CLOTHO: Directed Test Generation for Weakly Consistent Database Systems

KIA RAHMANI, Purdue University, USA
KARTIK NAGAR, Purdue University, USA
BENJAMIN DELAWARE, Purdue University, USA
SURESH JAGANNATHAN, Purdue University, USA

Relational database applications are notoriously difficult to test and debug. Concurrent execution of database transactions may violate complex structural invariants that constraint how changes to the contents of one (shared) table affect the contents of another. Simplifying the underlying concurrency model is one way to ameliorate the difficulty of understanding how concurrent accesses and updates can affect database state with respect to these sophisticated properties. Enforcing serializable execution of all transactions achieves this simplification, but it comes at a significant price in performance, especially at scale, where database state is often replicated to improve latency and availability.

To address these challenges, this paper presents a novel testing framework for detecting serializability violations in (SQL) database-backed Java applications executing on weakly-consistent storage systems. We manifest our approach in a tool named CLOTHO, that combines a static analyzer and a model checker to generate abstract executions, discover serializability violations in these executions, and translate them back into concrete test inputs suitable for deployment in a test environment. To the best of our knowledge, CLOTHO is the first automated test generation facility for identifying serializability anomalies of Java applications intended to operate in geo-replicated distributed environments. An experimental evaluation on a set of industry-standard benchmarks demonstrates the utility of our approach.

CCS Concepts: • **Software and its engineering → Formal software verification**;

Additional Key Words and Phrases: Static Analysis, Serializability, Weak Consistency

ACM Reference Format:
Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systems. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2019), 28 pages.

1 INTRODUCTION

Realistic SQL-based databases typically have sophisticated structural relationships (schemas), and clients must ensure that every possible use of a database operation preserves these relationships. Testing that client applications do so is particularly challenging, as the operations they perform depends on the control structure and initial state of the application. Generating appropriate test inputs to discover violations of these relationships necessarily requires reasoning about the behaviors of these operations in the context of the program's execution. Further complicating matters is that the initial state of the database itself needs to be chosen so that tests expose interesting integrity and assertion violations. One important simplification that helps reduce the complexity of developing a useful testing strategy for these applications is to treat database transactions as *serializable*, restricting the set of interleavings that must be considered to those that maintain transaction atomicity and isolation.

Authors' addresses: Kia Rahmani, Department of Computer Science, Purdue University, West Lafayette, Indiana, USA, rahmank@purdue.edu; Kartik Nagar, Department of Computer Science, Purdue University, USA, nagark@purdue.edu; Benjamin Delaware, Department of Computer Science, Purdue University, USA, bendy@purdue.edu; Suresh Jagannathan, Department of Computer Science, Purdue University, USA, suresh@cs.purdue.edu.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

As an example, consider the transactions `txnWrite` and `txnRead` from the simple pseudocode application presented in Figure 1. Transaction `txnWrite` updates rows from `table1` and `table2` with the value `val` where both rows are accessed using the given `id`. Similarly, transaction `txnRead` reads rows from both tables that are associated with the given `id` and asserts that they are equal. This is a very common pattern in database applications, e.g. when backup versions of tables are maintained. Under the guarantees afforded by serializability, an automated testing framework need only consider interleavings of entire transactions in order to completely cover the state of possible valid concurrent executions. This is because interleavings that expose intermediate transaction states (failure of *atomicity*), or which allow an already executing transaction to observe the effects of other completed transactions (failure of *isolation*) are prohibited, and hence the space of executions that must be considered to validate an application's correctness is greatly reduced. While conceptually simple to reason about, maintaining serializability imposes strong constraints on the underlying database and storage system that hinders performance at scale. In large distributed environments, especially those use replication to improve latency, the cost of enforcing serializability can be prohibitive, requiring global synchronization to enforce atomic updates and maintain a single global view of database state. Consequently, modern-day cloud systems often implement and support storage abstractions that provide significantly weaker consistency and isolation guarantees than those necessary to enforce serializable executions of database programs.

```
txnWrite(id,val):
 write(table1,id,val)
 write(table2,id,val)
```

(a) two writes

```
txnRead(id):
 v1=read(table1,id)
 v2=read(table2,id)
 assert:(v1=v2)
```

(b) two reads

Fig. 1

For example, when the transactions in the above example are executed on a replicated database systems offering eventual consistency (EC), each individual read and write operation can be executed independently making the space of possible executions significantly larger than before. Not surprisingly, new anomalous executions are also introduced under such weak semantics. These executions can trigger assertion violations that would otherwise not happen, for example, if both read operations occur between the two writes and vice versa. To make matters worse, these violations are only triggered if the initial state of the database for both rows differs from what the `txnWrite` transaction is writing. Given these complexities, the obvious question that arises is whether a database application, originally written assuming the underlying storage system provides strong consistency and isolation guarantees, will still operate correctly (i.e., behave as though serializability were still maintained) in weaker environments where these guarantees are no longer assured.

Given these concerns, the need for a practical automated testing framework for database applications executing in weakly consistent environments becomes is particularly acute. This paper takes an important first step towards addressing this need by presenting a tool (CLOTHO) that systematically and efficiently explores abstract executions of database-backed Java programs for serializability violations, assuming a weakly-consistent storage abstraction. To do so, we employ a bounded model-checking serializability violation detector that is parameterized by a specification of the underlying storage model. Abstract executions capture various visibility and ordering relations among read and write operations on the database generated by queries; the structure of these relations is informed by the underlying data consistency model. Potential serializability violations in an abstract execution manifest as cycles in a dependency graph induced by these relations. When such violations are discovered, CLOTHO synthesizes concrete tests that can be used to drive executions that manifest the problem for assessment by application developers.

Through our experimental evaluation, we demonstrate that CLOTHO can efficiently, automatically, and reliably detect and concretely manifest serializability anomalies of database programs executing on top of weakly-consistent data stores. An important by-product of its design is that CLOTHO does not bake-in any specific consistency or isolation assumptions about the underlying storage infrastructure, allowing users to strengthen visibility and ordering constraints as desired.

The contributions of this paper are as follows:

(1) We present an abstract representation and a novel encoding of database applications capable of modeling a diverse range of weak guarantees offered by real-world distributed databases.
(2) We develop a search algorithm capable of effectively identifying unique serializability anomalies accompanied by a complete execution history that leads to that anomaly.
(3) We bridge the gap between abstract and concrete executions by developing a front-end compiler and a test administration environment for database-backed Java applications that admits a wide range distributed database system behaviors and features.
(4) We demonstrate the utility of our approach on a set of standard benchmarks and present a comprehensive empirical study comparing CLOTHO's effectiveness against a tuned dynamic (random) testing approach.

The remaining of this paper is organized as follows. A brief overview of our solution using a motivational example is presented in Section 2. Sections 3 and 4 respectively define the operational semantics of our abstract programming model and the notion anomalies in that model. Section 5 presents our precise encoding of anomalies in database applications. Section 6 introduces our effective search algorithm and the implementation details of CLOTHO. We present two sets of empirical results in Section 7 showing applicability and effectiveness of our approach and its performance compared to a state of the art dynamic testing framework. Lastly we summarize related works in section 8 before concluding the paper in Section 9.

```
1  public void payment(int c_id)
2          throws CustNotFoundException{
3   stmt = prepareStatement("SELECT
          C_PAY_CNT FROM CUST WHERE C_ID=?");
4   stmt.setInt(1, c_id);
5   rs = stmt.executeQuery();
6   if(!rs.next())
7    throw new CustNotFoundException(c_id);
8   int c_pay_cnt = rs.getInt("C_PAY_CNT");
9   c_pay_cnt++;
10  stmt = prepareStatement("UPDATE CUST SET
           C_PAY_CNT=? WHERE C_ID=?");
11  stmt.setInt(1, c_pay_cnt);
12  stmt.setInt(2, c_id);
13  stmt.executeUpdate();
14 }
```

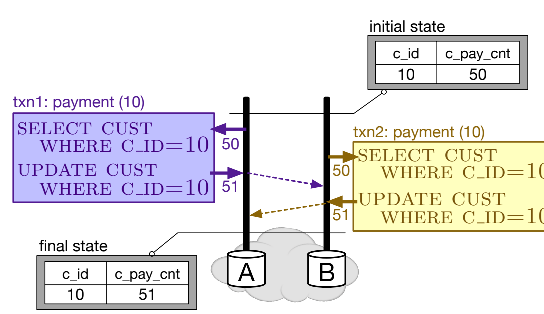

Fig. 2. A transaction from TPC-C benchmark in Java (left) and an anomalous execution (right)

2 OVERVIEW

TPC-C is a canonical Online Transaction Processing (OLTP) benchmark application that emulates a warehouse management and order entry/delivery system. The benchmark was originally designed to validate a mixture of requirements (including transactional safety) expected from relational database systems [tpc 2010]. To illustrate our approach, consider Figure 2 (left) that presents a pruned code snippet implementing a transaction from TPC-C named `payment`. The snippet shows a procedure that updates the total number of successful payments from a customer, where the customer's associated row from CUST table is initially retrieved (lines 3-5) and is then rewritten with an incremented value (lines 10-13). Now, consider the deployment of TPC-C on a replicated cloud database presented in Figure 2 (right) where two `payment` transactions are concurrently executed on weakly-consistent replicas A and B that do not enforce serializability. The initial database state in this execution consists of the table CUST(id, c_pay_cnt) with a single row $r_0 := (10, 50)$. Additionally, the transaction instances txn1 and txn2 are given arguments such that both transactions internally access and update r_0. Both transactions concurrently read the initial value of r_0 at different replicas (depicted as incoming labeled arrows to the transaction) and then both locally replace it with updated values $r_0^A := (10, 51)$ and $r_0^B := (10, 51)$ (depicted by outgoing labeled arrows). Each replica then propagates the locally updated row to the other replica (depicted by dashed arrows), where the written value at B supersedes value written at A, leaving the database with the final state of $r_0 := (10, 51)$.

Because the above scenario is not equivalent to any sequential execution of the two transaction instances, this behavior is properly classified as a *serializability anomaly*. Unfortunately, buggy executions like this are often permitted by modern cloud databases which drop support for atomic and isolated transactions in favor of fault tolerance, scalability, and availability. This foists a significant challenge onto OLTP application developers, requiring them to detect the potential anomalies that can occur in a particular application and validate them against different database systems in order to diagnose and remedy undesired behaviors.

Developing a testing framework that discovers anomalies of this kind is challenging, however. Part of the problem is due to the large set of possible interleaved executions that are possible. For example, the serializability anomaly described above is crucially dependent on the exact order in which operations are transmitted to different replicas and occurs only when the same customer (out of 30000 customers) is accessed by both instances. Combined with the possibility that operations within a single transaction instance can also be unreachable in some program paths, and that the execution paths taken are highly dependent on the initial database state, the chance of randomly encountering conflicts of this kind becomes even smaller.

In addition to the above challenges, the high computational cost of detecting serializability anomalies at runtime[Papadimitriou 1979] means that practical testing methods often leverage a set of user-provided application-level invariants and assertions to check for serializability failure-induced violations. But, these assertions are often underspecified. For example, the anomaly depicted in Figure 2, which most developers would classify as a bug, does not directly violate any of the twelve officially specified invariants of the TPC-C benchmark.

CLOTHO

To overcome these challenges, we have developed a principled test construction and administration framework called CLOTHO that targets database applications intended to be deployed on weakly-consistent storage environments. Our approach reasons over *abstract executions* of a database

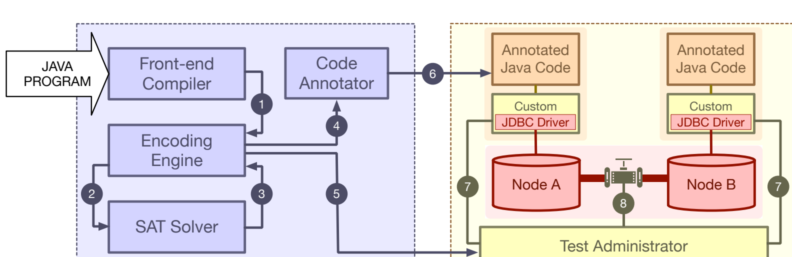

Fig. 3. Pipeline of CLOTHO

```java
@Parameters(10)
public void payment ... {
 ...
 @Sched(node="A", order=1)
 rs = stmt.executeQuery();
 ...
 @Sched(node="A", order=2)
 stmt.executeUpdate();
}
```

A1_Ins1.java

```java
@Parameters(10)
public void payment ... {
 ...
 @Sched(node="B", order=1)
 rs = stmt.executeQuery();
 ...
 @Sched(node="B", order=2)
 stmt.executeUpdate();
}
```

A1_Ins2.java

```
# initialize:
INSERT INTO
  CUST(c_id,c_pay_cnt)
  VALUES (10,50);
# schedule:
@T1@partitions{A,B}: Ins1-O1
@T2@partitions{A,B}: Ins2-O1
@T3@partitions{A,B}: Ins1-O2
@T4@partitions{A,B}: Ins2-O2
```

A1.conf

Fig. 4. Annotated code of two transaction instances and the test configuration file

application. Serializability anomalies manifest as cycles in such executions. CLOTHO translates the discovered abstract anomalies back to concrete executions which can be then automatically replayed and validated.

The test configuration details produced by CLOTHO include (i) concrete transaction instances and parameters, (ii) concurrent execution schedules, (iii) network partitionings throughout the execution, (iv) session management, and (v) initial database state. Our test administration framework allows arbitrary distributed or centralized databases to be plugged into its managed containers. Given a set of test configurations, CLOTHO automatically executes multiple application instances according to each test configuration, in order to effectively manifest and validate the intended serializability anomaly against a variety of actual database systems.

Figure 3 presents an overview of CLOTHO's design. CLOTHO's static analysis backend is based on an intermediate abstract representation of database programs, automatically generated from input Java source code via a front-end compiler. The abstract program representation is passed to an encoding engine (❶) that constructs FOL formulae that capture the necessary conditions under which a dependency cycle forms over instances of database operations. This fine-grained SAT representation of the problem is then passed to an off-the-shelf theorem prover (❷); responses with a satisfying solution (❸) are recorded and re-encoded into the context to be passed to the solver again with the intention of finding another anomaly. This iterative approach allows for a complete search of the space of anomalies within a configurable cycle-detection length. Once the bounded space is completely covered (i.e., there are no more satisfying solutions), test configuration files (❺) are generated from the collected abstract anomalies that provide details about concrete executions that can potentially manifest the intended anomaly. Additionally, the source code of the program is passed through a code annotator (❹,❻) which returns multiple instances of the source code including concrete function parameters and annotations on all database access operations.

For example, A1.conf in Figure 4 is the configuration file that specifies the initial state of the database needed to manifest the previously discussed anomaly on the CUST table. It also specifies the execution order of operations from the two `payment` transaction instances and the network status at each logical step (T1 to T4). The configuration is completed with annotated source code that specifies the input parameters to the `payment` transaction and marks the database operations with their relative local orders (A1_Ins1.java and A1_Ins2.java).

CLOTHO additionally offers a fully automated test administration framework, which consists of centrally managed wrappers for arbitrary database drivers which can temporarily block database access requests in order to enforce a specific execution order (❼). Similarly, database replicas are deployed in managed containers, communication among which can be throttled (❽) in order to induce temporary network partitionings. This enables CLOTHO to efficiently administer a wide range of statically constructed potential serializability anomalies, allowing developers to witness

$$a \in \texttt{Arguments} \quad v \in \texttt{Variables} \quad f \in \texttt{Fields} \quad A \in \{\texttt{min}, \texttt{max}\}$$
$$T \in \texttt{TxnNames} \quad \oplus \in \{+, -, \times, /\} \quad \odot \in \{<, \leq, =, >, \geq\} \quad \circ \in \{\wedge, \vee\}$$

$$\begin{array}{lcl}
\phi & := & e \odot e \mid \neg\phi \mid \phi \circ \phi \mid \texttt{true} \mid \texttt{false} \\
e & := & \mathbb{Z} \mid a \mid e \oplus e \mid \texttt{any}(\phi) \mid \texttt{iter} \mid \texttt{size}(v) \mid \texttt{proj}(f, v, e) \mid \texttt{this}.f \\
q & := & \texttt{SELECT}\ f\ \texttt{AS}\ v\ \texttt{WHERE}\ \phi \mid \texttt{SELECT}\ A(f)\ \texttt{AS}\ v\ \texttt{WHERE}\ \phi \mid \\
 & & \texttt{UPDATE SET}\ f = e\ \texttt{WHERE}\ \phi \mid \texttt{INSERT VALUES}\ \overline{f} = \overline{e} \mid \texttt{DELETE WHERE}\ \phi \\
c & := & q \mid \texttt{if}(\phi)\{c\} \mid \texttt{iterate}(e)\{c\} \mid c; c \mid \texttt{skip} \\
\mathbb{P} & := & \overline{T(\overline{a})\{c\}}
\end{array}$$

Fig. 5. Syntax of transactional programs written in the abstract representaiton ($\mathcal{AR}$)

and study undesired application behaviors that frequently occur in the wild but which are extremely difficult to catch using existing testing frameworks.

3 ABSTRACT MODEL

$\mathcal{AR}$ is the target language of CLOTHO's front-end compiler and is designed to capture key features of database-backed applications, including schemas, transactions, and data retrieval and modification operations. We begin by introducing the syntax of $\mathcal{AR}$ programs and presenting an overview of its semantics. A complete account of the semantics of $\mathcal{AR}$ can be found in [Rahmani et al. 2019].

3.1 Abstract Representation

$\mathcal{AR}$ programs are parameterized over a fixed schema that describes a set of database tables. Each table is defined as an ordered list of field names from the set `Fields`. In order to uniquely identify individual records, each table is assigned a non-empty subset of its fields that acts as a *primary key*. For simplicity, our presentation assumes a single table with only integer-valued fields, but CLOTHO relaxes this restriction and supports an arbitrary number of tables and other Java primitive types.

The syntax of $\mathcal{AR}$ is given in Figure 5. Programs in $\mathcal{AR}$ consist of a set of parameterized transactions, each of which has a unique name drawn from the set `TxnNames`. The body of a transaction is a command, denoted by c, which can be either a database query, a guarded command, a loop, or a sequence of commands. Database queries, denoted by q, retrieve (`SELECT`) or modify (`UPDATE`, `INSERT` and `DELETE`) records in the database. The result of each `SELECT` query is stored as an ordered list of records in a unique variable v.

Boolean expressions of $\mathcal{AR}$, denoted by ϕ, consist of standard arithmetic comparison and boolean operators.

Arithmetic expressions, denoted by e, include integer constants, transaction arguments, arithmetic operations, non-deterministic values, iteration counters, the number of results held in a variable, and field accesses, denoted as `proj`. For example, $\texttt{proj}(age, v, 3)$ returns the value of *age* field in the third record stored in the variable v. The expression $\texttt{any}(\phi)$ can evaluate to any value satisfying the predicate ϕ and is used to model source language features such as unknown function calls and updates to program variables within non-deterministic loops. The iteration counter `iter` represents the number of times a loop has been repeated during a program execution. Field names can also appear in an arithmetic expression as $\texttt{this}.f$, which can only be used in conditionals in the `WHERE` clauses of queries and denote the field values of records that are being evaluated in that clause.

JAVA CODE

```
stmt=prepareStatement("SELECT sal WHERE age<35");
rs = stmt.executeQuery();
while(rs.next()){
  int emp_id=rs.getInt("id");
  int old_sal=rs.getInt("sal");
  stmt=prepareStatement("UPDATE SET sal=? WHERE id=?");
  stmt.setInt(1,old_sal+1);
  stmt.setInt(2,emp_id);
  stmt.executeUpdate();
}
```

ENCODING IN $\mathcal{AR}$

```
SELECT sal AS v WHERE this.age<35;
iterate(size(v)){
  UPDATE SET sal=proj(sal,v,iter)+1
  WHERE this.id=proj(id,v,iter)
}
```

Fig. 6. A Java code block and its encoding in $\mathcal{AR}$

As an example, Figure 6 presents a Java code snippet and its $\mathcal{AR}$ encoding. The program scans a table and retrieves all records representing employees younger than 35 and then increases their salaries within a **while** loop.

3.2 System Configurations

The operational semantics of $\mathcal{AR}$ is defined by a small-step reduction relation, $\rightarrow \subseteq \Sigma \times \Gamma \times \Sigma \times \Gamma$, on the current *system state* of the underlying database, denoted as Σ, and a set of concurrently running transaction instances, denoted as Γ. Mimicking the execution of database-backed programs on replicated stores, $\mathcal{AR}$ programs are interpreted on a finite number of *partitions*, each of which has its own copy of the database. System states are therefore represented as a triple, $(\mathtt{str}, \mathtt{ar}, \mathtt{vis})$, which holds a history of database reads and writes executed at each partition and a pair of orderings on these events. Executing a database operation generates a set of *read effects*, $\mathsf{rd}(r, f)$, witnessing that the field f of record r was accessed, and *write effects*, $\mathsf{wr}(r, f, n)$, recording that the field f of record r was set to the value n. Read effects may optionally include the value read, e.g. $\mathsf{rd}(r, f, n)$. Each effect is implicitly tagged with a unique identifier, so that we can distinguish, for example, between the same field being read by multiple queries. Given a finite set $\mathcal{P}$ of partition names, $\mathtt{str}$ represents the set of effects generated on each partition as a mapping from partition names to disjoint sets of effects. When used without an argument, $\mathtt{str}$ refers to the complete set of effects from all partitions, i.e. $\mathtt{str} := \bigcup_{p \in \mathcal{P}} \mathtt{str}(p)$.

The $\mathtt{ar}$ component of the system state records the exact sequence of database operations that have been executed as an order between effects called an *arbitration relation*. This relation is however too coarse-grained to capture causal relationships between effects: an effect η_1 created at partition p_1, does not necessarily influence the execution of a query at partition p_2 that creates another effect η_2, even though η_1 may be arbitrated before η_2. The system state therefore maintains a more refined *visibility* relation, $\mathtt{vis} \subseteq \mathtt{ar}$, between effects, which only relates two effects if one actually witnesses the other at the time of creation.

It is possible to construct a current copy, or *local view*, of the database at each partition p from a system state by "applying" the effects of query operations stored in $\mathtt{str}(p)$ according to the order in $\mathtt{ar}$. Such local views are denoted as σ and are modeled as functions from primary keys to records. To ensure σ is total, all tables include a special field, *alive* $\in \mathtt{Fields}$, whose value determines if a record is present in the table or not.

3.3 Operational Semantics

Figure 7 presents the rules defining the operational semantics of $\mathcal{AR}$ programs and commands. These rules are parameterized over an $\mathcal{AR}$ program, $\mathbb{P}$. Each step either adds a new instance to the

$$\text{(E-SPAWN)}\ \frac{T(a)\{c\} \in \mathbb{P} \qquad n \in \mathbb{Z}}{\Sigma, \Gamma \longrightarrow \Sigma, \Gamma \cup \{c[a/n]\}} \qquad \text{(E-STEP)}\ \frac{\Sigma, c \longrightarrow \Sigma', c'}{\Sigma, \{c\} \cup \Gamma \longrightarrow \Sigma', \{c'\} \cup \Gamma} \qquad \text{(E-SKIP)}\ \frac{}{\Sigma, \mathsf{skip}; c \longrightarrow \Sigma, c} \qquad \text{(E-SEQ)}\ \frac{\Sigma, c \longrightarrow \Sigma', c''}{\Sigma, c; c' \longrightarrow \Sigma', c''; c'}$$

$$\text{(E-COND-T)}\ \frac{\Sigma \equiv (\mathsf{str}, \mathsf{ar}, \mathsf{vis}) \quad p \in \mathcal{P} \qquad \sigma = \Delta(\mathsf{ar}, \mathsf{str}(p)) \quad \sigma, \phi \Downarrow \mathsf{true}}{\Sigma, \mathtt{if}\ \phi\ \{c\} \longrightarrow \Sigma, c} \qquad \text{(E-COND-F)}\ \frac{\Sigma \equiv (\mathsf{str}, \mathsf{ar}, \mathsf{vis}) \quad p \in \mathcal{P} \qquad \sigma = \Delta(\mathsf{ar}, \mathsf{str}(p)) \quad \sigma, \phi \Downarrow \mathsf{false}}{\Sigma, \mathtt{if}\ \phi\ \{c\} \longrightarrow \Sigma, \mathsf{skip}}$$

$$\text{(E-ITER)}\ \frac{\Sigma \equiv (\mathsf{str}, \mathsf{ar}, \mathsf{vis}) \quad p \in \mathcal{P} \qquad \sigma = \Delta(\mathsf{ar}, \mathsf{str}(p)) \quad \sigma, e \Downarrow n}{\Sigma, \mathtt{iterate}(e)\{c\} \longrightarrow \Sigma, \mathsf{concat}(n, c)}$$

Fig. 7. Operational semantics of $\mathcal{AR}$ programs and commands

set of currently running transactions via E-SPAWN[1] or executes the body of one of the currently running transactions via E-STEP. The reduction rules for commands non-deterministically select a partition p to execute the command on. This partition is used by an auxiliary function Δ to construct a local view of the database. These local views are used by evaluation relation for boolean and arithmetic expressions, $\Downarrow$. The rules for non-query commands are straightforward outside for the (E-ITER) rule, which uses the $\mathtt{concat}(n, c)$ function that sequences n copies of the command c, with any occurrences of `iter` being instantiated as expected.

$$\boxed{\text{E-SELECT}}\ \frac{\begin{array}{c} q \equiv \mathtt{SELECT}\ f\ \mathtt{AS}\ x\ \mathtt{WHERE}\ \phi \qquad p \in \mathcal{P} \qquad \sigma = \Delta(\mathsf{ar}, \mathsf{str}(p)) \\ \varepsilon_1 = \{\mathsf{rd}(r, f) \mid \sigma, \phi[\mathtt{this}.f' \mapsto r(f')] \Downarrow \mathsf{true}\} \qquad \varepsilon_2 = \{\mathsf{rd}(r, f') \mid f' \in \mathcal{F}(\phi)\} \qquad \varepsilon = \varepsilon_1 \cup \varepsilon_2 \\ \mathsf{vis}' = \mathsf{vis} \cup \{(\eta, \eta') \mid \eta' \in \varepsilon \land \eta \in \mathsf{str}(p)\} \qquad \mathsf{ar}' = \mathsf{ar} \cup \{(\eta, \eta') \mid \eta' \in \varepsilon \land \eta \in \mathsf{str}\} \end{array}}{(\mathsf{str}, \mathsf{vis}, \mathsf{ar}), q \longrightarrow (\mathsf{str}[p \mapsto \mathsf{str}(p) \cup \varepsilon], \mathsf{vis}', \mathsf{ar}'), \mathsf{skip}}$$

$$\boxed{\text{E-UPDATE}}\ \frac{\begin{array}{c} q \equiv \mathtt{UPDATE}\ \mathtt{SET}\ f = v\ \mathtt{WHERE}\ \phi \qquad p \in \mathcal{P} \qquad \sigma = \Delta(\mathsf{ar}, \mathsf{str}(p)) \qquad \sigma, v \Downarrow n \\ \varepsilon_1 = \{\mathsf{wr}(r, f, n) \mid \sigma, \phi[\mathtt{this}.f' \mapsto r(f')] \Downarrow \mathsf{true}\} \qquad \varepsilon_2 = \{\mathsf{rd}(r, f') \mid f' \in \mathcal{F}(\phi)\} \qquad \varepsilon = \varepsilon_1 \cup \varepsilon_2 \\ \mathsf{vis}' = \mathsf{vis} \cup \{(\eta, \eta') \mid \eta' \in \varepsilon \land \eta \in \mathsf{str}(p)\} \qquad \mathsf{ar}' = \mathsf{ar} \cup \{(\eta, \eta') \mid \eta' \in \varepsilon \land \eta \in \mathsf{str}\} \end{array}}{(\mathsf{str}, \mathsf{vis}, \mathsf{ar}), q \longrightarrow (\mathsf{str}[p \mapsto \mathsf{str}(p) \cup \varepsilon], \mathsf{vis}', \mathsf{ar}'), \mathsf{skip}}$$

$$\boxed{\text{E-DELETE}}\ \frac{\begin{array}{c} q \equiv \mathtt{DELETE}\ \mathtt{WHERE}\ \phi \qquad p \in \mathcal{P} \qquad \sigma = \Delta(\mathsf{ar}, \mathsf{str}(p)) \\ \varepsilon_1 = \{\mathsf{wr}(r, \mathit{alive}, 0) \mid \sigma, \phi[\mathtt{this}.f' \mapsto r(f')] \Downarrow \mathsf{true}\} \qquad \varepsilon_2 = \{\mathsf{rd}(r, f') \mid f' \in \mathcal{F}(\phi)\} \qquad \varepsilon = \varepsilon_1 \cup \varepsilon_2 \\ \mathsf{vis}' = \mathsf{vis} \cup \{(\eta, \eta') \mid \eta' \in \varepsilon \land \eta \in \mathsf{str}(p)\} \qquad \mathsf{ar}' = \mathsf{ar} \cup \{(\eta, \eta') \mid \eta' \in \varepsilon \land \eta \in \mathsf{str}\} \end{array}}{(\mathsf{str}, \mathsf{vis}, \mathsf{ar}), q \longrightarrow (\mathsf{str}[p \mapsto \mathsf{str}(p) \cup \varepsilon], \mathsf{vis}', \mathsf{ar}'), \mathsf{skip}}$$

Fig. 8. Operational semantics of $\mathcal{AR}$ queries.

The operational semantics for queries, presented in Figure 8, are more interesting. The key component of each of these rules is how they construct the set of new effects generated by each database operation, denoted by ε_1 and ε_2. In the rule for `SELECT`, ε_1 includes appropriate read effects for each record satisfying the operation's `WHERE` condition. Any occurrence of $\mathtt{this}.f$ construct in the conditional ϕ, is substituted with the value of the corresponding field in each record instance under evaluation. The rule for `UPDATE` similarly includes a write effect in ε_1 for each record satisfying the operation's `WHERE` condition. To model the deletion of records, the rule for `DELETE` creates write effects with a constant value 0 for the *alive* fields of all records that satisfy its `WHERE` condition.

[1]For simplicity of presentation, we assume the spawned transaction has only one argument

In order to capture the data accesses that occur during database-wide scans, all three rules include new read effects in ε_2 for every field that is predicated in the operation's `WHERE` clause. To do so, these rules use the auxiliary function $\mathcal{F}$ which extracts any such fields from boolean expressions. $\mathcal{F}$ additionally always includes the *alive* field. As an example, the result of this function for the where clause from the query in Figure 6 is, $\mathcal{F}(age < 35) = \{age, alive\}$.

Finally, each rule updates the `ar` (resp. `vis`) relation to reflect the relationship between the newly created effects in $\varepsilon_1 \cup \varepsilon_2$ and already existing effects in `str` (resp. `str`(p)).

Figure 9 depicts a concrete example of two consecutive execution steps of the $\mathcal{AR}$ program from Figure 6. We depict an execution in which arithmetic expressions in the update operation (q_2) have been fully evaluated and both operations execute on the same partition. The example furthermore simplifies `str` to be just the set of effects in that particular partition. The initial system state encodes a database with three employee records, only two of which are alive. Because the `SELECT` query does not modify any database record, the local views σ and σ' are identical.

The `SELECT` query constructs a set of new effects in ε_2, including three read effects on the *age* field of all records (since *age* is constrained by its `WHERE` clause) and three read effects on all *alive* fields. Since only one of the witnessed records is both alive and satisfies the `WHERE` condition (i.e. $age < 35$), a single read effect on that record, $\mathsf{rd}(1, sal)$, is also created and included in ε_1. Similarly, ε'_1 is defined for the execution of `UPDATE` query, which includes only a single write effect, $\mathsf{wr}(1, sal, 86)$, capturing the modified salary value for the previously selected employee record. ε'_2 is empty, because the `UPDATE` query uses the primary key of the table to access the record, without any database-wide scans. Note that both steps update the arbitration and visibility relations identically, since the example assumes only a single partition.

$$(\mathsf{str}, \mathsf{ar}, \mathsf{vis}), q_1 \rightarrow (\mathsf{str}', \mathsf{ar}', \mathsf{vis}'), \texttt{skip}$$

$$q_1 \equiv \mathsf{SELECT}\ sal\ \mathsf{AS}\ x\ \mathsf{WHERE}\ \texttt{this}.age < 35$$

$$\begin{aligned}\sigma = \{&(id:1, age:32, sal:85, alive:\mathsf{true}),\\ &(id:2, age:55, sal:120, alive:\mathsf{true}),\\ &(id:3, age:28, sal:100, alive:\mathsf{false})\}\end{aligned}$$

$$\begin{aligned}\varepsilon_1 &= \{\mathsf{rd}(1, sal)\}\\ \varepsilon_2 &= \{\mathsf{rd}(1, age), \mathsf{rd}(2, age), \mathsf{rd}(3, age),\\ &\quad\ \mathsf{rd}(1, alive), \mathsf{rd}(2, alive), \mathsf{rd}(3, alive)\}\end{aligned}$$

$$\begin{aligned}\mathsf{str}' &= \mathsf{str} \cup \varepsilon_1 \cup \varepsilon_2\\ \mathsf{vis}' &= \mathsf{vis} \cup \{(\eta, \eta') | \eta \in \mathsf{str} \wedge \eta' \in \varepsilon_1 \cup \varepsilon_2\}\\ \mathsf{ar}' &= \mathsf{ar} \cup \{(\eta, \eta') | \eta \in \mathsf{str} \wedge \eta' \in \varepsilon_1 \cup \varepsilon_2\}\end{aligned}$$

$$(\mathsf{str}', \mathsf{ar}', \mathsf{vis}'), q_2 \rightarrow (\mathsf{str}'', \mathsf{ar}'', \mathsf{vis}''), \texttt{skip}$$

$$q_2 \equiv \mathsf{UPDATE}\ sal = 86\ \mathsf{WHERE}\ \texttt{this}.id = 1$$

$$\begin{aligned}\sigma' = \{&(id:1, age:32, sal:85, alive:\mathsf{true}),\\ &(id:2, age:55, sal:120, alive:\mathsf{true}),\\ &(id:3, age:28, sal:100, alive:\mathsf{false})\}\end{aligned}$$

$$\varepsilon'_1 = \{\mathsf{wr}(1, sal, 86)\} \qquad \varepsilon'_2 = \{\}$$

$$\begin{aligned}\mathsf{str}'' &= \mathsf{str}' \cup \{\mathsf{wr}(1, sal, 86)\}\\ \mathsf{ar}'' &= \mathsf{ar}' \cup \{(\eta, \mathsf{wr}(1, sal, 86)) | \eta \in \mathsf{str}'\}\\ \mathsf{vis}'' &= \mathsf{vis}' \cup \{(\eta, \mathsf{wr}(1, sal, 86)) | \eta \in \mathsf{str}'\}\end{aligned}$$

Fig. 9. Execution of the program in Figure 6

4 SERIALIZABILITY ANOMALIES

We now turn to the question of identifying and statically detecting undesirable executions in $\mathcal{AR}$ programs.

4.1 Serial(izable) Executions

An *execution history* of an $\mathcal{AR}$ program is simply a finite sequence of system configurations allowed by the reduction relation: $H \equiv (\Sigma_0, \Gamma_0) \longrightarrow (\Sigma_1, \Gamma_1) \longrightarrow \cdots \longrightarrow (\Sigma_k, \Gamma_k)$. An execution history is said to be *valid*, if all the spawned transactions are fully executed, i.e. Γ_0 and Γ_k are both empty. A valid history furthermore requires Σ_0 to be consistent with any user-defined constraints on the initial state of a database, e.g. that certain tables are initially empty.

Following the literature on weak consistency and isolation semantics [Burckhardt et al. 2014] we can specify constraints on `vis` and `ar` relations which identify when system states are consistent with a particular database's consistency guarantees. Table 1 presents some well-known instances of such guarantees and their specifications, where the predicate `st` simply relates effects created by queries from the **s**ame **t**ransaction instance. System states can be thought of as finite models

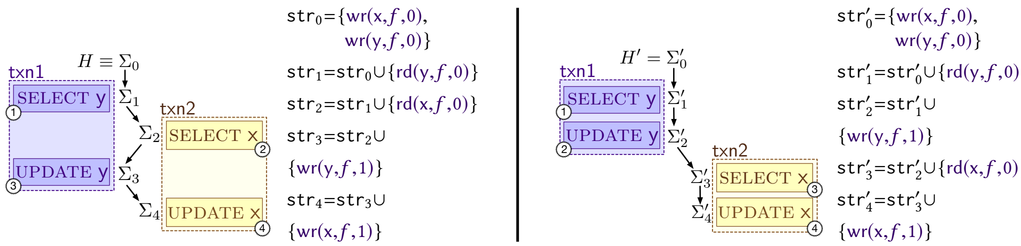

Fig. 10. A serializable execution (left) and its equivalent serial execution (right)

for these guarantees. For example, a system state Σ is said to be *strictly serial* if and only if Ψ_{SER} is satisfied by Σ, denoted as $\Sigma \models \Psi_{\text{SER}}$. Since both vis and ar only grow during the execution of a program, if the final system state satisfies a particular specification, so will every system state in the execution history. We correspondingly extend this notion to execution histories and define a history H to conform to a guarantee Ψ, denoted by $H \models \Psi$, if and only if Ψ is satisfied in the final system state of H.

Although strictly serial executions are easily comprehensible and desirable, forcing the database to execute transactions one-by-one is unnecessarily restrictive and must usually rely on poorly scalable pessimistic locking mechanisms to enforce [Sivaramakrishnan et al. 2015]. We instead define an execution history H to be *serializable* if there exists another strictly serial history H', that is constructed by reordering execution steps of H, such that the final set of effects in both histories are equivalent.

For example, Figure 10 depicts the system states from a history H (left), and another serial history H' (right) which is constructed by reordering steps ② and ③ of H. Although H is not strictly serial (since both Ψ_{RC} and Ψ_{RR} are violated in Σ_4), its final set of events, $\mathtt{str}_4$, is the same as $\mathtt{str}'_4$, the final set of events in H', modulo renaming of their implicit unique identifiers, and is therefore serializable.

Given an $\mathcal{AR}$ program, our goal is to statically find non-serializable execution histories that are consistent with the guarantees of the underlying database, i.e. to detect any *serializability anomalies* in the program. As a first step, observe that execution histories that contain a serializability anomaly can be decomposed into a serial execution history followed by a (usually smaller) non-serializable history. For instance, a non-serializable execution of ten transactions, may be decomposed into eight serially executed transactions which insert appropriate records into initially empty tables, needed for the remaining two (concurrently executed) transactions that actually manifest the serializability anomaly. Following this observation, CLOTHO systematically prunes the space of all possible interleavings of transactions involved in lengthy anomalous executions and instead

Guarantee	Specification
Causal Visibility	$\Psi_{\text{CV}} \equiv \forall \eta_1\eta_2\eta_3.\ \mathtt{vis}(\eta_1,\eta_2) \land \mathtt{vis}(\eta_2,\eta_3) \Rightarrow \mathtt{vis}(\eta_1,\eta_3)$
Causal Consistency	$\Psi_{\text{CC}} \equiv \forall \eta_1\eta_2.\ \Psi_{\text{CV}} \land (\mathtt{st}(\eta_1,\eta_2) \Rightarrow \mathtt{vis}(\eta_1,\eta_2) \lor \mathtt{vis}(\eta_2,\eta_1))$
Read Committed	$\Psi_{\text{RC}} \equiv \forall \eta_1\eta_2\eta_3.\ \mathtt{st}(\eta_1,\eta_2) \land \mathtt{vis}(\eta_1,\eta_3) \Rightarrow \mathtt{vis}(\eta_2,\eta_3)$
Repeatable Read	$\Psi_{\text{RR}} \equiv \forall \eta_1\eta_2\eta_3.\ \mathtt{st}(\eta_1,\eta_2) \land \mathtt{vis}(\eta_3,\eta_1) \Rightarrow \mathtt{vis}(\eta_3,\eta_2)$
Linearizable	$\Psi_{\text{LIN}} \equiv \mathtt{ar} \subseteq \mathtt{vis}$
Strictly Serial	$\Psi_{\text{SER}} \equiv \Psi_{\text{RC}} \land \Psi_{\text{RR}} \land \Psi_{\text{LIN}}$

Table 1. Consistency and isolation guarantees

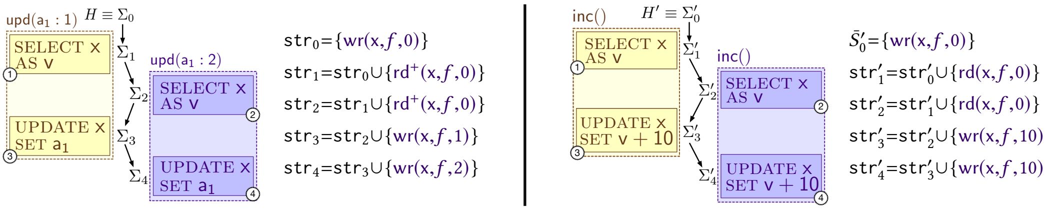

Fig. 11. External (left) and internal (right) lost update anomalies.

searches for *independent* serializability anomalies, defined as two disjoint histories $H_1 \longrightarrow H_2$, where H_1 is a valid serial execution and H_2 is a non-serializable execution history.

4.2 Internal and External Serializability

During the course of our experiments, we observed that a subset of serializability anomalies could be considered benign, in that they always leave the final state of the database indistinguishable from the state after some serializable execution of the same instances. As an example, consider the executions H and H' presented in the left and right of Figure 11. H consists of two instances of the upd transaction, which reads the variable x and then updates it to a fresh value. Similarly, H' includes two instances of the inc transaction which reads the value of x and then *increments* it by 10. Both of these executions are non-serializable as they manifest the classic *lost update* anomaly [Adya 1999]. However, only H' leaves the database in a state that is inconsistent with any serializable execution. We dub the sort of benign anomalies seen in H *external serializability anomalies*.

The key observation is that it is possible for some of the read effects in a non-serializable execution to not impact a transaction's control flow or any of later write effects. As a result, those read events could be excluded from the correctness analyses, in order to focus on *internal serializability anomalies*. Such anomalies are more harmful in that they leave the database state permanently diverged from a serializable execution state and should be carefully studied and addressed by developers. We denote such *unused* read effects as rd^+, which can be detected through a straightforward analysis of the source program. For these reasons, by default, CLOTHO only identifies internal serializabilty anomalies.

4.3 Dependency Cycles

Following the approach of Adya et al. [2000], we reduce the problem of determining serializability of an execution history to the detection of cycles in the *dependency graph* of its final state. To this end, we first define three *dependency relations* over the set of effects in an execution state: (i) Read dependency, WR, which relates two effects if one witnesses a value in a record's field that is written by the other. (ii) Write dependency, WW, which relates two effects if one overwrites the value written by the other. (iii) Read anti-dependency, RW, which relates two effects if one witnesses a value in a field that is later overwritten by the other effect. For example, in Figure 11 (left) both dependency relations $\mathsf{WW}(\mathsf{wr}(x, f, 1), \mathsf{wr}(x, f, 2))$ and $\mathsf{RW}(\mathsf{rd}^+(x, f, 0), \mathsf{wr}(x, f, 2))$ hold.

Recalling that serializability is defined by a reordering of database operations, we lift the definition of the above dependency relations (and previously defined vis and ar) from effects to query instances. Given an execution history H and for all relations $\mathsf{R} \in \{\mathsf{ar}, \mathsf{vis}, \mathsf{RW}, \mathsf{WR}, \mathsf{WW}\}$, if effects η and η' are created respectively by queries q and q' in H, then $\mathsf{R}(\eta, \eta') \Rightarrow \mathsf{R}(q, q')$. This lifting is necessary because in order to determine serializability of a history H, the execution order of queries in H must be altered such that the new history H' is serial (see section 4.1). Intuitively, such reordering cannot not exist when some queries in H are cyclically dependent to each other.

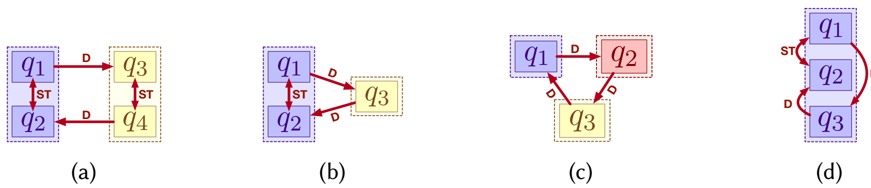

Fig. 12. Valid (a,b) and invalid (c,d) cycles where $\mathsf{D} \in \{\mathsf{WR}, \mathsf{RW}, \mathsf{WW}\}$

We are now ready to formally define the dependency graph G of a system state Σ as a directed multigraph whose nodes are query instances (that created the effects in Σ) and edges are in $\{\mathsf{RW}, \mathsf{WR}, \mathsf{WW}, \mathsf{ST}\}$. The relation ST is simply defined to relate query instances originated from the same transaction. We further define a valid dependency cycle as a 2-regular subgraph of G, that includes at least two dependency edges (i.e. $\mathsf{WR}, \mathsf{RW}, \mathsf{WW}$) which are connected by an ST edge. We do not allow dependency edges between query instances from the same transaction and w.l.o.g assume they are replaced with ST edges. Figure 12 presents schematic examples of valid and invalid cycle structures in dependency graphs.

We conclude this section by describing the connection between the dependency graph of the final state in an execution to its serializability. To this end, an execution history $(\Sigma_0, \Gamma_0) \longrightarrow \cdots \longrightarrow (\Sigma_k, \Gamma_k)$ is serializable if and only if Σ_k's dependency graph is acyclic [Adya et al. 2000]. As we will explain in the next section, this connection enables CLOTHO to encode potential dependency cycles in a given $\mathcal{AR}$ program into a decidable first-order formula and statically construct an execution history that manifests a serializability anomaly.

5 STATIC ANOMALY GENERATION

We now turn to the question of how to statically detect dependency cycles and construct independent serializability anomalies in $\mathcal{AR}$ programs. We adopt the approach presented in [Nagar and Jagannathan 2018] and [Brutschy et al. 2018] and reduce (the bounded version of) our problem to checking the satisfiability of an FOL formula $\varphi_{\mathbb{C}}$, constructed from a given $\mathcal{AR}$ program. This formula includes variables for each of the dependency, visibility, and arbitration constraints that can appear during the program's execution, and is designed such that the assignments to these variables in any satisfying model can be used to reconstruct an anomalous execution of the original program. This allows us to use an off-the-shelf SMT solver to efficiently check for anomalies in the given program. Given bounds max_p, max_t and max_c, the shape of the full formula is a conjunction of five clauses, each encoding a different aspect of the program:

$$\varphi_{\mathbb{C}} \stackrel{\Delta}{\equiv} \varphi_{\text{CONTEXT}} \wedge \varphi_{\text{DB}} \wedge \varphi_{\text{DEP}\rightarrow} \wedge \varphi_{\rightarrow\text{DEP}} \wedge \bigvee_{\substack{0 \le i \le max_p \\ 2 \le j \le max_t \\ 3 \le k \le max_c}} \varphi^{i,j,k}_{\text{ANOMALY}} \tag{1}$$

In the above formula, φ_{CONTEXT} represents a set of constraints on variables and functions, which ensures that a satisfying assignment corresponds to a valid execution of any database program. Clause φ_{DB} enforces a set of user-defined validity constraints on database records. Clauses $\varphi_{\text{DEP}\rightarrow}$ and $\varphi_{\rightarrow\text{DEP}}$ capture the necessary and sufficient conditions for establishing dependency relations between queries in the given program.

Following our discussion of execution histories which manifest an independent serializability anomaly in section 4.1, $\varphi^{i,j,k}_{\text{ANOMALY}}$ forces the solver

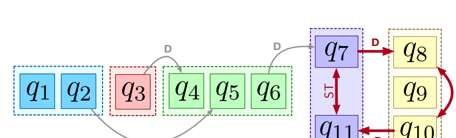

Fig. 13. An anomaly constructed by CLOTHO

to instantiate a serial execution of i transactions that lead to concurrent execution of j transactions within which a dependency cycle of length k is formed. Figure 13 presents an example of such an instantiation where $i = 3$, $j = 2$ and $k = 4$. The remainder of this section lays the formal groundwork for the encoding and explains each of the above clauses in more detail. The complete details of the encoding can be found in the extended version of this paper [Rahmani et al. 2019].

5.1 Components of FOL Encoding

We assume uninterpreted sorts for transactions ($\mathfrak{t}$), queries ($\mathfrak{q}$), partitions ($\mathfrak{p}$), and records ($\mathfrak{r}$). We define functions ($\mathtt{txn} : \mathfrak{q} \longrightarrow \mathfrak{t}$) and ($\tau : \mathfrak{q} \longrightarrow \mathfrak{p}$) to relate a query to the transaction instance calling it and the partition it is executed on. We use an auxiliary function $\mathtt{Alive} : \mathfrak{r} \times \mathfrak{q} \longrightarrow \mathbb{B}$ to capture the liveness of a record in a particular query instance. Predicates `vis`, `ar`, `ST`, `WR`, `RW` and `WW`, all with the signature $\mathfrak{q} \times \mathfrak{q} \longrightarrow \mathbb{B}$ are also defined to capture relations discussed in section 3. The disjunction of dependency relations `WW`, `WR` and `RW` is denoted by `D`.

φ_{ANOMALY}. This clause specifies that an independent serializability anomaly exists in the program, that consistes of i serially executed transactions and j concurrent transactions that manifest a dependency cycle of length k. It is also required that all serializable transactions are executed before the transactions involved in the cycle.

$$\varphi^{i,j,k}_{\text{ANOMALY}} \stackrel{\Delta}{\equiv} \exists t_1, \ldots, t_i. \bigwedge_{1 \leq m \leq i} \varphi_{\text{SER}}(t_m) \wedge \exists t'_1, \ldots, t'_j.\ \varphi^{k}_{\text{CYCLE}}(t'_1, \ldots, t'_j) \wedge \bigwedge_{\substack{1 \leq m \leq i \\ 1 \leq n \leq j}} \varphi_{\text{ORDER}}(t_m, t'_n)$$

In the above formula, $\varphi_{\text{SER}}(t)$ encodes the serializability of t with respect to all other transaction instances and $\varphi_{\text{ORDER}}(t, t')$ enforces that all queries from t are arbitrated before the queries of t'. Proposition $\varphi^{k}_{\text{CYCLE}}(t_1, \ldots, t_j)$ forces the solver to instantiate k queries within the given transactions, such that a cycle of length k exists between them, at least two of its edges are dependency relations in $\{\mathtt{WR}, \mathtt{RW}, \mathtt{WW}\}$ and one is an `ST` relation. The rest of the edges can be of either kind.

$$\begin{aligned}\varphi^{k}_{\text{CYCLE}}(t_1, \ldots, t_j) \stackrel{\Delta}{\equiv} \exists q_1, \ldots, q_k. & \bigwedge_{1 \leq m \leq k} \mathtt{txn}(q_m) \in \{t_1, \ldots, t_j\} \wedge \\ & \mathtt{D}(q_1, q_2) \wedge \mathtt{D}(q_{k-1}, q_k) \wedge \mathtt{ST}(q_1, q_k) \wedge \bigwedge_{2 \leq m < k-1} (\mathtt{ST}(q_m, q_{m+1}) \vee \mathtt{D}(q_m, q_{m+1}))\end{aligned}$$

Figure 14 presents concrete examples of dependency cycles detected by CLOTHO. Figure 14a depicts a scenario where the `SELECT` query from `oneRead` transaction witnesses the intermediate value written by `twoWrites` transaction. CLOTHO automatically determines the execution order of queries (shown as circled numbers) and values of input arguments required for manifestation of the anomaly. In this example, it suffices that all queries access the record with the same key, i.e. $\mathsf{k}_1 = \mathsf{k}_2 = \mathsf{k}_3$. The anomaly depicted in figure 14b occurs when an update from transaction `twoWrites'` is executed at a node which then becomes disconnected from the network to form partition 1. Assuming that clients can access the rest of the network on partition 2, the remaining queries need to be submitted to available nodes in the specified order in order to trigger the depicted serializability anomaly. This anomaly does not occur when network partitioning is not possible. Figures 14c and 14d respectively present the cycles detected for external and internal lost update anomaly from the example discussed in section 4.2. Predicate $\mathtt{ST}^{+}$ is defined similarly to `ST` but also requires that no data dependency exists between the predicated query instances. CLOTHO users

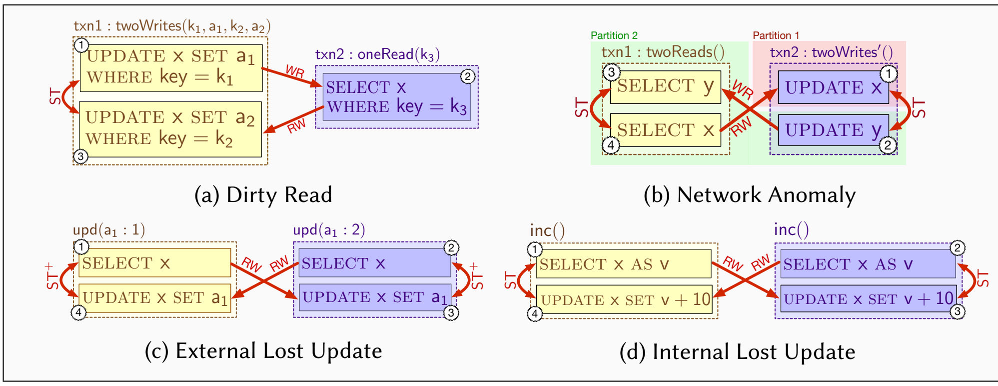

Fig. 14. Examples of dependency cycles generated by CLOTHO

may optionally choose to only detect internal serializability anomalies in their programs, in which case ST will be replaced with ST^+ in φ_{CYCLE}.

φ_{CONTEXT}. This clause ensures that a satisfying assignment corresponds to a valid execution of database programs. It uses a collection of functions $\mathtt{init}^f : \mathbb{r} \to \mathbb{Z}$ and $\mathtt{val}^f : \mathbb{q} \times \mathbb{r} \to \mathbb{Z}$ which are defined for all fields f in the given schema. These functions are used to identify the initial values of records and their values at each query operation, respectively. These functions are constrained so that the values read by operations are from the initial state if the fields have not yet been written to. We make this explicit via the following formula:

$$\bigwedge_{q_1,q_2 \in \mathbb{q}} \bigwedge_{r \in \mathbb{r}} \neg\mathsf{WR}(q_2, q_1) \Rightarrow \mathtt{val}^f(q_1, r) = \mathtt{init}^f(r) \tag{2}$$

Next, φ_{CONTEXT} constrains the values of all records stored in a variable v by a `SELECT` query to satisfy the `WHERE` condition of that query.

The next set of restrictions encoded in φ_{CONTEXT} constrains the fields of records involved in a dependency cycle in order to make the corresponding anomaly more understandable. As a concrete example, consider the twoWrites transaction in figure 14a. Without explicit constraints on the updated values in this transaction, a SMT solver could trivially instantiate parameters $\mathsf{a}_1 = \mathsf{a}_2 = 0$, making the dirty value read by oneRead transaction the same as the value read in a serializable execution of the two transactions. This could falsely appear benign to developers, so φ_{CONTEXT} constrains the values of records to avoid such scenarios. φ_{CONTEXT} forces the values of field f used in queries q_1 and q_2 to be **identical** if there is a read dependency between them, i.e. $\mathsf{WR}(q_1, q_2)$ holds. Similarly, if there is an anti-dependency (RW) or a write dependency (WW) between two queries, the values of any fields written or read by those queries are forced to be **different**.

Finally, we constrain the visibility relation to capture how effects in a partition are witnessed by any later query executed in that partition, i.e. we enforce *causal visibility within each partition*:

$$\forall q_1, q_2.\ \mathsf{ar}(q_1, q_2) \wedge (\tau(q_1) = \tau(q_2)) \Rightarrow \mathtt{vis}(q_1, q_2) \tag{3}$$

φ_{DB}. This clause enforces any consistency or isolation guarantees (e.g. those from Table 1) provided by the database under test, as well as any user-defined constraints on record instances, e.g. requiring a table to be initially empty or requiring *age* field to be always greater than 21.

$\varphi_{\text{DEP}\rightarrow}$. This clause ensures that every database operation captures the effects of any query it is related to by the dependency relation D. This is accomplished via a proposition, $\mu^{\mathsf{D}\rightarrow}_{q,q'}$, which asserts that (a) there exists a concrete record instance that both queries access and (b) both queries are reached by the control flow of their transactions.

To this end, we first introduce function $[\![]\!]^{\mathbb{B}}_{t} : \phi \to \mathbb{B}$ which constructs an FOL formula corresponding a boolean expression in $\mathcal{AR}$, alongside any additional conditions that must be satisfied for a successful construction. When the input is a WHERE condition of a query, a record instance (that must be checked if satisfies the condition) is also passed to the function as an extra argument, e.g. $[\![]\!]^{\mathbb{B}}_{t,r}$. We also assume a function $\Lambda : \mathfrak{q} \to \phi$ which returns the conjunction of all conditionals which must be satisfied in order for a query to be reached by the program's control flow.

$$\boxed{\text{RW-SELECT-UPDATE}}\quad \frac{\begin{array}{c} q \equiv \texttt{SELECT}\ f\ \texttt{AS}\ x\ \texttt{WHERE}\ \phi \\ q' \equiv \texttt{UPDATE SET}\ f = v\ \texttt{WHERE}\ \phi' \\ \texttt{txn}(q) = t \qquad \texttt{txn}(q') = t' \qquad t \neq t' \end{array}}{\begin{array}{c} \mu^{\mathsf{RW}\rightarrow}_{q,q'} = \exists r.\ [\![\phi]\!]^{\mathbb{B}}_{t,r} \wedge [\![\phi']\!]^{\mathbb{B}}_{t',r} \wedge \texttt{Alive}(r, q) \wedge \\ \texttt{Alive}(r, q') \wedge [\![\Lambda(q)]\!]^{\mathbb{B}}_{t} \wedge [\![\Lambda(q')]\!]^{\mathbb{B}}_{t'} \end{array}}$$

Fig. 15. An example of necessary conditions for a dependency relation

Figure 15 gives a rule defining a formula that enforces these conditions when an anti-dependency relation $\mathsf{RW}(q, q')$ is established between a SELECT query q and an UPDATE query q' which access the same field f. The transactions containing the two query are distinct. The rule enforces that there is a record which satisfies the WHERE clause of both queries and is perceived alive by both of them. Both queries are also forced to be reachable in their corresponding transactions. Assuming the full set of rules defining $\mu^{\mathsf{D}\rightarrow}$ for a given program, the $\varphi_{\text{DEP}\rightarrow}$ clause is simply defined as their conjunction:

$$\varphi_{\text{DEP}\rightarrow} \triangleq \bigwedge_{\mathsf{D} \in \{\mathsf{WR},\mathsf{RW},\mathsf{WW}\}} \ \bigwedge_{q,q' \in \mathfrak{q}} (\mathsf{D}(q, q') \Rightarrow \mu^{\mathsf{D}\rightarrow}_{q,q'})$$

$\varphi_{\rightarrow DEP}$. Examining the values of `ar`, τ, `init` and `val` in a satisfying assignment to $\varphi_{\mathbb{C}}$, gives us enough information to recover a concrete execution of all the queries involved in a dependency cycle. This suffices for constructing valid test cases for all the examples discussed so far, all of which consist of transactions that only include queries involved in the dependency cycle. In practice, though, we observed that for the majority of detected anomalies, transactions include operations which occur before the start of the cycle. Such operations may cause the database's pre-cycle state to diverge from the state built by the conjunction of the clauses previously described, preventing CLOTHO from constructing a valid test configuration that realizes the anomaly.

As a concrete example, consider the `delivery` transaction from the TPC-C benchmark, where records from the tables n_order and cust are retrieved and updated. As presented in Figure 16 (left), a lost update anomaly on cust table may occur if two instances of `delivery` transaction concurrently select and update the same cust record (depicted by red RW edges). Note that, the primary key for the cust record involved in this anomaly, is determined based on the n_order records previously retrieved during steps ① and ③.

Mapping this cycle to a non-serializable execution requires finding transaction arguments that force the retrieval of n_order records at steps ① and ③ that refer to the same cust record. Following proposition (2), our encoding thus far constrains the values of the n_order records (including values in the *alive* field) retrieved by the queries at ① and ③ to be the same. Both queries will witness the initial state of the records, since there does not exist any WR pointing at them. As a result, a satisfying assignment could select the same value as the argument to both transactions, i.e. $a_1 = a_2$. When replaying of this execution according to the specified arbitration orders is attempted, however,

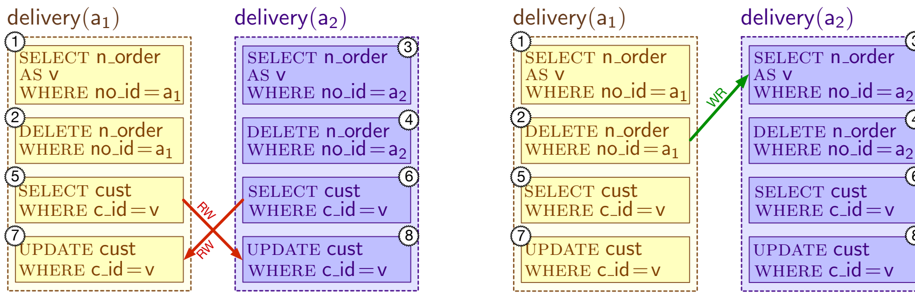

Fig. 16. Lost update anomaly on Cust table in delivery transaction of TPC-C

the n_order record retrieved at step ① is deleted at step ② and cannot be retrieved at step ③. Consequently, the execution will either terminate at step ③ or fail to form the desired dependency cycle, since two different cust records would be selected in steps ⑤ and ⑥.

The $\varphi_{\to \mathrm{DEP}}$ clause remedies this problem by forcing WR edges to exist between operations that are not part of the dependency cycle. This clause forces the WR edge depicted by the green arrow in Figure 16 (right) to be established. When combined with the previous arbitration ordering, this prevents the operation at step ③ from selecting the same n_order record as the one deleted at ② and eliminates the spurious dependency cycle between steps ⑤, ⑥, ⑦ and ⑧. A satisfying assignment must now ensure that *two* records exist in the n_order table, both of which refer to the same cust record, in order to subsequently manifest a valid lost update anomaly on that record.

The clause $\varphi_{\to \mathrm{DEP}}$, using predicates $\mu^{\to \mathsf{D}}_{q,q'}$, defines sufficient conditions under which a dependency relation D between q and q' must exist. Figure 17 presents the rule that defines the predicate to force a WR dependency when q is a SELECT query and q' is an UPDATE such that: (1) the update query is visible to the select query, (2) both queries access the same alive row, and (3) both q and q' will be executed.

UPDATE-SELECT-WR

$$\frac{q \equiv \texttt{SELECT}\ f\ \texttt{AS}\ x\ \texttt{WHERE}\ \phi \qquad q' \equiv \texttt{UPDATE SET}\ f = v\ \texttt{WHERE}\ \phi' \qquad \mathtt{txn}(q) = t \quad \mathtt{txn}(q') = t' \quad t \neq t'}{\mu^{\to \mathsf{WR}}_{q',q} = \mathtt{vis}(q', q) \wedge \exists r.\ [\![\phi]\!]^{\mathbb{B}}_{t,r} \wedge [\![\phi']\!]^{\mathbb{B}}_{t',r} \wedge \mathtt{Alive}(r, q) \wedge \mathtt{Alive}(r, q') \wedge [\![\Lambda(q)]\!]^{\mathbb{B}}_{t} \wedge [\![\Lambda(q')]\!]^{\mathbb{B}}_{t'}}$$

Fig. 17. An example of sufficient conditions for a dependency relation

Using the complete set of rules for $\mu^{\to \mathsf{D}}$, we can give the full definition of the clause that forces the solver to establish dependency edges also between operations outside of a cycle:

$$\varphi_{\to \mathrm{DEP}} \stackrel{\Delta}{\equiv} \bigwedge_{\mathsf{D} \in \{\mathsf{WR},\mathsf{RW},\mathsf{WW}\}} \bigwedge_{q_1, q_2 \in \mathfrak{q}} (\mu^{\to \mathsf{D}}_{q_1,q_2} \Rightarrow \mathsf{D}(q_1, q_2))$$

We describe refinements to this encoding scheme in the following section that allow CLOTHO to manifest serializability anomalies for realistic database programs, as evidenced by our experimental results. These results support our contention that our encoding is sufficiently precise to map abstract executions to realizable non-serializable concrete ones.

function : FindAnomalies($s, a, \varphi_{\text{DB}}, max_p, max_t, max_c$)
Input: s: data schema, a: $\mathcal{AR}$ application, φ_{DB}: database constraints,
max_p: maximum serial transactions, max_t: maximum concurrent transactions,
max_c: maximum cycle length
Output: *anoms*: a set of satisfying assignments
Initialize: $\varphi_{\text{APP}} \leftarrow$ EncApp(s, a), $anoms \leftarrow \emptyset$, $cycles \leftarrow \emptyset$

1 **for** $t \in [2, max_t]$ **do**
2 $\quad c \leftarrow 3$
3 $\quad$ **while** $c \leq max_c$ **do**
4 $\qquad \varphi_{\text{NEG}} \leftarrow$ EncNeg$(cycles)$
5 $\qquad new\_cyc \leftarrow$ isSAT$(\exists t_1, ..., t_t.\ \varphi^c_{\text{CYCLE}}(t_1, ...t_t) \wedge \varphi_{\text{DB}} \wedge \varphi_{\text{APP}} \wedge \varphi_{\text{NEG}})$
6 $\qquad$ **if** new_cyc = UNSAT **then** $c \leftarrow c + 1$; **continue**;
7 $\qquad cycles \leftarrow cycles \cup \{new\_cyc\}$
8 $\qquad \varphi_{\text{STCT}} \leftarrow$ EncStruct(new_cyc)
9 $\qquad$ **do**
10 $\qquad\quad \varphi_{\text{NEG}} \leftarrow$ EncNeg$(cycles)$
11 $\qquad\quad new\_cyc \leftarrow$ isSAT$(\exists t_1, ...t_t.\ \varphi^c_{\text{CYCLE}}(t_1, ...t_t) \wedge \varphi_{\text{DB}} \wedge \varphi_{\text{APP}} \wedge \varphi_{\text{NEG}} \wedge \varphi_{\text{STCT}})$
12 $\qquad\quad$ **if** new_cyc = UNSAT **then** **break** **else** $cycles \leftarrow cycles \cup \{new\_cyc\}$;
13 $\qquad$ **while** *true*;
14 **for** $cyc \in cycles$ **do**
15 $\quad$ **for** $p \in [0, max_p]$ **do**
16 $\qquad \varphi_{\text{PATH}} \leftarrow$ EncPath(cyc)
17 $\qquad new\_anml \leftarrow$ isSAT$(\exists t_1, ..., t_p.\ \varphi_{\text{PATH}})$
18 $\qquad$ **if** $new\_anml \neq$ UNSAT **then** $anoms \leftarrow anoms \cup \{new\_anml\}$; **break**;

Fig. 18. Search Algorithm

6 IMPLEMENTATION

In this section, we first present our algorithm for exhaustively finding the set of anomalous execution in a given application and then discuss details related to CLOTHO's implementation.

6.1 Search Algorithm

The naïve algorithm described in the previous sections that iteratively queries a SMT solver to find all solutions to $\varphi_{\mathbb{C}}$ is unfortunately too inefficient to be effective in practice. In our initial experiments, we found that the SMT solver would often fail to find *any* satisfying assignment to $\varphi_{\mathbb{C}}$ within a reasonable amount of time. We hypothesized that the reason for this was that the solver had the flexibility to instantiate φ_{ANOMALY} in (exponentially many) ways that did not force a cycle. Exploiting this intuition, we designed a two-stage search algorithm, presented in Figure 18, which iteratively guides the solver towards cycles via a series of successively more constrained SMT queries and then attempts to construct independent anomalies based upon those cycles.

The FindAnomalies algorithm takes as input a data schema s, an $\mathcal{AR}$ application a, a set of user-defined constraints on the database and its consistency model φ_{DB}, and max_p, max_t, max_c, which are bounds on the space of independent anomalies, as described in section 5. The algorithm also outputs *anoms*, the set of all satisfying assignments to $\varphi_{\mathbb{C}}$. The algorithm constructs *anoms* by repeatedly querying a SMT solver, denoted by isSAT. Satisfying assignments to $\varphi_{\mathbb{C}}$ are constructed by first iteratively finding the set of all bounded cycles using the SMT query in line 5, and then constructing independent anomalies atop of each cycle using the SMT query in line 17.

The formula in the first query (line 5) is a conjunction of four clauses quantified over t variables which represent the transactions on a dependency cycle of length c. φ_{APP} is constructed before the loop begins using function EncApp and corresponds to $\varphi_{\text{DEP}\rightarrow}$ and $\varphi_{\rightarrow\,\text{DEP}}$ in the given program. Finally, the formula also includes a clause φ_{NEG}, representing the negated conjunction of all previously found assignments (stored in *cycles*) so that the solver is forced to find new cycles at each iteration. The resulting satisfying assignment is stored in the *new_cyc* variable, which is then added to *cycles* in line 7. Similarly, the query in line 17, encoded by function EncPath, is quantified over p serially executed transactions that are intended to update the initial state of the database such that the pre-conditions for the given cycle can be satisfied.

The above steps describe how to efficiently construct a satisfying assignment to $\varphi_{\mathbb{C}}$. When experimenting with CLOTHO , we observed that many anomalies share a similar structure, in that they all include the same sorts of dependency relations between the operations in the transactions involved in the cycle. As an example, consider the program in Figure 19 where five different anomalies (depicted using dependency edges of different colors) can form on the two transaction instance, all sharing the structure: $\mathsf{txn1} \underset{\mathsf{RW}}{\overset{\mathsf{WR}}{\rightleftarrows}} \mathsf{txn2}$.

We refined our algorithm to exploit this observation by guiding the solver to find all such structurally similar anomalies once one of them is detected. It does so via an inner loop (lines 9-13) which repeats the steps discussed above, but first adds a clause, φ_{STCT}, that forces the solver to look for cycles that are structurally similar to the one found on line 5. φ_{STCT} is constructed by the function EncStruct which constrains the dependency edges and transactions that the solver considers on a cycle. The inner loop repeats until all such cycles are found. When the algorithm can no longer find any new anomaly with the shared structure, it breaks the loop in line 12, and the normal execution of the outer loop is resumed. We empirically demonstrate in section 7 that this optimized procedure is more performant than one that does not include the inner loop.

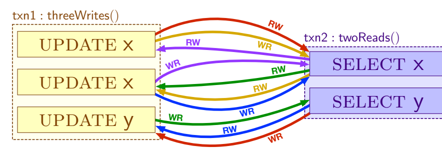

Fig. 19. Structurally Similar Anomalies

6.2 Prototype

CLOTHO[2] is a complete directed testing framework that combines the SAT-based approach of section 5 with a front-end compiler and test replay environment. The front-end compiler takes as input an arbitrary Java class which manipulates a database through the standard JDBC API, where each method is treated as a transaction. The compiler supports queries written in the subset of SQL given in Figure 5. Consequently, it does not allow nested queries, joins or update operations with implicit reads (e.g. `UPDATE T SET f=f+1`). These operations are not typically supported by distributed, weakly consistent databases, and translating application logic to avoid them is a necessary and standard step of porting a database application to a distributed systems.

Our compiler is implemented as a pass on Shimple programs which are produced by the Soot static analysis framework [Vallée-Rai et al. 1999]. The compiler initially performs a simple goto analysis to determine loops and conditional structures and marks all variables that are modified or read within them. It then detects all database access points, e.g. `executeUpdate()` function calls. For each access point, it then constructs an $\mathcal{AR}$ query that may contain typed holes. The compiler then walks backward through all assignments of variables used in the query and tries to fill the

[2] https://github.com/Kiarahmani/CLOTHO

holes with $\mathcal{AR}$ values. In the case that a variable is assigned a value produced by an unknown function call, the compiler creates a new abstract variable to use in place of the variable in the original query. CLOTHO then uses the algorithm presented in Figure 18 to search for anomalies, using Z3 [de Moura and Bjørner 2008] as a backend.

For each satisfying assignment found, CLOTHO extracts a set of test configuration folders, each of which contains a database initialization file and a set of Java class files annotated with execution orders and input parameters. CLOTHO's test administration system uses this information to replicate anomalous executions on a concrete environment. The test environment is initialized by creating a cluster of Docker containers running replicated database instances that have been instantiated according to the test configuration. The replayer supports any distributed or centralized database that has JDBC drivers. The replayer also spawns a set of Java worker containers executing transactions from the test configuration with the specified parameters. Partitioning requirements are implemented using the open-source Blockade [2019] tool which artificially induces network partitions between replica or worker containers as required by the test configuration. Transaction schedules are enforced by a custom JDBC driver that receives requests from application workers and acts as a *managed wrapper* to the driver for the underlying database. Requests are blocked until the driver communicates with the central scheduling unit using the RMI functionality in Java, and are only passed on to the actual driver once the permission to proceed has been acquired.

7 EVALUATION

The goal of our experimental evaluation was to investigate the following questions:

- **(R1)** Is CLOTHO *effective*, i.e. capable of detecting serializability anomalies in real weakly consistent database applications?
- **(R2)** Is CLOTHO *practical*, i.e. how many anomalies does it find and how long does it take to find them?
- **(R3)** Is our FOL encoding rich enough for cycles to be mapped to unserializable executions on a real datastore?
- **(R4)** How does CLOTHO compare to undirected or random testing at finding bugs?

Local evaluation of CLOTHO was carried out on a Macbook Pro with a 2.9GHz Intel Core i5 CPU and 16GB of memory, running Java 8 and equipped with Z3 v4.5.1.

7.1 Detecting Serialization Anomalies

To answer questions **(R1)-(R3)**, we carried out three experiments on the set of benchmarks shown in Table 2. These applications were drawn from the OLTP-Bench test suite [Difallah et al. 2013], and were designed to be representative of the heavy workloads seen by typical transaction processing and web-based applications that can also be deployed on weakly consistent data stores [Bailis et al. 2014b; Shasha and Bonnet 2003].

A few of these benchmarks have distinctive features that deserve special mention. SEATS is an online flight search and ticketing system with both read and write intensive transactions. TATP simulates a caller location system with non-conflicting transactions that are prone to be rolled back, which complicates compilation. SmallBank simulates a financial account management system comprised of transactions that repeatedly access the same objects and therefore heavily depend on transactional isolation guarantees. Twitter and Wikipedia are inspired from the popular online services featuring micro-blogging and collaborative editing facilities. The data model and the transactional logic of Wikipedia is notably complex, where transactions manipulate a large set of backed-up tables for user-authentication and concurrent edit history management. To apply CLOTHO to this test suite, we had to manually replace any unsupported database operations with equivalent operations, per section 6. Out of the 131 distinct queries found in these benchmarks, only 11 used `JOIN`, which we replaced with other supported queries and application-level loops. While we made these edits by hand, this process has been shown to be mechanizable [Tahboub et al. 2018].

Our first experiment investigated questions **(R1)** and **(R2)** by applying CLOTHO to detect serializability anomalies of a fixed length in the benchmark applications. The first eight columns of Table 2 presents the results of this experiment. For each benchmark the table lists the number of transactions (#Txns) and number of tables (#Tables) it includes, the maximum bound on cycle length considered (Max. Length), the total number of anomalies found (#Anmls), unique number of structures (#Dist Strcts), the average time needed to detect a single anomaly (Avg Anlys), and the total analysis time (Total Anlys). For each benchmark, we initially limited the length of cycles to be at most 4, as all the canonical serializability anomalies, e.g. dirty reads and lost updates are of this length [Berenson et al. 1995]. As section 7.2 will discuss in more detail, this bound is also sufficient to discover executions violating every correctness criteria in the specification of the TPC-C benchmark. For the two benchmarks in which no anomaly of length 4 was found, TATP and Voter, we iteratively increased the bound but were unable to find any longer anomalies. These results demonstrate that CLOTHO can effectively detect anomalies in a reasonable amount of time on our benchmark applications, needing about an hour to analyze our most complex application.

Benchmark	#Txns	#Tables	Max. Length	#Anmls	#Dist. Strcts	Avg Anlys	Total Anlys	Auto Replay	Man. Replay
SEATS	6	10	4	32	18	26s	1970s	26	6
TATP	7	4	15	0	0	-	55s	-	-
TPC-C	5	9	4	22	13	118s	3270s	18	4
SmallBank	6	3	4	60	15	3s	264s	50	10
Voter	1	3	20	0	0	-	8s	-	-
Twitter	5	5	4	2	2	19s	211s	2	0
Wikipedia	5	12	4	3	3	227s	4343s	2	1

Table 2. Serializability anomalies found and manifested in OLTP benchmark suite.

Our next experiment tackled **(R3)** by using the test configuration information generated by CLOTHO to manifest potential anomalies on the popular Cassandra database storage engine. We chose Cassandra because it only guarantees eventual consistency and provides no transactional support, allowing for all serializability anomalies to potentially occur and be caught.

Column (Auto Replay) in Table 2 lists the number of unserializable executions CLOTHO's test administration framework was able to exhibit from the test configurations generated by CLOTHO, accounting for over 80% of detected anomalies. There were two main reasons this process failed for the other anomalies: underspecified loop boundaries in the encoding and the mismatch between the order of fetched rows in the encoding and at the runtime. Nevertheless, we were able to edit the generated test configurations to include enough information to exhibit an unserializable executions in all the remaining cases. In particular, by specifying the loop boundaries in the encoding and modifying the order at which initial database rows are inserted. Column (Man. Replay) lists the number of anomalies that required such interventions.

To test the effectiveness of the optimized search algorithm presented in section 6.1, we created an unoptimized variant of FindAnomalies by removing its inner do-while loop. To compare the two algorithms, we applied the unoptimized version to the three benchmarks with the most anomalies, using the total analysis of the optimized

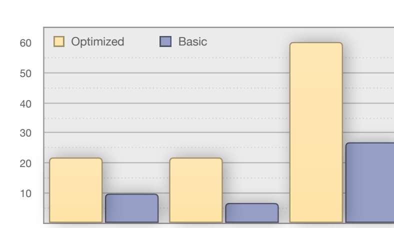

Fig. 20. Optimized vs basic analysis

version as a timeout. Figure 20 presents the number of anomalies found by the basic version against the optimized. We can see that the optimized algorithm performs better than the unoptimized version, reporting more than double the anomalies in the same amount of time for each benchmark.

7.2 Comparison with Random Testing

Our final experiment was designed to compare CLOTHO's ability to find "real" bugs over automated blackbox testing. To do so, we used the TPC-C benchmark from the previous section, as it specifies a set of twelve *consistency requirements* (CR) that must be preserved by any sound database system. These CRs give us objective notions of buggy behaviors that can arise in realistic OLTP applications, which we used to compare the relative effectiveness of CLOTHO and random testing. The rows (CR1 – CR12) of Table 3 list all of TPC-C consistency requirements. We have broken "if and only if" requirements into separate rows.

Inv.	Source	Description	Broken by rand testing? 10c	50c	100c
CR1	spec	W_YTD = sum(D_YTD)	✓	✓	✓
CR2	spec	D_NEXT_O_ID-1=max(O_ID)=max(NO_O_ID)	✗	✗	✓
CR3	spec	max(NO_O_ID)-min(NO_O_ID)+1=[number of corresponding rows in NEW-ORDER]	✗	✗	✓
CR4	spec	sum(O_OL_CNT) = [number of corresponding rows in ORDER-LINE]	✓	✓	✓
CR5A	spec	For any row in NEW_ORDER there is a row in ORDER with O_CARRIER_ID set to null	✗	✗	✗
CR5B	spec	For any row in ODER with O_CARRIER_ID set as null there is row in NEW_ORDER	✗	✗	✗
CR6	spec	O_OL_CNT is equal to the number of rows in ORDER-LINE for that order	✓	✓	✓
CR7A	spec	O_CARRIER_ID=null if all rows in ORDER_LINE have OL_DELIVERY_D=null	✗	✗	✗
CR7B	spec	all rows in ORDER_LINE have OL_DELIVERY_D=null if O_CARRIER_ID=null	✗	✗	✗
CR8	spec	W_YTD = sum(H_AMOUNT)	✓	✓	✓
CR9	spec	D_YTD = sum(H_AMOUNT)	✓	✓	✓
CR10	spec	C_BALANCE = sum(OL_AMOUNT) - sum(H_AMOUNT)	✗	✓	✓
CR11	spec	(count(*) from ORDER) - (count(*) from NEW-ORDER) = 2100	✓	✓	✓
CR12	spec	C_BALANCE + C_YTD_PAYMENT = sum(OL_AMOUNT)	✗	✓	✓
NCR1	anlys	s_ytd=sum_exec(ol_quantity)	✗	✓	✓
NCR2	anlys	sum(s_order_cnt) = sum_exec(o_ol_cnt)	✗	✓	✓
NCR3	anlys	In the absence of DELIVERY, (c_ytd_payment+c_balance) must be constant for each cust	✗	✗	✗
NCR4	anlys	c_payment_cnt = initial_value + #PAYMENT on that customer in execution log	✗	✗	✗
NCR5	anlys	sum(C_DELIVERY_CNT) = initial_value + number of times DELIVERY is called	✓	✓	✓
NCR6	anlys	ratio of different o_carrier_id must respect the corresponding ratio on DELIVERY's inputs	✗	✓	✓
NCR7	anlys	H_AMOUNT for each customer = sum(payment_amounts) for that customer in log	✗	✗	✗

Table 3. Targeted random testing results

In addition to CRs, we identified seven new consistency requirements (NCR1-NCR7) that should also be preserved by any serializable execution of TPC-C. We identified these invariants after examining the serializability anomalies generated by CLOTHO. These requirements capture more subtle safety properties of the application, descriptions of each of which is also shown in Table 3. As an example, NCR4 captures that a customer's `c_payment_cnt` field is uniquely determined by its initial value and the number of times `payment` transaction has been called on that customer.

To the best of our knowledge, no random testing tool exists that can be deployed on weakly consistent databases and that can also handle our targeted benchmark suit. Thus, we were forced to develop our own tool for testing high-level application properties. Our framework builds upon Jepsen [2018], a popular open source library for testing distributed databases and systems. In addition to test automation and logging infrastructure, Jepsen is capable of systematically injecting faults (e.g. network partitions or node clock skews) during a test. Our framework extends Jepson to:

(i) manage concurrent Java clients that share a JDBC connection pool, (ii) support enhanced logging of client-side generated information and (iii) perform a set of customized database-wide safety checks, each of which is required for targeted testing of application-level properties in TPC-C. We evaluated our testing framework on a fully replicated Apache Cassandra cluster running on a configurable number of `t2.xlarge` EC2 instances.

In order to ensure a fair comparison and mimic how applications are tested in practice, we ensured that: (i) the initial state of the database was realistic (ii) each transaction call's arguments were realistic (iii) the maximum Cassandra throughput is achieved and reported for each test. To achieve (i) and (ii), all our tests used initial database states and transaction parameters taken directly from OLTP-Bench. In addition, we used the smallest available tables in order to make sure that the probability of conflicts was maximized. Our tool utilizes Cassandra's sstableloader tool to automatically and efficiently bulk-load snapshots into the cluster before each test. Lastly, we made sure that none of our design decisions, e.g. the choice of Cassandra's secondary indices would compromise throughput, resulting in an unrealistically low load that would minimize the chance for conflicts.

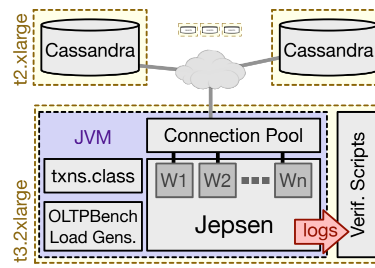

Fig. 21. Testing framework

We repeated each experiment with 10, 50 and 100 concurrent clients, each submitting 6 transactions per second, where the frequency of transactions adhered to TPC-C's workload requirements. These settings achieved approximately 20%, 80% and 100% of the maximum throughput we witnessed on the deployment. Each experiment was repeated with a 10 minute and then 30 minute timeout. Table 3 presents the results of these experiments, with ✓ and ✗ recording when an invariant was violated and was not violated, respectively. As the table shows, our random testing framework was only able to violate 14 out of 21 (67%) requirements.

#	Len	Txns	Tables	Type	Broken Invariants	Anlys	Rply
1	4	pm/pm	WAREHOUSE	lost update	CR1, CR8	5s	7s
2	4	pm/pm	DISTRICT	lost update	CR1, CR9	19s	12s
3	4	no/no	DISTRICT	lost update	CR2, ..., CR7, CR11, NCR1	15s	13s
4	4	dv/dv	CUSTOMER	lost update	CR10, CR12, NCR5	171s	12s
5	4	dv/dv	NEW_ORDER	lost update	NCR6	16s	6s
6	4	pm/pm	HISTORY	lost update	CR8,CR9, CR10, NCR7	125s	11s
7	4	no/no/no	STOCK	unclassified	NCR1, NCR2	697s	13s
8	4	no/no	STOCK	lost update	NCR1, NCR2	70s	12s
9	4	dv/os/no	ORDER_LINE	unclassified	none	11s	12s
10	4	dv/os/no	ORDER_LINE	unclassified	none	9s	15s
11	4	dv/pm	CUSTOMER	lost update	none	231s	9s
12	4	dv/pm	CUSTOMER	lost update	NCR3,NCR4	12s	11s
13	4	pm/pm	CUSTOMER	lost update	NCR3,NCR4,CR12	108s	8s

Table 4. Internal serializability anomalies found on TPC-C

Alternatively, we mapped each of TPC-C's internal serializability anomalies found by CLOTHO to the set of invariants that could be broken in scenarios arising from those anomalies. Table 4 lists the details of the internal serializability anomalies of length 4 CLOTHO finds for TPC-C along with the set of requirements each anomaly breaks. We found that every single one of TPC-C's consistency requirements was broken as a result of at least one serializability anomaly.

These results evidence the importance of (internal) serializability anomalies for detection of application-level bugs and showcase CLOTHO's ability to efficiently construct many of such anomalies that are present in a realistic database. CLOTHO reports these anomalies as slowed-down interleaved execution steps in a form comprehensible to the developers.

8 RELATED WORK

Serializability is a well-studied problem in the database community, and there have been a number of efforts over the years to develop static techniques to discover serializability anomalies in concurrent database systems. For example, early works by Fekete et al. [2005] and Jorwekar et al. [2007] proposed lightweight syntactic analyses to check for serializability anomalies in centralized databases, by looking for dangerous structures in the over-approximated static graph of all possible dynamic dependency conflicts. Several recent works [Bernardi and Gotsman 2016; Cerone et al. 2015; Cerone and Gotsman 2016; Cerone et al. 2017; Warszawski and Bailis 2017; Zhang et al. 2013] have continued along this line, by deriving different types of problematic structures in dependency graphs that are possible under different weak consistency mechanisms, and then checking for these structures on static dependency graphs.

However, as we have also demonstrated in this paper, using just static dependency graphs yields 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 efforts in this space [Bernardi and Gotsman 2016; Cerone and Gotsman 2016; Cerone et al. 2017] recognize this and propose 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.

Like our work, [Kaki et al. 2018] and [Nagar and Jagannathan 2018] also model application logic and consistency specifications using a decidable fragment of first-order logic (FOL), so that an underlying solver could 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. Although their core language supports SQL operators, they do not incorporate their techniques into a full test-and-reply environment that allows mapping anomalies identified in abstract executions to be translated to concrete inputs that can be executed in a test environment.

Brutschy et al. [2017, 2018] recently proposed an analysis technique to find serializability violations using a dependency graph-based approach. Their notion of dependencies is very conservative and is reliant upon accurate characterization of commutativity and so-called "absorption" relations between operations. Further, in a bid to make dependency characterizations local i.e. relying only on the operations involved in the dependency, they over-approximate the impact of other operations on commutativity and absorption (resulting in what they call as "far-commutativity" and "far-absorption").

The localization strategy presented in [Brutschy et al. 2018], however, does not suit query-based models where dependences between two operations cannot be decided locally, but are reliant on other operations. For example, following their approach, there will always be a dependence between `UPDATE` queries and `SELECT` queries that access the same field *on any row*, because there is always the possibility that the values from the row chosen by the `UPDATE` is copied into the row chosen by the `SELECT`. It is unclear how to accurately express far-commutativity and far-absorption between SQL statements performing predicate read and writes. Handling such fine-grained dependencies forms an important contribution of our work.

There have been several recent proposals to reason about programs executing under weak consistency [Alvaro et al. 2011; Bailis et al. 2014a; Balegas et al. 2015; Gotsman et al. 2016; Li et al. 2014, 2012]. All of them assume a system model that offers a choice between a *coordination-free* weak consistency level (*e.g.*, eventual consistency [Alvaro et al. 2011; Bailis et al. 2014a; Balegas et al. 2015; Li et al. 2014, 2012]) or causal consistency [Gotsman et al. 2016; Lesani et al. 2016]). In contrast, our focus is agnostic to the particular consistency guarantees provided by the underlying storage system, and is concerned with statically identifying violations of a particular class of invariants, namely those that hold precisely when transactions exhibit serializable behavior.

[Warszawski and Bailis 2017] presents a dynamic analysis for weak isolation that attempts to discover weak isolation anomalies from SQL log files. Their solution, while capable of identifying database attacks due to the use of incorrect isolation levels, does not consider how to help ascertain application correctness, in particular determining if applications executing on storage systems that expose guarantees weaker than serializability are actually correct.

In addition to the mentioned threads of work on database applications, in the recent years several proposals have been made addressing similar challenges in the context of shared memory concurrent programming models. For example, static [Flanagan and Qadeer 2003] and dynamic [Flanagan et al. 2004] approaches have been introduced to determine if a particular locking policy in such programs actually enforces a desired non-interference property. Using code annotation and custom memory allocations, Rajamani et al. [2009] extended these works and proposed a framework which allows detection of violations even in the presence of ill-behaved threads that disobey the locking discipline.

In order to liberate users from writing low-level synchronization mechanisms, a whole-program analysis is offered by McCloskey et al. [2006] which compiles user-declared pessimistic atomic sets and blocks, similar to transactional memory models, into performant locking policies.

While all above approaches rely on the programmer to annotate atomic regions in their code, as a solution to an orthogonal problem, [Lu et al. 2007] presents MUVI, a static analysis tool for inferring correlated variables which must be accessed together, i.e. atomically. Several dynamic frameworks also attempted to detect potentially erroneous program executions without depending on user annotations and by looking for dangerous access patters at runtime, either for a single shared variable [Xu et al. 2005] or multiple variables [Hammer et al. 2008].

Similar to our approach, [Huang et al. 2013] and [Machado et al. 2015] rely on symbolic constraint solving to construct full, failing, multithreaded schedules that manifest concurrency bugs. However, unlike these approaches which depend on dynamic path profiling to detect conflicting operations, we analyze programs in an intermediate language which can be generated from any source language using a proper compiler. Our approach has the additional completeness guarantee that no anomaly within the given bounds is missed.

It is worth noting that all shared-memory systems assume a coherence order on writes (a property that is not enforced in weakly consistent systems) rendering any existing analysis framework for such systems inapplicable to replicated database models. Such shared-memory systems do not permit certain executions which can occur in eventually consistent data-stores while CLOTHO's semantics is rich enough to model these sorts of executions. As an example, consider the simple execution in Figure 22 with two concurrent programs. Assuming x is initially 0, models of shared-memory systems disallow this execution, as they force the second read event in P_1 to witness P_0's write. No such restriction applies in a weakly consistent environment, which may allow the second read to be routed to a replica that has not yet received the update from P_0's write event.

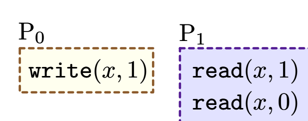

Fig. 22. An impossible execution in a shared memory system

Several violation detection mechanisms of developers' atomicity assumption for code regions in shared memory programs have been recently introduced where certain conflicting access patterns have been deemed safe due to lack of data flow between the read and write operations involved in such conflicts [Lu et al. 2006; Lucia et al. 2010]. Unlike their purely structural approach which only considers patterns within individual operations, our notion of internal serializability (to the best of our knowledge) is the first formulation of a semantic property of whole programs which takes into account actual data-dependencies between arbitrary read and write operations that form a conflicting cycle. For example, following the approach presented in these works, the external serializability anomaly presented Figure 14c and the internal serializability anomaly presented in Figure 14d are both considered harmful, as they include conflicting read and writes from distinct operations.

There have been a number of testing frameworks developed for distributed systems over the years. Some notable examples include MaceMC [Killian et al. 2007] a model-checker that discovers liveness bugs in distributed programs, and Jepsen [2018], a random testing tool that checks partition tolerance of NoSQL distributed database systems with varying consistency levels to enable high-availability. CLOTHO differs from these in a number of important ways. In particular, MaceMC does not consider safety issues related to replication, while Jepsen is purely a dynamic analysis that does not leverage semantic properties of the application in searching for faulty executions. Both of these considerations have played an important role in shaping CLOTHO's design and implementation.

9 CONCLUSIONS AND FUTURE WORKS

This paper presents CLOTHO, an automated test generation and replaying tool for applications of weakly consistent distributed databases. CLOTHO is backed by a precise encoding of database applications allowing it to maintain fine-grained dependency relations between complex SQL query and update operations. Notably, the encoding enforces special relationships on operations that capture concrete execution details required for an automated and complete replay of anomalous scenarios without any input from the users. We applied CLOTHO on a suite of real-world database benchmarks where it successfully constructed and replayed a wide range of bugs, providing strong evidence for its applicability. Additionally, we compared our approach to a state-of-the-art random testing tool where CLOTHO performed more efficiently and more reliably.

Concurrency anomalies detected by CLOTHO can be straightforwardly fixed by using stronger database-offered consistency/isolation guarantees (e.g. by using serializable transactions). Prescribing more optimal fixes, based on the structure of dependency cycles that CLOTHO generates is an intriguing research problem which we leave for the future works.

REFERENCES

2010. TPC-C Benchmark. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf. Online; Accessed 20 April 2018.

Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. Cambridge, MA, USA. AAI0800775.

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

Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings.* 249–260.

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014a. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014b. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14).* ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2588555.2588562

Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting the Consistency back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer System (EuroSys '15).* Bordeaux, France. http://lip6.fr/Marc.Shapiro/papers/putting-consistency-back-EuroSys-2015.pdf

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

Giovanni Bernardi and Alexey Gotsman. 2016. Robustness against Consistency Models with Atomic Visibility. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*. 7:1–7:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.7

Blockade. 2019. Blockade. https://github.com/worstcase/blockade accessed on 2019-03-29.

Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. 2017. 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*. 458–472. http://dl.acm.org/citation.cfm?id=3009895

Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. 2018. 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*. 90–104. https://doi.org/10.1145/3192366.3192415

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

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

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2017. Algebraic Laws for Weak Consistency. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*. 26:1–26:18. https://doi.org/10.4230/LIPIcs.CONCUR.2017.26

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288. https://doi.org/10.14778/2732240.2732246

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

Cormac Flanagan, Cormac Flanagan, and Stephen N Freund. 2004. Atomizer: A Dynamic Atomicity Checker for Multi-threaded Programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 256–267. https://doi.org/10.1145/964001.964023

Cormac Flanagan and Shaz Qadeer. 2003. A Type and Effect System for Atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 338–349. https://doi.org/10.1145/781131.781169

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. '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)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. 2008. Dynamic Detection of Atomic-set-serializability Violations. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 231–240. https://doi.org/10.1145/1368088.1368120

Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/2491956.2462167

Jepsen 2018. https://jepsen.io/

Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. 2007. 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*. 1263–1274. http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf

Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe Replication Through Bounded Concurrency Verification. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 164 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276534

Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &*

Implementation (NSDI'07). USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1973430.1973448

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 357–370. https://doi.org/10.1145/2837614.2837622

Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 281–292. http://dl.acm.org/citation.cfm?id=2643634.2643664

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 103–116. https://doi.org/10.1145/1294261.1294272

Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1168857.1168864

Brandon Lucia, Luis Ceze, and Karin Strauss. 2010. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-variable Atomicity Violations. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 222–233. https://doi.org/10.1145/1815961.1815988

Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015. Concurrency Debugging with Differential Schedule Projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 586–595. https://doi.org/10.1145/2737924.2737973

Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. 2006. Autolocker: Synchronization Inference for Atomic Sections. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 346–358. https://doi.org/10.1145/1111037.1111068

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*. 41:1–41:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.41

Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. https://doi.org/10.1145/322154.322158

Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannatha. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systtems (Extended Version). https://arxiv.org/abs/1806.08416

Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. 2009. ISOLATOR: Dynamically Ensuring Isolation in Comcurrent Programs. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/1508244.1508266

Dennis Shasha and Philippe Bonnet. 2003. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 307–322. https://doi.org/10.1145/3183713.3196893

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

Todd Warszawski and Peter Bailis. 2017. 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*. 5–20. https://doi.org/10.1145/3035918.3064037

Min Xu, Rastislav Bodík, and Mark D. Hill. 2005. A Serializability Violation Detector for Shared-memory Server Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/1065010.1065013

Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. 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*. 276–291. https://doi.org/10.1145/2517349.2522729