Checking Observational Correctness of Database Systems

LAUREN PICK, The Chinese University of Hong Kong, Hong Kong
AMANDA XU, University of Wisconsin-Madison, USA
ANKUSH DESAI, Amazon Web Services, USA
SANJIT A. SESHIA, University of California, Berkeley, USA
AWS ALBARGHOUTHI, University of Wisconsin-Madison, USA

Clients rely on database systems to be correct, which requires the system not only to implement transactions' semantics correctly but also to provide isolation guarantees for the transactions. This paper presents a client-centric technique for checking *both* semantic correctness and isolation-level guarantees for black-box database systems based on observations collected from running transactions on these systems. Our technique verifies *observational correctness* with respect to a given set of transactions and observations for them, which holds iff there exists a possible correct execution of the transactions under a given isolation level that could result in these observations. Our technique relies on novel symbolic encodings of (1) the semantic correctness of database transactions in the presence of weak isolation and (2) isolation-level guarantees. These are used by the checker to query a Satisfiability Modulo Theories solver. We applied our tool Troubadour to verify observational correctness of several database systems, including PostgreSQL and an industrial system under development, in which the tool helped detect two new bugs. We also demonstrate that Troubadour is able to find known semantic correctness bugs and detect isolation-related anomalies.

CCS Concepts: • **Information systems → Parallel and distributed DBMSs**; • **Theory of computation → Verification by model checking**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: observational correctness, isolation levels, distributed databases, automated verification

ACM Reference Format:
Lauren Pick, Amanda Xu, Ankush Desai, Sanjit A. Seshia, and Aws Albarghouthi. 2025. Checking Observational Correctness of Database Systems. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 139 (April 2025), 28 pages. https://doi.org/10.1145/3720504

1 Introduction

Clients typically treat database management systems (DBMSs) as black boxes that they interact with by issuing transactions and receiving responses. These DBMSs are assumed to be *correct*. That is, they are assumed to correctly implement the semantics of transactions' operations on the underlying database *and* to correctly enforce the isolation levels – specifications that restrict concurrent transactions' ability to see and influence each others' effects – that they claim to provide. Examples of common isolation levels provided by DBMSs include snapshot isolation, which allows transactions to read only from a single "snapshot" of the database state and disallows writes to any values that have changed since the snapshot, and read committed, which allows a transaction to read any previous committed values from any database state. While ensuring the correctness of

Authors' Contact Information: Lauren Pick, The Chinese University of Hong Kong, Hong Kong, Hong Kong, pick@cse.cuhk.edu.hk; Amanda Xu, University of Wisconsin-Madison, Madison, USA, axu44@wisc.edu; Ankush Desai, Amazon Web Services, Cupertino, USA, ankushpd@amazon.com; Sanjit A. Seshia, University of California, Berkeley, Berkeley, USA, sseshia@berkeley.edu; Aws Albarghouthi, University of Wisconsin-Madison, Madison, USA, aws@cs.wisc.edu.

ACM 2475-1421/2025/4-ART139
https://doi.org/10.1145/3720504

DBMS executions is possible with a verified DBMS [38], this is not practical for clients who cannot tolerate performance overheads or otherwise wish to use proprietary DBMSs.

In the absence of a verified DBMS, we would like to be able to *audit* responses from the DBMS to ensure that implementation bugs do not affect the behavior of a client application. In particular, for a given log of a DBMS responses, we would like a guarantee of *observational correctness* – i.e., a guarantee that the client has seen a correct operation of the DBMS. To solve the *DBMS correctness auditing problem*, in this paper, we answer the following question:

How do we verify that a DBMS's observed responses are correct for a set of arbitrary transactions?

Existing tools. Existing tools for bug-finding and isolation checking are ill-suited for solving this general DBMS correctness auditing problem. Testing tools, while effective for identifying bugs during DBMS development, cannot verify observational correctness for arbitrary logs. Furthermore, most (but not all [8]) testing tools for DBMSs focus only on finding semantic bugs *within* transactions and are thus unable to find isolation-related bugs. On the other hand, many isolation checking tools can perform some form of auditing for isolation correctness, but they assume semantic correctness and do not support arbitrary transactions. Many techniques based on Adya's formalism for describing isolation levels rely on seeing all transactions' read and written values [6, 8, 21, 27, 34, 40], which are not always available in practice. For example, a conditional update in SQL will not return the values read and written during the statement's execution.

DBMS correctness auditor. We propose a method for verifying *observational correctness* of a set of arbitrary transactions run on a DBMS by reduction to Satisfiability Modulo Theories (SMT) solving. This method must show that observations collected (e.g., in client-side logs) for the transactions are consistent with both the semantics of the transactions *and* the DBMS-specified isolation level.

Our approach is based on the observation that semantic correctness and isolation levels are related: the former relies on correct values being read, and the latter influences what values can be read. Semantic correctness can be viewed informally as the property that *if correct values are read for the given transaction*, then the implementation of the transaction's behavior is correct according to the semantics for the language in which the transactions are written. Isolation levels can then be viewed as constraints on what values can be read or written by a transaction.

This observation motivates the two key components behind our approach: (1) a symbolic encoding of the nondeterminism in which database states a transaction reads from, which we use to generate symbolic semantic correctness constraints for transactions, and (2) a symbolic encoding of isolation constraints based on the constraints from the formulation by Crooks et al. [12]. We can conjoin the semantic and isolation constraints to yield an SMT formula that is satisfiable iff the overall observational correctness property holds.

Directly posing this formula as an SMT query, however, performs poorly in practice because the formula often requires complex theory reasoning and contains irrelevant constraints. To address this, we present a method for generating a simpler SMT encoding of observational correctness based on a novel fine-grained representation of database states and an operational semantics that works over this representation to generate constraints.

We implemented an automated checker Troubadour based on our symbolic encoding of observational correctness. It can handle a fragment of SQL and common isolation levels. In particular, in addition to the snapshot isolation (SI) and read comitted (RC) levels mentioned above, it supports serializability (S), which allows a transaction only to read from the previous transaction's committed database state for some sequential ordering of transactions; strict serializability (SS), which is like serializability except that the sequential ordering of transactions must be according to transactions' start and end times; and strict snapshot isolation (SSI), which is like SI except that the snapshot must be taken at the transaction start time. As generation of semantic and isolation constraints are

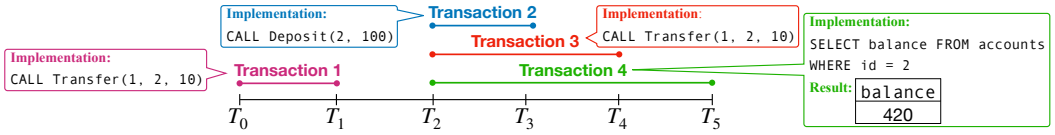


Fig. 1. Sample set of committed transactions and their observations. Each transaction's leftmost and rightmost times respectively indicate its start and commit times.

decoupled, it is easy to extend TROUBADOUR to support other transaction description languages or isolation levels. Experiments demonstrate TROUBADOUR can detect violations of *both* semantic and isolation correctness as well as prove observational correctness for logged transactions. We used TROUBADOUR to check widely-used DBMS PostgreSQL [29] as well as an under-development industrial DBMS, in which its use led to the discovery of two new bugs.

Contributions. We make the following main contributions:

- We present a formal definition of observational correctness for a black-box DBMS (§3). This is to our knowledge the first such definition encompassing both semantic correctness and isolation guarantees.
- We formulate a symbolic encoding of observational correctness for DBMSs, which relies on a lifting of the constraints proposed by Crooks et al., to the symbolic setting (§6) and encoding the nondeterminism in transaction reads (§5).
- We propose a method that uses a novel partially-symbolic representation of database states to generate a simpler encoding of observational correctness (§5.2), and an incremental checking technique based on it (§7).
- We implemented a tool TROUBADOUR that can handle a sizable fragment of SQL and used it to check both semantic- and isolation-related correctness properties of DBMSs (§8).

2 Motivating Example

Let us consider a simple banking application backed by a relational database. We consider three kinds of transactions: *transfers*, which transfer a specified amount of money from a source account to a target one; *deposits*, which deposit a specified amount into a target account; and *balance-checks*, which return the balance of a specified account. Account information is stored in a table with two columns: `id`, which contains bank account id numbers, and `balance`, which contains the bank balances. Bank accounts are identified with their id numbers. We assume the initial state is one where account 1 has balance 100 and 2 has balance 300 and that the DBMS claims to provide the isolation level SSI. This level imposes two constraints on transactions' reads and writes: constraint RMR specifies that each transaction takes a snapshot of the committed database state at its start time and operates over the data in that snapshot; and constraint NWC specifies that, for a transaction to commit, it must not modify any data that has been modified and committed in between its start and commit time. Formal definitions of SSI and other isolation levels are provided in §6.

We want to prove that the DBMS exhibits *observational correctness* for given a set of transactions and observations for them that were collected from a run of the DBMS; i.e., we want to show that there exists a possible correct execution trace of the DBMS that could produce the observed responses. If no such trace exists, there is no possible correct behavior of the DBMS that could have led to the observations, indicating a bug. In practice, observations, which include DBMS responses resulting from issued transactions, can be collected in a client-side log.

Let us consider checking four committed transactions with start and commit times as shown in Fig. 1. We assume time T_i precedes time T_j for $i < j$ and associate each transaction n with

```
CREATE PROCEDURE Deposit (IN tgt INT,
     IN amt DECIMAL(12,2))
    BEGIN
        UPDATE accounts
          SET balance = balance + @amt
          WHERE acc_id = @tgt;
    END;
```

```
CREATE PROCEDURE Transfer (IN src INT, IN tgt INT,
     IN amt DECIMAL(12,2))
    BEGIN
      CALL Deposit(@tgt, @amt);
      UPDATE accounts SET balance = balance - @amt
        WHERE acc_id = @src;
    END;
```

Fig. 2. Procedures implementing deposits and transfers in MySQL syntax.

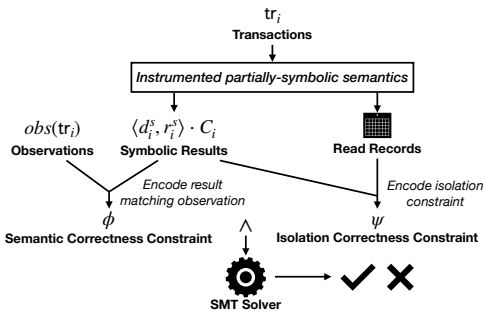


Fig. 3. Verifying observational correctness of a DBMS.

transaction ID n. We assume start and commit times are observable, as these are necessary to check the constraints for SSI. (These start and commit times are not required for non-strict isolation levels.) The behaviors of these transactions as well as the information in the observations (start and commit times, as well as results from the DBMS) that we collect from running them on our hypothetical DBMS are shown in the bubbles in Fig. 1. The figure omits the commit status of the transactions, which is also collected as part of the observations, as the transactions that have all committed. Behaviors are specified using calls to SQL stored procedures shown in Fig. 2, except for balance-check transactions, which consist of `SELECT` statements: Transaction 1 transfers 10 from account 1 to 2, transaction 2 deposits 100 into account 2, transaction 3 transfers 10 from account 1 to 2, and transaction 4 returns the balance of account 2.

Observations in Fig. 1 come from an *incorrect* execution of the transactions that violates the isolation guarantees for SSI: as transactions 2 and 3 both modify the balance for account 2, we can see that transaction 3 should not be able to commit without violating NWC.

Note that other than balance-check transactions, which return the results of the `SELECT` statement, none of the transactions return any information about read or written values. For example, for the `Transfer` transaction implemented as shown in Fig. 2, the typical behavior of a DBMS would be to return neither the rows written by the `UPDATE` statement nor the values written to them, requiring reasoning about transaction semantics to detect this isolation guarantee violation. We will demonstrate how our approach uses knowledge of the transactions' semantics *and* isolation level guarantees to conclude this is the case.

2.1 Verifying Observational Correctness

Verifying observational correctness is challenging in the presence of different isolation levels, which differ in the extent to which they isolate transactions from each others' effects. They may, for example, allow transactions to see results of not-yet-committed transactions, leading to nondeterminism in transactions' reads and writes. For a serializable isolation level, where transactions must behave as if they execute one after another, we could verify observational

correctness of a DBMS for a log by finding an order of the transactions such that executing the transactions in order yields results that match the collected observations in the log. For non-serializable isolation levels, though, we cannot directly use such an approach because transactions may read from (potentially several) *past* database states. In this work, we enable a similar approach to the one just described for logs of serializable transactions by using a semantics that *symbolically* encodes the nondeterminism in which database state is read from.

Fig. 3 shows our overall approach. We rely on a *partially*-symbolic semantics that maintains some database state information concretely for efficiency. This semantics is instrumented to collect additional information about which states each transaction read from in the form of *read records*. Running the semantics to completion for a transaction yields a partially-symbolic representation of sequences of system states d^s and expected database system response r^s after the transaction, a formula C encoding semantic constraints on the symbolic variables in $\langle d^s, r^s \rangle$, and the transaction's read record. These are then used to produce, respectively, the semantic correctness constraint ϕ – which includes C but also encodes that the expected results match those observed in actual DBMS responses – and the isolation correctness constraint ψ, which expresses whether the read records are consistent with a given isolation level. These constraints are then conjoined and given to an SMT solver. If the conjunction is satisfiable (indicated by ✓), then the transactions are observationally correct. If unsatisfiable (indicated by ✗), then they are not.

We use identifiers to represent states that were read from. These state identifiers appear both in read records and the symbolic representations of values in our DBMS state representation. All the states we need to refer to result from transaction commits, so we will identify a state using index i if it resulted from the commit of transaction i. For now, to be compatible with all the isolation levels we consider in this work (SS, S, SSI, SI, and RC), the semantics considers a transaction as being able to read from *all* possible committed states. (It is fairly straightfoward to extend our approach to also handle other isolation levels that allow transactions to read from uncommitted states as well by also keeping track of uncommitted states in the semantics.) We consider further isolation-level-related constraints after encoding the semantic constraints.

Transaction 1. There is only one state to read from: the initial one at T_0. The commit results in a new state at time T_1 where account 1 has balance 90 and 2 has balance 310. That these accounts have these balances at this time is captured in the representation of system states d_1^s produced by the partially-symbolic semantics. The semantics also produces semantic constraint C_1, which is true. There are no observations so far, so the semantic correctness constraint for the log containing only this transaction would be true. The isolation correctness constraint would similarly be true.

Transaction 2. This deposit transaction reads from and writes to the balance of account 2. There are two possible states that can be read from: the initial one at T_0 and one at T_1. We capture this nondeterminism in the partially-symbolic semantics by modeling the read as being from the state associated with *symbolic* state identifier id^s. The read balance is then represented as $\mathit{ite}(\mathrm{id}^s \geq 1, 310, 300)$, with *ite* meaning "if-then-else." This expresses that if the transaction reads the balance from a state resulting from transaction 1's commit or later, it reads value 310, but otherwise it reads value 300. The value the transction writes is thus $\mathit{ite}(\mathrm{id}^s \geq 1, 310, 300) + 100$, which we simplify to $\mathit{ite}(\mathrm{id}^s \geq 1, 410, 400)$. This symbolic written value is stored in d_2^s produced by the partially-symbolic semantics. The constraint C_2 generated by the semantics is simply true, and because there are no observations for this or the previous transaction, the semantic correctness constraint is just true for the first two transactions in the log.

The read records indicate that a read was performed on symbolic state id^s. The isolation constraint that we generate for SSI should require that id^s identifies the latest committed state before time T_2 – the start time of transaction 2 (for the RMR constraint). Since no transaction committed between

T_1 and T_2, we do not need to check if there has been a conflicting write to the balance of account 2 (NWC is trivially true). The isolation constraint for the log up to this point is thus $\mathsf{id}^s = 1$.

As the conjunction $\mathsf{true} \wedge \mathsf{id}^s = 1$ of the semantic and isolation correctness constraints is *satisfiable*, we conclude that the execution of transactions 1 and 2 is observationally correct. Furthermore, a satisfying assignment for this conjunction determines a possible concrete execution trace that is correct. In this case, the only satisfying assignment has $\mathsf{id}^s = 1$ and the balance of account 2 after the commit as 410, as expected for SSI.

Transaction 3. This transaction makes a transfer from account 1 to 2, requiring reads of balances for both accounts 1 and 2 at respective symbolic states with symbolic identifiers id_1^s and id_2^s. Let us focus particularly on the balance of account 2. The partially-symbolic semantics determines that transaction 3 reads $\mathsf{balance}_1$ as account 2's balance, where we define $\mathsf{balance}_1$ to be the following:

$$\mathsf{balance}_1 \equiv \mathit{ite}(\mathsf{id}_2^s \geq 2, \mathit{ite}(\mathsf{id}^s \geq 1, 410, 400), \mathit{ite}(\mathsf{id}_2^s \geq 1, 310, 300))$$

When transaction 3 commits, the balance will therefore be $\mathsf{balance}_1 + 10$. This information is recorded in d_3^s produced by the symbolic semantics.

The isolation constraint for SSI will contain conjunct $\mathsf{id}_2^s = 1$ for the RMR constraint. It also needs to express that the balance read for account 2 by transaction 2 should be the same as it was for the state with identifier id_2^s. The only state resulting from a commit between T_2 and T_4 is the one committed at T_3, where the balance for account 2 is $\mathit{ite}(\mathsf{id}^s \geq 1, 410, 400)$, so we have the following conjunct in NWC: $\mathit{ite}(\mathsf{id}^s \geq 1, 410, 400) = \mathsf{balance}_1$.

When conjoined with the other conjuncts $\mathsf{id}^s = 1$ (learned from checking transaction 2) and $\mathsf{id}_2^s = 1$ (the RMR constraint) to yield the isolation constraint for the first three transactions in the log, the result is *unsatisfiable*, indicating the observations came from an *incorrect* DBMS implementation that violates guarantees for SSI. While it is clear that we had to use knowledge of isolation guarantees to detect this bug, we *also* had to reason about semantics to determine if and when account 2 had its balance written to – i.e., we had to consider the possible values of $\mathsf{balance}_1$, since it occurs in our isolation constraint.

Returned results. To see how we use the returned results in DBMS responses to check observational correctness, let us modify our example slightly so that our set of transactions now excludes transaction 3. Behaviors of transactions 1 and 2 are as above. We now consider transaction 4.

The partially-symbolic semantics produces a partially-symbolic expected DBMS response r^s that represents the read balance of account 2 as expression $\mathsf{balance}_2$, defined below:

$$\mathsf{balance}_2 \equiv \mathit{ite}\big(\mathsf{id}_4^s \geq 3, \mathit{ite}(\mathsf{id}^s \geq 1, 410, 400), \mathit{ite}(\mathsf{id}_4^s \geq 1, 310, 300)\big)$$

for a fresh symbolic state id_4^s. There are three possible states id_4^s could identify: the initial one, the one with identifier 1, and the one with identifier 3. Expression $\mathsf{balance}_2$ captures the result for each possibility. The semantic correctness constraint ϕ must encode that the observed return result and the one computed using the symbolic semantics are the same, expressed by conjunct $420 = \mathsf{balance}_2$. As this is *unsatisfiable*, we conclude the DBMS implemented the transaction semantics incorrectly.

3 Observational Correctness

In this section, we formally define observational correctness.

3.1 States, Transactions, and Execution Traces

We first introduce several notions that we will use to define observational correctness.

Database state. A *database state* is a map $\ell \in \mathcal{L}$ from keys to values. Maps ℓ clearly allow us to model databases that are key-value stores, but we can also use them to model relational databases using key-value pairs in ℓ, by letting a key k be a pair of a table name $k.\mathbf{tbl}$ and a row id $k.\mathbf{id}$ and

letting values be rows for the particular row id in the table. Each row itself is a map from a column name to that column's value for the row.

As in related work [8], our row ids are the result of associating every row of the table with a unique id. A table is then a map from keys to values in which all keys k have the same table name $k.\mathbf{tbl}$ and all rows have the same type. A table in ℓ with name *tbl* is the subset ℓ of keys k with $k.\mathbf{tbl} = tbl$. For simplicity, we have assumed tables are never created nor deleted and any needed tables exist in the initial database state. To support creation/deletion, database states should instead be *pairs* of maps: one to describe whether the table exists and one that is ℓ as described. While in our implementation, we use this formulation of database states and model relational databases, for simplicity, we will treat database states as just maps ℓ.

System state. A *system state* $s = \langle \ell, r \rangle$ consists of database state ℓ and possibly-empty (denoted ε) *read result* r, which is also a map from keys to values. The read result r is either the observable result of a transaction or an intermediate result to be consumed in subsequent transaction operations.

Transaction description language. A *transaction description language* (TDL) is a programming language (e.g., SQL) used to specify a transaction's behavior; it is assumed to have an associated operational semantics, assumptions on which we describe in §3.2.

Transaction. A *transaction* tr is a pair of a unique transaction identifier and a set of statements $\mathsf{stmt}(\mathrm{tr}_i)$ in a given TDL. We use $\mathcal{T}$ to denote sets of transactions. Our definition of transaction differs from other work in that we do not know a priori the read and write set of the transaction, and instead must reason about the semantics of $\mathsf{stmt}(\mathrm{tr}_i)$.

Transition. A system state s_{i-1} can *transition* to s_i and produce output $o_i \in \mathcal{O}$ via a transaction $\mathrm{tr}_i \in \mathcal{T}$, denoted $s_{i-1} \xrightarrow{\mathrm{tr}_i : o_i} s_i$. An output o of a DBMS is a record $(r, c, \mathit{start}, \mathit{end})$, where read result $\mathsf{r}(o) = r$ is the observed result of a transaction as returned in a DBMS response, $\mathsf{committed}(o) = c$ is a Boolean indicating if the transaction committed successfully, $\mathsf{start}(o) = \mathit{start}$ is the transaction's start time if available or else is $\bot$, and $\mathsf{end}(o) = \mathit{end}$ is the transaction's end time or else is $\bot$. If $\mathsf{committed}(o_i)$, then state s_i is a *committed state*, and is in particular the state committed by tr_i.

Execution trace. An *execution trace* (also called a *trace*) e for a set of transactions $\{\mathrm{tr}_i\}_i$ is a sequence of transitions $s_0 \xrightarrow{\mathrm{tr}_1 : o_1} \cdots \xrightarrow{\mathrm{tr}_n : o_n} s_n$.

We use traces to describe the logged behavior of a (potentially buggy) DBMS implementation on a set of transactions, where system state s_i, reached via a transition via transaction tr_i, represents the DBMS state after executing transaction tr_i. The definition of trace allows for *arbitrary*, possibly semantically-incorrect traces, as traces may result from buggy runs of DBMS implementations. We define semantic correctness of a trace in §3.3.

3.2 Version-Aware TDL Semantics

Correctness of a trace is defined with respect to a given TDL semantics, which describes TDL operations on single database states. Under weaker isolation levels, transactions may be allowed read from from any of a set of *readable* database states, so we adapt the TDL semantics to allow for this, yielding a version-aware transaction description language (VA-TDL) semantics.

To reason about the semantic correctness in the presence of non-serializable isolation levels, we need to allow a transaction read from potentially *all* possible committed database states in the trace. In general, these need not be only committed database states but also intermediate ones that occur during the execution of transactions, including ones that abort. For simplicity, we consider allowing a transaction to read from only *committed* database states, allowing us to reason about behaviors for RC and stronger isolation levels.

Assumptions on TDL semantics. We assume the TDL semantics is an operational semantics with steps $(\mathit{stmts}, s) \hookrightarrow (\mathit{stmts}', s')$, where *stmts* is a sequence of TDL statements. An execution of

READ

$$\frac{is = \{k \mapsto i \mid k \in \mathrm{Dom}(d), i \in \mathrm{Dom}(d[k])\}^{(c1)} \quad \ell = \{k \mapsto \mathrm{readVal}(w, d, k, is) \mid k \in \mathrm{Dom}(d)\}^{(c2)} \quad (stmts, \langle \ell, r \rangle) \hookrightarrow (stmts', \langle \ell, r' \rangle)^{(c3)}}{d \vdash (stmts, \langle w, r \rangle) \rightsquigarrow (stmts', \langle w, r' \rangle)}$$

READ-FREE

$$\frac{\ell = \{k \mapsto \mathrm{dummy} \mid k \in \mathrm{Dom}(d) \setminus \mathrm{Dom}(w)\} \cup w^{(c4)} \quad (stmts, \langle \ell, r \rangle) \hookrightarrow (stmts', \langle \ell', \varepsilon \rangle)^{(c5)} \quad w' = \{k \mapsto \ell'[k] \mid k \in \mathrm{Dom}(\ell') \wedge \ell'[k] \neq \mathrm{dummy}\}^{(c6)}}{d \vdash (stmts, \langle w, r \rangle) \rightsquigarrow (stmts', \langle w', \varepsilon \rangle)}$$

Fig. 4. VA-TDL semantics rules.

the semantics completes when the sequence of statements is empty, denoted ϵ. We assume semantic steps can be divided into *read* steps, which do no writes (so $s.\ell = s'.\ell$) and, *read-free* steps, which do no reads and consume intermediate read results (so $s'.r = \varepsilon$).

State maps. As mentioned previously, our VA-TDL semantics must allow transactions to read nondeterministically from sets of database states. We use unique identifiers to identify readable states and model the choice in which state to read from as a choice over identifiers.

We represent a set $\{\ell_i\}_{i=1}^n$ of different versions of database states (identified by their indices) as *state maps* d. The state map d maps from keys and state identifiers i to values such that $\ell_i[k] = d[k][i]$. We use $D(\{\ell_i\}_{i=1}^n)$ to denote the state map that represents database states $\{\ell_i\}_{i=1}^n$.

Example 3.1. The initial database state and the one after transaction 1 in §2 are captured by the state map d, with relevant entries for account 1 for state identifiers 0 and 1 as follows:

$$d[\texttt{accounts}, 1][0] = \{\texttt{id} \mapsto 1, \texttt{balance} \mapsto 100\}$$
$$d[\texttt{accounts}, 1][1] = \{\texttt{id} \mapsto 1, \texttt{balance} \mapsto 90\}$$

VA-TDL semantics. Given a TDL semantics that satisfies our assumptions, we can derive a VA-TDL semantics with steps $d \vdash (stmts, \langle w, r \rangle) \rightsquigarrow (stmts', \langle w', r' \rangle)$, where the context state map d represents all readable database states for the transaction, *write map* w is a map from keys to values written to them so far by the transaction, and r is a read result.

The derivation of the VA-TDL semantics is shown in Fig. 4. The READ rule shows how to handle the case in which the corresponding next TDL step would perform a read. A state identifier $i \in \mathrm{Dom}(d[k])$ is nondeterministically chosen per key k in the database (c1) to indicate the database state from which it should be read. A database state ℓ is then constructed based on these identifiers using readVal (c2), which ensures that transactions read their own writes:

$$\mathrm{readVal}(w, d, k, is) \equiv \begin{cases} w[k] & \text{if } k \in \mathrm{Dom}(w) \\ d[k][is[k]] & \text{otherwise} \end{cases}$$

If the current transaction has not written to key k, then the read value is the one at the database state with identifier $is[k]$, but otherwise the read value is the latest one written by the transaction. The TDL semantics then updates the read result to yield r' (c3).

The READ-FREE rule handles the case in which the corresponding next TDL step would be read-free. It constructs a stand-in database state ℓ to be updated by the TDL semantics by mapping every key not in w to a distinguished dummy value not part of the TDL (note that these values will not be read) and every key k in w to $w[k]$ (c4). The original TDL semantics is run on ℓ and r (c5) to yield a new system state $\langle \ell', \varepsilon \rangle$. Writes in ℓ' are then added to w to get w' (c6); any key with a non-dummy value has been written by the transaction.

3.3 Semantic Correctness of Traces

As mentioned, the definition of a trace does not impose any semantic constraints on a transaction tr_i used to transition from state s_{i-1} to state s_i, since s_i may result from a buggy execution of tr_i. We use VA-TDL semantics to define what it means for a transition to be semantically correct.

Definition 3.1 (Semantic correctness of a transition)**.** A transition $s_{i-1} \xrightarrow{\mathrm{tr}_i:o_i} s_i$ in a trace is semantically correct iff according to VA-TDL semantics, the following two conditions hold, where $\rightsquigarrow^*$ is the reflexive transitive closure of $\rightsquigarrow$, and d represents the set $\{s_j\}_j$ of initial and committed database states in the trace with $0 \leq j < i$:

- **S1**: $d \vdash \left(\mathrm{stmt}(\mathrm{tr}_i), \langle \emptyset, \varepsilon \rangle\right) \rightsquigarrow^* \left(\epsilon, \langle w_i, r_i \rangle\right)$, and s_i is the same as s_{i-1} but with keys whose values are the same as in w_i and $s_i.r = r_i$
- **S2**: $s_i.r = \mathrm{r}(o_i)$

Condition **S1** captures that the semantics *terminates without error* when run on the transactions, and condition **S2** captures that the transaction's read result according to the semantics *matches the output read result* for the transition.

Definition 3.2 (Semantic correctness of a trace)**.** A trace e is semantically correct iff every transition in e is semantically correct.

Example 3.2. Consider the set of transactions $\{\mathrm{tr}_i\}_{i=1}^4$, which have corresponding outputs $\{o_i\}_{i=1}^4$ as observed in Fig. 1. The trace $s_0 \xrightarrow{\mathrm{tr}_1:o_1} \cdots \xrightarrow{\mathrm{tr}_4:o_4} s_4$ is incorrect for any system states s_1, s_2, s_3, s_4, where s_0 is the initial state where bank account 1 has balance 100 and account 2 has balance 300.

3.4 Isolation Correctness of Traces

When the appropriate isolation-related constraints hold for a trace, it is considered to be *isolation correct*. Similarly to the case of semantic correctness, there is nothing in the definition of a trace that imposes isolation-level constraints. We define isolation correctness of a trace and a given isolation level I in terms of isolation constraints C_I on traces, described in detail in §6, where these constraints are formulated in terms of read records produced by an instrumented version of the VA-TDL semantics for the transitions in the trace.

Definition 3.3 (Isolation correctness)**.** Trace e is isolation correct iff the isolation constraint for I holds for e, i.e., $C_I(e)$.

Our isolation correctness definition and constraints are based on a the approach by Crooks et al. [12], in which checking isolation guarantees boils down to constructing a trace that is consistent with observations and satisfies isolation constraints.

3.5 Observational Correctness

For a *black-box* DBMS, we cannot reason about all possible execution traces that a DBMS can produce to verify its correctness. We do not even have access to actual execution traces of the DBMS in general, as we do not have access to database states. Instead, we have access only to *observations*, observed outputs from running transactions on the DBMS.

We thus formulate a notion of *observational correctness* of a DBMS with respect to collected observations. To determine if it holds, we reason about the underlying execution traces the DBMS *could have produced* based on the observations. Observations are collected from an actual run of the DBMS, and are represented as a map *obs* from transactions tr to observed outputs $o \in O$ for tr.

Definition 3.4 (Observational DBMS correctness)**.** A DBMS that claims to provide isolation level I is *observationally correct* with respect to the observations *obs* gathered from running the transactions

in $\mathcal{T}$ iff there exists a trace e of transitions for $\mathcal{T}$ such that (**O1**) every transition $s_{i-1} \xrightarrow{\mathrm{tr}_i:o_i} s_i$ in e has $o_i = obs(\mathrm{tr}_i)$ *and* (**O2**) e is both semantically and isolation correct.

Note that in the trace e, that must exist for observational correctness, the transitions via transactions in $\mathcal{T}$ may appear in *any* order so long as e is consistent with observations (**O1**) and semantics and isolation constraints (**O2**). Also note that we can always construct a trace satisfying **O1** using arbitrary s_i and choosing $o_i = obs(\mathrm{tr}_i)$, but this trace may violate **O2**.

For convenience, we treat transactions in $\mathcal{T}$ as all having committed. In practice, we simply consider the subset of $\mathcal{T}$ that committed, as aborted transactions should not affect results of committed ones for isolation levels RC and stronger.

4 Conditions for Correctness

Now that we have defined observational correctness, let us consider necessary and sufficient conditions for demonstrating it with respect to given observations $obs : \mathcal{T} \to O$ for transactions $\mathcal{T}$. From the definition of observational correctness (Definition 3.4), we need to demonstrate the existence of a trace that meets two requirements: (**O1**) its transition labels are consistent with the observations *obs*, and (**O2**) it is both semantically and isolation correct.

Given a total ordering $\prec$ on $\mathcal{T}$ and a sequence of states $(s_i)_{i=0}^n$, we can generate a unique trace

$$s_0 \xrightarrow{\mathrm{tr}_1:obs(\mathrm{tr}_1)} s_1 \xrightarrow{tr_2:obs(\mathrm{tr}_2)} \cdots \xrightarrow{\mathrm{tr}_n:obs(\mathrm{tr}_n)} s_n$$

where $\mathrm{tr}_i \prec \mathrm{tr}_j$ for $i < j$, $\mathrm{tr}_i, \mathrm{tr}_j \in \mathcal{T}$. We refer to such an execution trace constructed from $(s_i)_{i=0}^n$, $\mathcal{T}$, $\prec$, *obs* as $\textsc{Tr}((s_i)_{i=0}^n, \mathcal{T}, \prec, obs)$. Note that such a trace always meets **O1** (i.e., transition labels are consistent with the observations *obs*) by construction. We can thus demonstrate the existence of a trace that meets the both requirements for observational correctness by finding $\prec$ and $(s_i)_{i=0}^n$ such that $\textsc{Tr}((s_i)_{i=0}^n, \mathcal{T}, \prec, obs)$ is semantically and isolation correct, therefore meeting **O2**.

4.1 Checking Approach

In this paper, we assume that for the observations *obs*, we are given an ordering $\prec$ and a set of possible initial database states $\mathcal{L}_0$. In practice, we can construct $\prec$ using commit timestamps returned by the DBMS, where $\mathrm{tr}_i \prec \mathrm{tr}_j$ if tr_i commits before tr_j. These timestamps are often available, though they may subject to clock drift. If we fail to show observational correctness for a given ordering, we can use information from the failure to construct a new one; however, we have often found the initial ordering to be sufficient in practice. We thus focus on addressing the remaining problem of showing observational correctness by finding the sequence of states $(s_i)_{i=0}^n$ (with $s_0 = \langle \ell_0, \varepsilon \rangle$, $\ell_0 \in \mathcal{L}_0$) such that $\textsc{Tr}\big((s_i)_{i=0}^n, \mathcal{T}, \prec, obs\big)$ is semantically and isolation correct.

In the sequel, we uniquely identify both these states and transitions with indices derived from their (would-be) position in the constructed trace. In particular, for the sequence $(\mathrm{tr}_i)_{i=1}^n$ of transactions in $\mathcal{T}$, where $\mathrm{tr}_i \prec \mathrm{tr}_j$ if $i < j$, we identify tr_i by its index i. Similarly, for the sequence of system states $(s_i)_{i=0}^n$, we identify s_i and its components by index i. Thus, in the constructed trace, the system or database state with index i results from a transition via the transaction with index i.

To verify observational correctness with respect to a set of transactions and their observations, we search for a sequence $(s_i)_{i=0}^n$ such that for trace $e = \textsc{Tr}\big((s_i)_{i=0}^n, \mathcal{T}, \prec, obs\big)$, (1) every transition in e is semantically correct, and (2) e is isolation correct. We reduce this to SMT solving by generating encodings of semantic (§5) and isolation (§6) correctness.

5 Encoding Semantic Correctness

We describe in §5.1 how to encode observational semantic correctness of a trace into SMT constraints in the style of bounded model checking (BMC) [11]. While this encoding is relatively straightforward,

it yields formulas ill-suited for SMT solving because of the need to do complex theory reasoning as well as the inclusion of irrelevant constraints on state components for proving observational correctness. We thus introduce a *partially-symbolic state representation* and an improved encoding technique based on this representation in §5.2.

5.1 BMC-Based Encoding

We first provide a brief overview of the necessary background on SMT-based model checking and then describe a straightforward encoding of observational semantic correctness.

Transition system. A transition system M is a tuple (Σ, I, R, L), where Σ is a set of states (note that these states are independent of the database and system states introduced earlier), $I \subseteq \Sigma$ is a set of initial states, $R \subseteq \Sigma \times \Sigma$ is a transition relation, and L is a mapping from states to a set of atomic predicates A that take state variables V as arguments. We denote by L_σ the set of atomic predicates for state $\sigma \in \Sigma$. For property p and state σ, if $\bigwedge L_\sigma(V) \Rightarrow p(V)$ holds, we call σ a p *state*.

Paths. A *path* π of a transition system M is a sequence of states $\sigma_0\sigma_1\sigma_2 \ldots (\sigma_i \in \Sigma)$ such that $\sigma_0 \in I$ and for each $R(\sigma_i, \sigma_{i+1})$ for each pair of sequential states σ_i, σ_{i+1} in the path. If all states in a path π are p states for a property p, we call π a p path.

Symbolic model checking. For a transition system containing only finite paths, we can prove that there exists a p path in the system by explicitly enumerating all paths π of M and checking if for any of them, whether the implication $\bigwedge L_\sigma(V) \Rightarrow p(V)$ holds for every state σ in the path. A more scalable approach is to use *symbolic model checking*, in which we implicitly represent sets of states using logical formulas and perform checking by constructing a logical formula that is satisfiable iff there exists a p path in the transition system. We consider encoding the set of initial states and transition relation using predicates in first-order logic, consider *unrolling* the symbolic transition relation to get all paths reachable in k steps from an initial state, and then finally add a conjunct requiring that p holds for all of states along the paths.

For state variables V, we let $Init(V)$ be a formula capturing all the states in I, i.e., $L_{\sigma_0}(V)$ iff $Init(V)$. For state variables V and their primed versions V', we also let the symbolic transition relation $\rho(V, V')$ be such that for any $(\sigma, \sigma') \in R$, $\rho(V, V')$ iff $L_\sigma(V)$ and $L_{\sigma'}(V')$ hold.

The k-BMC formula for the system is given below for $k+1$ versions of the state variables $\{V_i\}_{i=0}^{k}$:

$$Init(\sigma_0) \wedge \bigwedge_{i=0}^{k-1} \rho(V_i, V_{i+1}) \wedge \bigwedge_{i=0}^{k} p(V_i) \qquad (k\text{-BMC})$$

The first two conjuncts are a *k-unrolling* of the symbolic transition relation ρ, which is satisfiable iff there is an assignment of the variables $\bigcup_{i=0}^{k}\{V_i\}$ to constants such that for some path $\sigma_0 \ldots \sigma_k$ in M, $L_{\sigma_i}(V_i)$ holds for $i \in \{0, \ldots, k\}$. This assignment is called an *interpretation* $\mathcal{I}$, and we denote a FOL formula F being satisfiable with an interpretation $\mathcal{I}$ as $\mathcal{I} \vdash F$.

The final conjunct ensures that this formula is satisfiable with interpretation $\mathcal{I}$ iff there exists a path $\sigma_0 \ldots \sigma_k$ in M where for each $i \in \{0, \ldots, k\}$, $\bigwedge \mathcal{I}(V_i) \Leftrightarrow L_{\sigma_i}(V_i)$ *and* $p(\mathcal{I}(V_i))$ holds. In other words, the formula is satisfiable iff there is a p path in the original transition system. For completeness, we choose k to be equal to the largest length of any path π of M.

Encoding observational semantic correctness. To reduce the checking of observational semantic correctness to symbolic model checking, we need to encode the necessary parts of the VA-TDL semantics in the transition system M as well as the property p. We let the system states and transitions in traces be the states and transitions in the transition system M and define ρ according to the VA-TDL semantics so that condition **S1** holds for all trace transitions iff the unrolling of ρ is satisfiable. We define p so that condition **S2** holds for all trace transitions iff formula k-BMC is satisfiable for k chosen to be the length of the trace. (**S1** and **S2** are as defined in Definition 3.1).

We capture condition **S2** by defining p to be a predicate eq that constrains corresponding values in the i^{th} observed read result $\mathsf{r}(o_i)$ and in the trace's i^{th} read result $\mathsf{s_i}.r$ to be equal. For a key-value store, $\mathrm{eq}(\mathsf{r}(o_i), \mathsf{s_i}.r)$ is defined to be as follows:

$$\mathrm{Dom}(\mathsf{r}(o_i)) = \mathrm{Dom}(\mathsf{s_i}.r) \wedge \bigwedge_{k \in \mathrm{Dom}(\mathsf{s_i}.r)} \mathsf{r}(o_i)[k] = \mathsf{s_i}.r[k]$$

For the set of state identifiers ID in the trace, $p(V)$ is then $\bigwedge_{i \in ID} \mathrm{eq}(\mathsf{r}(o_i), \mathsf{s_i}.r)$

To preserve the types of the state variables in the SMT encoding, formulas are over a multi-sorted first-order theory that includes all needed datatypes for the full state representation. To encode partial maps using the theory of arrays, which only allows for the representation of total maps, we introduce, for each partial map state variable, a corresponding map ranging over Booleans, which we use to encode whether a particular element is in the map's domain.

5.2 Improved Encoding

In practice, SMT solvers are unable to solve the formulas yielded by the above encoding because of the need to do theory reasoning over arrays. We can avoid needing to do array theory reasoning by flattening arrays and then eliminating them altogether by introducing a state variable for each needed element of each array. This array-free encoding is possible in general so long database states are of a known, finite maximum size; however, even this encoding is unideal.

A drawback to the removal of arrays is that the number of constraints that are added to the SMT solver increases. While needed to fully encode the current database state, they may not be needed for proving observational correctness. For example, constraints on a value that never influences an observed DBMS response are irrelevant for proving observational correctness. We would like to simplify and reduce the number of constraints given to the SMT solver to improve solving times.

Rather than maintaining the entire state in the symbolic encoding, and leaving all state reasoning to the backend solver, we can simplify the encoding by maintaining the state *outside* the solver and generating constraints only involving relevant parts of the database state. We can also do concrete semantic reasoning when possible to avoid generating some constraints at all. In practice, this encoding approach leads to a large improvement in solving times (see §8).

5.3 Partially-Symbolic State Representation

To generate simpler SMT encodings, we rely on a fine-grained *partially*-symbolic state representation, in which primitive values are represented concretely as much as possible. We use a superscript s to indicate a possibly- (in the case of primitives) or partially- (in the case of composite data structures) symbolic representation. A possibly-symbolic primitive v^s is either a concrete value or an SMT expression of the appropriate sort.

System states and write maps. We represent system states and write maps in a partially-symbolic way by letting the value components of maps such as ℓ, r, and w be represented by possibly-symbolic guarded values $\langle g, val^s \rangle$. Boolean formula guard g gives the condition under which the map entry exists, and val^s represents the value itself. For maps whose values are already primitives, these val^s representations are either concrete primitives or SMT expressions. When values are composite data structures, we use a fine-grained representation where only primitives can be represented directly with SMT expressions; composite data structures are represented with data structures *containing* these SMT expressions. For example, for relational databases where the value components of the database state map ℓ are rows, the value components of ℓ^s are guarded maps from from column names to concrete primitives or symbolic expressions. For simplicity of exposition, outside of examples, we will assume that such value components are primitives.

State maps. Similarly to concrete database states (§3.2), we represent a set of partially-symbolic database states $\{\ell_i^s\}_{i=1}^n$ as a partially-symbolic state map d^s, which is such that such that $\ell_i^s[k] = d^s[k][i]$ for all database keys k. Each $d^s[k]$ maps from state identifiers to symbolic guarded values. To reduce the nondeterminism in the choice of which states we can read from, we use a sparse representation of maps $d^s[k]$, where state identifier id is only a key in $d^s[k]$ if transaction $\mathrm{tr}_{\mathrm{id}}$ wrote to key k. Partially-symbolic state maps d^s have only state identifiers for committed states.

Example 5.1. Assume that the transfer transactions for the example in §2 are instead *conditional* and only transfer money iff a condition θ holds, otherwise acting as no-ops. The database state after transaction 1 is then represented by $\langle d^s, \varepsilon\rangle$, which has the following entries for account 2:

$$d^s[\texttt{accounts}, 2] = \{0 \mapsto \{\texttt{id} \mapsto 2, \texttt{balance} \mapsto \langle \mathsf{true}, 300\rangle\}, 1 \mapsto \{\texttt{id} \mapsto 2, \texttt{balance} \mapsto \langle \theta, 310\rangle\}\}$$

5.4 Semantic Constraint Generation

We now present a method for generating a simpler encoding of the observational semantic correctness constraint in the form of a partially-symbolic version of the VA-TDL semantics. It relies on a partially-symbolic TDL semantics analogously to how the concrete VA-TDL semantics relies on the TDL semantics. We present this partially-symbolic TDL semantics first.

5.4.1 Partially-Symbolic TDL Semantics. We assume that for our TDL, we can construct a partially-symbolic semantics with steps of the form $(\mathit{stmts}, s^s \cdot C) \hookrightarrow (\mathit{stmts}', s^{s\prime} \cdot C')$, where $s^s, s^{s\prime}$ are partially-symbolic database states and constraints. At the end of the execution of a sequence of statements, the final constraint C' must hold in order for the sequence of statements to have had a semantically correct execution. This constraint C' should be equisatisfiable to a BMC formula capturing the behaviors of the statements.

We assume the partially-symbolic semantics is sound, precise, and meets the concrete execution condition with respect to the concrete TDL semantics, where these notions are defined below:

Definition 5.1 (Soundness and Precision of Partially-Symbolic TDL Semantics)**.** A partially-symbolic TDL semantics is *sound* (resp. *precise*) for a concrete TDL semantics iff there exists a step $(\mathit{stmts}, s) \hookrightarrow^* (\mathit{stmts}', s')$ in the concrete semantics only if (resp. if) there is a corresponding step in the partially-symbolic one $(\mathit{stmts}, s^s \cdot \mathsf{true}) \hookrightarrow^* (\mathit{stmts}', s^{s\prime} \cdot C)$, where there exists interpretation $\mathcal{I}$ such that $\mathcal{I} \vdash C$, and the replacement of all expressions in s^s and $s^{s\prime}$ with their interpretations in $\mathcal{I}$ respectively yields states s and s'.

Definition 5.2 (Concrete Execution Condition)**.** Let ℓ^s, r^s be partially-symbolic maps, and let $\ell^s = \ell_1 \cup \ell_2^{ss}$ and $r^s = r_1 \cup r_2^{ss}$ where the pairs ℓ_1, ℓ_2^{ss} and r_2, r_2^{ss} are of maps with disjoint domains, ℓ_1 and r_1 have values that are all concrete, and ℓ_2^{ss} and r_2^{ss} have values that are symbolic. If there exists a function **upd** such that for all concrete maps ℓ_2, r_2 with the same domains as ℓ_2^{ss} and r_2^{ss} respectively, $(\mathit{stmts}, \langle \ell_1 \cup \ell_2, r_1 \cup r_2\rangle) \hookrightarrow (\mathit{stmts}', \mathbf{upd}(\ell_1, r_1))$ is a step in the concrete TDL semantics, then $(\mathit{stmts}, \langle \ell^s, r^s\rangle \cdot C) \hookrightarrow (\mathit{stmts}', \mathbf{upd}(\ell_1, r_1) \cdot C)$ is a step in the partially-symbolic execution.

Soundness and precision requirements ensure that the partially-symbolic semantics captures exactly the behaviors of the concrete semantics. The concrete execution condition ensures that the partially-symbolic semantics performs concrete execution (and does not generate any constraints) whenever the next semantic step depends only on concrete values in the partially-symbolic state.

5.4.2 Partially-Symbolic VA-TDL Semantics. Given a partially-symbolic TDL semantics, we can construct a partially-symbolic VA-TDL semantics via the rule shown in Fig. 5. The rules have premises that correspond to those in Fig. 4, but where operations have been lifted to use the partially-symbolic representations or TDL as appropriate. We will explain premises (c1) and (c2) in more detail as they involve encoding nondeterministic reads symbolically.

READ

$$\frac{is^s = \mathit{ReadMap}(\mathrm{Dom}(d^s))^{(c1)} \quad \ell^s = \{k \mapsto \mathrm{readVal}(w^s, d^s, k, rm) \mid k \in \mathrm{Dom}(is^s)\}^{(c2)} \quad (stmts, \langle \ell^s, r^s\rangle \cdot C) \hookrightarrow (stmts', \langle \ell^s, r^{s\prime}\rangle \cdot C')^{(c3)}}{d^s \vdash (stmts, \langle w^s, r^s\rangle \cdot C) \rightsquigarrow (stmts', \langle w^s, r^{s\prime}\rangle \cdot C')}$$

READ-FREE

$$\frac{\ell^s = \{k \mapsto \mathrm{dummy} \mid k \in \mathrm{Dom}(d^s) \setminus \mathrm{Dom}(w^s)\} \cup w^{s\,(c4)} \quad (stmts, \langle \ell^s, r^s\rangle \cdot C) \hookrightarrow (stmts', \langle \ell^{s\prime}, \varepsilon^s\rangle \cdot C')^{(c5)} \quad w^{s\prime} = \{k \mapsto \ell^{s\prime}[k] \mid k \in \mathrm{Dom}(\ell^{s\prime}) \wedge \ell^{s\prime}[k] \neq \mathrm{dummy}\}^{(c6)}}{d^s \vdash (stmts, \langle w^s, r^s\rangle \cdot C) \rightsquigarrow (stmts', \langle w^{s\prime}, \varepsilon^s\rangle \cdot C')}$$

Fig. 5. Partially-symbolic VA-TDL semantics rules.

$\mathit{ReadMap}(\mathrm{Dom}(d^s))$ maps from every key k in $\mathrm{Dom}(d^s)$ to a *fresh symbolic identifier* for the state that the value for k is read from. Instead of choosing concrete state identifiers i indicating which states to read from for each key k, the READ rule chooses *symbolic* state identifiers instead (c1). These effectively capture *all* possible nondeterministic choices of is, yielding its symbolic version of is^s.

We lift readVal (defined in §3.2) to the symbolic setting (c2) and let $\mathrm{readVal}(w^s, d^s, k, is^s)$ be the read of a key k from d^s, with the values read per key specified by the map is^s unless the key was written by transaction $\mathrm{tr}_{\mathrm{id}}$, in which case that written value is read:

$$\mathrm{readVal}(w^s, d^s, k, is^s) \equiv \begin{cases} \mathsf{ITE}\big(\mathsf{true}, w^s[k], \mathsf{at}(d^s[k], is^s[k])\big) & \text{if } k \in \mathrm{Dom}(w) \\ \mathsf{at}(d^s[k], is^s[k]) & \text{otherwise} \end{cases}$$

where

$$\mathsf{ITE}(c, \langle g_1, v_1\rangle, \langle g_2, v_2\rangle) \equiv \begin{cases} \langle \mathsf{smp}(c \wedge g_1 \vee g_2), v_1\rangle & v_1 = v_2 \\ \langle g_2, v_2\rangle & v_1 \neq v_2, \mathsf{smp}(c \wedge g_1) = \mathsf{false} \\ \langle \mathsf{smp}(c \wedge g_1), v_1\rangle & v_1 \neq v_2, g_2 = \mathsf{false} \\ \langle \mathsf{smp}(g_1 \vee g_2), \mathit{ite}(\mathsf{smp}(c \wedge g_1), v_1, v_2)\rangle & \text{otherwise} \end{cases}$$

and smp syntactically simplifies formulas by concretely evaluating any equalities or inequalities, and then simplifies conjunctions and disjunctions based on any true or false conjuncts.

The at operator encodes the value in a map $d[k]$ at potentially symbolic state identifier id^s, producing a guarded value capturing all possible values components of $d[k]$. The guard constrains id^s to identify a state in which the value exists, and the value component encodes the value of $d[k]$:

$$\mathsf{at}(d^s[k], \mathrm{id}^s) \equiv \langle \min(\mathrm{Dom}(d^s[k])) \leq \mathrm{id}^s \wedge g, \mathit{val}\rangle$$

where $\langle g, \mathit{val}\rangle$ encodes $d^s[k]$ for id^s and is defined as $\mathrm{expr}(d^s[k], \mathrm{id}^s)$, where for $v = d^s[k]$,

$$\mathrm{expr}(v, \mathrm{id}^s) \equiv \begin{cases} v[\mathrm{id}] & \text{if } \mathrm{Dom}(v) = \{\mathrm{id}\} \\ \mathsf{ITE}\big(\mathrm{id}^s \geq i, v[i], \mathrm{expr}(v[i \mapsto \bot], \mathrm{id}^s)\big) & i = \max(\mathrm{Dom}(v)) \end{cases}$$

Example 5.2. Continuing from Example 5.1, after running the partially-symbolic VA-TDL semantics for transaction 2 in §2, the database state is then represented by $\langle d^s, \varepsilon\rangle$ where we have $d^s[\mathtt{accounts}, 2][3] = \{\mathtt{id} \mapsto 2, \mathtt{balance} \mapsto \langle 0 \leq \mathit{id}^s, 100 + \mathit{ite}(\mathrm{id}^s \geq 1 \wedge \theta, 310, 300)\rangle\}$.

Alg. 1 shows how can use the partially-symbolic VA-TDL semantics to generate semantic constraint $\bigwedge_{i=0}^{n} F_i$ for a given sequence of transactions, their observations, and an encoding of the initial states *Init*. At line 2, the initial constraint is $F_0 = \mathit{Init}$. (There is no read result for the initial state, so there is no need to generate a constraint for p.) The initial symbolic system state

Algorithm 1 Encoding semantic constraints

1: **procedure** EncodeSem($\{\mathrm{tr}_i\}_{i=1}^n$, *obs*, $\prec$, ℓ_0^s, *Init*)
2: $F_0, w_0^s, r_0^s, d_0^s \leftarrow \mathit{Init}, \emptyset, \varepsilon, \{k \mapsto \{0 \mapsto \ell_0^s[k]\} \mid k \in \mathrm{Dom}(\ell_0^s)\}$
3: **for** each tr_i in order of $\prec$
4: **if** $d_{i-1}^s \vdash (\mathrm{stmt}(\mathrm{tr}_i), \langle \emptyset, \varepsilon \rangle \cdot F_{i-1}) \rightsquigarrow^* (\epsilon, \langle w_i^s, r_i^s \rangle \cdot C_i)$
5: $F_i \leftarrow C_i \wedge \mathrm{eq}(\mathrm{r}(\mathit{obs}(\mathrm{tr}_i)), r_i^s)$
6: $d_i^s \leftarrow d_{i-1}^s$ but updated with writes in w_i
7: **else return** false
8: **return** $\bigwedge_{i=0}^n F_i$

has empty read result ε, and the initial symbolic state map contains initial symbolic system state ℓ_0^s, which can be defined to correspond to an arbitrary system state in a set of such initial states $\mathcal{L}_0$. Line 4 generates constraint F_i for $i > 0$ by applying the partially-symbolic VA-TDL semantics incrementally to the statements of transaction tr_i, the partially-symbolic database state generated so far, and the previous constraint F_{i-1}. If we can run the partially-symbolic VA-TDL semantics to completion, then we construct F_i (line 5) as the conjunction of the constraint C_i and the encoding of p for tr_i. We also construct the next state map (line 6). On the other hand, if the semantics gets stuck, there is no semantically correct behavior for the arguments to Alg. 1 (line 7).

6 Isolation Constraints

We now consider isolation constraints over traces and their symbolic encodings. To reason about isolation guarantees, we need to know which database states were read by each transaction for each key. We do not have this information a priori as the choice of value to read is nondeterministic and depends on the semantics of the transactions' statements; however, it is straightforward to collect this information while executing the VA-TDL semantics. We accordingly instrument VA-TDL semantics to collect this information in the form of read records m, which map from each key in database state ℓ to sets of identifiers for *all* states in which the key had its value read for that key. We also instrument our partially-symbolic VA-TDL semantics so that it collects *symbolic* read records m^s, which map from keys to sets of *symbolic* state identifiers.

To encode isolation constraints, we first adapt Crooks et al.'s commit tests to be in terms of concrete read records, at which point the symbolic encoding is straightforward. We overload the notation C_I and define these concrete isolation constraints $C_I(e)$ as being equal to corresponding constraints $C_I(e, (m_i)_i)$ over not only the execution but reads records $(m_i)_i$ collected from running the instrumented VA-TDL semantics for the transactions in e. We can then symbolically encode these isolation constraints and conjoin them with the semantic correctness formula ϕ for the trace to yield a formula that is satisfiable iff there is an observationally correct execution.

6.1 Instrumented VA-TDL Semantics

As the instrumentation of the semantics is straightforward, here we just define requirements on the reads records returned by the instrumented semantics.

For a Read step in the VA-TDL semantics (Fig. 4) with conclusion $d \vdash (\mathit{stmts}, \langle w, r \rangle \cdot C) \rightsquigarrow (\mathit{stmts}', \langle w, r' \rangle)$, a key k is *read* if it is part of the read result, i.e., in $\mathrm{Dom}(r')$. If the map *is* is generated as part of the Read step's premises (c1), then the *state identifier read for the key k* is id if the transaction reads its own write on k, or else it is $\mathit{is}[k]$.

Read Most Recent $\mathsf{RMR}(i, m)$:$\forall \mathsf{id} \in \mathsf{ids}(m).\mathsf{id} = i$
Read One Snapshot $\mathsf{ROS}(m)$: $\forall \mathsf{id}_i, \mathsf{id}_j \in \mathsf{ids}(m).\mathsf{id}_i = \mathsf{id}_j$
Read One Value $\mathsf{ROV}(m)$: $k \in \mathrm{Dom}(m) \implies \forall \mathsf{id}_i, \mathsf{id}_j \in m[k].\mathsf{id}_i = \mathsf{id}_j$
No Future Reads $\mathsf{NFR}(i, m)$: $\forall \mathsf{id} \in \mathsf{ids}(m), \mathsf{id} \leq i$
No Write Conflict $\mathsf{NWC}((\ell_i)_i, i, m) : \forall k \in \mathrm{Dom}(m), \mathsf{id} \in m[k]\ .\ \ell_i[k] \neq \ell_{\mathsf{id}}[k] \implies \ell_i[k] = \ell_{i-1}[k]$

Fig. 6. Conditions for isolation constraints

Read Most Recent $\Psi_{\mathsf{RMR}}(i, m^s)$: $\bigwedge_{\mathsf{id}^s \in \mathsf{ids}(m^s)} \mathsf{id}^s = i$
Read One Snapshot $\Psi_{\mathsf{ROS}}(m^s)$: $\bigwedge_{\mathsf{id}^s_i, \mathsf{id}^s_j \in \mathsf{ids}(m^s)} \mathsf{id}^s_i = \mathsf{id}^s_j$
Read One Value $\Psi_{\mathsf{ROV}}(m^s)$: $\bigwedge_{k \in \mathrm{Dom}(m^s)} \bigwedge_{\mathsf{id}^s_i, \mathsf{id}^s_j \in m[k]} \mathsf{id}^s_i = \mathsf{id}^s_j$
No Future Reads $\Psi_{\mathsf{NFR}}(i, m^s)$: $\bigwedge_{\mathsf{id}^s \in \mathsf{ids}(m^s)} \mathsf{id}^s \leq i$
No Write Conflict $\Psi_{\mathsf{NWC}}(d^s, w^s, i, m^s)$, where $src = \mathsf{at}(d^s[k], \mathsf{id}^s)$:

$$\bigwedge_{k \in \mathrm{Dom}(w^s)} \bigwedge_{\mathsf{id}^s \in m^s[k]} \neg \mathsf{eq}(w^s[k], src) \implies \mathsf{eq}(src, d^s[k][i-1])$$

Fig. 7. Encodings of conditions for isolation constraints

Definition 6.1 (Read Records)**.** For a complete run of the VA-TDL semantics, a *read record* m is a map whose domain is exactly the set of keys that were read by some Read step in the run. Furthermore, $\mathsf{id} \in m[k]$ iff id was a state identifier read for key k for some Read step in the run.

We analogously instrument the partially-symbolic VA-TDL semantics to yield a corresponding partially-symbolic read records m^s.

6.2 Isolation Constraints on Execution Traces

Our isolation constraint definitions adapt the commit tests from Crooks et al. to work over pairs of a trace and a sequence of corresponding concrete read records $(e, (m_i)_i)$ instead of whole states and known read values. Constraints C_I for isolation levels are defined in terms of the conditions in Fig. 6, where $\mathsf{ids}(m)$ is the set of *all* the state identifiers kept track of in m, i.e., $\mathsf{ids}(m) = \bigcup \mathit{Range}(m)$.

We use the phrase "i for which there is transaction in e" to refer to any i such that there is a transition in trace e taken via transaction tr_i. For convenience, we let $\mathsf{start}(\mathrm{tr}_i) = \mathsf{start}(\mathit{obs}(\mathrm{tr}_i))$ and $\mathsf{end}(\mathrm{tr}_i) = \mathsf{end}(\mathit{obs}(\mathrm{tr}_i))$. For strict isolation levels (SS, SSI), we assume access to commit times of transactions ($\mathsf{end}(\mathrm{tr}_i) \neq \bot$) and that $\prec$ is such that $\mathrm{tr}_i \prec \mathrm{tr}_j$ if $\mathsf{end}(o_i) < \mathsf{end}(o_j)$, since this is necessary for the level's time-related guarantees to be met. For SS, we also assume access to start times ($\mathsf{start}(\mathrm{tr}_i) \neq \bot$) and that they always precede corresponding commit times $\mathsf{end}(\mathrm{tr}_i)$.
Serializability $C_S\big(e, (m_i)_i\big)$: for all i for which there is a transition in e, we need that $\mathsf{RMR}(i-1, m_i)$, so that the values observed by each transaction tr_i are consistent with those that would have been observed in a sequential execution of transactions in the order that they occur in e.
Strict Serializability $C_{SS}\big(e, (m_i)_i\big)$: for all i for which there is a transaction in e, we need that $\mathsf{RMR}(i-1, m_i)$ and that transactions happen sequentially according to their start and end times, a property which we will refer to as $\mathit{seq}\big((\mathrm{tr}_i)_{i=1}^n\big)$.

Snapshot Isolation $C_{SI}(e, (m_i)_i)$: for all i for which there is a transaction in e, we need that $\mathsf{ROS}(m_i)$, $\mathsf{NFR}(i, m_i)$, and $\mathsf{NWC}(s_i.\ell, \mathrm{tr}_i, m_i)$. ROS ensures that all reads by a transaction tr_i come from the same snapshot, NFR prevents reads on states come after the commit of the transaction, and NWC ensures that for any key written by transaction tr_i, it has not been (observably) written by any intervening transaction tr_j that committed after the snapshot.
Strict Snapshot Isolation $C_{SSI}(e, (m_i)_i)$: for all i for which there is a transaction in e, we need that $\mathsf{RMR}(j, m_i)$ and $\mathsf{NWC}(d_i, i, m_i)$ for tr_j with the latest end time $\mathrm{end}(\mathrm{tr}_j)$ before $\mathrm{start}(\mathrm{tr}_i)$. RMR guarantees all reads by transaction tr_i come from the snapshot taken at the start of the transaction.
Read Committed $C_{RC}(e, (m_i)_i)$: for all i for which there is a transaction in e, we need $\mathsf{NFR}(i, m_i)$.

6.3 Symbolic Encoding of Isolation Constraints

It is straightforward to encode the isolation constraints above as formulas over the variables in partially-symbolic read records m^s. Fig. 7 shows symbolic encodings of conditions in Fig. 6. Formula $\psi_I((d_i^s)_i, (w_i^s)_i, (m_i^s)_i)$ encoding a constraint C_I is constructed by replacing each condition *cond* in C_I's definition with its symbolic encoding Ψ_{cond} and conjoining the constraints as appropriate.

Theorem 6.1. For given $\mathcal{T}$, *obs*, $\prec$, and ℓ_0^s, if ϕ is the formula returned by running Alg. 1 on them and $(d_i^s)_i$, $(w_i^s)_i$, and $(m_i^s)_i$ are the database states, write maps, and read records produced by running the partially-symbolic semantics as part of Alg. 1, then $\phi \wedge \psi_I((d_i^s)_i, (w_i^s)_i, (m_i^s)_i)$ is satisfiable iff there exists an observationally correct execution of $\mathcal{T}$ for isolation I starting from some system state in the set of system states represented by ℓ_0^s.

6.4 Isolation-Level-Aware Reads

So far, we have decoupled the generation of semantic constraints from the isolation level; however, for isolation levels like strict serializability, serializability, and SSI, a transaction can only read from either the latest version (for the serializability levels) or the version associated with its start time (for SSI). For these levels, rather than allowing reads from *all* possible previous state identifiers, we can modify the partially-symbolic VA-TDL semantics to permit only reads from states that are allowed under the isolation level and thereby further simplify the constraints given to the SMT solver. We modify the generation of the ℓ^s in (c2) of the Read rule by using a version of the at operation that captures only reads from the state the current transaction can read from: instead of considering the entire domain of $d^s[k]$, we consider a reduced domain consisting of only the state identifier for right before the current transaction's commit time (for strict serializable and serializable), or the state identifier for the transaction's start time (for SSI).

Example 6.1. Using isolation-level-aware reads, rather than the new database state entry shown in Example 5.2, we would instead have $d^s[\texttt{accounts}, 2][3][\texttt{balance}] = 100 + ite(\theta, 310, 300)$. Furthermore, without conditional transfers (or equivalently, if $\theta = \texttt{true}$) the guarded value would instead be $\langle \texttt{true}, 410 \rangle$, since we could concretely evaluate the addition in the TDL semantics.

7 Incremental Checking

Rather than directly checking the monolithic SMT formula constructed by conjoining semantic and isolation constraints, we instead use an *incremental* approach, where we check prefixes of the sequence of transactions resulting from ordering $\mathcal{T}$ by $\prec$. When an error is detected, this approach provides the smallest prefix of the sequence that is not observationally correct, helping localize errors. It also increases opportunities to apply heuristics like H_4 (§7.1).

To perform incremental checking, we simply construct and issue an SMT query IC_i, which is defined as $IC_i \equiv \bigwedge_{j=0}^{j} F_j \wedge \psi_I((d_j^s)_{j=0}^i, (w_j^s)_{j=0}^i, (m_j^s)_{j=0}^i)$ at each step i, where F_i is as in Alg. 1, and $(d_j^s)_{j=0}^i$, $(w_j^s)_{j=0}^i$ $(m_j^s)_{j=0}^i$ respectively give the partially-symbolic database states, write maps, and

read records up to the i^{th} transaction. This formula captures the semantic and isolation constraints of the trace up to the i^{th} transaction. To avoid redundant work from repeated conjuncts, we use incremental SMT solving, adding only new conjuncts to the solver at each iteration. After performing checking for the full trace, we can recover concrete system states by finding an interpretation $\mathcal{I}$ for the formula extracting the interpretation for each state variable of each state s_i.

Theorem 7.1. For a set of n transactions $\mathcal{T}$ ordered by $\prec$, if $\bigwedge_i^n IC_i$ is satisfiable with interpretation $\mathcal{I}$, then the sequence of states $(s_i)_{i=0}^n$ recovered from $\mathcal{I}$ always produces a correct trace $\textsc{Tr}((s_i)_{i=0}^n, \mathcal{T}, \prec, obs)$. Furthermore, if there exists a correct trace, then $\bigwedge_i^n IC_i$ is satisfiable.

7.1 Heuristics

We now consider strategies for improving performance. Note that all the heuristics either preserve soundness or are applied in a way that preserves soundness.

Indexing (H_1): Inspired by traditional database indexing, for relational databases, we can use a simple indexing scheme for equality predicates with a constant on one side to avoid full table scans on evaluating `WHERE` clause predicates. For each table, we maintain a map from the constant to the set of rows that satisfy the predicate and use this to avoid generating symbolic constraints checking predicate satisfaction as well as expressions for rows that do not satisfy the predicate.

Sliding window (H_2): For weaker isolation levels, because of the large amount of nondeterminism in possible read states, it can be more efficient to check *stronger* properties that restrict the states from which reads can happen. If the stronger check is passed, then the weaker check for the isolation level will also pass. This observation inspired a heuristic that uses a version of the at operation that captures only reads from the most recent k committed states for some parameter k. If a satisfiability check fails, we backtrack to consider reads from the previous k committed states.

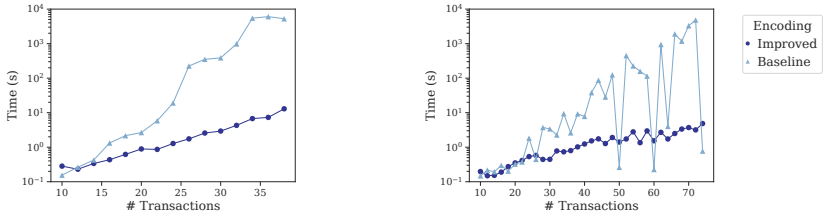
Concretization (H_3): To improve scalability, we *concretize* expressions in symbolic database state d_i^s before using it to construct the next F_i and IC_i. Symbolic expression *expr* can be concretized for SMT formula F as follows: given $\mathcal{I} \vdash F$, if $F \wedge \neg expr = \mathcal{I}(expr)$ is unsatisfiable, replace *expr* with $\mathcal{I}(expr)$. We use a recursive technique to concretize the maximum number of expressions: we initially try to concretize all expressions in the set E of symbolic state variable expressions. If E is empty, we return. Otherwise, we issue query $F \wedge \neg \bigwedge_{expr \in E} expr = \mathcal{I}(expr)$. If the query is unsatisfiable, we concretize all expressions with their interpretations in $\mathcal{I}$. Otherwise, we divide E into two (arbitrary) halves and recursively try to concretize the halves.

Stronger isolation levels (H_4): As stronger isolation levels allow for less nondeterminism in read states, they can be more efficient to check. To take advantage of this, when performing a step of incremental checking for isolation level I, we first use a suitable stronger isolation level than I (if one exists) to construct a stronger IC_i. If the satisfiability check fails at any point, we backtrack to the last i for which IC_i was constructed using a stronger isolation level than I, construct a new IC_i using level I, and resume checking from that point. In the worst case, we have to backtrack for each i, but in practice, this helps scale checking for weak isolation levels like RC.

8 Evaluation and Case Study

We implemented a tool Troubadour, which uses the Z3 SMT solver [13] to check observational correctness of logs of transactions in a fragment of SQL. It supports `SELECT`s, `UPDATE`s, `INSERT`s, `DELETE`s, `JOIN`s, predicates and `WHERE` clauses, table creation/deletion, stored procedure definitions and calls, cursor creation and usage, `WHILE`- and `FOR`-loops, conditional statements, `ORDER BY`, and aggregate functions `SUM`, `MAX`, and `MIN`. We used it to answer the following research questions:

- **(RQ1)** To what extent does our improved encoding help?
- **(RQ2)** How does our technique compare with others in terms of classes of bugs considered?



(a) Buggy (b) Correct

Fig. 8. Effect of different encoding approaches on solving times. The Y-axis is log scaled.

(RQ3) How does our tool scale on a variety of workloads?
(RQ4) What is the impact of our heuristics?

We ran experiments for RQ1 and RQ2 on a Macbook Pro with an M1 processor and 16GB RAM and ran experiments for RQ3 and RQ4 on a cluster of AMD EPYC™ 7763 64-Core CPUs.

Benchmarks. For RQ1, we considered a modified version of the example from §2.

For RQ2, we considered examples for real semantic bugs in DBMSs, the example in §2, and synthetic randomly-generated correct logs. For the remainder of the evaluation, we used a set of comprehensive benchmarks commonly used for DBMSs and studied in prior work [34, 40].

We considered the bank-server benchmark, which is similar to the example in §2, but in addition to *balance-checks*, it includes four transactions to manage bank account balances: `Deposit`, `Withdraw`, `Transfer`, and `WithdrawOverdraft`. The benchmark is complex enough to manifest interesting anomalies and has no pre-defined workload, so we adjusted it to explore the scalability of Troubadour on different workloads. Next, we considered Twitter [1] and RUBiS [2], which simulate real application workloads like that of a social media platform and an eBay-like bidding system, respectively. These benchmarks contain more complicated SQL logic compared to the bank-server benchmark and have pre-defined workloads used in prior work [34, 40]. The workload for Twitter is `NewTweet` (20%), `Follow` (40%), `Timeline` (10%), `ShowFollow` (10%), and `ShowTweets` (20%). The workload for RUBiS is `RegisterUser` (10%), `RegisterItem` (15%), `StoreBuyNow` (15%), `StoreComment` (20%), and `RandomBrowse` (40%). Finally, we considered TPC-C [35], a standard online transaction processing benchmark also considered in prior work [34, 40]. It specifies a workload distribution of five transactions simulating a wholesale supplier: `NewOrder` (45%), `Payment` (43%), `Delivery` (4%), `OrderStatus` (4%), and `StockLevel` (4%). These transactions are the most complex and use constructs such as cursors, `JOIN`s, and `ORDER BY`.

8.1 RQ1: Impact of the Improved Encoding

Fig. 8 shows the reduction in checking time using the improved encoding strategy when compared to what results from completely removing arrays by introducing state variables per array element (Baseline). We do not consider the fully-naive encoding from §5.1 here because Z3 reports `unknown` for even just *Init* of the direct BMC encoding. Both strategies encode observational correctness under the RC isolation level for traces of transactions that start from the same initial state of database in the example in §2, perform $k-1$ transfer transactions of 10 from account 1 to 2, and then finally read the balance of account 2. We compare the performance of the two encodings for $k = 10$ and for values of k achieved by incremented it by 2 until either checking for the baseline encoding exceeded 60 minutes once (for the buggy case) or exceeded 40 minutes for three consecutive k values (for the correct case). For Fig. 8a, the final read-out value is observed to be 66, indicating a semantic bug, and for Fig. 8b, the final read-out value is 400. Because proving observational correctness for a set of transactions' observations involves finding *any* satisfying assignment, there

is more variability in solver performance for the correct example. Solver heuristics lead to faster verification for certain values of k.

Summary. *Our improved encoding leads to a significant reduction in checking times for most example sizes and results in less variability in the solving times. While for correct examples, the baseline sometimes outperforms the improved encoding, the improved encoding consistently leads to solving times under two seconds, even where solving times exceed 40 minutes for the baseline.*

8.2 RQ2: Classes of Bugs Considered

Semantic. Troubadour can verify that a trace does not demonstrate any of the classes of bugs detected by DBMS fuzzer SQLancer for the fragment of SQL that Troubadour supports as well as detect the presence of such bugs. We used Troubadour to check correctness of queries by generating a transaction for the query and a correct observation for the transaction. We then ran Troubadour on the transaction and observation. It showed that all 11 of the considered examples were observationally correct in under a second total. We also ran the tool on the query and the original incorrect response observed by SQLancer, and Troubadour reported errors for all the examples in under a second total as well, demonstrating its ability to detect semantic bugs. We considered benchmarks based on MySQL queries 2, 3, 9, 15, 16, 18, and 19 and TiDB queries 1, 3, 10, and 13 generated by SQLancer[1], which detected real bugs in MySQL and TiDB DBMS implementations. Other queries used features (e.g., views) not currently supported by Troubadour.

Isolation and data anomalies. Troubadour can also be used to ensure the absence of observable isolation level violations and detect anomalies as isolation checking tools can. Many tools are specialized for finding anomalies for specific isolation levels. For example, Cobra supports only anomalies for the serializable isolation level [34], and Viper and PolySI support only anomalies for SI [21, 40]. Meanwhile, Troubadour and Elle [6] can detect subsets of anomalies that correspond to different isolation levels, including those handled by more specialized checkers. Troubadour, similarly to Elle, can detect when there has been an anomaly that violates isolation guarantees.

To demonstrate this, we generated logs of 2-4 transactions demonstrating common anomalies: dirty update [6], duplicate write [6], dirty write [5], aborted read [5], garbage read [6], intermediate read [5], and circular information flow [5] (permitted by none of the isolation levels we consider here); phantom read [37], lost update [17], read skew [37], and internal inconsistency [6] (permitted by RC); and write skew [37] (permitted by SI and RC). When verifying observational correctness for a log containing an anomaly not allowed by the specified isolation level (e.g., a log containing a dirty write for isolation level RC), Troubadour reported that the log demonstrated a bug in under half a second for each log. When run on a log containing an anomaly *allowed* by the provided isolation level (e.g., a log containing a write skew for isolation level SI), it verified observational correctness in under half a second for each log.

Observational correctness. While Troubadour can detect semantics- or isolation-related bugs in logs separately, its purpose is primarily to *verify* the absence of observable semantic or isolation correctness errors with respect to logged transactions. Testing tools like SQLancer and TxCheck can detect certain semantic bugs but clearly cannot perform verification of observational correctness [22, 31]. These tools further cannot catch bugs that require reasoning about isolation levels, such as the bug for the example in §2.

Techniques that reason only about isolation levels similarly cannot show the absence of semantic bugs and may also miss errors because they lack information about which transactions write which values. Tools like Elle, Cobra, Viper, and PolySI can only be applied in settings where transactions' reads and writes are all known, since they construct dependency graphs from reads. They would

[1]Found here: https://www.manuelrigger.at/dbms-bugs/

not, for example, be applicable for reasoning about the transactions in Fig. 1 with only the given observations, since they do not record the read and written values of transactions. Furthermore, even if such information were available from instrumentation, the tools do not check the correctness of such information. If an incorrect implementation or instrumentation of `UPDATE` statements led the only observed write to Transaction 3 to be to account 1's balance, then isolation checking tools would not detect any errors as having occurred within the first three transactions in Fig. 1.

Even when in cases where all transactions' reads and write are known, isolation checkers, while faster than Troubadour (which *always* checks semantic constraints and can detect *all* bugs), may not be as useful for checking arbitrary logs for correctness because of their incompleteness as bug-finders. To investigate, we compared Troubadour against Elle. on logs of key-value store transactions. We generated 100 correct synthetic logs of 50 transactions for isolation level RC. Each transaction is either a read of a single random value or a write of a single random value to a random key. There were 10 possible keys and 100 possible values. For Troubadour, we modeled the key-value store using a relational database table.

As expected, Elle is faster, taking an average of 2.9 seconds per log where Troubadour takes an average of 9.3 seconds. On the other hand, Elle reported only 47 logs correct, reporting "unknown" for the rest. Like other checkers that construct dependency graphs, Elle's anomaly detection is complete only when written values are unique, which is not the case for arbitrary logs. Troubadour was able to prove all 100 logs correct as it does not require any up-front inference of dependencies.

Summary. *Our approach can detect the same semantic and isolation bugs as prior work as well as show that there is no observable violation of these errors in a log of transactions. Furthermore, it can detect and show the absence of observable bugs involving both semantic and isolation-level guarantees.*

8.3 RQ3: Scaling Across Workloads

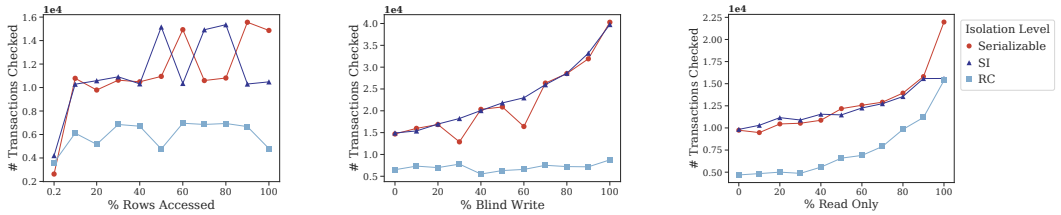


Fig. 9. The effect of varying bank-server workloads on performance. Y-axis labels in scientific notation.

Experimental setup. We consider workloads that vary in the level of transaction concurrency. We expect more concurrency will lead to longer checking times, as the SMT solver must explore a larger search space. To explore this question, we first conducted a controlled study using three different configurations of the bank-server benchmark with 1000 accounts. The default configuration has 100% rows accessed, 0% blind writes, and 20% read-only transactions. Then we ran our tool on the Twitter, RUBiS, and TPC-C benchmarks to explore benchmarks that have diverse pre-defined workloads and SQL logic. We configured Twitter with 1000 users, RUBiS with 1000 users and 4000 items, and TPC-C with 1 warehouse, 2 districts, 3 customers per district, and 1 item. We generated logs of transactions, responses, and commit times using a local instance of PostgreSQL 14 with two concurrent users executing 50,000 random transactions each. PostgreSQL exposes commit timestamps via the `pg_xact_commit_timestamp()` system function. We ordered transactions by commit times, breaking ties arbitrarily. The set and order of transactions for each user was the same across the different PostgreSQL isolation levels (Serializable, SI, RC).

By default, Troubadour has all heuristics enabled and checks each log against the isolation level used to generate it. Each benchmark was allotted 16GB of Java heap space and run with a 5-minute timeout. We measured the number of transactions checked within the timeout under varying workloads.

Rows accessed. The first configuration varies the percentage of rows in the table that users are allowed to access in their transactions. As this percentage increases, the number of transactions per key decreases, resulting in less nondeterminism in the values for a particular row. We thus expect that each incremental check should be faster at higher percentages and that the checker should be able to process more transactions. Fig. 9a shows that this is the case: we observe a significant increase after 0.2% of the rows accessed, where performance remains consistently better than at 0.2% thereafter, with variation in performance resulting from the variation in solving time overheads from heuristic H_3.

Blind write. A *blind write* is a write that is performed without being inferrable from the transaction read result. Non-blind writes can resolve some of the nondeterminism in the symbolic state and help the checker concretize but may also slow checking down if there are more read results to check. Fig. 9b shows that the latter is the case for Serializable and SI, where the cost of checking additional read results outweighs the benefit from concretization, but for RC, concretization helps enough that performance is largely unaffected.

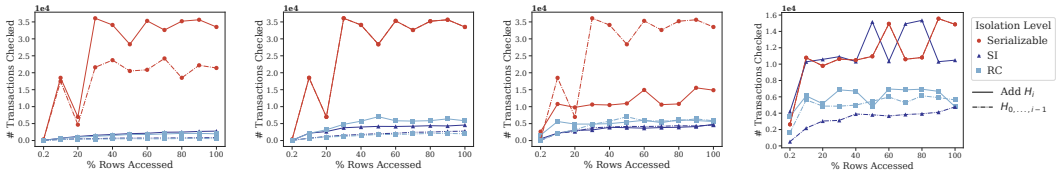
Read only. The final configuration varies the percentage of read-only transactions. These transactions only read the balance of an account. We expect these to be easier to check as this percentage increases because fewer updates lead to fewer possible values that can be read. Indeed, Fig. 9c shows that this is the case: Troubadour can check exponentially more transactions as the percentage of read-only transactions increases linearly. When all transactions are read-only, Troubadour can check on the order of 10^4 transactions within the timeout for all isolation levels. This scalability is promising given what is achieved by isolation-only checkers. For example, the Viper checker for SI was shown to check ten thousand transactions in roughly 7.3 minutes [40], where 50% of the transactions are read-only. With 50% read-only transactions, Troubadour can check 15% more transactions in five minutes. We note that this is not an apples-to-apples comparison for several reasons, including that (1) our tool was run on more powerful hardware than Viper, and (2) Viper, like other isolation checkers, does not handle (and consequently was not evaluated on) conditional writes, which are present in our workloads and lead to growing read nondeterminism.

Benchmarks with pre-defined workloads. We considered three other standard benchmarks that, unlike bank-server, have pre-defined workloads used in prior work [34, 40]. These benchmarks use a richer subset of SQL and have a high amount of read nondeterminism, increasing reasoning complexity and impacting Troubadour's runtime. The number of rows returned by transactions in these benchmarks is also significantly higher, increasing the time required to verify condition **S2**. For example, bank-server transactions return at most two rows while the `RandomBrowse` transaction, which makes up 40% of the RUBiS workload, returns twenty. Table 1 shows the number of transactions Troubadour checked within five minutes for each benchmark and isolation level. We observe the same expected performance trend across isolation levels as for bank-server in Fig. 9.

Summary. *Troubadour can check observational correctness for thousands of transactions within a 5-minute timeout across different workloads for the bank-server benchmark. Read nondeterminism is the greatest factor in performance. When run on more complex benchmarks, Troubadour can still check hundreds of transactions within the timeout.*

Table 1. Number of transactions TROUBADOUR can check within 5 minutes for more complex benchmarks.

Benchmark	Serializable	SI	RC
Twitter [1]	2860 transactions	2810 transactions	496 transactions
RUBiS [2]	451 transactions	397 transactions	311 transactions
TPCC [35]	445 transactions	394 transcations	121 transactions



(a) H_1: Indexing (b) H_2: Window = 4 (c) H_3: Concretization (d) H_4: Try Stronger

Fig. 10. The effect of incrementally adding heuristics on performance. Y-axis labels in scientific notation.

8.4 RQ4: Impact of Heuristics

Experimental setup. We progressively add each heuristic and observe the effect for varying percentages of rows accessed. Otherwise, the setup is as for the bank-server part of RQ3. Each plot in Fig. 10 shows the results after adding the heuristic with a solid line and results with only the prior heuristics with a dashed line. Enabling H_i is beneficial for an isolation level if its solid line is above its dashed line. H_2 and H_4 do not apply to Serializable, so both lines overlap.

As shown in Fig. 10, almost all heuristics benefit performance for all isolation levels where applicable. One exception is that H_3 helps only for low percentages of rows accessed and can be detrimental (especially for serializable) otherwise because of overheads. This is consistent with RQ3; with more rows accessed and stronger isolation, there is less nondeterminism that H_3 could help resolve. Another exception is that H_4 sometimes does not help for RC because the logs fail to be strict RC relatively early on, and TROUBADOUR proceeds to check RC using the provided window. For a remote database (recall that our logs were generated using a local database with extremely short latency), H_4 should be more consistently useful for RC; transactions would be subjected to additional delays such as network transit time, making it more likely they would satisfy stronger levels enforcing real-time dependencies. Indeed, if we add a sufficiently long delay between queries in the local setting, the logs all satisfy the strict version of the isolation level. These results demonstrate that it is best to enable all heuristics for workloads where there is a very high amount of nondeterminism, and that otherwise, it is best to enable all heuristics but H_3.

Summary. *All heuristics greatly improve performance and interact well with one another when there is a high amount of nondeterminism. For example, turning on all heuristics leads TROUBADOUR to check 18x more transactions on average for SI.*

8.5 Industrial Case Study

We applied TROUBADOUR to logs generated using an unreleased (non-production) database that offers SSI, built by our industrial collaborator. As our approach works for black-box DBMSs, no modifications were required to apply TROUBADOUR to these logs. TROUBADOUR will eventually be run continuously in a canary as a step in the CI/CD pipeline to monitor for correctness violations.

Semantic correctness. TROUBADOUR was able to catch semantic correctness violations in the query processor. The first violation was caused by an indexing logic bug: when a table was created

and inserted into in the same transaction, `SELECT...WHERE...` statements in subsequent transactions would return no results when at least one row satisfied the `WHERE` predicate. However, `SELECT` statements with no `WHERE` gave expected results. The second violation manifested in the form of multiple rows returned when selecting based on a primary key predicate. We also used Troubadour to check that known bugs in the query processor that had already been fixed.

Isolation correctness. We generated and checked 24 hours worth of logs for the bank-server and TPC-C benchmarks. Logs were generated with two concurrent users, and each had their own query processor. Bank-server logs had 1000 accounts, 100% rows accessed, 0% blind writes, and 10% reads. We configured TPC-C with 1 warehouse, 2 districts, 3 customers per district, and 1 item. The setup was as in RQ3 but with H_3 disabled. Troubadour demonstrated observational correctness for a total of 1,096,548 transactions for the bank-server benchmark in 1.76 hours. Because TPC-C is a more complex benchmark, checking observational correctness for a total of 60,022 transactions took 89.2 hours. No semantic correctness nor isolation level violations were found.

Summary. *Troubadour was applied directly to 24 hours of logs for bank-server and TPC-C benchmarks for a database under development in industry. It found new semantic bugs in and verified observational correctness of the logs.*

9 Related Work

Below we describe related work by category. Additional related work includes Litmus [38], which constructs a verified DBMS by modifying interactions with a (potentially black-box) backend DBMS.

Automated consistency verification. Different isolation levels result in different *consistency levels* being provided by the DBMS. Consistency levels describe the visibility of and constraints on the order of operations performed in a concurrent or distributed system, and there has been much work automating verification of consistency levels such as linearizability [26, 41], serializability [32], sequential consistency [30], and causal consistency [9, 39]. As with isolation levels, this requires reasoning about nondeterministic reads for weaker levels like sequential and causal consistency.

Many of these verification techniques do not only verify observational correctness but check that *all* executions of concurrent operations provide the specified consistency guarantees, especially for stronger consistency levels like linearizability [26]. For weaker consistency levels, this verification is difficult – it has been shown to be undecidable for causal consistency [9] – there are thus also techniques that consider verifying consistency levels of only *single* executions at a time [4, 9, 39].

Our setting is distinct from all these techniques in two main ways: (1) our technique is parameterizable by isolation level, where many of the levels we can check result in weaker consistency guarantees than considered by automated verifiers for consistency levels; and (2) we operate in a black-box setting without access to the state of the system, meaning that we must reason about semantics to *infer whether there exists* a sequence of states such that an execution is correct.

Testing. The testing approaches for DBMSs that we discuss here generate test cases that expose bugs in the DBMS implementation, and do not test arbitrary transactions and client-side transaction logs for correctness. They therefore solve a much different problem than DBMS correctness auditing.

There has been much work on DBMS fuzzing for SQL using a variety of techniques [3, 15, 31, 42], such as ternary logic partitioning and mutation-based fuzzing. These tools focus exclusively on semantic correctness of intra-transactional operations as they do not reason about isolation levels. ADUSA [23, 24] uses constraint solving in Alloy to systematically test DBMSs. The TxCheck fuzzer employs some limited reasoning about isolation levels [22]. The MonkeyDB work not only tests for semantic bugs but also for isolation bugs in DBMSs [8]. It proposes operational semantics (based on previous work [7]) and handles a limited subset of SQL by compiling to key-value store

operations [8]. The semantics is used to test DBMS correctness with respect to a set of transactions for which all read values are observable (i.e., there is no read nondeterminism).

Checking isolation levels. Most work on checking isolation guarantees solves the following problem: *given an isolation level I and set of transactions $\mathcal{T}$ with known reads and writes, does there exist an execution of $\mathcal{T}$ that meets constraints for I?* This problem is similar to the problem of proving observational correctness with respect to a log, but importantly includes no reasoning about the semantics of the transactions. The assumption that *all* read and written values are observable either limits the kinds of transactions that can be considered (e.g., the `Deposit` and `Transfer` transactions from Fig. 2 could not be used as-is), requires a non-black-box DBMS, or leads to bugs being missed. While these values may be gotten by instrumenting transactions [22], instrumentation implicitly relies on the semantic correctness of the DBMS and may not be possible for certain DBMSs [16].

Elle deduces dependencies between transactions based on read and written values, and then reasons possible transaction dependency graphs according to Adya's formalism [5, 6]. Related techniques do specialized checking of certain isolation levels [21, 27, 34, 40], such as serializability and SI. Some of these further make the assumption that each value in the database is unique [21, 34]. Several of these techniques rely on checking for graph properties, where this checking is encoded as a SAT or SMT problem. While we also reduce DBMS correctness auditing to SMT solving, the queries we pose are more complex as we also require the solver to reason about the semantics of transactions. Leopard is a much different approach that exploits implementation details of commercial DBMSs to check they enforce isolation levels correctly [25].

Database query and program verification. Work on verification of database query equivalence is also concerned with the semantics of database queries [10, 14, 20]; however, it does not consider transactional settings nor SQL statements that modify the underlying database. Work on equivalence of database-backed programs does consider transactions that may modify the database [36], but it does not consider that the underlying DBMS may provide anything but full isolation of transactions.

Automated verification of concurrent programs. Automated verification of concurrent programs makes a similar distinction between semantics and consistency levels that we do for semantics and isolation levels [18, 19, 33]: semantics are encoded into BMC(-like) formulas and separate constraints are generated to constrain executions to conform to the consistency level. These approaches also aim to generate fewer SMT constraints and perform reasoning during solving to reduce the number of consistency constraints generated. This idea is similar in spirit to our on-demand adding of constraints to reduce the complexity of the SMT problem; however, these approaches focus on consistency constraints and rely on new theory solvers to perform this reduction. In contrast, we do not rely on new theory solvers and aim to reduce *both* semantic and isolation constraints generated by evaluating the VA-TDL semantics concretely where possible, maintaining database states' representations outside the solver, and passing only relevant constraints to the solver on-demand. It is also worth noting that these techniques are not compatible with the state-based formalism of isolation we consider here, which is not formulated in terms of relations on read and write events.

10 Conclusion

We proposed a novel formal definition of observational correctness for black-box DBMSs and an SMT-based checking approach for it that is parameterizable over different transaction description languages and isolation levels. We demonstrated experimentally that our implementation Troubadour can verify *full* observational correctness (including both semantic and isolation constraints), and that it can scale to practical log sizes. Future work includes developing further heuristics or optimizations for improving scalability for non-serializable isolation levels and extensions to handle more permissive isolation levels that allow for reading of uncommitted states.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant # 2127309 to the Computing Reserach Association for the CIFellows project.

Data-Availability Statement

The artifact, containing software that supports §8, is available on Zenodo [28]. It also contains two Docker images (ARM and AMD) with all dependencies installed. Performance of the tool can vary depending on the architecture of the machine being run on.

References

[1] 2011. Big data in real time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter.
[2] 2017. RUBiS. https://projects.ow2.org/view/rubis/.
[3] 2022. SQLsmith: a random sql query generator. https://github.com/anse1/sqlsmith. Accessed: 2023-11-15.
[4] 2024. Jepsen. https://jepsen.io/. Accessed: 2024-04-05.
[5] Atul Adya. 1999. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. (1999).
[6] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280. https://doi.org/10.5555/3430915.3442427
[7] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28. https://doi.org/10.1145/3360591
[8] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: effectively testing correctness under weak isolation levels. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485546
[9] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 626–638. https://doi.org/10.1145/3009837.3009888
[10] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 510–524.
[11] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods Syst. Des.* 19, 1 (2001), 7–34. https://doi.org/10.1023/A:1011276507260
[12] 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. https://doi.org/10.1145/3087801.3087802
[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.
[14] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proc. ACM Manag. Data* 1, 4 (2023), 227:1–227:26.
[15] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin : Grammar-Free DBMS Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 49:1–49:12. https://doi.org/10.1145/3551349.3560431
[16] Google Cloud. 2024. BigQuery: Multi-statement transactions. https://cloud.google.com/bigquery/docs/transactions. Accessed: 2024-04-02.
[17] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.
[18] Thomas Haas, Roland Meyer, and Hernán Ponce de León. 2022. CAAT: consistency as a theory. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 114–144. https://doi.org/10.1145/3563292
[19] Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1264–1279. https://doi.org/10.1145/3453483.3454108
[20] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 1071–1099. https:

//doi.org/10.1145/3649849

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

[22] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 397–417. https://www.usenix.org/conference/osdi23/presentation/jiang

[23] Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL query generation for systematic testing of database engines. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 329–332. https://doi.org/10.1145/1858996.1859063

[24] Shadi Abdul Khalek and Sarfraz Khurshid. 2011. Systematic Testing of Database Engines Using a Relational Constraint Solver. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*. IEEE Computer Society, 50–59. https://doi.org/10.1109/ICST.2011.21

[25] 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. https://doi.org/10.1109/ICDE55515.2023.00061

[26] Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. 2009. Model Checking Linearizability via Refinement. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5850)*, Ana Cavalcanti and Dennis Dams (Eds.). Springer, 321–337. https://doi.org/10.1007/978-3-642-05089-3_21

[27] Kartik Nagar and Suresh Jagannathan. 2018. Automated Detection of Serializability Violations Under Weak Consistency. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs, Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 41:1–41:18. https://doi.org/10.4230/LIPICS.CONCUR.2018.41

[28] Lauren Pick, Amanda Xu, Ankush Desai, Sanjit Seshia, and Aws Albarghouthi. 2024. *Checking Observational Correctness of Database Systems Artifact (Troubadour)*. https://doi.org/10.5281/zenodo.14918621

[29] PostgreSQL Global Development Group. 2023. PostgreSQL. http://www.postgresql.org.

[30] Shaz Qadeer. 2003. Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking. *IEEE Trans. Parallel Distributed Syst.* 14, 8 (2003), 730–741. https://doi.org/10.1109/TPDS.2003.1225053

[31] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 211:1–211:30. https://doi.org/10.1145/3428279

[32] Arnab Sinha and Sharad Malik. 2010. Runtime checking of serializability in software transactional memory. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 1–12. https://doi.org/10.1109/IPDPS.2010.5470389

[33] Zhihang Sun, Hongyu Fan, and Fei He. 2022. Consistency-preserving propagation for SMT solving of concurrent program verification. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 929–956. https://doi.org/10.1145/3563321

[34] 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

[35] Transaction Processing Performance Council. 2010. TPC-C Benchmark Revision 5.11.0. https://www.tpc.org/tpcc/.

[36] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.* 2, POPL (2018), 56:1–56:29.

[37] ANSI X3.135-1992. November 1992. American National Standard for Information Systems – Database Language – SQL.

[38] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. 2022. Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1478–1492. https://doi.org/10.1145/3514221.3517851

[39] Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2022. Checking causal consistency of distributed databases. *Computing* 104, 10 (2022), 2181–2201. https://doi.org/10.1007/S00607-021-00911-3

[40] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 654–671. https://doi.org/10.1145/3552326.3567492

[41] Shao Jie Zhang. 2011. Scalable automatic linearizability checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 1185–1187. https://doi.org/10.1145/1985793.1986037

[42] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 955–970. https://doi.org/10.1145/3372297.3417260

Received 2024-10-16; accepted 2025-02-18