Repairing Serializability Bugs in Distributed Database Programs via Automated Schema Refactoring

Kia Rahmani
Purdue University, USA
rahmank@purdue.edu

Kartik Nagar
IIT Madras, India
nagark@cse.iitm.ac.in

Benjamin Delaware
Purdue University, USA
bendy@purdue.edu

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

Abstract

Serializability is a well-understood concurrency control mechanism that eases reasoning about highly-concurrent database programs. Unfortunately, enforcing serializability has a high performance cost, especially on geographically distributed database clusters. Consequently, many databases allow programmers to *choose* when a transaction must be executed under serializability, with the expectation that transactions would only be so marked when necessary to avoid serious concurrency bugs. However, this is a significant burden to impose on developers, requiring them to (a) reason about subtle concurrent interactions among potentially interfering transactions, (b) determine when such interactions would violate desired invariants, and (c) then identify the minimum number of transactions whose executions should be serialized to prevent these violations. To mitigate this burden, this paper presents a sound and fully automated schema refactoring procedure that transforms a program's data layout – rather than its concurrency control logic – to eliminate statically identified concurrency bugs, allowing more transactions to be safely executed under weaker and more performant database guarantees. Experimental results over a range of realistic database benchmarks indicate that our approach is highly effective in eliminating concurrency bugs, with safe refactored programs showing an average of 120% higher throughput and 45% lower latency compared to a serialized baseline.

CCS Concepts: • **Software and its engineering** → *System modeling languages*; *Application specific development environments*; • **Computing methodologies** → *Distributed computing methodologies.*

Keywords: Schema Refactoring, Serializability, Weak Consistency, Weak Isolation

ACM Reference Format:
Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2021. Repairing Serializability Bugs in Distributed Database Programs via Automated Schema Refactoring. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454028

1 Introduction

Programs that concurrently access shared data are ubiquitous: bank accounts, shopping carts, inventories, and social media applications all rely on a shared database to store information. For performance and fault tolerance reasons, the underlying databases that manage state in these applications are often replicated and distributed across multiple, geographically distant locations [34, 36, 48, 51]. Writing programs which interact with such databases is notoriously difficult, because the programmer has to consider an exponential space of possible interleavings of database operations in order to ensure that a client program behaves correctly. One approach to simplifying this task is to assume that sets of operations, or *transactions*, executed by the program are *serializable* [40], i.e. that the state of the database is always consistent with some sequential ordering of those transactions. One way to achieve this is to rely on the underlying database system to seamlessly enforce this property. Unfortunately, such a strategy typically comes at a considerable performance cost. This cost is particularly significant for distributed databases, where the system must rely on expensive coordination mechanisms between different replicas, in effect limiting *when* a transaction can see the effects of another in a way that is consistent with a serializable execution [5]. This cost is so high that developers default to weaker consistency guarantees, using careful design and testing to ensure correctness, only relying on the underlying system to enforce serializable transactions when serious bugs are discovered [27, 37, 45, 50].

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454028

Uncovering such bugs is a delicate and highly error-prone task even in centralized environments: in one recent study, Warszawski and Bailis [56] examined 12 popular E-Commerce applications used by over two million well-known websites and discovered 22 security vulnerabilities and invariant violations that were directly attributable to non-serializable transactions. To help developers identify such bugs, the community has developed multiple program analyses that report potential *serializability anomalies* [12, 13, 27, 31, 39]. Automatically repairing these anomalies, however, has remained a challenging open problem: in many cases full application safety is only achievable by relying on the system to enforce strong consistency of all operations. Such an approach results in developers either having to sacrifice performance for the sake of correctness, or conceding to operate within a potentially restricted ecosystem with specialized services and APIs [4] architectured with strong consistency in mind.

In this paper, we propose a novel language-centric approach to resolving concurrency bugs that arise in these distributed environments. Our solution is to alter the *schema*, or data layout, of the data maintained by the database, rather than the consistency levels of the transactions that access that data. Our key insight is that it is possible to modify shared state to remove opportunities for transactions to witness changes that are inconsistent with serializable executions. We, therefore, investigate automated schema transformations that change *how* client programs access data to ensure the absence of concurrency bugs, in contrast to using expensive coordination mechanisms to limit *when* transactions can concurrently access the database.

For example, to prevent transactions from observing non-atomic updates to different rows in different tables, we can fuse the offending fields into a single row in a single table whose updates are guaranteed to be atomic under any consistency guarantee. Similarly, consecutive reads and writes on a row can be refactored into "functional" inserts into a new table, which removes the race condition between concurrently running instances of the program. By changing the schema (and altering how transactions access data accordingly), without altering a transaction's atomicity and isolation levels, we can make clients of distributed databases *safer* without *sacrificing performance*. In our experimental evaluation, we were able to fix on average 74% of all identified serializability anomalies with only a minimal impact (less than 3% on average) on performance in an environment that provides only weak eventually consistent guarantees [14]. For the remaining 26% of anomalies that were not eliminated by our refactoring approach, simply marking the offending transactions as serializable yields a provably safe program that nonetheless improves the throughput (resp. latency) of its fully serialized counterpart by 120% (resp. 45%) on average.

This paper makes the following contributions:

1. We observe that serializability violations in database programs can be eliminated by changing the schema of the underlying database and the client programs in order to eliminate problematic accesses to shared database state.
2. Using this observation, we develop an automated refactoring algorithm that iteratively repairs statically identified serializability anomalies in distributed database clients. We show this algorithm both preserves the semantics of the original program and eliminates many identified serializability anomalies.
3. We develop a tool, Atropos[1], implementing these ideas, and demonstrate its ability to reduce the number of serializability anomalies in a corpus of standard benchmarks with minimal performance impact over the original program, but with substantially stronger safety guarantees.

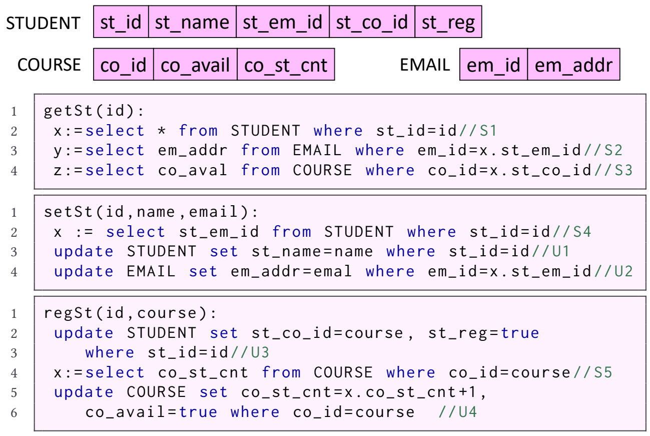
The remainder of the paper is structured as follows. The next section presents an overview of our approach. Section 3 defines our programming model and formalizes the notion of concurrency bugs. Section 4 provides a formal treatment of our schema refactoring strategy. Sections 5 and 6 describe our repair algorithm and its implementation, respectively. Section 7 describes our experimental evaluation. Related work and conclusions are given in Section 8 and Section 9.

2 Overview

To illustrate our approach, consider an online course management program that uses a database to manage a list of course offerings and registered students. Figure 1 presents a simplified code snippet implementing such a program. The database consists of three tables, maintaining information regarding courses, students, and their email addresses. The `STUDENT` table maintains a reference to a student's email entry in schema `EMAIL` (via secondary key `st_em_id`) and a reference to a course entry in table `COURSE` (via secondary key `st_co_id`) that the student has registered for. A student's registration status is stored in field `st_reg`. Each entry in table `COURSE` also stores information about the availability of a course and the number of enrolled students.

The program includes three sets of database operations or *transactions*. Transaction `getSt`, given a student's id, first retrieves all information for that student (`S1`). It then performs two queries, (`S2` and `S3`), on the other tables to retrieve their email address and course availability. Transaction `setSt` takes a student's id and updates their name and email address. It includes a query (`S4`) and an update (`U1`) to table `STUDENT` and an update to the `EMAIL` table (`U2`). Finally, transaction `regSt` registers a student in a course. It consists of an update to the student's entry (`U3`), a query to `COURSE` to determine the number of students enrolled in the course they

[1] https://github.com/Kiarahmani/AtroposTool



STUDENT st_id st_name st_em_id st_co_id st_reg

COURSE co_id co_avail co_st_cnt EMAIL em_id em_addr

```
getSt(id):
 x:=select * from STUDENT where st_id=id//S1
 y:=select em_addr from EMAIL where em_id=x.st_em_id//S2
 z:=select co_aval from COURSE where co_id=x.st_co_id//S3
```

```
setSt(id,name,email):
 x := select st_em_id from STUDENT where st_id=id//S4
 update STUDENT set st_name=name where st_id=id//U1
 update EMAIL set em_addr=emal where em_id=x.st_em_id//U2
```

```
regSt(id,course):
 update STUDENT set st_co_id=course, st_reg=true
    where st_id=id//U3
 x:=select co_st_cnt from COURSE where co_id=course//S5
 update COURSE set co_st_cnt=x.co_st_cnt+1,
    co_avail=true where co_id=course  //U4
```

Figure 1. Database schemas and code snippets from an online course management program

wish to register for (S5), and an update to that course's availability (U4) indicating that it is available now that a student has registered for it.

The desired semantics of this program is these transactions should be performed *atomically* and in *isolation*. Atomicity guarantees that a transaction never observes intermediate updates of another transaction. Isolation guarantees that a transaction never observes changes to the database by other committed transactions once it begins executing. Taken together, these properties ensure that all executions of this program are *serializable*, yielding behavior that corresponds to some sequential interleaving of these transaction instances.

While serializability is highly desirable, it requires using costly centralized locks [25] or complex version management systems [10], which severely reduce the system's available concurrency, especially in distributed environments where database state may be replicated or partitioned to improve availability. In such environments, enforcing serializability typically either requires coordination among all replicas whenever shared data is accessed or updated, or ensuring replicas always witness the same consistent order of operations [16]. As a result, in most modern database systems, transactions can be executed under weaker isolation levels, e.g. permitting them to observe updates of other committed transactions during their execution [34, 38, 43, 48]. Unfortunately, these weaker guarantees can result in *serializability anomalies*, or behaviors that would not occur in a serial execution. To illustrate, Figure 2 presents three concurrent executions of this program's transaction instances that exhibit non-serializable behaviors.

The execution on the left shows instances of the getSt and setSet transactions. Following the order in which operations execute (denoted by red arrows), observe that (S2) witnesses the update to a student's email address, but (S1) does not see their updated name. This anomaly is known as a *non-repeatable read*. The execution in the center depicts the concurrent execution of instances of getSt and regSt. Here, (S1) witnesses the effect of (U3) observing that the student is registered, but (S3) sees that the course is unavailable, since it does not witness the effect of (U4). This is an instance of a *dirty-read* anomaly. Lastly, the execution on the right shows two instances of regSt that attempt to increment the number of students in a course. This undesirable behavior, known as a *lost update*, leaves the database in a state inconsistent with any sequential execution of the two transaction instances. All of these anomalies can arise if the strong atomicity and isolation guarantees afforded by serializability are weakened.

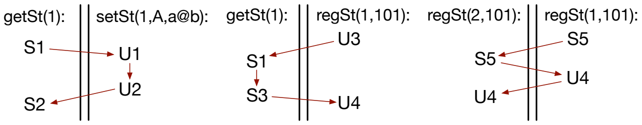


Figure 2. Serializability Anomalies

Several recent proposals attempt to identify such undesirable behaviors in programs using a variety of static or dynamic program analysis and monitoring techniques [12, 13, 39, 56]. Given potential serializability violations, the standard solution is to strengthen the atomicity and isolation requirements on the offending transactions to eliminate the undesirable behaviour, at the cost of increased synchronization overhead or reduced availability [7, 27, 50].

Atropos. Are developers obligated to sacrifice concurrency and availability in order to recover the pleasant safety properties afforded by serializability? Surprisingly, we are able to answer this question in the negative. To see why, observe that a database program consists of two main components - a set of computations that includes transactions, SQL operations (e.g., selects and updates), locks, isolation-level annotations, etc.; and a memory abstraction expressed as a relational schema that defines the layout of tables and the relationship between them. The traditional candidates picked for repairing a serializability anomaly are the transactions from the computational component: by injecting additional concurrency control through the use of locks or isolation-strengthening annotations, developers can control the degree of concurrency permitted, albeit at the expense of performance and availability.

This paper investigates the under-explored alternative of transforming the program's schema to reduce the number of potentially conflicting accesses to shared state. For example, by *aggregating* information found in multiple tables into a single row on a single table, we can exploit built-in *row-level atomicity* properties to eliminate concurrency bugs that arise because of multiple non-atomic accesses to different table state. Row-level atomicity, a feature supported in most database systems, guarantees that other concurrently executing transactions never observe partial updates to a

STUDENT	**st_id**	st_name	st_em_id	st_em_addr	st_co_id	st_co_avail	st_reg

COURSE_CO_ST_CNT_LOG	**co_id**	**log_id**	co_st_cnt_log

```
getSt(id):
 x:=select * from STUDENT where st_id=id //RS1,RS2,RS3
```

```
setSt(id,name,email):
 update STUDENT set st_name=name,st_em_addr=email
    where st_id=id //RU1,RU2
```

```
regSt(id,course):
 update STUDENT set st_co_id=course, st_co_avail=true,
    st_reg=true where st_id=id //RU3
 insert into COURSE_CO_ST_CNT_LOG values
    (co_id=course,log_id=uuid(),co_st_cnt_log=1) //RU4
```

Figure 3. Refactored transactions and database schemas

particular row. Alternatively, it is possible to *decompose* database state to minimize the number of distinct updates to a field, for example by logging state changes via table *inserts*, rather than recording such changes via *updates*. The former effectively acts as a functional update to a table. To be sure, these transformations affect read and write performance to database tables and change the memory footprint, but they notably impose no additional synchronization costs. In scalable settings such as replicated distributed environments, this is a highly favorable trade-off since the cost of global concurrency control or coordination is often problematic in these settings, an observation that is borne our in our experimental results.

To illustrate the intuition behind our approach, consider the database program depicted in Figure 3. This program behaves like our previous example, despite featuring very different database schemas and transactions. The first of the two tables maintained by the program, STUDENT, removes the references to other tables from the original STUDENT table, instead maintaining independent fields for the student's email address and their course availability. These changes make the original course and email tables obsolete, so they have been removed. In addition, the number of students in each course is now stored in a dedicated table COURSE_CO_ST_CNT_LOG. Each time the enrollment of a course changes, a new record is inserted into this table to record the change. Subsequent queries can retrieve all corresponding records in the table and aggregate them in the program itself to determine the number of students in a course.

The transactions in the refactored program are also modified to reflect the changes in the data model. The transaction getSt now simply selects a single record from the student table to retrieve all the requested information for a student. The transaction setSt similarly updates a single record. Note that both these operations are executed atomically, thus eliminating the problematic data accesses in the original program. Similarly, regSt updates the student's st_co_id field and inserts a new record into the schema COURSE_CO_ST_CNT_LOG. Using the function uuid() ensures that a new record is inserted every time the transaction is called. These updates remove potential serializability anomalies by replacing the disjoint updates to fields in different tables from the original with a simple atomic row insertion. Notably, the refactored program can be shown to be a meaningful *refinement* of the original program, despite eliminating problematic serializability errors found in it. Program refinement ensures that the refactored program maintains *all* information maintained by the original program without exhibiting *any* new behaviour.

The program shown in Figure 3 is the result of several database schema refactorings [3, 21, 24], incremental changes to a database program's data model along with corresponding semantic-preserving modifications to its logic. Manually searching for such a refactored program is unlikely to be successful. On one hand, the set of potential solutions is large [3], rendering any manual exploration infeasible. On the other hand, the process of rewriting an application for a (even incrementally) refactored schema is extremely tedious and error-prone [55].

We have implemented a tool named Atropos that, given a database program, explores the space of its possible schema and program refactorings, and returns a new version with possibly many fewer concurrency bugs. The refactored program described above, for example, is automatically generated by Atropos from the original shown in Figure 1. Figure 4 presents the Atropos pipeline. A static analysis engine is used to identify potential serializability anomalies in a given program. The program is then pre-processed to extract the components which are involved in at least one anomaly, in order to put it into a form amenable for our analysis. Next, a refactoring engine applies a variety of transformations in an attempt to eliminate the bugs identified by our static analysis. Finally, the program is analyzed to eliminate dead code, and the refactored version is then reintegrated into the program from which it was extracted.

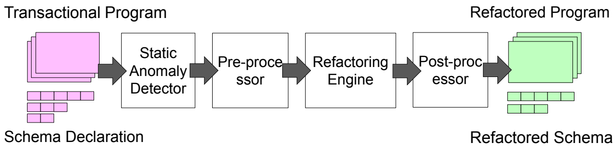


Figure 4. Schematic overview of Atropos

3 Database Programs

We adopt a commonly-used model for database applications [41, 42, 44], in which programs consist of a statically known set of *transactions* that are comprised of a combination of control flow and database operations. The syntax of our database programs is given in Figure 5. A program P is defined in terms of a set of database schemas ($\overline{R}$), and a set

$$
\begin{array}{lcl@{\qquad}lcl}
f & \in & \texttt{FldName} & \oplus & \in & \{+, -, \times, /\} \\
\rho & \in & \texttt{SchmName} & \odot & \in & \{<, \leq, =, >, \geq\} \\
t & \in & \texttt{TxnName} & \circ & \in & \{\wedge, \vee\} \\
a & \in & \texttt{Arg} & T & := & t(\overline{a})\{c; \texttt{return}\ e\} \\
x & \in & \texttt{Var} & R & := & \rho : \overline{f} \\
n & \in & \texttt{Val} & F & := & \langle \overline{f : n} \rangle \\
\texttt{agg} & \in & \{\texttt{sum}, \texttt{min}, \texttt{max}\} & P & := & (\overline{R}, \overline{T})
\end{array}
$$

$$
\begin{array}{lcl}
e & := & n \mid a \mid e \oplus e \mid e \odot e \mid e \circ e \mid \texttt{iter} \mid \texttt{agg}(x.f) \mid \texttt{at}^{e}(x.f) \\
\phi & := & \texttt{this}.f \odot e \mid \phi \circ \phi \\
q & := & x := \texttt{SELECT}\ \overline{f}\ \texttt{FROM}\ R\ \texttt{WHERE}\ \phi \mid \texttt{UPDATE}\ R\ \texttt{SET}\ \overline{f = e}\ \texttt{WHERE}\ \phi \\
c & := & q \mid \texttt{iterate(e)\{c\}} \mid \texttt{if(e)}\{c\} \mid \texttt{skip} \mid c; c
\end{array}
$$

Figure 5. Syntax of database programs

of transactions ($\overline{T}$). A database schema consists of a schema name (ρ) and a set of field names ($\overline{f}$). A database *record* (F) for schema R is comprised of a set of value bindings to R's fields. A database table is a set of records. Associated with each schema is a non-empty subset of its fields that act as a *primary key*. Each assignment to these fields identifies a unique record in the table. In the following, we write $R_{\texttt{id}}$ to denote the set of all possible primary key values for the schema R. In our model, a table includes a record corresponding to *every* primary key. Every schema includes a special Boolean field, *alive* $\in$ `FldName`, whose value determines if a record is actually present in the table. This field allows us to model `DELETE` and `INSERT` commands without explicitly including them in our program syntax.

Transactions are uniquely named, and are defined by a sequence of parameters, a body, and a return expression. The body of a transaction (c) is a sequence of *database commands* (q) and *control commands*. A database command either modifies or retrieves a subset of records in a database table. The records retrieved by a database query are stored locally and can be used in subsequent commands. Control commands consist of conditional guards, loops, and return statements. Both database commands (`SELECT` and `UPDATE`) require an explicit *where clause* (ϕ) to filter the records they retrieve or update. $\phi_{\texttt{fld}}$ denotes the set of fields appearing in a clause ϕ.

Expressions (e) include constants, transaction arguments, arithmetic and Boolean operations and comparisons, iteration counters and field accessors. The values of field f of records stored in a variable x can be aggregated using $\texttt{agg}(x.f)$, or accessed individually, using $\texttt{at}^{e}(x.f)$.

3.1 Data Store Semantics

Database states Σ are modeled as a triple $(\texttt{str}, \texttt{vis}, \texttt{cnt})$, where `str` is a set of *database events* (η) that captures the history of all reads and writes performed by a program operating over the database, and `vis` is a partial order on those events. The execution counter, `cnt`, is an integer that represents a global timestamp that is incremented every time a database command is executed; it is used to resolve conflicts among concurrent operations performed on the same elements, which can be used to define a *linearization* or *arbitration order* on updates [15]. Given a database state (Σ), and a primary key $r \in R_{\texttt{id}}$, it is possible to reconstruct each field f of a record r, which we denote as $\Sigma(r.f)$.

Retrieving a record from a table R generates a set of *read events*, $\texttt{rd}(\tau, r, f)$, which witness that the field f of the record with the primary key $r \in R_{\texttt{id}}$ was accessed when the value of the execution counter was τ. Similarly, a *write event*, $\texttt{wr}(\tau, r, f, n)$, records that the field f of record r was assigned the value n at timestamp τ. The timestamp (resp. record) associated with an event η is denoted by η_τ (resp. η_r).

Our semantics enforces record-level atomicity guarantees: transactions never witness intermediate (non-committed) updates to a record in a table by another concurrently executing one. Thus, all updates to fields in a record from a database command happen atomically. This form of atomicity is offered by most commercial database systems, and is easily realized through the judicious use of locks. Enforcing stronger multi-record atomicity guarantees is more challenging, especially in distributed environments with replicated database state [6, 9, 20, 35, 57]. In this paper, we consider behaviors induced when the database guarantees only a very weak form of consistency and isolation that allows transactions to see an *arbitrary subset* of committed updates by other transactions. Thus, a transaction which accesses multiple records in a table is not obligated to witness *all* updates performed by another transaction on these records.

To capture these behaviors, we use a *visibility* relation between events, `vis`, that relates two events when one witnesses the other in its *local view* of the database at the time of its creation. A local view is captured by the relation $\lhd \subseteq \Sigma \times \Sigma$ between database states, which is constrained as follows:

$$
\textsc{(ConstructView)}\quad
\frac{\texttt{str}' \subseteq \texttt{str} \qquad \forall_{\eta' \in \texttt{str}'} \forall_{\eta \in \texttt{str}} (\eta_r = \eta'_r \wedge \eta_\tau = \eta'_\tau) \Rightarrow (\eta \in \texttt{str}') \qquad \texttt{vis}' = \texttt{vis}|_{\texttt{str}'} \qquad \texttt{cnt}' = \texttt{cnt}}{(\texttt{str}', \texttt{vis}', \texttt{cnt}') \lhd (\texttt{str}, \texttt{vis}, \texttt{cnt})}
$$

The above definition ensures that an event can only be present in a local view, `str′`, if all other events **on the same record with the same counter value** are also present in `str′` (ensuring record-level atomicity). Additionally, the visibility relation permitted on the local view, `vis′`, must be consistent with the global visibility relation, `vis`.

Figure 6 presents the operational semantics of our language, which is defined by a small-step reduction relation, $\Rightarrow \subseteq \Sigma \times \Gamma \times \Sigma \times \Gamma$, between tuples of data-store states (Σ) and a set of currently executing transaction instances ($\Gamma \subseteq c \times e \times (\texttt{Var} \rightharpoonup \overline{R_{\texttt{id}} \times F})$). A transaction instance is a tuple consisting of the unexecuted portion of the transaction body (i.e., its continuation), the transaction's return expression, and a local store holding the results of previously processed query commands. The rules are parameterized over a program P containing a set of transactions, $P_{\texttt{txn}}$. At

$$\text{(TXN-INVOKE)}\ \frac{n \in \mathtt{Val} \qquad t(\overline{a})\,\{c; \mathtt{return}\ e\} \in P_{\mathtt{txn}}}{\Sigma, \Gamma \Rightarrow \Sigma, \Gamma \cup \{t : c[\overline{a/n}]; \mathtt{skip}\ ; e[\overline{a/n}]; \varnothing\}}
\qquad
\text{(TXN-STEP)}\ \frac{\Sigma, \Delta, c \longrightarrow \Sigma', \Delta', c'}{\Sigma, \{t : c; e; \Delta\} \cup \Gamma \Rightarrow \Sigma', \{t : c'; e; \Delta'\} \cup \Gamma}
\qquad
\text{(TXN-RET)}\ \frac{e \notin \mathtt{Val} \qquad \Delta, e \Downarrow m}{\Sigma, \{t : \mathtt{skip}; e; \Delta\} \cup \Gamma \Rightarrow \Sigma, \{t : \mathtt{skip}; m; \Delta\} \cup \Gamma}$$

$$\text{(SEQ)}\ \frac{\Sigma, \Delta, c \longrightarrow \Sigma', \Delta', c''}{\Sigma, \Delta, c; c' \longrightarrow \Sigma', \Delta', c''; c'}
\qquad
\text{(SKIP)}\ \frac{}{\Sigma, \Delta, \mathtt{skip}; c \longrightarrow \Sigma, \Delta, c}
\qquad
\text{(COND-T)}\ \frac{\Delta, e \Downarrow \mathtt{true}}{\Sigma, \Delta, \mathtt{if}(e)\{c\} \longrightarrow \Sigma, \Delta, c}
\qquad
\text{(COND-F)}\ \frac{\Delta, e \Downarrow \mathtt{false}}{\Sigma, \Delta, \mathtt{if}(e)\{c\} \longrightarrow \Sigma, \Delta, \mathtt{skip}}
\qquad
\text{(ITER)}\ \frac{\Delta, e \Downarrow n}{\Sigma, \Delta, \mathtt{iterate}(e)\{c\} \longrightarrow \Sigma, \Delta, \mathtt{concat}(n, c)}$$

$$\text{(SELECT)}\ \frac{\begin{array}{c} \Sigma' \lhd \Sigma \qquad \varepsilon_1 = \{\mathtt{rd}(\mathtt{cnt}, r, f) \mid r \in R_{\mathtt{id}} \wedge f \in \phi_{\mathtt{fld}}\} \\ \mathtt{results} = \{(r, \langle \overline{f : n} \rangle) \mid r \in R_{\mathtt{id}} \wedge \Sigma'(r) = \langle \overline{f' : n'} \rangle \wedge \\ \Delta, \phi(\langle \overline{f' : n'} \rangle) \Downarrow \mathtt{true} \wedge \overline{f : n} \subseteq \overline{f' : n'}\} \\ \varepsilon_2 = \{\mathtt{rd}(\mathtt{cnt}, r, f_i') \mid (r, \langle \overline{f' : n'} \rangle) \in \mathtt{results} \wedge f_i' \in \overline{f}\} \\ \mathtt{str}' = \Sigma.\mathtt{str} \cup \varepsilon_1 \cup \varepsilon_2 \qquad \mathtt{vis}' = \Sigma.\mathtt{vis} \cup \{(\eta, \eta') \mid \eta' \in \varepsilon_1 \cup \varepsilon_2 \wedge \eta \in \Sigma'.\mathtt{str})\} \end{array}}{\Sigma, \Delta, x := \mathtt{SELECT}\ \overline{f}\ \mathtt{FROM}\ R\ \mathtt{WHERE}\ \phi \longrightarrow (\mathtt{str}', \mathtt{vis}', \mathtt{cnt} + 1), \Delta[x \mapsto \mathtt{results}], \mathtt{skip}}$$

$$\text{(UPDATE)}\ \frac{\begin{array}{c} \Sigma' \lhd \Sigma \\ \varepsilon = \{\mathtt{wr}(\mathtt{cnt}, r, f_i, m) \mid r \in R_{\mathtt{id}} \wedge \Sigma'(r) = \langle \overline{f : n} \rangle \wedge \\ \Delta, \phi(\langle \overline{f : n} \rangle) \Downarrow \mathtt{true} \wedge (f_i = e_i) \in \overline{f = e} \wedge \Delta, e_i \Downarrow m\} \\ \mathtt{str}' = \Sigma.\mathtt{str} \cup \varepsilon \qquad \mathtt{vis}' = \Sigma.\mathtt{vis} \cup \{(\eta, \eta') \mid \eta' \in \varepsilon \wedge \eta \in \Sigma'.\mathtt{str}\} \end{array}}{\Sigma, \Delta, \mathtt{UPDATE}\ R\ \mathtt{SET}\ \overline{f = e}\ \mathtt{WHERE}\ \phi \longrightarrow (\mathtt{str}', \mathtt{vis}', \mathtt{cnt} + 1), \Delta, \mathtt{skip}}$$

Figure 6. Operational semantics of weakly-isolated database programs.

every step, a new transaction instance can be added to the set of currently running transactions via (TXN-INVOKE). Alternatively, a currently running transaction instance can be processed via (TXN-STEP). Finally, if the body of a transaction has been completely processed, its `return` expression is evaluated via (TXN-RET); the resulting instance simply records the binding between the transaction instance (t) and its return value (m).

The semantics of commands are defined using a local reduction relation ($\longrightarrow$) on database states, local states, and commands. The semantics for control commands are straightforward outside of the (ITER) rule, which uses an auxiliary function $\mathtt{concat}(n, c)$ to sequence n copies of the command c. Expression evaluation is defined using the big-step relation $\Downarrow \subseteq (\mathtt{Var} \rightharpoonup R_{\mathtt{id}} \times F) \times e \times \mathtt{Val}$ which, given a store holding the results of previous query commands, determines the final value of the expression. The full definition of $\Downarrow$ can be found in the supplementary material.

The semantics of database commands, given by the (SELECT) and (UPDATE) rules, expose the interplay between global and local views of the database. Both rules construct a local view of the database ($\Sigma' \lhd \Sigma$) that is used to select or update the contents of records. Neither rule imposes any restrictions on Σ' other than the consistency constraints defined by (ConstructView). The key component of each rule is how it defines the set of new events (ε) that are added to the database. In the SELECT rule, ε_1 captures the retrievals that occur on database-wide scans to identify records satisfying the `SELECT` command's where clause. In an abuse of notation, we write $\Delta, \phi(\langle \overline{f : n} \rangle) \Downarrow n$ as shorthand for $\Delta, \phi[\overline{\mathtt{this.f}/n}] \Downarrow n$. ε_2 constructs the appropriate read events of these retrieved records. The (UPDATE) rule similarly defines ε, the set of write events on the appropriate fields of the records that satisfy the where clause of the `UPDATE` command under an arbitrary (but consistent) local view (Σ') of the global store (Σ). Both rules increment the local timestamp, and establish new global visibility constraints reflecting the dependencies introduced by the database command, i.e., all the generated read and write events depending upon the events in the local view. All updates are performed atomically, as the set of corresponding write events all have the same timestamp value, however, other transactions are not obligated to see all the effects of an update since their local view may only capture a subset of these events.

3.2 Anomalous Data Access Pairs

We reason about concurrency bugs on transactions induced by our data store programming model using *execution histories*; finite traces of the form: $\Sigma_1, \Gamma_1 \Rightarrow \Sigma_2, \Gamma_2 \Rightarrow \cdots \Rightarrow \Sigma_k, \Gamma_k$ that capture interleaved execution of concurrently executing transactions. A *complete* history is one in which all transactions have finished, i.e., the final Γ in the trace is of the form: $\{t_1 : \mathtt{skip}; m_1, \Delta_1\} \cup \ldots \cup \{t_k : \mathtt{skip}; m_k, \Delta_k\}$. As a shorthand, we refer to the final state in a history h as $h_{\mathtt{fin}}$. A *serial* execution history satisfies two important properties:

Strong Atomicity: $(\forall \eta, \eta'.\ \eta_{\mathtt{cnt}} < \eta'_{\mathtt{cnt}} \Rightarrow \mathtt{vis}(\eta, \eta')) \wedge \forall \eta, \eta', \eta''.\ \mathtt{st}(\eta, \eta') \wedge (\mathtt{vis}(\eta, \eta'') \Rightarrow \mathtt{vis}(\eta', \eta''))$
Strong Isolation: $\forall \eta, \eta', \eta''.\ \mathtt{st}(\eta, \eta') \wedge \mathtt{vis}(\eta'', \eta') \Rightarrow \mathtt{vis}(\eta'', \eta)$.

The strong atomicity property prevents non-atomic interleavings of concurrently executing transactions. The first constraint linearizes events, relating timestamp ordering of events to visibility. The second generalizes this notion to multiple events, obligating *all* effects from the same transaction (identified by the `st` relation) to be visible to another if any of them are; in particular, any recorded event of a transaction T_1 that precedes an event in T_2 requires all of $T_1's$ events to precede all of T_2's.

The strong isolation property prevents a transaction from observing the commits of other transactions once it begins execution. It does so through visibility constraints on a transaction T that require any event η'' generated by any other transaction that is visible to an event η' generated by T to be visible to any event η that precedes it in T's execution.

A *serializability anomaly* is an execution history with a final state that violates at least one of the above constraints.

These sorts of anomalies capture when the events of a transaction instance are either not made visible to other events in totality (in the case of a violation of strong atomicity) or which themselves witness different events (in the case of a violation of strong isolation). Both kinds of anomalies can be eliminated by identifying commands which generate sets of problematic events and altering them to ensure *atomic execution.* Two events are executed atomically if they witness the same set of events and they are both made visible to other events simultaneously, i.e. $\mathtt{atomic}(\eta, \eta') \equiv \forall \eta''. (\mathtt{vis}(\eta, \eta'') \Rightarrow \mathtt{vis}(\eta', \eta'')) \wedge (\mathtt{vis}(\eta'', \eta) \Rightarrow \mathtt{vis}(\eta'', \eta'))$.

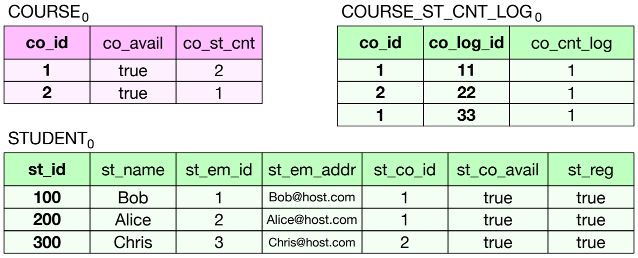
Given a program P, we define a *database access pair* (χ) as a quadruple $(c_1, \overline{f}_1, c_2, \overline{f}_2)$ where c_1 and c_2 are database commands from a transaction in P, and $\overline{f}_1$ (resp. $\overline{f}_2$) is a subset of the fields that are accessed by c_1 (resp. c_2). An access pair is anomalous if there is at least one execution in the execution history of P that results in an event generated by c_1 accessing a field $f_1 \in \overline{f}_1$ which induces a serializability anomaly with another event generated by c_2 accessing field $f_2 \in \overline{f}_2$. An example of an anomalous access pair for the program from in Section 2, is $(S1, \{\mathtt{st\_name}\}, S2, \{\mathtt{em\_addr}\})$ and $(U1, \{\mathtt{st\_name}\}, U2, \{\mathtt{em\_addr}\})$; this pair contributes to that program's non-repeatable read anomaly from Figure 2.

We now turn to the development of an automated static repair strategy that given a program P and a set of anomalous access pairs produces a semantically equivalent program P' with fewer anomalous access pairs. In particular, we repair programs by *refactoring* their database schemas in order to benefit from record-level atomicity guarantees offered by most databases, without introducing new observable behaviors. We elide the details of how anomalous access pairs are discovered, but note that existing tools [13, 46] can be adapted for this purpose. Section 6 provides more details about how this works in Atropos.

4 Refactoring Database Programs

In this section, we establish the soundness properties on the space of database program refactorings and then introduce our particular choice of sound refactoring rules. Similar refactorings are typically applied by developers when migrating traditional database programs to distributed database systems [28, 58]. Our approach to repair can be thought of as automating this manual process in a way that eliminates serializability anomalies.

The correctness of our approach relies on being able to show that each program transformation maintains the invariant that *at every step in any history of a refactored program, it is possible to completely recover the state of the data-store for a corresponding history of the original program.* To establish this property, we begin by formalizing the notion of a *containment* relation between tables.



COURSE$_0$

co_id	co_avail	co_st_cnt
1	true	2
2	true	1

COURSE_ST_CNT_LOG$_0$

co_id	co_log_id	co_cnt_log
1	**11**	1
2	**22**	1
1	**33**	1

STUDENT$_0$

st_id	st_name	st_em_id	st_em_addr	st_co_id	st_co_avail	st_reg
100	Bob	1	Bob@host.com	1	true	true
200	Alice	2	Alice@host.com	1	true	true
300	Chris	3	Chris@host.com	2	true	true

Figure 7. An example illustrating value correspondences.

4.1 Database Containment

Consider the tables in Figure 7, which are instances of the schemas from Section 2. Note that every field of $\mathtt{COURSE}_0$ can be computed from the values of some other field in either the $\mathtt{STUDENT}_0$ or $\mathtt{COURSE\_ST\_CNT\_LOG}_0$ tables: `co_avail` corresponds to the value of the `st_co_avail` field of a record in $\mathtt{STUDENT}_0$, while `co_st_cnt` can be recovered by summing up the values of the `co_cnt_log` field of the records in $\mathtt{COURSE\_ST\_CNT\_LOG}_0$ whose `co_id` field has the same value as the original table.

The containment relation between a table (e.g. $\mathtt{COURSE}_0$) and a set of tables (e.g. $\mathtt{STUDENT}_0$ and $\mathtt{COURSE\_ST\_CNT\_LOG}_0$) is defined using a set of mappings called *value correspondences* [55]. A value correspondence captures how to compute a field in the contained table from the fields of the containing set of tables. Formally, a value correspondence between field f of schema R and field f' of schema R' is defined as a tuple $(R, R', f, f', \theta, \alpha)$ in which: (i) a total *record correspondence function*, denoted by $\theta : R_{\mathtt{id}} \rightarrow \overline{R'_{\mathtt{id}}}$, relates every record of any instance of R to a *set* of records in any instance of R' and (ii) a fold function on values, denoted by $\alpha : \overline{\mathtt{Val}} \rightarrow \mathtt{Val}$ is used to aggregate a set of values. We say that a table X is contained by a set of tables $\overline{X}$ under a set of value correspondences V, if V accurately explains how to compute X from $\overline{X}$, i.e.

$$\begin{aligned} X \sqsubseteq_V \overline{X} \;\; \equiv \;\; & \forall f \in X_{\mathtt{fld}}. \exists (R, R', f, f', \theta, \alpha) \in V. \exists X' \in \overline{X}. \\ & \forall r \in R_{\mathtt{id}}. X(r.f) \; = \; \alpha(\{m \mid r' \in \theta(r) \wedge X'(r'.f') = m\}) \end{aligned}$$

For example, the table $\mathtt{COURSE}_0$ is contained in the set of tables $\{\mathtt{STUDENT}_0, \mathtt{COURSE\_ST\_CNT\_LOG}_0\}$ under the pair of value correspondences, $(\mathtt{COURSE}, \mathtt{STUDENT}, \mathtt{co\_avail}, \mathtt{st\_co\_avail}, \theta_1, \mathtt{any})$ and $(\mathtt{COURSE}, \mathtt{COURSE\_ST\_CNT\_LOG}, \mathtt{co\_st\_cnt}, \mathtt{co\_cnt\_log}, \theta_2, \mathtt{sum})$, where $\theta_1(1) = \{100, 200\}$, $\theta_1(2) = \{300\}$, $\theta_2(1) = \{(1, 11), (1, 33)\}$ and $\theta_2(2) = \{(2, 22)\}$. The aggregator function $\mathtt{any} : \overline{\mathtt{Val}} \rightarrow \mathtt{Val}$ returns a non-deterministically chosen value from a set of values. The containment relation on tables is straightforwardly lifted to data store states, denoted by $\Sigma \sqsubseteq_V \Sigma'$, if all tables in Σ are contained by the set of tables in Σ'.

We define the soundness of our program refactorings using a pair of *refinement* relations between execution histories and between programs. An execution history h' (where $h'_{\mathtt{fin}} = (\Sigma', \Gamma')$) is a refinement of an execution h (where

$$\textsc{(intro } \rho) \quad \frac{\rho \notin \overline{R}_{\texttt{RelNames}}}{\overline{V}, (\overline{R}, \overline{T}) \hookrightarrow \overline{V}, (\overline{R} \cup \{\rho : \emptyset\}, \overline{T})} \qquad \textsc{(intro } \rho.f) \quad \frac{R = \rho : \overline{f} \quad f \notin \overline{f} \quad R' = \rho : \overline{f} \cup \{f\}}{\overline{V}, (\{R\} \cup \overline{R}, \overline{T}) \hookrightarrow \overline{V}, (\{R'\} \cup \overline{R}, \overline{T})}$$

$$\textsc{(intro } v) \quad \frac{v \notin \overline{V} \qquad \overline{T'} = \{t(\overline{a})\{[[c]]_v; \texttt{return } [[e]]_v\} \mid t(\overline{a})\{c; \texttt{return } e\} \in \overline{T}\}}{\overline{V}, (\overline{R}, \overline{T}) \hookrightarrow \overline{V} \cup \{v\}, (\overline{R}, \overline{T'})}$$

Figure 8. Refactoring Rules

$h_{\texttt{fin}} = (\Sigma, \Gamma)$), denoted by $h' \preceq_V h$, if and only if Γ' and Γ have the same collection of finalized transaction instances and there is a set of value correspondences V under which Σ is contained in Σ', i.e. $\Sigma \sqsubseteq_V \Sigma'$. Intuitively, any refinement of a history h maintains the same set of records and spawns the same set of transaction instances as h, with each instance producing the same result as it does in h. Lastly, we define a refactored program P' to be a refinement of the original program P, denoted by $P' \preceq_V P$, if the following conditions are satisfied:

(I) Every history h' of P' has a corresponding history h in P such that h' is a refinement of h.

(II) Every serializable history h of P has a corresponding history h' in P' such that h' is a refinement of h.

The first condition ensures that P' does not introduce any new behaviors over P, while the second ensures that P' does not remove any desirable behavior exhibited by P.

4.2 Refactoring Rules

We describe Atropos's refactorings using a relation $\hookrightarrow \subseteq \overline{V} \times P \times \overline{V} \times P$, between programs and sets of value correspondences. The rules in Figure 8 are templates of the three categories of transformations employed by Atropos. These categories are: (1) adding a new schema to the program, captured by the rule (intro ρ); (2) adding a new field to an existing schema ρ, captured by rule (intro $\rho.f$); and, (3) relocating certain data from one table to another while modifying the way it is accessed by the program, captured by the rule (intro v).

The refactorings represented by (intro v) introduce a new value correspondence v, and modify the body and return expressions of a programs transactions via a rewrite function, $[[.]]_v$. A particular instantiation of $[[.]]_v$ must ensure the same data is accessed and modified by the resulting program, in order to guarantee that the refactored program refines the original. At a high-level, it is sufficient for $[[\cdot]]_v$ to ensure the following relationship between the original (P) and refactored programs (P') :

(R1) P' accesses the same data as P, which may be maintained by different schemas;

(R2) P' returns the same final value as P;

(R3) and, P' properly updates all data maintained by P.

To see how a rewrite function might ensure R1 to R3, consider the original (top) and refactored (bottom) programs presented in Figure 9. This example depicts a refactoring of transactions `getSt` and `setSt` to utilize a value correspondence from `em_addr` to `st_em_addr`, moving email addresses to the `STUDENT` table, as described in Section 2. The select commands `S1` and `S3` in `getS` remain unchanged after the refactoring, as they do not access the affected table. However, the query `S2`, which originally accessed the `EMAIL` table is redirected to the `STUDENT` table.

More generally, in order to take advantage of a newly added value correspondence v, $[[.]]_v$ must alter every query on the source table and field in v to use the target table of v instead, so that the new query accesses the same data as the original. This rewrite has the general form:

$$\begin{aligned} &[[x := \texttt{SELECT } f \texttt{ FROM } R \texttt{ WHERE } \phi]]_v \equiv \\ &\qquad x := \texttt{SELECT } f' \texttt{ FROM } R' \texttt{ WHERE } \texttt{redirect}(\phi, \theta) \end{aligned} \tag{1}$$

Intuitively, in order for this transformation to ensure R1, the `redirect` function must return a new where clause on the target table which selects a set of records corresponding to set selected by the original clause.

In order to preserve R2, program expressions also need to be rewritten to evaluate to the same value as in the original program. For example, observe that the `return` expression in `getSt` is updated to reflect that the records held in the variable `y` now adhere to a different schema.

The transformation performed in Figure 9 also rewrites the update (`U2`) of transaction `setSt`. In this case, the update is rewritten using the same redirection strategy as (`S2`), so that it correctly reflects the updates that would be performed by the original program to the `EMAIL` record.

Taken together, $R1 - R3$ are sufficient to ensure that a particular instance of intro v is sound[2]:

Theorem 4.1. *Any instance of* intro v *whose instantiation of* $[[\cdot]]_v$ *satisfies* $R1 - R3$ *is guaranteed to produce a refactored program that refines the original, i.e.*

$$\forall_{P,P',V,vww}.\ (V, P) \hookrightarrow (V \cup \{v\}, P') \Rightarrow P' \preceq_{V \cup \{v\}} P$$

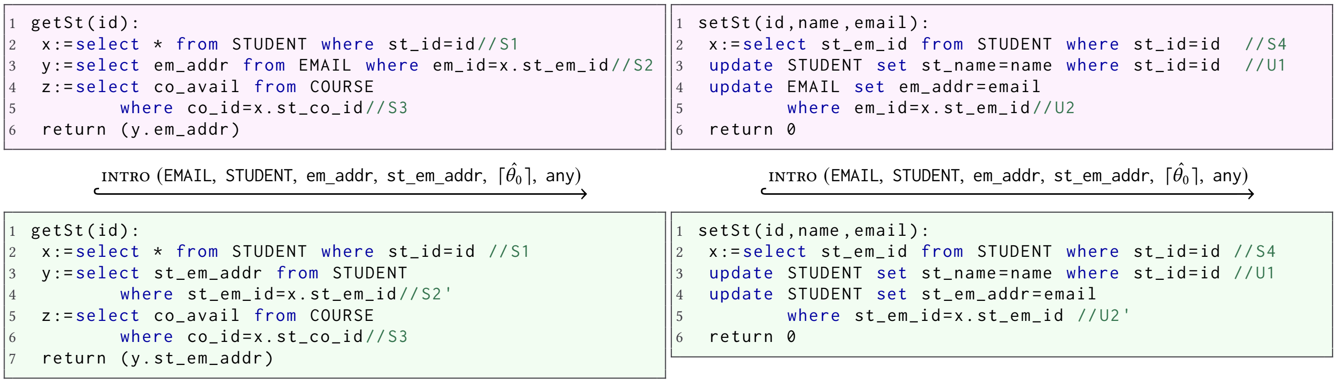
Although our focus has been on preserving the semantics of refactored programs, note that as a direct consequence of our definition of program refinement, this theorem implies that sound transformations do not introduce any new anomalies.

We now present the instantiations of intro v used by Atropos, explaining along the way how they ensure R1-R3.

4.2.1 Redirect Rule.

Our first refactoring rule is parameterized over the choice of schemas and fields and uses the aggregator `any`. Given data store states Σ and Σ', the record correspondence is defined as: $\lceil\hat{\theta}\rceil(r) \equiv \{r' \mid r' \in R'_{\texttt{id}} \land \forall_{f \in R_{\texttt{id}}} \forall_n.\ \Sigma(r.f) \Downarrow n \Rightarrow \Sigma'(r'.\hat{\theta}(f)) \Downarrow n\}$. In essence,

[2] A complete formalization of all three refactoring rules, their correctness criteria, and proofs of soundness is presented in the extended version of this paper [47].



```
getSt(id):
  x:=select * from STUDENT where st_id=id//S1
  y:=select em_addr from EMAIL where em_id=x.st_em_id//S2
  z:=select co_avail from COURSE
         where co_id=x.st_co_id//S3
  return (y.em_addr)
```

```
setSt(id,name,email):
  x:=select st_em_id from STUDENT where st_id=id   //S4
  update STUDENT set st_name=name where st_id=id   //U1
  update EMAIL set em_addr=email
         where em_id=x.st_em_id//U2
  return 0
```

$\xrightarrow{\textsc{intro}\ (\texttt{EMAIL},\ \texttt{STUDENT},\ \texttt{em\_addr},\ \texttt{st\_em\_addr},\ \lceil\hat{\theta}_0\rceil,\ \texttt{any})}$

$\xrightarrow{\textsc{intro}\ (\texttt{EMAIL},\ \texttt{STUDENT},\ \texttt{em\_addr},\ \texttt{st\_em\_addr},\ \lceil\hat{\theta}_0\rceil,\ \texttt{any})}$

```
getSt(id):
  x:=select * from STUDENT where st_id=id //S1
  y:=select st_em_addr from STUDENT
         where st_em_id=x.st_em_id//S2'
  z:=select co_avail from COURSE
         where co_id=x.st_co_id//S3
  return (y.st_em_addr)
```

```
setSt(id,name,email):
  x:=select st_em_id from STUDENT where st_id=id //S4
  update STUDENT set st_name=name where st_id=id //U1
  update STUDENT set st_em_addr=email
         where st_em_id=x.st_em_id //U2'
  return 0
```

Figure 9. A single program refactoring step, where $\hat{\theta}_0(\texttt{EMAIL.em\_addr}) = \texttt{STUDENT.st\_em\_addr}$

the lifted function $\hat{\theta}$ identifies how the value of the primary key f of a record r can be used to constrain the value of field $\hat{\theta}(f)$ in the target schema to recover the set of records corresponding to r, i.e. $\theta(r)$. The record correspondences from Section 4.1 were defined in this manner, where

$$\hat{\theta}_1(\texttt{COURSE.co\_id}) \equiv \texttt{STUDENT.st\_co\_id}, \quad \text{and}$$
$$\hat{\theta}_2(\texttt{COURSE.co\_id}) \equiv \texttt{COURSE\_CO\_ST\_CNT\_LOG.co\_id}.$$

Defining the record correspondence this way ensures that if a record r is selected in Σ, the corresponding set of records in Σ' can be determined by identifying the values that were used to select r, without depending on any particular instance of the tables. Our choice of record correspondence function makes the definition of $[\![\cdot]\!]$ for select statements a straightforward instantiation of (1) with the following definition of `redirect`:

$$\texttt{redirect}(\phi, \lceil\hat{\theta}\rceil) \equiv \bigwedge_{f \in \phi_{\texttt{fld}}} \texttt{this}.\hat{\theta}(f) = \phi[f]_{\texttt{exp}} \qquad (2)$$

The one wrinkle in this definition of `redirect` is that it is only defined when the where clause ϕ is *well-formed*, i.e. ϕ only consists of conjunctions of equality constraints on primary key fields. The expressions used in such a constraint is denoted by $\phi[f]_{\texttt{exp}}$. As an example, the where clause of command (S2) in Figure 9 (left) is well-formed, where $\phi[\texttt{em\_id}]_{\texttt{exp}} \equiv \texttt{x.st\_e\_id}$. However, the where clause in (S2′) after the refactoring step is not well-formed, since it does not constrain the primary key of the `STUDENT` table. This restriction ensures that only queries that access a single record of the original table will be rewritten. Expressions using variables containing the results of such queries are rewritten by substituting the source field name with the target field name, e.g. $[\![\texttt{at}^1(x.f)]\!]_v \equiv \texttt{at}^1(x.f')$.

Redirecting updates is similarly defined using the definition of $\texttt{redirect}(\phi, \theta)$ from 2:

$$[\![\texttt{UPDATE}\ R\ \texttt{SET}\ f = e\ \texttt{WHERE}\ \phi]\!]_v \equiv$$
$$\texttt{UPDATE}\ R'\ \texttt{SET}\ (f' = [\![e]\!]_v)\ \texttt{WHERE}\ \texttt{redirect}(\phi, \theta)$$

4.2.2 Logger Rule.

Unfortunately, instantiating $\textsc{intro}\ v$ is not so straightforward when we want to utilize value correspondences with more complicated aggregation functions than `any`. To see why, consider how we would need to modify an `UPDATE` when $\alpha \equiv \texttt{sum}$ is used. In this case, our rule transforms the program to insert *a new record* corresponding to each update performed by the original program. Hence, the set of corresponding records in the target table always grows and cannot be statically identified.

We enable these sorts of transformations by using *logging* schema for the target schema. A logging schema for source schema R and the field f is defined as follows: (i) the target schema ($LogR$) has a primary key field, corresponding to every primary key field of the original schema (R); (ii) the schema has one additional primary key field, denoted by $LogR.\texttt{log\_id}$, which allows a *set* of records in $LogR$ to represent each record in R; and (iii) the schema $LogR$ has a single field corresponding to the original field $R.f$, denoted by $LogR.f'$.

Intuitively, a logging schema captures the *history* of updates performed on a record, instead of simply replacing old values with new ones. Program-level aggregators can then be utilized to determine the final value of each record, by observing all corresponding entries in the logging schema. The schema `COURSE_CO_ST_CNT_LOG` from Section 2 is an example of a logging schema for the source schema and field `COURSE.co_st_cnt`.

Under these restrictions, we can define an implementation of $[\![\cdot]\!]$ for the logger rule using `sum` as an aggregator. This refactoring also uses a lifted function $\lceil\hat{\theta}\rceil$ for its value correspondence, which allows $[\![\cdot]\!]$ to reuse our earlier definition of `redirect`. We define $[\![\cdot]\!]$ on accesses to f to use program-level aggregators, e.g. $[\![\texttt{at}^1(x.f)]\!]_v \equiv \texttt{sum}(x.f')$.

Finally, the rewritten `UPDATE` commands simply need to log any updates to the field f, so its original value can be recovered in the transformed program, e.g.

$$[\![\texttt{UPDATE}\ R\ \texttt{SET}\ f = e + \texttt{at}^1(x.f)\ \texttt{WHERE}\ \phi]\!]_v \equiv \texttt{UPDATE}\ R'$$
$$\texttt{SET}\ f' = [\![e]\!]_v \texttt{WHERE}\ \texttt{redirect}(\phi, \theta) \wedge R'.\texttt{log\_id} = \texttt{uuid}().$$

Having introduced the particular refactoring rules instantiated in Atropos, we are now ready to establish the soundness of those refactorings:

Theorem 4.2. *The rewrite rules described in this section satisfy the correctness properties (R1), (R2) and (R3).*

Corollary 4.3. *(Soundness) Any sequence of refactorings performed by Atropos is sound, i.e. the refactored program is a refinement of the original program.*

Proof. Direct consequence of theorems 4.1 and 4.2. □

5 Repair Procedure

Figure 10 presents our algorithm for eliminating serializability anomalies using the refactoring rules from the previous section. The algorithm (`repair`) begins by applying an anomaly detector $\mathcal{O}$ to a program to identify a set of anomalous access pairs. As an example, consider `regSt` from our running example. For this transaction, the anomaly oracle identifies two anomalous access pairs:

$$(\text{U3}, \{\texttt{st\_co\_id}, \texttt{st\_reg}\}, \text{U4}, \{\texttt{co\_avail}\}) \qquad (\chi_1)$$

$$(\text{S5}, \{\texttt{co\_st\_cnt}\}, \text{U4}, \{\texttt{co\_st\_cnt}\}) \qquad (\chi_2)$$

The first of these is involved in the dirty read anomaly from Section 2, while the second is involved in the lost update anomaly.

```
Function: repair(P)
1 $\overline{\chi} \leftarrow \mathcal{O}(P)$;   $P \leftarrow$ pre_process($P, \overline{\chi}$)
2 for $\chi \in \overline{\chi}$ do
3     if try_repair($P, \chi$) = $P'$ then  $P \leftarrow P'$
4 return post_process($P$)

Function: try_repair($P, \chi$)
1 $c_1 \leftarrow \chi.c_1$;   $c_2 \leftarrow \chi.c_2$
2 if same_kind($c_1, c_2$) then
3     if same_schema($c_1, c_2$) then
4         return try_merging($P, c_1, c_2$)
5     else if try_redirect($P, c_1, c_2$) = $P'$ then
6         return try_merging($P', c_1, c_2$)
7 return try_logging($P, c_1, c_2$)
```

Figure 10. The repair algorithm

The repair procedure next performs a pre-processing phase, where database commands are split into multiple commands such that each command is involved in at most one anomalous access pair. For example, the first step of repairing the `regSt` transaction is to split command U4 into two update commands, as shown in Figure 11 (top). Note that we only perform this step if the split fields are not accessed together in other parts of the program; this is to ensure that the splitting does not introduce new unwanted serializability anomalies.

After pre-processing, the algorithm iterates over all detected anomalous access pairs ($\overline{\chi}$) and greedily attempts to repair them one by one using `try_repair`. This function attempts to eliminate a given anomaly in two different ways; either by merging anomalous database commands into a single command, and/or by removing one of them by making it obsolete. In the remainder of this section, we present these two strategies in more detail, using the running example from Figure 11.

```
regSt(id,course):
 update STUDENT set st_co_id=course,st_reg=true
        where st_id=id//U3
 x:=select co_st_cnt from COURSE
        where co_id=course//S5
 update COURSE set co_st_cnt=x.co_st_cnt+1
        where co_id=course//U4.1
 update COURSE set co_avail=true
        where co_id=course//U4.2
```

$$\xrightarrow{\textsc{intro}\ (\text{COURSE, STUDENT, co\_avail, st\_co\_avail, } \lceil\hat{\theta}_1\rceil\text{, any})}$$

```
regSt(id,course):
 update STUDENT set st_co_id=course,st_reg=true
        where st_id=id//U3
 x:=select co_st_cnt from COURSE
        where co_id=course//S5
 update COURSE set co_st_cnt=x.co_st_cnt+1
        where co_id=course//U4.1
 update STUDENT set st_co_avail=true
        where st_co_id=course//U4.2'
```

$$\xrightarrow{\textsc{intro}\ \text{COURSE\_CO\_ST\_CNT\_LOG}} \ \cdots$$

$$\xrightarrow{\textsc{intro}\ (\text{COURSE, COURSE\_ST\_CNT\_LOG, co\_st\_cnt, co\_cnt\_log, } \lceil\hat{\theta}_2\rceil\text{,sum})}$$

```
regSt(id,course):
 update STUDENT set st_co_id=course, st_reg=true
        where st_id=id//U3
 x:=select co_st_cnt from COURSE
        where co_id=course//S5
 insert into COURSE_CO_ST_CNT_LOG values //U4.1'
    (co_id=course,log_id=uuid(),co_st_cnt_log=1)
 update STUDENT set st_co_avail=true
    where st_co_id=course//U4.2'
```

Figure 11. Repair steps of transaction `regSt`

We first explain the merging approach. Two database commands can only be merged if they are of the same kind (e.g. both are selects and if they both access the same schema. These conditions are checked in lines 2-3. Function `try_merge` attempts to merge the commands if it can establish that their where clauses always select the exact same set of records, i.e. condition (R1) described in Section 4.2.

Unfortunately, database commands involved in anomalies are rarely on the same schema and cannot be merged as they originally are. Using the refactoring rules discussed earlier, Atropos attempts to introduce value correspondences so that the anomalous commands are redirected to the same table in the refactored program and thus mergeable. This is captured by the call to the procedure `try_redirect`. This procedure first introduces a set of fields into the schema accessed by c_1, each corresponding a field accessed by c_2. Next, it attempts to introduce a sequence of value correspondences between the two schemas using the redirect rule, such that c_2 is redirected to the same table as c_1. The record correspondence is constructed by analyzing the commands' where clauses and identifying equivalent expressions used in their constraints. If redirection is successful, `try_merge` is invoked on the commands and the result is returned (line 6).

For example, consider commands `U3` and `U4.2` in Figure 11 (top), which are involved in the anomaly χ_1. By introducing a value correspondence from `COURSE` to `STUDENT`, Atropos refactors the program into a refined version where `U4.2` is transformed into `U4.2'` and is mergeable with `U3`.

Merging is sufficient to fix χ_1, but fails to eliminate χ_2. The repair algorithm next tries to translate database updates into an equivalent insert into a logging table using the `try_logging` procedure.This procedure first introduces a new logging schema (using the intro ρ rule) and then introduces fields into that schema (using intro $\rho.f$). It then attempts to introduce a value correspondence from the schema involved in the anomaly to the newly introduced schema using the logger rule. The function returns successfully if such a translation exists and if the select command involved in the anomaly becomes obsolete, i.e., the command is dead-code. For example, in Figure 11, a value correspondence from `COURSE` to the logger table `COURSE_CO_ST_CNT` is introduced, which translates the `update` command involved in the anomaly to an `insert` command. The select command is obsolete in the final version, since variable x is never used.

Once all anomalies have been iterated over, Atropos performs a post-processing phase on the program to remove any remaining dead code and merge commands whenever possible. For example, the transaction `regSt` is refactored into its final version depicted in Figure 3 after post-processing. Both anomalous accesses (χ_1 and χ_2) are eliminated in the final version of the transaction.

6 Implementation

Atropos is a fully automated static analyzer and program repair tool implemented in Java. Its input programs are written in a DSL similar to the one described in Figure 5, but it would be straightforward to extend the front-end to support popular database programming APIs, e.g. JDBC or Python's DB-API. Atropos consists of a static anomaly detection engine and a program refactoring engine and outputs the repaired program. The static anomaly detector in Atropos adapts existing techniques to reason about serializability violations over abstract executions of a database application [13, 39]. In this approach, detecting a serializability violation is reduced to checking the satisfiability of an FOL formula constructed from the input program. This formula includes variables for each of the transactional dependencies, as well as the visibility and global time-stamps that can appear during a program's execution. The assignments to these variables in any satisfying model can be used to reconstruct an anomalous execution. We use an off-the-shelf SMT solver, Z3 [17], to check for anomalies in the input program and identify a set of anomalous access pairs. These access pairs are then used by an implementation of the repair algorithm build a repaired version of the input program.

Table 1. Statically identified anomalous access pairs in the original and refactored benchmark programs

Benchmark	#Txns	#Tables	EC	AT	CC	RR	Time (s)
TPC-C [1, 18, 33]	5	9, 16	33	8	33	33	81.2
SEATS [18, 52]	6	8, 12	35	10	35	33	61.5
SmallBank [18, 50]	6	3, 5	24	8	21	20	68.7
Twitter [18]	5	4, 5	6	1	6	5	3.6
SIBench [18]	2	1, 2	1	0	1	1	0.3
Wikipedia [18]	5	12, 13	2	1	2	2	9.0
FMKe [53]	7	7, 9	6	2	6	6	33.6
Killrchat [2, 13]	5	3, 4	6	3	6	6	42.9
Courseware [27, 32]	5	3, 2	5	0	5	5	12.7

7 Evaluation

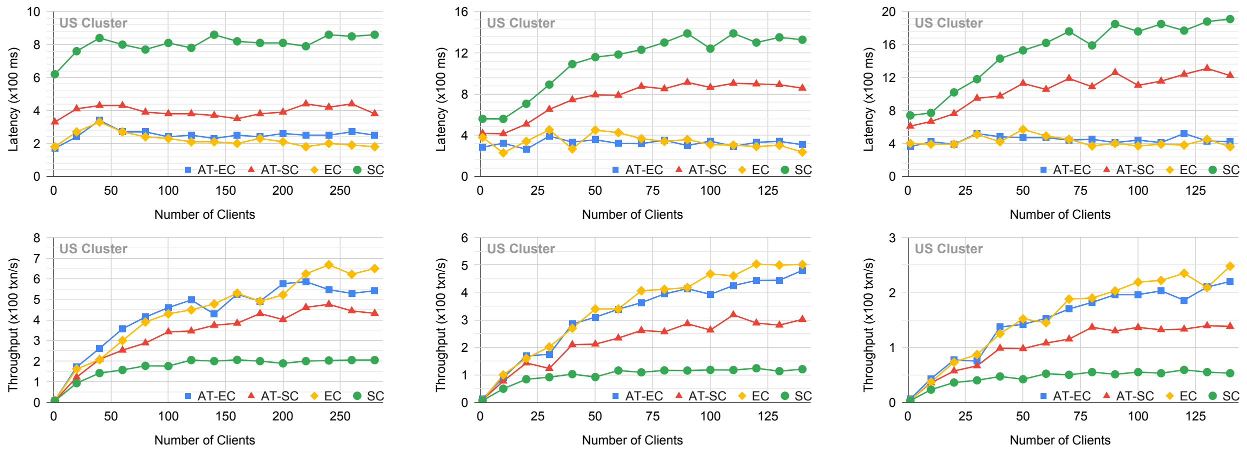
This section evaluates Atropos along two dimensions:

1. **Effectiveness:** Does schema refactoring eliminate serializability anomalies in real-world database applications? Is Atropos capable of repairing meaningful concurrency bugs?
2. **Performance:** What impact does Atropos have on the performance of refactored programs? How does Atropos compare to other solutions to eliminating serializability anomalies, in particular by relying on stronger database-provided consistency guarantees?

7.1 Effectiveness

To assess Atropos' effectiveness, we applied it to a corpus of standard benchmarks from the database community. Table 1 presents the results for each program. The first six programs come from the ten benchmarks defined in the OLTPBench project [18]. We did not consider the remaining four benchmarks because they do not exhibit any serializability anomalies. The last three programs are drawn from papers that deal with the consistency of distributed systems [13, 27, 53].

The first four columns in Table 1 display the number of transactions (#Txns), the number of tables in the original and refactored schemas (#Tables), and the number of anomalies detected assuming eventually consistent guarantees for the original (EC) and refactored (AT) programs. For each benchmark, Atropos was able to repair at least half the anomalies, and in many cases substantially more, suggesting that many serializability bugs can be directly repaired by our schema refactoring technique. The total time needed to analyze and repair each benchmark is presented in the **Time(s)** column. The time spent on analysis dominates these numbers; repairing programs took under 50ms for every benchmark.



(a) SmallBank **(b)** SEATS **(c)** TPC-C

Figure 12. Performance evaluation of SmallBank, SEATS and TPC-C benchmarks running on US cluster (see the extended version [47] for experimental results for local and globally distributed clusters).

In order to compare our approach to other means of anomaly elimination – namely, by merely strengthening the consistency guarantees provided by the underlying database – we modified Atropos's anomaly oracle to only consider executions permitted under causal consistency and repeatable read; the former enforces causal ordering in the visibility relation, while the latter prevents results of a newly committed transaction T becoming visible to an executing transaction that has already read state that is written by T. The next two columns of Table 1, (CC) and (RR), show the result of this analysis: causal consistency was only able to reduce the number of anomalies in one benchmark (by 12%) and repeatable read in three (by 5%, 15% and 16%). This suggests that only relying on isolation guarantees between eventual and sequential consistency is not likely to significantly reduce the number of concurrency bugs that manifest in an EC execution.

As a final measure of Atropos's impact on correctness, we carried out a more in-depth analysis of the SmallBank benchmark, in order to understand Atropos's ability to repair meaningful concurrency bugs. This benchmark maintains the details of customers and their accounts, with dedicated tables holding checking and savings entries for each customer. By analyzing this and similar banking applications from the literature [27, 31, 56], we identified the following three invariants to be preserved by each transaction:

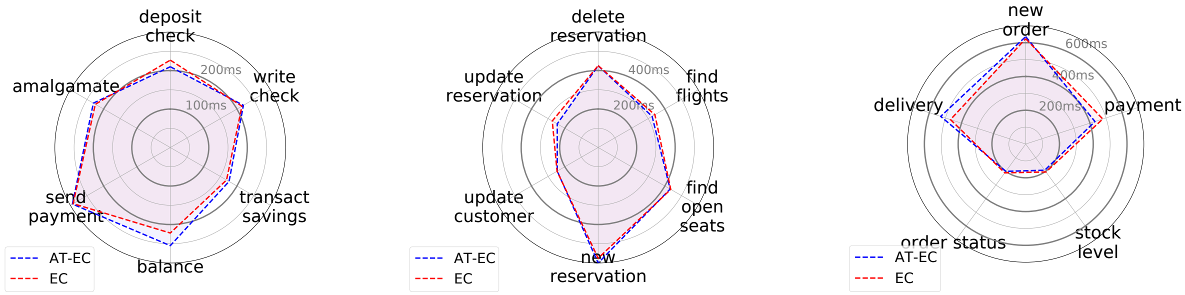
(i) Each account must accurately reflect the history of deposits to that account,
(ii) The balance of accounts must always be non-negative,
(iii) Each client must always witness a consistent state of her checking and savings accounts. For example, when transferring money between accounts, users should not see a state where the money is deducted from the checking account but not yet deposited into savings.

Interestingly, we were able to detect violations of *all three* invariants in the original program under EC, while the repaired program violated only invariant (ii). This is evidence that the statically identified serializability anomalies eliminated by Atropos are meaningful proxies to the application-level invariants that developers care about.

7.2 Performance

To evaluate the performance impact of schema refactoring, we conducted further experiments on a real-world geo-replicated database cluster, consisting of three AWS machines (M10 tier with 2 vCPUs and 2GB of memory) located across US in N. Virginia, Ohio and Oregon. Similar results were exhibited by experiments on a single data center and globally distributed clusters. Each node runs MongoDB (v.4.2.9), a modern document database management system that supports a variety of data-model design options and consistency enforcement levels. MongoDB documents are equivalent to records and a collection of documents is equivalent to a table instance, making all our techniques applicable to MongoDB clients.

Figure 12 presents the latency (top) and throughput (bottom) of concurrent executions of SmallBank (left), SEATS (middle) and TPC-C (right) benchmarks. These benchmarks are representative of the kind of OLTP applications best suited for our refactoring approach. Horizontal axes show the number of clients, where each client repeatedly submits transactions to the database according to each benchmark's specification. Each experiment was run for 90 seconds and the average performance results are presented. For each benchmark, performance of four different versions of the program are compared: (i) original version running under EC (♦ EC), (ii) refactored version running under EC (■ AT-EC), (iii) original version running under SC (● SC) and



(a) SmallBank **(b)** SEATS **(c)** TPC-C

Figure 13. Average latency breakdown for each transaction

(iv) refactored version where transactions with at least one anomaly are run under SC and the rest are run under EC (▲ AT-SC). Across all benchmarks, SC results in poor performance compared to EC, due to lower concurrency and additional synchronization required between the database nodes. On the other hand, AT-EC programs show negligible overhead with respect to their EC counterparts, despite having fewer anomalies. Most interestingly, refactored programs show an average of 120% higher throughput and 45% lower latency compared to their counterparts under SC, while offering the *same level of safety*. These results provide evidence that automated schema refactoring can play an important role in improving both the correctness and performance of modern database programs.

Lastly, in order to illuminate the impact of refactoring on the performance of *individual* transactions, Figure 13 presents the average latency across all experiments for each transaction in the original and the refactored programs running under EC. There are minor performance improvements due to fewer database operations (e.g. the update reservation transaction from SEATS or the payment transaction from TPC-C) and minor performance losses due to additional logging and aggregation (e.g. the balance transaction in SmallBank or the delivery transaction in TPC-C) witnessed after refactoring the benchmarks. The refactoring of our benchmarks has limited impact on the latency of individual transactions as evidenced by the close similarity of the shapes of the radar charts in the figure.

7.3 Discussion

It is well-known that *some* serializability anomalies cannot be eliminated without database-level enforcement of strong isolation and consistency semantics [26]. In particular, if an anomaly is caused by a read operation (R) followed by a write operation (W) that depends on the value returned by R, it cannot be eliminated without synchronization between clients. For example, the write check transaction in the Smallbank benchmark includes an anomaly caused by reading an account's balance and then performing writes to that account *depending on the original account balance*. This is a well-studied anomaly which has been proven to require strong consistency and isolation in order to be fully eliminated [27, 50].

Since Atropos is a synchronization-free solution, it cannot always repair every serializability anomaly in a program, as shown in Table 1. Nevertheless, by first using Atropos to repair anomalies that do not require synchronization and then relying on stronger consistency semantics to eliminate the remainder, it is possible to provide strong serializability guarantees with less performance impact than relying solely on database-level enforcement of strong isolation and consistency semantics.

8 Related Work

Wang et al. [55] describe a synthesis procedure for generating programs consistent with a database refactoring, as determined by a verification procedure that establishes database program equivalence [54]. Their synthesis procedure performs enumerative search over a template whose structure is derived by value correspondences extracted by reasoning over the structure of the original and refactored schemas. Our approach has several important differences. First, our search for a target program is driven by anomalous access pairs that identify serializability anomalies in the original program and does not involve enumerative search over the space of *all* equivalent candidate programs. This important distinction eliminates the need for generating arbitrarily-complex templates or sketches. Second, because we *simultaneously* search for a target schema and program consistent with that schema given these access pairs, our technique does not need to employ conflict-driven learning [23] or related mechanisms to guide a general synthesis procedure as it recovers from a failed synthesis attempt. Instead, value correspondences derived from anomalous access pairs help define a restricted class of schema refactorings (e.g., aggregation and logging) that directly informs the structure of the target program.

Identifying serializability anomalies in database systems is a well-studied topic that continues to garner attention [8, 11, 22, 30, 37], although the issue of automated repair is comparatively less explored. A common approach in all these techniques is to model interactions among concurrently executing database transactions as a graph, with edges connecting transactions that have a data dependency with one another; cycles in the graph indicate a possible serializability violation. Both dynamic [12, 56] and static [13, 39, 46] techniques have been developed to discover these violations in various domains and settings.

Various techniques have been developed to discover these violations dynamically. For example, Warszawski and Bailis [56] use program traces to identify potential vulnerabilitis in Web applications that exploit weak isolation while Brutschy et al. [12] present a dynamic analysis technique for discovering serializability in an eventually consistent distributed setting. Follow-on work [13] develops scalable static methods under stronger causally-consistent assumptions. Rahmani et al. [46] present a test generation tool for triggering serializability anomalies that builds upon a static detection framework described in [39].

An alternative approach to eliminating serializability anomalies is to develop correct-by-construction methods. For example, to safely develop applications for eventually-consistent distributed environments, conflict-free replicated data-types (CRDTs) [49] have been proposed. CRDTs are abstract datatypes (e.g. sets, counters) equipped with commutative operations whose semantics are invariant with respect to the order in which operations are applied on their state. Alternatively, there have been recent efforts which explore enriching specifications, rather than applications, with mechanisms that characterize notions of correctness in the presence of replication [29, 50], using these specifications to guide safe implementations. These techniques, however, have not been applied to reasoning on the correctness of concurrent relational database programs which have highly-specialized structure and semantics, centered on table-based operations over inter-related schema definitions, rather than control- and data-flow operations over a program heap.

The idea of altering the data structures used by a client program, rather than changing its control flow, is reminiscent of the data-centric synchronization proposed by Dolby et al. [19], which considers how to build atomic sets with associated units of work. The context of their investigation, concurrent Java programs, is quite different from ours; in particular, their solution does not consider sound schema refactorings, an integral part of our approach.

9 Conclusions and Future Work

There are several interesting future directions for Atropos. In particular, our repair algorithm greedily identifies the first refactoring that eliminates an anomaly. Integrating a cost model into this search could result in repaired programs with even better performance. In addition, though we were not able to identify any cases where the ordering of refactorings mattered in our experiments, investigating the potential of refactorings to enable additional beneficial transformations merits further investigation.

The techniques presented in this paper operate solely on the *database parts* of some larger program. Our refactorings are guaranteed to soundly preserve the semantics of these parts, and thus those of the surrounding program as well. A more holistic refactoring approach, which considers both the database parts and the surrounding application, may offer further opportunities for repairs and performance improvements.

We have presented Atropos, an approach for automatically eliminating serializability anomalies in the clients of distributed databases. By altering the data layout (i.e. schemas) of the underlying database and refactoring the client programs accordingly, we demonstrate that it is possible to repair many statically identified anomalies in those clients. Our experimental results showcase the utility of this approach, showing that the refactored programs perform comparably to the original programs, while exhibiting fewer serializability bugs. Furthermore, our evaluation shows that the combination of Atropos and stronger database-provided consistency semantics, enables clients of distributed databases to offer strong serializability guarantees with less performance impact than stronger consistency semantics alone.

Acknowledgments

We thank our shepherd, Dr. Kapil Vaswani, and the anonymous reviewers for their detailed and insightful comments. This material is based upon work supported by the NSF under Grant No. CCF-SHF 1717741.

References

[1] 2020. TPC-C Benchmark. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf. Online; Accessed April 2020.

[2] DuyHai Doan. KillrChat, a scalable chat with Cassandra, AngularJS & Spring Boot. https://github.com/doanduyhai/killrchat. Online; Accessed October 2020.

[3] Scott Ambler. 2006. *Refactoring databases : evolutionary database design.* Addison Wesley, Upper Saddle River, NJ.

[4] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 331–343. https://doi.org/10.1145/3035918.3056103

[5] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *PVLDB* 7, 3 (2013), 181–192. http://www.vldb.org/pvldb/vol7/p181-bailis.pdf

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

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

[8] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.* 1–10. https://doi.org/10.1145/223784.223785

[9] Philip A. Bernstein and Sudipto Das. 2013. Rethinking Eventual Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 923–928. https://doi.org/10.1145/2463676.2465339

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

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

[12] 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

[13] 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

[14] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Foundations and Trends in Programming Languages* 1, 1-2 (2014), 1–150.

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

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, Berkeley, CA, USA, 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905

[17] 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.

[18] 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

[19] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. 2012. A Data-Centric Approach to Synchronization. *ACM Trans. Program. Lang. Syst.* 34, 1, Article 4 (May 2012), 48 pages. https://doi.org/10.1145/2160910.2160913

[20] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (Nov. 1976), 624–633. https://doi.org/10.1145/360363.360369

[21] Stephane Faroult. 2008. *Refactoring SQL applications.* O'Reilly Media, Sebastopol, Calif.

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

[23] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[24] Martin Fowler. 2019. *Refactoring : improving the design of existing code.* Addison-Wesley, Boston.

[25] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[26] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

[27] 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, St. Petersburg, FL, USA, January 20 - 22, 2016.* 371–384. https://doi.org/10.1145/2837614.2837625

[28] Tyler Hobbs. 2015. Basic Rules of Cassandra Data Modeling. https://www.datastax.com/blog/basic-rules-cassandra-data-modeling [Online; accessed March-2021].

[29] Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. *PACMPL* 3, POPL (2019), 74:1–74:32. https://dl.acm.org/citation.cfm?id=3290387

[30] 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

[31] 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

[32] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2018. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *PACMPL* 2, POPL (2018), 27:1–27:34. https://doi.org/10.1145/3158115

[33] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable Replicated Data Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360580

[34] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[35] 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* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

[36] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated

Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(NSDI'13)*. USENIX Association, USA, 313–328.

[37] Shiyong Lu, Arthur Bernstein, and Philip Lewis. 2004. Correct Execution of Transactions at Different Isolation Levels. *IEEE Transactions on Knowledge and Data Engineering* 16, 9 (2004), 1070–1081.

[38] MySQL 2020. Transaction Isolation Levels. https://dev.mysql.com/doc/refman/5.6/en/innodb-transaction-isolation-levels.html Accessed: 2020-01-1 10:00:00.

[39] 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

[40] 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

[41] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844

[42] Andrew Pavlo, Evan P. C. Jones, and Stanley Zdonik. 2011. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *Proc. VLDB Endow.* 5, 2 (Oct. 2011), 85–96. https://doi.org/10.14778/2078324.2078325

[43] PostgreSQL 2020. Transaction Isolation. https://www.postgresql.org/docs/9.1/static/transaction-iso.html Accessed: 2020-01-1 10:00:00.

[44] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 430–441. https://doi.org/10.1145/2452376.2452427

[45] Kia Rahmani, Gowtham Kaki, and Suresh Jagannathan. 2018. Fine-grained Distributed Consistency Guarantees with Effect Orchestration. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data* (Porto, Portugal) *(PaPoC '18)*. ACM, New York, NY, USA, Article 6, 5 pages. https://doi.org/10.1145/3194261.3194267

[46] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 117 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360543

[47] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2021. Repairing Serializability Bugs in Distributed Database Programs via Automated Schema Refactoring (extended version). *CoRR* abs/2103.05573 (2021). arXiv:2103.05573 https://arxiv.org/abs/2103.05573

[48] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable Consistency in MongoDB. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2071–2081. https://doi.org/10.14778/3352063.3352125

[49] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Lecture Notes in Computer Science, Vol. 6976. Springer Berlin Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[50] 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* (Portland, OR, USA) *(PLDI 2015)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

[51] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23^{rd} ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. ACM, New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

[52] Michael Stonebraker and Andy Pavlo. 2012. The SEATS Airline Ticketing Systems Benchmark. http://hstore.cs.brown.edu/projects/seats

[53] Gonçalo Tomás, Peter Zeller, Valter Balegas, Deepthi Akkoorath, Annette Bieniusa, João Leitão, and Nuno Preguiça. 2017. FMKe: A Real-World Benchmark for Key-Value Data Stores. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data* (Belgrade, Serbia) *(PaPoC '17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/3064889.3064897

[54] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2017. Verifying Equivalence of Database-driven Applications. *Proc. ACM Program. Lang.* 2, POPL, Article 56 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158144

[55] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 286–300. https://doi.org/10.1145/3314221.3314588

[56] 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* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 5–20. https://doi.org/10.1145/3035918.3064037

[57] Kamal Zellag and Bettina Kemme. 2014. Consistency Anomalies in Multi-tier Architectures: Automatic Detection and Prevention. *The VLDB Journal* 23, 1 (Feb. 2014), 147–172. https://doi.org/10.1007/s00778-013-0318-x

[58] William Zola. 2014. 6 Rules of Thumb for MongoDB Schema. https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1 [Online; accessed March-2021].