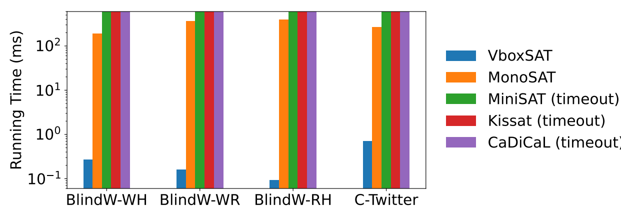


Fig. 9. Running time of SAT solvers.

SAT Solvers. We compare our customized solver VboxSAT with MiniSat [13], Kissat [15], CaDiCaL [7], and MonoSAT [3]. VboxSAT natively supports our SAT formulation. MonoSAT introduces a Boolean variable for each pair of transactions accessing the same object and encodes item constraints using Eq. (10). For MiniSat, Kissat, and CaDiCaL, acyclicity is enforced via a pure Boolean encoding [16]. Figure 9 reports solver runtimes, including formula generation. Overall, VboxSAT consistently achieves the best performance across all workloads, followed by MonoSAT; MiniSat, Kissat, and CaDiCaL all time out under our encoding (> 10min).

The superior performance of VboxSAT is due to its maintained and compact transitive closure for cycle detection, which enables efficient cycle checking and conflict analysis. In contrast, MiniSat, Kissat, and CaDiCaL rely on Boolean encodings of acyclicity whose size grows cubically with the number of transactions, leading to prohibitively large formulas even for moderately sized histories.

MonoSAT benefits from native support for acyclicity constraints and thus outperforms general-purpose SAT solvers, but remains slower than VboxSAT. While MonoSAT introduces Boolean variables for individual edges and relies on dynamic topological sorting with repeated graph reconstruction, VboxSAT can represent an entire edge set with a single variable, resulting in significantly fewer clauses.

5 Conclusion

Vbox is verified to be correct and more efficient and more capable of detecting more data anomalies than the existing methods, while not relying on any specific concurrency control protocols. The superiority of Vbox in anomaly detection results from its capability of characterizing anomalies related to predicate read and write operations. The advantages of Vbox in time and space efficiency are attributed to the application of client-side time information of transactions in known graph construction and constraint reduction, the adoption of the compact transitive closure, and the efficient formulation and solver for the SAT problem. Future work includes developing an online version of Vbox for real-world production scenarios and extending Vbox to support the verification of weak isolation levels such as repeatable read.

A Proof of Theorem 3.4

Let $\mathcal{T}$ be a set of transactions and $H' = (\mathcal{T}', \_)$ be an observed history of executing $\mathcal{T}$. Let C_I, C_P and C_T denote the sets of item, predicate and truth constraints constructed for H', respectively.

Let G' be the resulting compatible graph. According to Definition 2.3, the serializability of H' is equivalent to the existence of a complete history $H = (\mathcal{T}, \prec_v)$ compatible with H' that is serializable. Therefore, Theorem 3.4 can be stated in two parts.

Theorem A.1. *If there exists an assignment for C_I, C_P and C_T such that G' is acyclic and H' contains no aborted reads, then there exists a complete history H compatible with H' that is serializable.*

Theorem A.2. *If there exists a complete history H compatible with H' that is serializable, then there exists an assignment for C_I, C_P and C_T such that G' is acyclic and H' contains no aborted reads.*

Proof of Theorem A.1. We first process each invisible predicate write $w(x, \_, x_j)$ according to the assignment of C_T. For each such operation, the associated truth constraint is $\{\tau^{\theta}_{\text{true}}(x_j), \tau^{\theta}_{\text{false}}(x_j)\}$. If $\tau^{\theta}_{\text{true}}(x_j)$ is selected, then $w(x, \_, x_j)$ is interpreted as an invisible predicate read $r_\theta(x, \_)$ that evaluates to true, followed by an item write $w(x, x_j)$. Otherwise, it is interpreted as an invisible predicate read $r_\theta(x, \_)$ that evaluates to false.

We next construct the version order $\prec_v$ from the assignment of C_I. For any two distinct transactions t_i and t_j that create different versions x_m and x_n of the same object x, the corresponding item constraint is (E_{ij}, E_{ji}). If E_{ij} is selected, we define $x_m \prec_v x_n$; otherwise, $x_n \prec_v x_m$.

We then complete each invisible predicate read according to the assignment of C_P. For each $r_\theta(x, \_)$, the corresponding predicate constraint is $\{E^{x_i}_{\theta}(\delta) \mid x_i \in S^{x}_{\theta}(\delta)\}$. If $E^{x_i}_{\theta}(\delta)$ is selected, the read $r_\theta(x, \_)$ is completed with version x_i.

Lemma A.3 shows that $\prec_v$ is a total order. Using $\prec_v$ and replacing each invisible predicate read $r_\theta(x, \_)$ with its completed form $r_\theta(x, x_i)$, we obtain a complete history $H = (\mathcal{T}, \prec_v)$ that is compatible with H'. Lemma A.4 further guarantees that H is serializable, thereby completing the proof of Theorem A.1.

Lemma A.3. *For each object x, the version order $\prec_v$ is a total order over all versions of x.*

Proof. We show that $\prec_v$ satisfies the defining properties of a total order.

(Totality). For any two versions x_i and x_j of the same object x, either $x_i \preceq_v x_j$ or $x_j \preceq_v x_i$ holds by construction.

(Reflexivity). For every version x_i of x, we trivially have $x_i =_v x_i$.

(Transitivity). For any versions x_i, x_j, and x_k, if $x_i \prec_v x_j$ and $x_j \prec_v x_k$, then $x_i \prec_v x_k$. Otherwise, if $x_k \prec_v x_i$, the graph G' would contain a cycle among the transactions that created x_i, x_j, and x_k, contradicting the acyclicity of G'.

(Antisymmetry). If $x_i \preceq_v x_j$ and $x_j \preceq_v x_i$, then $x_i =_v x_j$. Otherwise, G' would contain a cycle involving the transactions that created x_i and x_j, again contradicting acyclicity.

Hence, $\prec_v$ is a total order. □

Lemma A.4. *The complete history $H = (\mathcal{T}, \prec_v)$ is serializable.*

Proof. Let G_1 be the SG of H, and let $e = (t_a, t_b)$ be any edge in G_1. For convenience, we use the same vertex notation for both G' and G_1. We analyze e according to its type.

(1) Item dependencies. If e is an item read-dependency, item write-dependency, or item anti-dependency edge, then e also appears in G' by construction.

(2) Predicate read-dependencies. If e is a predicate read-dependency edge, then there exists a predicate read $r_\theta(x, x_j)$ in transaction t_b of the complete history H. By Theorem 2.2, transaction t_a performs a write that creates a version x_i such that $\theta(x_i) = \theta(x_j)$. In H', the corresponding operation in t_b is the invisible predicate read $r_\theta(x, \_)$. For this read, we construct a predicate constraint, and the assignment selects $E^{x_j}_{\theta}(\delta)$. Consequently, an edge $(t_m, t_b) \in E^{x_j}_{\theta}(\delta)$ appears in G', where t_m creates version x_j and $x_i \preceq_v x_j$, which implies $(t_a, t_m) \in \mathbb{W}$. Therefore, there exists

a path from t_a to t_b in G' consisting of the item write-dependency edge (t_a, t_m) followed by the (candidate) predicate read-dependency edge (t_m, t_b).

(3) Predicate anti-dependencies. If e is a predicate anti-dependency edge, then t_a can also reach t_b in G'. The argument is analogous to the predicate read-dependency case and is omitted for brevity.

In all cases, reachability in G_1 implies reachability in G'. Since G' is acyclic, G_1 must also be acyclic. Moreover, because H' contains no intermediate or aborted reads, the complete history H inherits this property. Therefore, H is serializable. □

Proof of Theorem A.2. Let $H = (\mathcal{T}, \prec_v)$ be a complete history that is compatible with H' and serializable. We construct assignments for C_T, C_I, and C_P, and show that the resulting compatible graph G' is acyclic and that H' contains no aborted reads.

We first assign truth constraints in C_T. For each truth constraint $\{\tau^{\theta}_{\text{true}}(x_j), \tau^{\theta}_{\text{false}}(x_j)\}$ corresponding to an invisible predicate write $w(x, \_, x_j)$ in H', we inspect the evaluation result of the corresponding predicate write in H. If the predicate evaluates to true, we select $\tau^{\theta}_{\text{true}}(x_j)$ and interpret $w(x, \_, x_j)$ as an invisible predicate read $r_\theta(x, \_)$ followed by an item write $w(x, x_j)$; otherwise, we select $\tau^{\theta}_{\text{false}}(x_j)$.

We next assign item constraints in C_I. For each item constraint $(E_{ij}, E_{ji}) \in C_I$ associated with an object x, let x_m and x_n be the versions of x created by transactions t_i and t_j, respectively. If $x_m \prec_v x_n$, we select E_{ij}; otherwise, we select E_{ji}. The selected edges are added to the compatible graph G'.

We then assign predicate constraints in C_P. For each predicate constraint $\{E^{x_i}_{\theta}(\delta) \mid x_i \in S^{x}_{\theta}(\delta)\}$ associated with an invisible predicate read $r_\theta(x, \_)$ in H', let $r_\theta(x, x_i)$ be the corresponding completed predicate read in H. We select $E^{x_i}_{\theta}(\delta)$ and add its edges to G'.

Using the assignments for C_T, C_I, and C_P, we obtain the compatible graph G'. Lemma A.5 establishes that G' is acyclic. Moreover, since the complete history H contains no intermediate or aborted reads, the incomplete history H' also exhibits no such read anomalies. Therefore, Theorem A.2 holds.

Lemma A.5. *The compatible graph G' constructed from the assignments for C_I, C_P, and C_T is acyclic.*

Proof. Let G_1 be the SG of the complete history H, and let $e = (t_a, t_b)$ be an arbitrary edge in G'. We analyze e according to its type.

(1) Item dependencies. If e is an item read-dependency, item write-dependency, or item anti-dependency edge, then e is also an edge in G_1 by construction.

(2) Predicate read-dependencies. If e is a predicate read-dependency edge induced by an invisible predicate read, then e directly appears in G_1. If e is induced by a visible predicate read, then t_a can reach t_b in G_1 via a sequence of item-dependency edges.

(3) Predicate anti-dependencies. If e is a predicate anti-dependency edge, then there exist a version x_m read by an appropriate predicate read $r_\theta(x, x_m)$ in t_a and a version x_n created by t_b such that $x_m \prec_v x_n$. Let (t_a, t_p) be the predicate anti-dependency edge in G_1 corresponding to $r_\theta(x, x_m)$, and let x_w be the version created by t_p. By definition of predicate anti-dependency, $x_w \preceq_v x_m$, since x_w is the earliest version satisfying the anti-dependency condition. Consequently, there exists a path in G_1 from t_a to t_b consisting of the predicate anti-dependency edge (t_a, t_p) followed by a sequence of item write-dependency edges connecting the transactions that create the versions between x_m and x_n.

In all cases, reachability in G' implies reachability in G_1. Since G_1 is acyclic, G' must also be acyclic. □

B Proof of Theorem 3.5

Let $\mathcal{T}$ be a set of transactions, and let $H' = (\mathcal{T}', \_)$ be the observed history of executing the transactions in $\mathcal{T}$. These transactions are executed under one of the following concurrency control protocols: Serializable Snapshot Isolation (SSI), Two-Phase Locking (2PL), Optimistic Concurrency Control (OCC), Timestamp Ordering (TO), or Percolator. The time dependency is constructed as follows: for any pair of transactions t_i and t_j, if $\mathrm{e}^r(t_i) \le \mathrm{s}^r(t_j)$, the time dependency edge (t_i, t_j) is added to the edge set $\mathbb{T}^r$. We call a SG is compatible with the observed history H' if there is a complete history H compatible with H', and the SG is generated from H. Theorem 3.5 can be divided into the following two parts:

Theorem B.1. *If there is no acyclic SG compatible with the observed history H', there is no SG compatible with H' that remains acyclic after adding all time-dependency edges in $\mathbb{T}^r$.*

Theorem B.2. *If there is an acyclic SG that is compatible with the observed history H', there exists a SG compatible with H' that remains acyclic after adding all the time-dependency edges in $\mathbb{T}^r$.*

Proof of Theorem B.1. If a SG compatible with the observed history H' remains acyclic after adding all the time-dependency edges in $\mathbb{T}^r$, the SG must be acyclic. This contradicts with the initial assumption that no acyclic SG exists.

Proof of Theorem B.2. We provide the proof for the case where the SG includes only item dependencies. The extension to cases involving predicate dependencies is conceptually straightforward and therefore omitted.

The proof of Theorem B.2 relies on the following lemmas.

Lemma B.3. *For three committed transaction t_i, t_j and t_k, if $(t_i, t_j) \in \mathbb{T}^r$ and $(t_j, t_k) \in \mathbb{T}^r$, we have $(t_i, t_k) \in \mathbb{T}^r$.*

Lemma B.4. *For two committed transaction t_i and t_j, if t_j item write/read/anti-depends on t_i, we have $s^r(t_i) < e^r(t_j)$.*

Lemma B.5. *A directed graph $G_t = (V, \mathbb{T}^r)$ consisting of only the edges in $\mathbb{T}^r$ is acyclic.*

Proof. Sort the vertices in V by the start timestamps of their corresponding transactions, $\mathrm{s}^r(t)$, yielding a total order $\mathbb{T}^s$ on V. For any pair $t_i, t_j \in V$, if $(t_i, t_j) \in \mathbb{T}^r$, we have $\mathrm{e}^r(t_i) \le \mathrm{s}^r(t_j)$ and clearly $\mathrm{s}^r(t_i) \le \mathrm{s}^r(t_j)$, so $(t_i, t_j) \in \mathbb{T}^s$. Hence, $\mathbb{T}^r \subseteq \mathbb{T}^s$. Since $\mathbb{T}^s$ is a total order, the subgraph formed by $\mathbb{T}^r$ is acyclic. □

The following lemmas introduce the time properties of observed histories under different concurrency control protocols. Since these properties are directly derived from the protocols themselves, the proofs are omitted.

Lemma B.6. *The observed history h under SSI satisfies that, for two committed transactions t_i and t_j, if t_j write/read-depends t_i, we have $e^r(t_i) < s^r(t_j)$.*

Lemma B.7. *The observed history h under 2PL satisfies that, for two committed transactions t_i and t_j, if t_j write/reads/ anti-depends on t_i, we have $e^r(t_i) < e^r(t_j)$.*

Lemma B.8. *The observed history h under OCC satisfies that, for two committed transactions t_i and t_j, if t_j read/anti-depends on t_i, we have $e^r(t_i) < s^r(t_j)$. If t_j write-depends on t_i, we have $e^r(t_i) < e^r(t_j)$.*

Lemma B.9. *The observed history h under TO satisfies that, for two committed transactions t_i and t_j, if t_j write/reads/anti-depends on t_i, we have $s^r(t_i) < s^r(t_j)$.*

Lemma B.10. *The observed history h under Percolator satisfies that, for two committed transactions t_i and t_j, if t_j write/read-depends on t_i, we have $s^r(t_i) < e^r(t_j)$. If t_j anti-depends on t_i, we have $e^r(t_i) < e^r(t_j)$.*

Let G_1 be an acyclic SG compatible with the observed history H'. Assume that adding the edges in $\mathbb{T}^r$ to G_1 makes the generated graph G_1' contain one or more cycles. Due to Lemma B.5 and the acyclicity of G_1, each cycle must contain at least one time-dependency edge and one write/read/anti-dependency edge. If the cycle contains two edges, there must be a time dependency edge and a write/read/anti-dependency edge. This conflicts with Lemma B.4. Thus, each cycle must have three or more edges. Next, we prove that Theorem B.2 holds under different protocols.

(1) *Proof of Theorem B.2 under SSI.* According to Lemma B.6, G_1' must contain a cycle consisting of only time-dependency edges and anti-dependency edges. SSI prohibits contiguous anti-dependency edges, and the time dependency edges are transitive, so there is a shortest cycle c in the SG, and the time-dependency edges and the anti-dependency edges appear alternately in c. Let (t_i, t_j) be a time-dependency edge in c. We have $e^r(t_i) \leq s^r(t_j)$. The remaining part of c is a path from t_j to t_i, that is, $p_{ji} = (t_j, t_{j+1}) \to \cdots \to (t_{j+n-1}, t_{j+n}) \to (t_{j+n}, t_i)$. Because time-dependency edges and anti-dependency edges appear alternately in this path, we have $(t_{j+n}, t_i) \in \mathbb{A}$ and $(t_{j+n-1}, t_{j+n}) \in \mathbb{T}^r$. For the anti-dependency edge (t_{j+n}, t_i), we have $s^r(t_{j+n}) < e^r(t_i)$ by Lemma B.4. For time-dependency edge (t_{j+n-1}, t_{j+n}), we have $e^r(t_{j+n-1}) \leq s^r(t_{j+n})$. Thus, $e^r(t_{j+n-1}) \leq s^r(t_{j+n}) \leq e^r(t_i) \leq s^r(t_j)$, that is, $(t_{j+n-1}, t_j) \in \mathbb{T}^r$. Now, we obtain a new cycle induced by the edges $(t_j, t_{j+1}), \ldots, (t_{j+n-2}, t_{j+n-1})$, $(t_{j+n-1}, t_j)\}$. This is inconsistent with the fact that c is one of the shortest cycles. Thus, Theorem B.2 holds under SSI.

(2) *Proof of Theorem B.2 under 2PL.* Let c be a cycle in G_1'. Let (t_i, t_j) be a time-dependency edge in c. We have $e^r(t_i) \leq s^r(t_j)$. The remaining part of the cycle c is a path from t_j to t_i, that is, $p_{ji} = (t_j, t_{j+1}) \to \cdots \to (t_{j+n-1}, t_{j+n}) \to (t_{j+n}, t_i)$. Each edge (u, v) in p_{ji} must satisfy that $e^r(u) < e^r(v)$ according to Lemma B.7. Thus, $e^r(t_j) < e^r(t_i)$ holds. However, it conflicts with the fact that $e^r(t_i) \leq s^r(t_j) < e^r(t_j)$. Thus, Theorem B.2 holds under 2PL.

(3) The proofs of Theorem B.2 under OCC, Percolator and TO are very similar to the proof of Theorem B.2 under 2PL. Therefore, we omit these proofs.

Consequently, Theorem B.2 holds.

References

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 67–78. doi:10.1109/ICDE.2000.839388

[2] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280. doi:10.5555/3430915.3442427

[3] Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. 2015. SAT Modulo Monotonic Theories. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, Blai Bonet and Sven Koenig (Eds.). AAAI Press, 3702–3709. doi:10.1609/AAAI.V29I1.9755

[4] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221. doi:10.1145/356842.356846

[5] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (1983), 465–483. doi:10.1145/319996.319998

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[7] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. 2024. CaDiCaL 2.0. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14681)*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer, 133–152. doi:10.1007/978-3-031-65627-9_7

[8] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28. doi:10.1145/3360591

[9] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015 (LIPIcs, Vol. 42)*, Luca Aceto and David de Frutos-Escrig (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 58–71. doi:10.4230/LIPICS.CONCUR.2015.58

[10] 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 2017, Washington, DC, USA, July 25-27, 2017*, Elad Michael Schiller and Alexander A. Schwarzmann (Eds.). ACM, 73–82. doi:10.1145/3087801.3087802

[11] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, Jun Wei, and Tao Huang. 2024. Understanding Transaction Bugs in Database Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 163:1–163:13. doi:10.1145/3597503.3639207

[12] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1123–1135. doi:10.1109/ICSE48619.2023.00101

[13] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers (Lecture Notes in Computer Science, Vol. 2919)*, Enrico Giunchiglia and Armando Tacchella (Eds.). Springer, 502–518. doi:10.1007/978-3-540-24605-3_37

[14] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528. doi:10.1145/1071610.1071615

[15] ABKFM Fleury and Maximilian Heisinger. 2020. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *Sat Competition* 2020 (2020), 50.

[16] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. 2014. SAT Modulo Graphs: Acyclicity. In *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8761)*, Eduardo Fermé and João Leite (Eds.). Springer, 137–151. doi:10.1007/978-3-319-11558-0_10

[17] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[18] Giuseppe F. Italiano. 1986. Amortized Efficiency of a Path Retrieval Data Structure. *Theor. Comput. Sci.* 48, 3 (1986), 273–281. doi:10.1016/0304-3975(86)90098-8

[19] Paul Walton Purdom Jr. 1970. A Transitive Closure Algorithm. *BIT* 10 (1970), 76–94.

[20] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (1981), 213–226. doi:10.1145/319566.319567

[21] Haixiang Li, Yuxing Chen, and Xiaoyan Li. 2022. Coo: Consistency Check for Transactional Databases. *CoRR* abs/2206.14602 (2022). arXiv:2206.14602 doi:10.48550/ARXIV.2206.14602

[22] Keqiang Li, Siyang Weng, Peiyuan Liu, Lyu Ni, Chengcheng Yang, Rong Zhang, Xuan Zhou, Jianghang Lou, Gui Huang, Weining Qian, and Aoying Zhou. 2023. Leopard: A Black-Box Approach for Efficiently Verifying Various Isolation Levels. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 722–735. doi:10.1109/ICDE55515.2023.00061

[23] Nick Kallen. 2011. Big Data in Real-Time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter/. Accessed: Jan. 2026.

[24] Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J. ACM* 26, 4 (1979), 631–653. doi:10.1145/322154.322158

[25] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 251–264. http://www.usenix.org/events/osdi10/tech/full_papers/Peng.pdf

[26] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861. doi:10.14778/2367502.2367523

[27] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis. 2013. System Level Concurrency Control for Distributed Database Systems. In *Fundamental Problems in Computing, Essays in Honor of Professor Daniel J. Rosenkrantz*, S. S. Ravi and Sandeep K. Shukla (Eds.). Springer, 71–98. doi:10.1007/978-1-4020-9688-4_4

[28] 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

[29] Transaction Processing Performance Council (TPC). 2010. TPC-C Benchmark (Version 5.11). https://www.tpc.org/tpcc/.

[30] Stephen Warshall. 1962. A Theorem on Boolean Matrices. *J. ACM* 9, 1 (1962), 11–12. doi:10.1145/321105.321107

[31] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. 2001. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, Rolf Ernst (Ed.). IEEE Computer Society, 279–285. doi:10.1109/ICCAD.2001.968634

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009