Automated Detection of Serializability Violations Under Weak Consistency

Kartik Nagar
Purdue University, USA
nagark@purdue.edu

Suresh Jagannathan
Purdue University, USA
suresh@cs.purdue.edu

Abstract

While a number of weak consistency mechanisms have been developed in recent years to improve performance and ensure availability in distributed, replicated systems, ensuring the correctness of transactional applications running on top of such systems remains a difficult and important problem. Serializability is a well-understood correctness criterion for transactional programs; understanding whether applications are serializable when executed in a weakly-consistent environment, however remains a challenging exercise. In this work, we combine a dependency graph-based characterization of serializability and leverage the framework of abstract executions to develop a fully-automated approach for statically finding bounded serializability violations under *any* weak consistency model. We reduce the problem of serializability to satisfiability of a formula in First-Order Logic (FOL), which allows us to harness the power of existing SMT solvers. We provide rules to automatically construct the FOL encoding from programs written in SQL (allowing loops and conditionals) and express consistency specifications as FOL formula. In addition to detecting bounded serializability violations, we also provide two orthogonal schemes to reason about unbounded executions by providing sufficient conditions (again, in the form of FOL formulae) whose satisfiability implies the absence of anomalies in any arbitrary execution. We have applied the proposed technique on TPC-C, a real-world database program with complex application logic, and were able to discover anomalies under Parallel Snapshot Isolation (PSI), and verify serializability for unbounded executions under Snapshot Isolation (SI), two consistency mechanisms substantially weaker than serializability.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Weak Consistency, Serializability, Database Applications

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2018.41

Related Version A full version of the paper is available at [24], `https://arxiv.org/abs/1806.08416`.

1 Introduction

We consider the problem of detecting serializability violations of transactional programs executing in a weakly-consistent replicated distributed database. An execution of such programs is said to be *serializable* if it is equivalent to some sequential execution of the transactions that comprise the program. Ensuring that all executions of such programs are serializable greatly simplifies reasoning about program correctness by reducing the complexity of understanding concurrent executions to the problem of understanding sequential ones. Unfortunately, *enforcing* serializability using runtime synchronization mechanisms

is problematic in geo-replicated distributed systems without sacrificing availability (low-latency) [18]. To reap the correctness benefits of serializability with the performance and scalability benefits of high-availability, we study the conditions under which transactional programs can be statically identified to always yield a serializable execution *without* the need for global synchronization. The challenge to realizing this goal stems from the complexity in reasoning about replicated state in which not all replicas share the same view of the data they hold.

To address this challenge, we present a fully automated static analysis that precisely encodes salient dependencies in the program as abstract executions defined in terms of an axiomatic specification of a particular weak consistency model (§4). The analysis then leverages a theorem prover to systematically search for the presence or absence of cycles in these executions consistent with these dependencies; the presence of a cycle indicates a serializability violation (§5.1). Notably, our approach can be applied to any weak consistency model whose specification can be expressed in first-order logic, a class that subsumes all realistic data stores we are aware of. More specifically, our approach constructs a dependency graph [2] from the input program containing a cycle and then asks whether there exists a valid execution under the given consistency specification that can result in this graph. To do this, we automatically extract dependency conditions from the transactional program, and relate these dependencies to artifacts in an event-based model to find whether there exists a valid abstract execution corresponding to the dependency graph. These dependencies are encoded in a first-order logic formula that is satisfiable only if there exists an execution that violates serializability.

Given a transactional program written in SQL, we discover serializability violations of bounded length under the given weak consistency model (with the bound limiting the number of concurrent transaction instances that are considered). We output the actual anomaly including the transactions involved and their inputs. This output can then be used to strengthen the consistency of the transactions involved in the anomaly (or even modifying the transactions themselves). Since the approach is parametric on a consistency policy, it can also be used to determine the weakest consistency policy for which the program is serializable. Similar to other bounded verification techniques used to detect bugs in e.g., concurrent programs [23], we posit that most serializability violations will manifest using a small number of transaction instances.

Nonetheless, we additionally provide two orthogonal schemes to reason about arbitrarily long executions with an unbounded number of transaction instances (§5.2, §5.3). The first scheme formalizes the argument that it is enough to check serializability violations in bounded executions by proving that longer violations beyond that bound would induce violations within the bound. The second scheme applies an inductive argument to check the absence of anomalies in arbitrarily long executions. Our approach is sound, but not complete - while all discovered anomalies are justified by counterexamples offered by the theorem prover, we cannot rule out the possibility of serializability violations appearing in unbounded executions that are not identified by these two schemes.

As serious case studies to assess the applicability of our approach, we have applied our technique on TPC-C, a real-world transactional program, and a course grading application [19] (§6). In both cases, we were able to detect multiple serializability violations under Eventual Consistency and a weaker variant of snapshot isolation (SI) called parallel snapshot isolation [26], and verified that these anamolies do not occur when using SI for unbounded executions. We now present an overview of our approach using a simple example.

```
withdraw (ID, Amount)
    SELECT Balance AS bal WHERE AccID=ID
    IF bal > Amount
        UPDATE SET Balance=bal-Amount WHERE AccID=ID
```

Figure 1 Example Application.

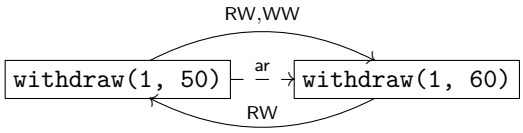

Figure 2 Abstract Execution E and its Dependency Graph.

2 Overview

In this section, we show how our approach discovers serializability violations, and how the output of our analysis can be used to repair violations using selective synchronization. Consider a simple banking application which maintains the balance of multiple accounts in a table `Account` which is indexed using the primary key `AccID` and contains the field `Balance`. Consider a `withdraw` operation (shown in Fig. 1) written in a SQL-style language, which takes `ID` and `Amount` as input, and deducts the amount from the account with account number `ID`, if the balance is sufficient. Suppose the application is deployed in a distributed, replicated environment which allows concurrent invocations of the `withdraw` operation at potentially different replicas, with the only guarantee provided being eventual consistency - eventually, all replicas will witness all updates to the `Balance` field. Under eventual consistency, the application is clearly not serializable, since concurrent withdraws operations to the same account–whose total withdrawn amount exceeds the balance of the account–could both succeed, which is not possible in a serializable execution.

A convenient way to express executions in such an environment is to use an axiomatic event-based representation. In this framework, an abstract execution [12] is expressed as the tuple $(T, \mathsf{vis}, \mathsf{ar})$, where T is the set of transaction invocations, $\mathsf{vis} \subseteq T \times T$ is a visibility relation such that if $t \xrightarrow{\mathsf{vis}} t'$ then updates of t are visible to t', and $\mathsf{ar} \subseteq T \times T$ is an arbitration relation which totally orders all writes to the same location and ensures eventual consistency [9]. For example, if $t_1 =$ `withdraw`(1,50), $t_2 =$ `withdraw`(1,60), then $E = (\{t_1, t_2\}, \{\}, \{(t_1, t_2)\})$ is an abstract execution which is not serializable, because the final value of `Balance` in the account number 1 will only reflect the `withdraw` operation t_2 (assuming an initial `Balance` of 100 in `AccID` 1), since there is no visibility constraint enforced between the two operations. This is an example of a *lost update* [5] anomaly. Our goal is to automatically construct such anomalous executions.

A useful technique to detect serializability violations is to build dependency graphs from abstract executions, and then search for cycles in the dependency graph. The nodes of the dependency graph are invocations, and edges indicate dependencies between them. There are three type of dependencies relevant to serializability detection: $t_1 \xrightarrow{\mathsf{WR}} t_2$ is a read dependency, which means that t_2 reads a value written by t_1, $t_1 \xrightarrow{\mathsf{WW}} t_2$ is a write dependency, which means that both t_1 and t_2 write to the same location, with the write of t_2 arbitrated after t_1, and $t_1 \xrightarrow{\mathsf{RW}} t_2$ is an anti-dependency, which means that t_1 does *not* read a value written by t_2 but instead reads an older version. For example, the dependency graph of the anomalous execution E described above is shown in Fig. 2.

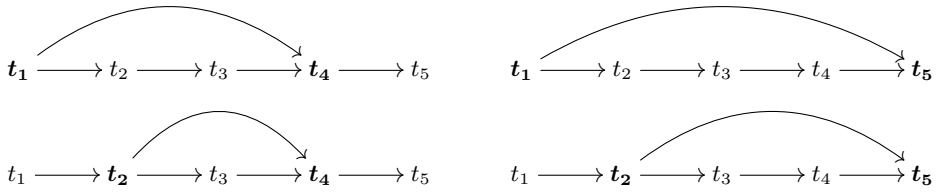

Figure 3 Different possibilities for paths of length 4 in the dependency graphs of the banking application. Note that transactions in bold perform writes.

In our approach, we start with a dependency graph containing a cycle, and then ask whether an execution corresponding to the dependency graph is possible. From the transaction code, we automatically extract the conditions under which a dependency edge can manifest between invocations of the transactions. In our running example, a dependency edge (of any type) between two `withdraw` invocations can only manifest if they are called with the same account ID. Further, we link the dependency edges with the relations vis and ar of the corresponding abstract execution. For example, $t_1 \xrightarrow{\mathsf{RW}} t_2 \Rightarrow \neg(t_2 \xrightarrow{vis} t_1)$, because otherwise, t_1 would read the value written by t_2. This is useful because different consistency schemes can be axiomatically expressed by placing constraints on the vis and ar relations.

In order to prevent the anomalous execution in our running example, we can use PSI [26] which ensures that if two invocations write to the same location, then they cannot be concurrent. While PSI is implemented using a complex, distributed protocol, in our abstract framework, it can be simply expressed using the following constraint: $\forall t, t'.\ t \xrightarrow{\mathsf{WW}} t' \implies t \xrightarrow{\mathsf{vis}} t'$. Now, the anomalous execution E is not possible, because $t_1 \xrightarrow{\mathsf{WW}} t_2 \Rightarrow t_1 \xrightarrow{\mathsf{vis}} t_2$, which contradicts $t_2 \xrightarrow{\mathsf{RW}} t_1$.

To summarize, the following is the relevant portion of formulae that we generate for the above application under PSI:

$$\forall t, t'.\ t \xrightarrow{\mathsf{RW}} t' \Rightarrow (\exists r.\ \mathtt{AccID}(r) = \mathtt{ID}(t) \wedge \mathtt{AccID}(r) = \mathtt{ID}(t') \wedge \mathtt{bal}(t') > \mathtt{Amount}(t')) \quad (1)$$

$$\forall t, t', r.\ (\mathtt{AccID}(r) = \mathtt{ID}(t) \wedge \mathtt{bal}(t) > \mathtt{Amount}(t) \wedge \mathtt{AccID}(r) = \mathtt{ID}(t') \wedge \mathtt{bal}(t') > \mathtt{Amount}(t') \wedge t \xrightarrow{\mathsf{ar}} t') \Rightarrow t \xrightarrow{\mathsf{WW}} t' \quad (2)$$

$$\forall t, t'.\ t \xrightarrow{\mathsf{RW}} t' \Rightarrow \neg(t' \xrightarrow{\mathsf{vis}} t) \quad (3)$$

$$\forall t, t'.\ t \xrightarrow{\mathsf{WW}} t' \Rightarrow t \xrightarrow{\mathsf{vis}} t' \quad (4)$$

We use t, t' to denote invocations of the transaction, and r to denote a record in the database. We define the function `AccID` to access the primary key of a record. Similarly, `ID`, `Amount`, etc. are functions which map an invocation to its parameters and local variables. The existence of a dependence between two invocations forces the existence of a record that both invocations must access, as well as conditions on the local variables required to perform the access (Eqn. 1). On the other hand, if two invocations are guaranteed to write to the same location, there must exist a WW dependency between them (Eqn. 2). Now, it is not possible to have invocations t_1 and t_2, obeying Eqns. (1)-(4) such that $t_1 \xrightarrow{\mathsf{RW}} t_2$ and $t_2 \xrightarrow{\mathsf{RW}} t_1$, the condition necessary to induce a cycle and thus manifest a serializability violation.

In fact, it is not possible to have a cycle of any arbitrary length in a dependency graph of this application under PSI. To show this, we use the following observation: any long path in a dependency graph generated by the above application will have chords in it, resulting in a shorter path. In fact, it can be shown that the shortest path between any two invocations in any dependency graph of the application (if there is a path) will always be less than or equal

to 3. This can shown by using the above constraints (1)-(4) (and adding similar constraints for WR edges), instantiating a path of length 4 such that there is no chord between any of the nodes involved in the path, and then showing the unsatisfiability of such an encoding. Since a cycle is also a path, it is now sufficient to only check for cycles of length 3, since any longer cycle will necessarily induce a cycle of length less than or equal to 3.

Intuitively, this is happening in the banking application because the presence of any dependency edge between two nodes implies that both invocations must access the same account, and at least one of them must perform a write. Further, any two writes are always related by a WW edge. Now, as shown in Figure 3, in any path of length 4 in the dependency graph, one of t_1 or t_2 and one of t_4 or t_5 must be a write, which implies a chord between the two writes. Hence, there will always be a shorter path of length less than or equal to 3 between t_1 and t_5.

3 Preliminaries

3.1 Input Language and Database Model

$$\texttt{v} \in \texttt{Variables} \quad \texttt{f} \in \texttt{Fields} \quad \mathcal{Q} \in \{\texttt{MIN}, \texttt{MAX}, \texttt{COUNT}\}$$
$$\oplus \in \{+, -, \times, /\} \quad \odot \in \{<, \leq, =, >, \geq\} \quad \circ \in \{\wedge, \vee\}$$

$$\begin{array}{lcl}
e_d & \coloneqq & \texttt{f} \mid \texttt{v} \mid e_d \oplus e_d \mid \mathbb{Z} \\
\phi_d & \coloneqq & \texttt{f} \odot e_d \mid \texttt{f} \in \texttt{v} \mid \neg\phi_d \mid \phi_d \circ \phi_d \\
e_c & \coloneqq & \texttt{v} \mid \texttt{CHOOSE v} \mid e_c \oplus e_c \mid \mathbb{Z} \\
\phi_c & \coloneqq & \texttt{v} \odot e_c \mid \texttt{v} = \texttt{NULL} \mid \texttt{v}_1 \in \texttt{v}_2 \mid \neg\phi_c \mid \phi_c \circ \phi_c \\
c & \coloneqq & \texttt{SELECT } \bar{\texttt{f}} \texttt{ AS v WHERE } \phi_\texttt{d} \mid \texttt{SELECT } \mathcal{Q}\ \texttt{f AS v WHERE } \phi_\texttt{d} \mid \texttt{UPDATE SET f} = \texttt{e}_\texttt{c} \texttt{ WHERE } \phi_\texttt{d} \mid \\
 & & \texttt{INSERT VALUES } \bar{\texttt{f}} = \bar{\texttt{e}}_\texttt{c} \mid \texttt{DELETE WHERE } \phi_\texttt{d} \mid \texttt{v} = \texttt{e}_\texttt{c} \mid \texttt{IF } \phi_\texttt{c} \texttt{ THEN c ELSE c} \mid \texttt{c ; c} \\
 & & \texttt{FOREACH } \texttt{v}_1 \texttt{ IN } \texttt{v}_2 \texttt{ DO c END} \mid \texttt{SKIP} \\
vlist & \coloneqq & \texttt{v} \mid vlist, vlist \\
\mathcal{T} & \coloneqq & \texttt{Tname}(vlist)\{\texttt{c}\}
\end{array}$$

We start with description of the language of transactional programs in our framework. We assume a database model, where data is organized in tables with multiple records, where each record has multiple fields and transactions can insert/delete records and read/modify fields in selected records. The grammar is essentially a simplified version of standard SQL, allowing SQL statements which access the database to be combined with usual program connectives such as conditionals, sequencing and loops. Every transactional program $\mathcal{T}$ has a set of parameter variables ($vlist$) which are instantiated with values on invocation, and a set of local variables which are used to store intermediate values from the database (typically as output of `SELECT` queries). For a transactional program $\mathcal{T}$, let $\texttt{Vars}(\mathcal{T})$ be the set of parameters and local variables of $\mathcal{T}$. Let $\texttt{Stmts}(\mathcal{T})$ be the set of SQL statements (i.e. `INSERT`, `DELETE`, `SELECT` or `UPDATE`) in $\mathcal{T}$.

To simplify the presentation, we will assume that there is only one table and each record is a set of values indexed by the set `Fields`. Furthermore all fields store integer values. The `FOREACH` loop iterates over a set of records in $\texttt{v}_2$, and assigns $\texttt{v}_1$ to an individual record during each iteration. We call $\texttt{v}_2$ as the loop variable. Let $\mathcal{D}(\texttt{v})$ denote the nesting depth of v, which is 0 if v is assigned a value outside any loop (or is a parameter variable), and otherwise is the number of enclosing loops. For a variable v assigned a value inside a loop, let $\texttt{LVar}(\texttt{v}, i)$ denote the loop variable at depth i, for all $1 \leq i \leq \mathcal{D}(\texttt{v})$.

SQL statements use predicates ϕ_d to select records that would be accessed/modified, where ϕ_d allows all boolean combinations of comparison predicates between fields and values. Conditionals used inside `IF` statements (ϕ_c) are only allowed to used local variables and

parameters. To check whether the output of a `SELECT` query is empty, we use the conditional expression `v = NULL`, where `v` stores the output of the query.

We assume a fixed non-empty subset of `Fields` to be the primary key `PK`. Any two records must have distinct values in at least one of their `PK` fields. Assume that there is a special field called `Alive` $\in$ `Fields` whose value is 1 if the record is in the database, 0 otherwise. Initially, all records are not `Alive`. When a record is inserted into the database, it becomes `Alive`, and when the record is deleted, it again becomes not `Alive`.

3.2 Abstract Executions

Executions of transactional programs in our framework are expressed using an event structure, which is based on the approach used in [5]. The execution of a transaction instance consists of events, which are database operations. A database operation is a read or write to a field of a record. Let $\mathcal{R} = \texttt{PK} \to \mathbb{Z}$ be the set of all possible primary keys. Then, the set of all database operations is $\mathcal{O} = \{\texttt{wri}(r, f, n) \mid r \in \mathcal{R}, f \in \texttt{Fields} \setminus \texttt{PK}, n \in \mathbb{Z}\} \cup \{\texttt{rd}(r, f, n) \mid r \in \mathcal{R}, f \in \texttt{Fields}, n \in \mathbb{Z}\}$.

To simplify the presentation, we assume that a transaction reads (writes) at most once from (to) a field of a record and does not read any record that it writes, inserts or deletes. These assumptions allow us to ignore the ordering among events of a single transaction instance. Our approach can be easily adapted if these assumptions are not satisfied.

▶ **Definition 1** (Transaction Instance)**.** A transaction instance is a tuple $\sigma = (\texttt{TID}, \varepsilon)$, where `TID` is a unique transaction instance-ID and $\varepsilon \subseteq \mathcal{O}$ is a set of events.

In this work, we assume that transactions are executed in an environment which guarantees atomicity and isolation (also called atomic visibility [12]). That is, either all events of a transaction are made visible to other transactions, or none are, and the same set of transactions are visible to all events in a transaction. Atomicity and isolation are crucial properties for transactional programs, and both can be implemented efficiently in a replicated, distributed environment [9, 3]. Note that atomicity and isolation does not guarantee serializability, as seen in example in §2, and our goal is to explore serializability in this context of weak consistency.

▶ **Definition 2** (Abstract Execution)**.** An abstract execution is a tuple $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$, where Σ is a set of transaction instances, $\mathsf{vis} \subseteq \Sigma \times \Sigma$ is an anti-symmetric, irreflexive relation, and $\mathsf{ar} \subseteq \Sigma \times \Sigma$ is a total order on Σ such that $\mathsf{vis} \subseteq \mathsf{ar}$.

Intuitively, given transaction instances σ, σ' in an abstract execution χ, if $\sigma \xrightarrow{\mathsf{vis}} \sigma'$, then all writes performed by σ are visible to σ' and hence may affect the output of the reads performed by σ'. ar is used to order all writes to the same location. We use the notation $\sigma \vdash o$ to specify that transaction instance σ performs a database operation o. The length of an abstract execution is defined to be the number of transaction instances involved in the execution (i.e. $|\Sigma|$).

Given a set of transaction instances Σ', we use the notation $[\Sigma']_{<\texttt{wri}(r,f)>} = \{\sigma \in \Sigma' \mid \sigma \vdash \texttt{wri}(r, f, n), n \in \mathbb{Z}\}$ to denote the set of transactions which are writing to field `f` of record `r`. We use the notation $\mathsf{MAX}_{\mathsf{ar}}(\Sigma')$ to denote $\sigma \in \Sigma'$ such that $\forall \sigma' \in \Sigma'.\ \sigma = \sigma' \vee \sigma' \xrightarrow{\mathsf{ar}} \sigma$. Given a transaction instance σ, we use $\mathsf{vis}^{-1}(\sigma)$ to denote the set $\{\sigma' \in \Sigma \mid \sigma' \xrightarrow{\mathsf{vis}} \sigma\}$. The last writer wins nature of the database dictates that a transaction reads the most recent value (according to ar) written by the transactions visible to it. Formally, this is specified as follows: $\sigma \vdash \texttt{rd}(r, f, n) \Rightarrow (f \notin \texttt{PK} \Rightarrow \mathsf{MAX}_{\mathsf{ar}}([\mathsf{vis}^{-1}(\sigma)]_{<\texttt{wri}(r,f)>}) \vdash \texttt{wri}(r, f, n)) \wedge (f \in \texttt{PK} \Rightarrow r(f) = n)$.

▶ **Definition 3** (Dependency Graph). Given an abstract execution $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$, the dependency graph $G_\chi = (\Sigma, E)$ is a directed, edge-labeled multigraph where the edges and their labels are defined as follows :

- $\sigma \xrightarrow{\mathsf{WR}_{r,f}} \sigma'$ if $\sigma' \vdash \mathtt{rd}(r, f, n)$ and $\sigma = \mathsf{MAX}_{\mathsf{ar}}([\mathsf{vis}^{-1}(\sigma')]_{<\mathtt{wri}(r,f)>})$.
- $\sigma \xrightarrow{\mathsf{WW}_{r,f}} \sigma'$ if $\sigma \vdash \mathtt{wri}(r, f, n)$, $\sigma' \vdash \mathtt{wri}(r, f, m)$ and $\sigma \xrightarrow{\mathsf{ar}} \sigma'$.
- $\sigma \xrightarrow{\mathsf{RW}_{r,f}} \sigma'$ if $\sigma \vdash \mathtt{rd}(r, f, n)$, $\sigma' \vdash \mathtt{wri}(r, f, m)$ and there exists another transaction instance σ'' such that $\sigma'' \xrightarrow{\mathsf{WR}_{r,f}} \sigma$ and $\sigma'' \xrightarrow{\mathsf{WW}_{r,f}} \sigma'$.

Edges in the dependency graph G_χ also induce corresponding binary relations on the transaction instances (we use the same notation for these relations). Let $\mathsf{WR}, \mathsf{WW}, \mathsf{RW}$ be the union of $\mathsf{WR}_{r,f}, \mathsf{WW}_{r,f}, \mathsf{RW}_{r,f}$ for all r, f respectively. The following lemma follows directly from the definition[1]:

▶ **Lemma 4.** *Given an abstract execution $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$ and its dependency graph $G_\chi = (\Sigma, E)$, the following are true:*

- *If $\sigma \xrightarrow{\mathsf{WR}_{r,f}} \sigma' \in E$, then $\sigma \xrightarrow{\mathsf{vis}} \sigma'$.*
- *If $\sigma \xrightarrow{\mathsf{WW}_{r,f}} \sigma' \in E$, then $\sigma \xrightarrow{\mathsf{ar}} \sigma'$.*
- *If $\sigma \xrightarrow{\mathsf{RW}_{r,f}} \sigma' \in E$, then $\neg(\sigma' \xrightarrow{\mathsf{vis}} \sigma)$.*

In our framework, transaction instances are generated by assigning values to all the parameter variables of a transactional program $\mathcal{T}$, written using the grammar specified in §3.1. We use the notation $\Gamma(\sigma)$ to denote the transactional program $\mathcal{T}$ associated with transaction instance σ.

Different weak consistency and weak isolation models can be expressed by placing constraints on vis and ar relations associated with an abstract execution. This gives rise to the notion of *valid* abstract executions under a specific model, which are executions satisfying the constraints associated with those models. Below, we provide examples of several known weak consistency and weak isolation models:

- Full Serializability : $\Psi_{\text{Ser}} \triangleq \mathsf{vis} = \mathsf{ar}$
- Selective Serializability for Transactional Programs $\mathcal{T}_1, \mathcal{T}_2$ [16] : $\Psi_{\text{Ser}(\mathcal{T}_1,\mathcal{T}_2)} \triangleq \forall \sigma_1, \sigma_2.$ $((\Gamma(\sigma_1) = \mathcal{T}_1 \wedge \Gamma(\sigma_2) = \mathcal{T}_2) \vee (\Gamma(\sigma_1) = \mathcal{T}_2 \wedge \Gamma(\sigma_2) = \mathcal{T}_1) \wedge \sigma_1 \xrightarrow{\mathsf{ar}} \sigma_2) \Rightarrow \sigma_1 \xrightarrow{\mathsf{vis}} \sigma_2$
- Causal Consistency (CC) [22] : $\Psi_{\text{CC}} \triangleq \forall \sigma_1, \sigma_2, \sigma_3.\ \sigma_1 \xrightarrow{\mathsf{vis}} \sigma_2 \wedge \sigma_2 \xrightarrow{\mathsf{vis}} \sigma_3 \Rightarrow \sigma_1 \xrightarrow{\mathsf{vis}} \sigma_3$
- Prefix Consistency (PC) (equivalent to *repeatable read* in centralized databases) [27, 10] : $\Psi_{\text{PC}} \triangleq \forall \sigma_1, \sigma_2, \sigma_3.\ \sigma_1 \xrightarrow{\mathsf{ar}} \sigma_2 \wedge \sigma_2 \xrightarrow{\mathsf{vis}} \sigma_3 \Rightarrow \sigma_1 \xrightarrow{\mathsf{vis}} \sigma_3$
- Parallel Snapshot Isolation (PSI) [26] : $\Psi_{\text{PSI}} \triangleq \forall \sigma_1, \sigma_2.\ \sigma_1 \xrightarrow{\mathsf{WW}} \sigma_2 \Rightarrow \sigma_1 \xrightarrow{\mathsf{vis}} \sigma_2$

Different models can be also be combined together to create a hybrid model. For example, $\Psi_{PSI} \wedge \Psi_{PC}$ is equivalent to Snapshot Isolation [4] in centralized databases. Below, we formalize the classical notion of conflict serializability [6] in our setting and then relate it to the presence of cycles in the dependency graph.

▶ **Definition 5** (Serializable Execution). An abstract execution $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$ is said to be serializable if there exists another abstract execution $\chi' = (\Sigma, \mathsf{vis}', \mathsf{ar}')$ which satisfies Ψ_{Ser} such that G_χ and $G_{\chi'}$ are isomorphic.

▶ **Theorem 6.** *Given an abstract execution $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$, if there is no cycle in the dependency graph G_χ, then χ is serializable.*

[1] All proofs can be found in Appendix C in the extended version of the paper[24].

3.3 Operational Semantics

We now propose an operational semantics to generate abstract executions from transactional programs under a consistency specification. The purpose of the operational semantics is to link SQL statements with abstract database operations, and to prove the soundness of our encoding in FOL. Here, we only provide an informal overview; the full operational semantics can be found in Appendix B of [24].

The semantics is a transition system $\mathcal{S}_{\mathbb{T},\Psi} = (\Delta, \rightarrow)$ parametrized over a set of transactional programs $\mathbb{T}$ and a consistency specification Ψ. The state ($\delta \in \Delta$) is stored as a tuple $(\Sigma, \mathsf{vis}, \mathsf{ar}, \mathcal{P})$ where Σ is the set of committed transaction instances, vis and ar are relations on Σ, and $\mathcal{P}$ is the running pool of transaction instances. The transitions are of two types : spawning a new instantiation of a transactional program $\mathcal{T} \in \mathbb{T}$ or executing a statement of a transaction instance in the running pool. When a new execution of a transaction instance begins, a subset of Σ is non-deterministically selected to be made visible to the new instance. A view of the database is constructed for the new instance based on the set of visible transactions and the ar relation (ensuring the last writer wins policy), and all queries of the transaction instance are answered on the basis of this view. At any point, any transaction instance from $\mathcal{P}$ can be non-deterministically selected for execution of its next statement. Any new event generated during the execution of a transaction instance is stored in the running pool. Finally, when a transaction instance wants to commit, it is checked whether the consistency specification (Ψ) is satisfied if the instance were to commit, and if yes, it is added to Σ. We can now define a valid abstract execution in terms of traces of the transition system:

▶ **Definition 7** (Valid execution of $\mathbb{T}$ under Ψ)**.** An abstract execution $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$ is said to be a valid execution produced by $\mathbb{T}$ under Ψ if there exists a trace $(\{\}, \{\}, \{\}, \{\}) \rightarrow^* (\Sigma, \mathsf{vis}, \mathsf{ar}, \{\})$ of the transition system $\mathcal{S}_{\mathbb{T},\Psi}$.

4 FOL Encoding

4.1 Vocabulary

Given a set of transactional programs $\mathbb{T}$ and a consistency specification Ψ we now show how to construct a formula in FOL such that any valid abstract execution χ of $\mathbb{T}$ under Ψ and its dependency graph G_χ is a satisfying model of the formula. The encoding is parametric over $\mathbb{T}$ and Ψ. We first describe the vocabulary of the encoding. We define two uninterpreted sorts τ and R, such that members of τ are transaction instances, and members of R are records. In addition, we also define a finite sort $\mathbb{T}$ which contains the transaction types, where each type is a transactional program.

The function $\Gamma : \tau \rightarrow \mathbb{T}$ associates each transaction instance with its type. For each transactional program $\mathcal{T} \in \mathbb{T}$ and for each variable $\mathtt{v} \in \mathtt{Vars}(\mathcal{T})$, the variable projection function $\rho_\mathtt{v}$ gives the value of $\mathtt{v}$ in a transaction instance. The signature of $\rho_\mathtt{v}$ depends upon the type of the variable and whether it is assigned inside a loop. First, let us consider variables which are assigned values outside any loop. In our framework, variables are of two types : a value or a set of values. Further, the value can be either an integer (e.g. the parameter `ID` of the `withdraw` transaction) or a record. Let $\mathbb{V} = \mathbb{Z} \cup R$. If $\mathtt{v}$ is a value, the $\rho_\mathtt{v}$ has the signature $\tau \rightarrow \mathbb{V}$. If $\mathtt{v}$ is a set of values, then $\rho_\mathtt{v}$ is a predicate with signature $\mathbb{V} \times \tau \rightarrow \mathbb{B}$, such that $\rho_\mathtt{v}(r, t)$ is true if r belongs to $\mathtt{v}$ in the transaction instance t.

Consider a loop of the form : `FOREACH` $\mathtt{v}_1$ `IN` $\mathtt{v}_2$ `DO` c `END`. All local variables which are assigned values inside the loop body (including $\mathtt{v}_1$) will be indexed by values in the set $\mathtt{v}_2$. Hence, if a local variable $\mathtt{v}_3$ is assigned inside the loop, and it is a value, then $\rho_{\mathtt{v}_3}$ will have the signature $\mathbb{V} \times \tau \to \mathbb{V}$. On the other hand, if $\mathtt{v}_3$ stores a set of values, then $\rho_{\mathtt{v}_3}$ will have the signature $\mathbb{V} \times \mathbb{V} \times \tau \to \mathbb{B}$, with the interpretation that $\rho_{\mathtt{v}_3}(r_1, r_2, t)$ is true if $\mathtt{v}_3$ contains r_2 in the iteration where $\mathtt{v}_1$ is $r_1 \in \mathtt{v}_2$. Similarly, nested loops will have local variables which are indexed by records in all enclosing loops.

To summarize, the signature of $\rho_{\mathtt{v}}$ is either $\mathbb{V}^{\mathcal{D}(\mathtt{v})} \times \tau \to \mathbb{V}$ or $\mathbb{V}^{\mathcal{D}(\mathtt{v})+1} \times \tau \to \mathbb{B}$. Similar to the variable projection function, the field projection function $\rho_{\mathtt{f}} : R \to \mathbb{Z}$ is defined for each field $\mathtt{f} \in \mathtt{Fields}$, such that $\rho_{\mathtt{f}}(r)$ gives the value of $\mathtt{f}$ in a record r.

We define predicates $\mathsf{WR}, \mathsf{WW}, \mathsf{RW}$ all of type $\tau \times \tau \to \mathbb{B}$ which specify the read, write and anti-dependency relations respectively between transaction instances. We also define predicates $\mathsf{WR}^R, \mathsf{RW}^R, \mathsf{WW}^R$ all of type $R \times \mathtt{Fields} \times \tau \times \tau \to \mathbb{B}$ which provide more context by also specifying the records and fields causing the dependencies. Predicates $\mathsf{vis}, \mathsf{ar}$ of type $\tau \times \tau \to \mathbb{B}$ specify the visibility and arbitration relation between transaction instances. The predicate $\mathtt{Alive} : R \times \tau \to \mathbb{B}$ indicates whether a record is `Alive` for a transaction instance.

4.2 Relating Dependences with Abstract Executions

By Lemma 4, in any abstract execution, the presence of a dependency edge between two transaction instances enforces constraints on the vis and/or ar relations between the two instances. The following formula encodes this along with basic constraints satisfied on vis and ar:

$$\begin{aligned}\varphi_{basic} = \textsc{TotalOrder}(\mathsf{ar}) \wedge \forall(t, s : \tau).\ (\mathsf{vis}(t, s) \Rightarrow \neg\mathsf{vis}(s, t)) \wedge (\mathsf{vis}(t, s) \Rightarrow \mathsf{ar}(t, s)) \\ \wedge (\mathsf{WR}(t, s) \Rightarrow \mathsf{vis}(t, s)) \wedge (\mathsf{WW}(t, s) \Rightarrow \mathsf{ar}(t, s)) \wedge (\mathsf{RW}(t, s) \Rightarrow \neg\mathsf{vis}(s, t)) \qquad (5)\end{aligned}$$

The following formula encodes a fundamental constraint involving the dependency relations on the same field of the same record due to the last writer wins nature of the database:

$$\varphi_{dep} = \bigwedge_{\mathtt{f} \in \mathtt{Fields}} \forall(t_1, t_2, t_3 : T)(r : R).\ \mathsf{WR}^R(r, \mathtt{f}, t_2, t_1) \wedge \mathsf{RW}^R(r, \mathtt{f}, t_1, t_3) \Rightarrow \mathsf{WW}^R(r, \mathtt{f}, t_2, t_3)$$

Finally, the consistency specification Ψ can be directly encoded using the relations and functions defined in our vocabulary (we denote this formula by φ_Ψ).

4.3 Relating dependences with transactional programs

The presence of a dependency edge between two transaction instances places constraints on the type of transactional programs generating the instances and their parameters. To automatically infer these constraints, we use the following strategy : if there is a dependency edge between two instances, then there must exist SQL statements in both transactions which access a common record.

To encode this, we first extract the conditions under which a SQL statement in a transactional program can be executed. By performing a simple syntactic analysis over the code of a transaction $\mathcal{T}$, we obtain a mapping $\Lambda_{\mathcal{T}}$ from each SQL statement in $\mathtt{Stmts}(\mathcal{T})$ to a conjunction of enclosing `IF` conditionals (the complete algorithm can be found in Appendix A of [24]).

The FOL encoding of all conditionals in a program and all `WHERE` clauses in a SQL statement is constructed by replacing variables and fields with the corresponding variable projection and field projection functions respectively. A representative set of rules for this

$$
\begin{array}{lcll}
[\![\mathtt{v} = \mathtt{NULL}]\!]_t & = & (\exists(r_1,\ldots,r_{\mathcal{D}(\mathtt{v})} : R).\ \bigwedge_{i=1}^{\mathcal{D}(\mathtt{v})} \mathcal{V}([\![r_i \in \mathtt{LVar}(\mathtt{v},i)]\!]_t), & \mathsf{fresh}(r_1,\ldots,r_{\mathcal{D}(\mathtt{v})},r) \\
 & & \forall(r:R).\neg\rho_{\mathtt{v}}(r_1,\ldots,r_{\mathcal{D}(\mathtt{v})},r,t)) & \\
[\![r \in \mathtt{v}]\!]_t & = & (\exists(r_1,r_2,\ldots,r_{\mathcal{D}(\mathtt{v})} : R).\ \bigwedge_{i=1}^{\mathcal{D}(\mathtt{v})} \mathcal{V}([\![r_i \in \mathtt{LVar}(\mathtt{v},i)]\!]_t), & \mathsf{fresh}(r_1,\ldots,r_{\mathcal{D}(\mathtt{v})}) \\
 & & \rho_{\mathtt{v}}(r_1,\ldots,r_{\mathcal{D}(\mathtt{v})},r,t)) & \\
[\![\mathtt{v}_1 \in \mathtt{v}_2]\!]_t & = & (\varphi_1 \wedge \varphi_2, \psi_2) & [\![\mathtt{v}_1]\!]_t = (\varphi_1,\psi_1) \\
 & & & [\![\psi_1 \in \mathtt{v}_2]\!]_t = (\varphi_2,\psi_2) \\
[\![\mathtt{f} \odot e]\!]_{t,r} & = & \begin{cases} (\varphi, \rho_{\mathtt{f}}(r) \odot \psi) & \text{if } \mathtt{f} \cup \mathcal{F}(e) \subseteq \mathtt{PK} \\ (\mathsf{true},\mathsf{true}) & \text{otherwise} \end{cases} & \\
 & & & [\![e]\!]_{t,r} = (\varphi,\psi)
\end{array}
$$

Figure 4 Encoding conditionals and `WHERE` clauses.

encoding are shown in Fig. 4. For conditionals ϕ used in `IF` statements, we use the notation $[\![\phi]\!]_t$ to describe the FOL encoding specialized to transaction instance t. The interpretation is that $[\![\phi]\!]_t$ is satisfiable only if the conditional ϕ is true in the transaction instance t. If ϕ is inside a loop, then $[\![\phi]\!]_t$ must be satisfiable if ϕ is true in any arbitrary iteration of the enclosing loop(s) in t. For this reason, $[\![\phi]\!]_t$ is actually represented as a tuple (φ,ψ), where φ chooses any arbitrary iteration of enclosing loops, and the formula ψ is the value of the conditional in that iteration. We define an evaluation function $\mathcal{V}(\varphi,\psi) = \varphi \wedge \psi$ which gives the final FOL encoding.

The formula φ chooses an iteration by instantiating records belonging to loop variables of all enclosing loops. For example, consider the encoding of $\mathtt{v} = \mathtt{NULL}$. Here, φ instantiates a record belonging to the loop variable of every enclosing loop of $\mathtt{v}$ (encoded as $\mathcal{V}([\![r_i \in \mathtt{LVar}(\mathtt{v},i)]\!]_t)$), and ψ encodes that $\rho_{\mathtt{v}}$ in the chosen iteration is false for every record. Similarly, in the encoding of $[\![r \in \mathtt{v}]\!]_t$, $\rho_{\mathtt{v}}$ must be true for the record r. In the encoding of $[\![\mathtt{v}_1 \in \mathtt{v}_2]\!]_t$, we first obtain the value of $\mathtt{v}_1$ (the second term in the tuple $[\![\mathtt{v}_1]\!]_t$), and then check whether it is present in $\mathtt{v}_2$.

A similar procedure is used to obtain the encoding of the `WHERE` clauses used inside SQL statements. Since `WHERE` clauses are evaluated on records, the encoding is specialized on both records and transaction instances, for which we use the notation $[\![\phi]\!]_{t,r}$. The interpretation is that $[\![\phi]\!]_{t,r}$ is satisfiable only if ϕ is true for transaction instance t on record r. The encoding replaces field accesses with the corresponding field projection function applied on r. Note that the field projection function is only used for primary key fields which are accessed within `WHERE` clauses (expressed as $\mathcal{F} \subseteq \mathtt{PK}$). The complete encoding for all types of conditionals and `WHERE` clauses can be found in the Appendix A of [24].

As stated earlier, our strategy is to encode the necessary condition for a dependency edge based on the access of a common record. For each pair of transaction types $\mathcal{T}_1, \mathcal{T}_2 \in \mathbb{T}$, each dependency type $\mathcal{R} \in \{\mathsf{WR},\mathsf{RW},\mathsf{WW}\}$, and each pair of SQL statements $\mathtt{c}_1 \in \mathtt{Stmts}(\mathcal{T}_1), \mathtt{c}_2 \in \mathtt{Stmts}(\mathcal{T}_2)$, we compute a necessary condition $\eta^{\mathcal{R}\rightarrow,\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2)$ for dependency $\mathcal{R}$ to exist between instances t_1 and t_2 of types $\mathcal{T}_1$ and $\mathcal{T}_2$ due to statements $\mathtt{c}_1$ and $\mathtt{c}_2$ respectively. The following formula encodes the fact that a dependency between two transaction instances can be caused due to a dependency between any two SQL statements in those transactions:

$$
\varphi_{\mathcal{R}\rightarrow,\mathcal{T}_1,\mathcal{T}_2} \triangleq \forall(t_1,t_2:\tau).(\Gamma(t_1) = \mathcal{T}_1 \wedge \Gamma(t_2) = \mathcal{T}_2 \wedge \mathcal{R}(t_1,t_2)) \Rightarrow \bigvee_{\substack{\mathtt{c}_1 \in \mathtt{Stmts}(\mathcal{T}_1) \\ \mathtt{c}_2 \in \mathtt{Stmts}(\mathcal{T}_2)}} \eta^{\mathcal{R}\rightarrow,\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2)
$$

The general format of $\eta^{\mathcal{R}\rightarrow,\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2)$ is a conjunction of the conditionals required to execute the statements $\mathtt{c}_1$ and $\mathtt{c}_2$ (i.e. $\Lambda_{\mathcal{T}_1}(\mathtt{c}_1)$ and $\Lambda_{\mathcal{T}_2}(\mathtt{c}_2)$) in t_1 and t_2 resp. and the `WHERE` clauses of the two statements evaluated on some record r. If they can never access the

same field of the same record, then $\eta^{\mathcal{R}\rightarrow,\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2)$ is simply false. While this is the general format of the clauses, in addition, we can also infer more information depending upon the type of the SQL statements. To illustrate this we present a sample rule below:

$$\frac{\begin{array}{c}\mathtt{c}_1 \equiv \texttt{SELECT MAX f AS v WHERE } \phi_1 \quad \mathtt{c}_2 \equiv \texttt{UPDATE SET f = e WHERE } \phi_2 \\ \mathtt{c}_1 \in \texttt{Stmts}(\mathcal{T}_1) \quad \mathtt{c}_2 \in \texttt{Stmts}(\mathcal{T}_2) \quad \Gamma(t_1)=\mathcal{T}_1 \quad \Gamma(t_2)=\mathcal{T}_2 \quad [\![\mathtt{v}]\!]_{t_1}=(\varphi_1,\psi_1) \quad [\![\mathtt{e}]\!]_{t_2}=(\varphi_2,\psi_2)\end{array}}{\begin{array}{c}\eta^{\mathsf{RW}\rightarrow,\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2) = (\exists r.\ \mathcal{V}([\![\Lambda_{\mathcal{T}_1}(\mathtt{c}_1)]\!]_{t_1}) \wedge \mathcal{V}([\![\phi_1]\!]_{t_1,r}) \wedge \mathcal{V}([\![\Lambda_{\mathcal{T}_2}(\mathtt{c}_2)]\!]_{t_2}) \wedge \mathcal{V}([\![\phi_2]\!]_{t_2,r}) \wedge \\ \texttt{Alive}(r,t_2) \wedge \varphi_1 \wedge \varphi_2 \wedge \psi_1 < \psi_2)\end{array}}$$

The rule encodes a necessary condition for an anti-dependency to exist from a `SELECT MAX` to a `UPDATE` statement. First, it encodes that the conflicting SQL statements actually execute in their respective transactions and there is a common record which satisfies the `WHERE` clauses of both statements. `SELECT MAX` selects the record with the maximum value in the field `f` among all records that satisfy ϕ_1, and stores the value in variable `v`. If there is an anti-dependency from `SELECT MAX` to `UPDATE`, then the updated value must be greater than the output of `SELECT MAX`, because otherwise, the update does not affect the output of `SELECT MAX`.

In addition, some transaction instances may be guaranteed to execute certain SQL statements, which forces the presence of a dependency edge between them. For example, if two transaction instances are guaranteed to update the same field of a record, then there must be a WW dependeny between them. For each pair of transaction types $\mathcal{T}_1, \mathcal{T}_2 \in \mathbb{T}$, each dependency type $\mathcal{R} \in \{\mathsf{WR}, \mathsf{RW}, \mathsf{WW}\}$, and each pair of SQL statements $\mathtt{c}_1 \in \texttt{Stmts}(\mathcal{T}_1), \mathtt{c}_2 \in \texttt{Stmts}(\mathcal{T}_2)$, we compute a condition $\eta^{\rightarrow\mathcal{R},\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2)$ which forces the dependency $\mathcal{R}$ to exist between instances t_1 and t_2 of types $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively due to $\mathtt{c}_1$ and $\mathtt{c}_2$. The following formula encodes this:

$$\varphi_{\rightarrow\mathcal{R},\mathcal{T}_1,\mathcal{T}_2} \triangleq \forall t_1,t_2.(\Gamma(t_1)=\mathcal{T}_1 \wedge \Gamma(t_2)=\mathcal{T}_2 \wedge \bigvee_{\substack{\mathtt{c}_1\in\texttt{Stmts}(\mathcal{T}_1)\\ \mathtt{c}_2\in\texttt{Stmts}(\mathcal{T}_2)}} \eta^{\rightarrow\mathcal{R},\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2)) \Rightarrow \mathcal{R}(t_1,t_2)$$

$$\frac{\begin{array}{c}\mathtt{c}_1 \equiv \texttt{UPDATE SET f} = \mathtt{e}_1 \texttt{ WHERE } \phi_1 \quad \mathtt{c}_2 \equiv \texttt{UPDATE SET f} = \mathtt{e}_2 \texttt{ WHERE } \phi_2 \\ \mathtt{c}_1 \in \texttt{Stmts}(\mathcal{T}_1) \quad \mathtt{c}_2 \in \texttt{Stmts}(\mathcal{T}_2) \quad \Gamma(t_1)=\mathcal{T}_1 \quad \Gamma(t_2)=\mathcal{T}_2\end{array}}{\begin{array}{c}\eta^{\rightarrow\mathsf{WW},\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2) = (\exists r.\ \mathcal{V}([\![\Lambda_{\mathcal{T}_1}(\mathtt{c}_1)]\!]_{t_1}) \wedge \mathcal{V}([\![\phi_1]\!]_{t_1,r}) \wedge \mathcal{V}([\![\Lambda_{\mathcal{T}_1}(\mathtt{c}_1)]\!]_{t_2}) \wedge \mathcal{V}([\![\phi_2]\!]_{t_2,r}) \wedge \\ \texttt{Alive}(r,t_1) \wedge \texttt{Alive}(r,t_2) \wedge \mathsf{ar}(t_1,t_2))\end{array}}$$

$\eta^{\rightarrow\mathcal{R},\mathcal{T}_1,\mathcal{T}_2}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2)$ is computed in the same manner as $\eta^{\mathcal{R},\mathcal{T}_1,\mathcal{T}_2\rightarrow}_{\mathtt{c}_1,\mathtt{c}_2}(t_1,t_2)$. As an example consider the above rule. Two `UPDATE` statements modifying the same field are guaranteed to cause a WW dependency if both statements actually execute in their respective transactions, and there exists a common record accessed by both statements which is `Alive` to both transactions.

In addition, there are some auxiliary facts which are satisfied by all abstract executions (which we encode as the formula φ_{aux}) such as a record present in the output variable of a `SELECT` query must satisfy the `WHERE` clause of the query, the value of the iterator variable in a loop must belong to the loop variable, etc. For more details, we again refer to the Appendix. The final encoding is defined as follows:

$$\varphi_{\mathbb{T},\Psi} \triangleq \varphi_{basic} \wedge \varphi_{dep} \wedge \bigwedge_{\mathcal{R}\in\{\mathsf{WR},\mathsf{RW},\mathsf{WW}\}} \bigwedge_{\mathcal{T}_1,\mathcal{T}_2\in\mathbb{T}} (\varphi_{\mathcal{R}\rightarrow,\mathcal{T}_1,\mathcal{T}_2} \wedge \varphi_{\rightarrow\mathcal{R},\mathcal{T}_1,\mathcal{T}_2}) \wedge \varphi_\Psi \wedge \varphi_{aux} \quad (6)$$

▶ **Theorem 8.** *Given a set of transactional programs $\mathbb{T}$ and a consistency specification Ψ, for any valid abstract execution $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$ generated by $\mathbb{T}$ under Ψ and its dependency*

graph G_χ, there exists a satisfying model of the formula $\varphi_{\mathbb{T},\Psi}$ with $\tau = \Sigma$ and the binary predicates $\mathsf{vis}, \mathsf{ar}, \mathsf{WR}, \mathsf{RW}, \mathsf{WW}$ *being equal to the corresponding relations in χ and G_χ.*

Note that $\varphi_{\mathbb{T},\Psi}$ is always satisfiable, since the empty abstract execution is a satisfying model.

5 Applications

5.1 Bounded Anomaly Detection

By Theorem 6, any execution which violates serializability must have a cycle in its dependency graph. We can directly instantiate a dependency graph which contains a cycle of bounded length and then ask for a satisfying model of the formula built in the previous section which contains the cycle. We introduce a new predicate $D : \tau \times \tau \to \mathbb{B}$ which represents the presence of any dependency edge between two transaction instances : $\varphi_D \triangleq \forall(t_1, t_2 : \tau).\mathsf{D}(t_1, t_2) \Leftrightarrow (t_1 = t_2) \vee \mathsf{WR}(t_1, t_2) \vee \mathsf{RW}(t_1, t_2) \vee \mathsf{WW}(t_1, t_2)$. A cycle of length less than or equal to k can now be directly encoded as follows: $\varphi_{\text{Cycle,k}} \triangleq \exists t_1, \ldots, t_k.\ \bigwedge_{i=1}^{k-1} \mathsf{D}(t_i, t_{i+1}) \wedge \mathsf{D}(t_k, t_1) \wedge (t_1 \neq t_k)$.

▶ **Theorem 9.** *Given a set of transactional programs $\mathbb{T}$ and a consistency specification Ψ, if $\varphi_{\mathbb{T},\Psi} \wedge \varphi_D \wedge \varphi_{\text{Cycle,k}}$ is* UNSAT*, then all valid abstract executions produced by $\mathbb{T}$ under Ψ of length less than or equal to* k *are serializable.*

5.2 Verifying Serializability: The Shortest Path Approach

We propose a condition, which can be also be encoded in FOL, and which if satisfied would imply that it is enough to check for violations of bounded length to prove the absence of violations of any arbitrary length.

The condition is based on the simple observation that any long path in the dependency graph could induce a short path due to chords among the nodes in the path (as demonstrated in the example in §2). This would imply that any long cycle would also induce a short cycle, and hence lack of short cycles would imply the lack of longer cycles. To check for this condition, we encode a shortest path of length k in the dependency graph and then ask whether there is a satisfying model:

$$\varphi_{\text{Shortest Path,k}} \triangleq \exists t_1, \ldots, t_k, t_{k+1}.\ \bigwedge_{i=1}^{k} \mathsf{D}(t_i, t_{i+1}) \wedge \bigwedge_{i=1}^{k-1} \bigwedge_{j=i+2}^{k+1} \neg\mathsf{D}(t_i, t_j) \wedge \bigwedge_{1 \leq i < j \leq k+1} t_i \neq t_j$$

The condition instantiates a path of length k in the dependency graph and also asserts the absence of any chord, which implies that the path is shortest. If there does not exist a shortest path of length k, then there also cannot exist a shortest path of greater length, because if not, such a path would necessarily contain a shortest path of length k. Now, it is enough to check for cycles of length less than or equal to k, because any longer cycle would contain a path of length at least k, which would imply the presence of a shorter path and thus a cycle of length less than or equal to k.

▶ **Theorem 10.** *Given a set of transactional programs $\mathbb{T}$ and a consistency specification Ψ, if both $\varphi_{\mathbb{T},\Psi} \wedge \varphi_D \wedge \varphi_{\text{Shortest Path,k}}$ and $\varphi_{\mathbb{T},\Psi} \wedge \varphi_D \wedge \varphi_{\text{Cycle,k}}$ are* UNSAT*, then all valid abstract executions produced by $\mathbb{T}$ under Ψ are serializable.*

5.3 Verifying Serializability: An Inductive Approach

We now present an alternative approach to verifying serializability which uses the transitivity and irreflexivity of the ar relation to show lack of cycles. In this approach, our goal is to show that if there is a path in the dependency graph from t_1 to t_2, then $t_1 \xrightarrow{\mathsf{ar}} t_2$. By the irreflexivity of ar, this would imply that there cannot be a cycle in the dependency graph. Since paths can be of arbitrary length, we will use the transitivity of ar and an inductive argument to obtain a simple condition which can be encoded in FOL.

▶ **Lemma 11.** *Given a set of transactional programs $\mathbb{T}$, a consistency specification Ψ and a subset of programs $\mathbb{T}' \subseteq \mathbb{T}$,* ***if*** *for all valid executions χ and their dependency graphs G_χ, the following conditions hold:*

1. *if $\sigma_1 \to \sigma_2$ in G_χ and $\Gamma(\sigma_1) \in \mathbb{T}'$, then $\sigma_1 \xrightarrow{\mathsf{ar}} \sigma_2$*
2. *if $\sigma_1 \to \sigma_2 \to \sigma_3$ in G_χ, then either $\sigma_1 \xrightarrow{\mathsf{ar}} \sigma_3$ or $\sigma_2 \xrightarrow{\mathsf{ar}} \sigma_3$*

then *all valid executions which contain at least one instance of a program in $\mathbb{T}'$ are serializable.*

The proof uses an inductive argument to show that if there is path from σ_1, an instance of a program in $\mathbb{T}'$ to any other instance σ_2, then $\sigma_1 \xrightarrow{\mathsf{ar}} \sigma_2$. This would imply that any instance of $\mathbb{T}'$ cannot be present in a cycle. The above conditions can be directly encoded in FOL:

$$\varphi_{\text{Inductive},\mathbb{T}'} \triangleq (\exists(t_1, t_2 : \tau).\ \Gamma(t_1) \in \mathbb{T}' \wedge \mathsf{D}(t_1, t_2) \wedge t_1 \neq t_2 \wedge \neg\mathsf{ar}(t_1, t_2)) \vee$$
$$(\exists(t_1, t_2, t_3 : \tau).\mathsf{D}(t_1, t_2) \wedge \mathsf{D}(t_2, t_3) \wedge \bigwedge_{1 \leq i < j \leq 3} t_i \neq t_j \wedge \neg\mathsf{ar}(t_1, t_3) \wedge \neg\mathsf{ar}(t_2, t_3)) \tag{7}$$

▶ **Theorem 12.** *Given a set of programs $\mathbb{T}$ and a consistency specification Ψ, if $\varphi_{\mathbb{T},\Psi} \wedge \varphi_D \wedge \varphi_{\text{Inductive},\mathbb{T}'}$ is* UNSAT*, then all valid executions of $\mathbb{T}$ under Ψ which contains at least one instance of a program in $\mathbb{T}'$ are serializable.*

If $\mathbb{T}' = \mathbb{T}$, then all valid executions of $\mathbb{T}$ are serializable, otherwise, we can focus only on programs in $\mathbb{T} \setminus \mathbb{T}'$, and re-apply the technique with $\varphi_{\mathbb{T}',\Psi} \wedge \varphi_D \wedge \varphi_{\text{Inductive},\mathbb{T}''}$ for $\mathbb{T}'' \subseteq \mathbb{T}'$. In the next section, we show how we use this technique to verify serializability of TPC-C, a real-world database benchmark.

6 Case Studies

We have developed a tool called ANODE which takes a set of programs written in the language presented in §3.1 and a consistency specification and uses the encoding rules presented in §4 to automatically generate an FOL encoding. We use the Z3 SMT solver to determine the satisfiabiliy of the generated formulae. In order to evaluate the effectiveness of our approach, we have applied the proposed technique on TPC-C [1], a well-known Online Transaction Processing (OLTP) benchmark widely used in the database community, and a Courseware application (used in [19]) which is a representative of course registration systems used in universities.

TPC-C. TPC-C has a complex database schema with 9 tables, and complex application logic in its 5 transactions. The transactions contain loops and conditionals, have multiple parameters and behave differently depending upon the values of the parameters; they also use complex queries such as `SELECT MIN` and `SELECT MAX`. To the best of our knowledge, this is the first automated static analysis for validating serializability of TPC-C under weak consistency.

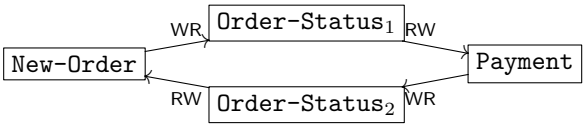

Figure 5 Long fork anomaly in TPC-C under PSI.

Under eventual consistency, TPC-C has a number of 'lost update' anomalies, similar to the anomaly in the banking application described in §2. These anomalies are small in length and were automatically detected using encoding presented in §5.1 (with $k = 2$) . To get rid of these anomalies, we upgraded the consistency specification to PSI [26]. Under PSI, we did not find any anomalies for $k = 2$ or $k = 3$, but for $k = 4$, the 'long fork' anomaly involving the New-Order, Payment and Order-Status transactions was discovered, as shown in Fig. 5.

This anomaly happens because the `New-Order` and `Payment` transactions update two different tables (`Order` and `Customer` table resp.) while the `Order-Status` transaction reads both those tables. Since there is no synchronization between `New-Order` and `Payment` transactions, it is possible for `Order-Status`$_1$ to see the update of `New-Order` but not `Payment`, and the vice versa for `Order-Status`$_2$. We also discovered a similar anomaly involving two instances of `New-Order` and two instances of `Stock-level` transactions.

To get rid of these anomalies, we further upgraded the consistency level to Snapshot Isolation (SI), after which we did not find any anomalies for $k = 4$. We then turned our attention to verifying serializability of TPC-C under SI. We first tried the Shortest Path approach (which worked well for the banking application), but we were able to discover a long path (which can be arbitrarily extended) without any chords. Next, we tried the inductive approach, which was successful in proving serializability of TPC-C. Specifically, with $\mathbb{T}' = \{\texttt{New-Order}, \texttt{Payment}\}$, the formula $\varphi_{\text{Inductive},\mathbb{T}'}$ was shown to be UNSAT, and with the remaining 3 transactions $\varphi_{\text{Inductive},\{\texttt{Delivery}\}}$ was UNSAT. The remaining two transactions do not have any dependencies between them, which implies that all executions of TPC-C under SI are serializable.

Courseware. The Courseware application maintains a database of courses and students, and provides the functionality of adding/removing students and courses, and enrolling students into courses subject to course capacities. Under EC, the following anomalies were discovered by our encoding : (1) two concurrent `Enroll` transactions may enroll students beyond the course capacity, (2) two courses with the same name or two students with the same name may be registered, (3) a student may be enrolled in a course which is being concurrently removed, or the student is being concurrently removed. Note that all these anomalies were discovered for $k = 2$.

In order to remove these violations, we upgraded the consistency model in a number of ways : the `Enroll` transaction was upgraded to PSI, while selective serializability was used for two instances of `AddCourse` and `AddStudent`, and for instances of `Enroll` and `RemCourse`, `Enroll` and `RemStudent`. While these upgrades took care of the above mentioned anomalies, we discovered a new long fork anomaly (for $k = 4$) as shown in Fig. 6. Here, two `Enroll` transactions trying to enroll a student (`s`) into a course (`c`) see conflicting views of the database, with one `Enroll` witnessing the student but not the course, and vice versa for the other. We note that while this is an actual serializability violation, it is completely harmless as both transactions which witness inconsistent database states ultimately fail, so that the final database state is the same as that which manifests at the end of an execution in which

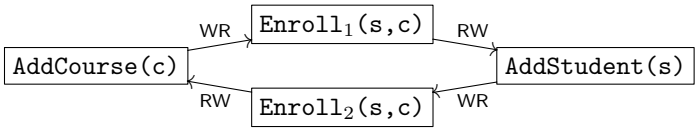

Figure 6 Long fork anomaly in the Courseware application under PSI.

the effects of neither of the two enroll transactions occur. This is a limitation of our analysis as it does not provide any way to ignore harmless serializability violations. We plan to address this issue as part of future work.

In order to remove this violation, we upgraded the consistency level of `Enroll` to SI, after which we did not find any anomalies. Next, we moved to verification, and here we were successfully able to use the Shortest Path approach and prove that there does not exist a shortest path in any dependency graph of the Courseware application of length greater than or equal to 8. Along with the fact there does not exist any cycle of length less than or equal to 8, this implies that any execution of the application is serializable. In all instances, the solver produced its output in a few (< 10) seconds.

7 Related Work and Conclusions

Serializability is a well-studied problem in the database community, but there is a lack of static automated techniques to check for serializability of database applications. Early work by Fekete *et al.* [17] and Jorwekar *et al.* [20] proposed lightweight syntactic analyses to check for serializability under SI in centralized databases, by looking for dangerous structures in the static dependency graph of an application (which is an over-approximation of all possible dynamic dependency graphs). Several recent works [5, 12, 13, 14, 30, 28] have continued along this line, by deriving different types of dangerous structures in dependency graphs that are possible under different weak consistency mechanisms, and then checking for these structures on static dependency graphs.

However, static dependency graphs are highly imprecise representations of actual executions, and any analysis reliant on these graphs is likely to yield a large number of false positives. Indeed, recent effortsin this space [5, 13, 14] recognize this and propose complex conditions to reduce false positives for specific consistency mechanisms, but these works do not provide any automated methodology to check those conditions on actual programs. Further, application logic could prevent these harmful structures from manifesting in actual executions, for example as in TPC-C, which has a harmful structure in its static dependency graph under SI, but which does not appear in any dynamic dependency graph. In our work, we precisely model the application logic and the consistency specification using FOL, so that the solver would automatically derive harmful structures which are possible under the given consistency specification and search for them in actual dependency graphs taking application logic into account.

[8] proposes a static analysis for serializability under causal consistency by constructing actual dependency graphs with cycles using a FOL encoding. While this work is similar to ours in spirit, their notion of serializability is stronger than ours, since they allow transactions to be grouped together in sessions, with the serial order forced to accommodate the chosen session order. While this eases the task of verifying serializability for unbounded executions, it also results in a large number of harmless serializability violations, for which they propose

various *ad hoc* filtering approaches. Further, their focus is on programs operating on high-level data types rather than SQL programs, and their analysis is not parametric on consistency specifications.

There are also dynamic anomaly detection techniques [29, 11, 7] which either build the dependency graphs at run-time and check for cycles, or analyze the trace of events after execution. These approaches do not provide any guarantee that all anomalies will be detected, even for bounded executions. A number of approaches have been proposed recently [25, 19, 21, 15] which attempt to verify that high-level application invariants are preserved under weak consistency. These approaches are also parametric on consistency specifications, but they are not completely automated as they require correctness conditions in the form of invariants from the user, and they do not tackle serializability.

To conclude, in this paper we take the first step towards building a precise, fully automated static analysis for veifying serializability of database applications under weak consistency. We leverage the acyclic dependency graph based characterization of serializability and the framework of abstract executions to develop a FOL based analysis which is parametric on the consistency specification. We show how our approach can be used to detect bounded anomalies, and to verify serializability under specific conditions for unbounded executions. We show the practicality of our approach by successfully applying it on several realistic database benchmarks.

References

1 Tpc-c benchmark. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`. Online; Accessed 20 April 2018.

2 Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78, 2000. `doi:10.1109/ICDE.2000.839388`.

3 Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. *ACM Trans. Database Syst.*, 41(3):15:1–15:45, 2016. `doi:10.1145/2909870`.

4 Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 1–10, 1995. `doi:10.1145/223784.223785`.

5 Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 7:1–7:15, 2016. `doi:10.4230/LIPIcs.CONCUR.2016.7`.

6 Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

7 Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 458–472, 2017. URL: `http://dl.acm.org/citation.cfm?id=3009895`.

8 Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. Static serializability analysis for causal consistency. In *Proceedings of the 39th ACM SIGPLAN Conference on*

Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018, pages 90–104, 2018. doi:10.1145/3192366.3192415.

9 Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 67–86, 2012. doi:10.1007/978-3-642-28869-2_4.

10 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 568–590, 2015. doi:10.4230/LIPIcs.ECOOP.2015.568.

11 Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, 2009. doi:10.1145/1620585.1620587.

12 Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 58–71, 2015. doi:10.4230/LIPIcs.CONCUR.2015.58.

13 Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 55–64, 2016. doi:10.1145/2933057.2933096.

14 Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Algebraic laws for weak consistency. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, pages 26:1–26:18, 2017. doi:10.4230/LIPIcs.CONCUR.2017.26.

15 Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 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*, pages 73–82, 2017. doi:10.1145/3087801.3087802.

16 Alan Fekete. Allocating isolation levels to transactions. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, pages 206–215, 2005. doi:10.1145/1065167.1065193.

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

18 Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. doi:10.1145/564585.564601.

19 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 371–384, 2016. doi:10.1145/2837614.2837625.

20 Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1263–1274, 2007. URL: http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf.

21 Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone together: compositional reasoning and inference for weak isolation. *PACMPL*, 2(POPL):27:1–27:34, 2018. doi:10.1145/3158115.

22 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 401–416, 2011. doi:10.1145/2043556.2043593.

23 Madan Musuvathi. Systematic concurrency testing using CHESS. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), PADTAD 2008, Seattle, Washington, USA, July 20-21, 2008*, page 10, 2008. doi:10.1145/1390841.1390851.

24 Kartik Nagar and Suresh Jagannathan. Automated Detection of Serializability Violations under Weak Consistency (Extended Version). arXiv:1806.08416.

25 K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 413–424, 2015. doi:10.1145/2737924.2737981.

26 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400, 2011. doi:10.1145/2043556.2043592.

27 Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP 13, Farmington, PA, USA, November 3-6, 2013*, pages 309–324, 2013. doi:10.1145/2517349.2522731.

28 Todd Warszawski and Peter Bailis. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 5–20, 2017. doi:10.1145/3035918.3064037.

29 Kamal Zellag and Bettina Kemme. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *VLDB J.*, 23(1):147–172, 2014. doi:10.1007/s00778-013-0318-x.

30 Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 276–291, 2013. doi:10.1145/2517349.2522729.