Repairing Serializability Bugs in Distributed Database Programs via Automated Schema Refactoring

Anonymous Author(s)

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.

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 [31, 33, 40, 43]. 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 [37], 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 [25, 34, 42].

Uncovering such bugs is a delicate and highly error-prone task even in centralized environments: in one recent study, Warszawski and Bailis [48] examined 12 popular eCommerce 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, 25, 28, 36]. 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] architected 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 new tables, 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, 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 schedule. 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 availability of the course they 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 that these transactions should be performed atomically and in isolation. Atomicity guarantees that a transaction never observes intermediate updates of another concurrently executing 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 [24] or complex version management systems [10], which severely reduce the system’s available concurrency, especially in distributed environments.

Figure 1. Database schemas and code snippets from an online course management program
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 [31, 35, 38, 40].

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 setSt 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](image)

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, 36, 48]. Given potential serializability violations, the standard solution is to strengthen the atomicity and isolation requirements on the offending transactions to eliminate the undesirable behavior, at the cost of increased synchronization overhead or reduced availability [7, 25, 42].

### 2.1 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 particular row. Alternatively, it is possible to decompose database state to minimize the number of distinct updates to different fields, 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 tradeoff since the cost of global concurrency control or coordination is often problematic in these settings, an observation that is borne out 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, in-
stead 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 modi-
ﬁed 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 elim-
inating the problematic data accesses in the original program.
Similarly, regSt updates the student’s st_co_id ﬁeld and in-
serts a new record into the schema COURSE_CO_ST_CNT_LOG.
Using the function uuid() ensures that a new record is in-
serted 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, i.e., it does not exhibit any new behav-
iors which were not originally possible, despite eliminating
problematic serializability errors found in the ﬁrst program.

This program is the result of several database schema
refactorings [3, 20, 23], 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.

We have implemented a tool named Arropos 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 pro-
gram described above, for example, is automatically gener-
ated by Arropos from the original shown in Figure 1. Figure 4
presents the Arropos pipeline. A static analysis engine is
used to identify potential serializability anomalies in a given
program. The program is then preprocessed 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
test 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.

3 Database Programs

The syntax of our database programs is given in Figure 5. A

Figure 5. Syntax of database programs

program $P$ is deﬁned in terms of a set of database schemas
$(\bar{R})$, and a set of transactions $(\bar{T})$. A database schema consists
of a schema name ($\rho$) and a set of ﬁeld names ($\bar{f}$). A data-
base record ($R$) 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 $R_{id}$ 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 $\in \text{FldName}$, whose
value determines if a record is actually present in the table.
This ﬁeld allows us to model DELETE and INSERT commands
without explicitely 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.

Figure 4. Schematic overview of Arropos
Both database commands (SELECT and UPDATE) require an explicit where clause \( \phi \) to filter the records they retrieve or update. \( \phi_{\text{id}} \) denotes the set of fields appearing in a clause \( \phi \).

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 \( \text{agg}(x.f) \), or accessed individually, using \( \text{at}^e(x.f) \).

### 3.1 Data Store Semantics

Database states \( \Sigma \) are modeled as a triple \( (\text{str}, \text{vis}, \text{cnt}) \), where \( \text{str} \) is a set of database events \( (\eta) \) that captures the history of all reads and writes performed by a program operating over the database, and \( \text{vis} \) is a partial order on those events. The execution counter, \( \text{cnt} \), 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 \( (\Sigma) \), and a primary key \( r \in R_{\text{id}} \), 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, \( \text{rd}(r,r,f) \), which witness that the field \( f \) of the record with the primary key \( r \) was accessed when the value of the execution counter was \( r \). Similarly, a write event, \( \text{wt}(r,r,f,n) \), records that the field \( f \) of record \( r \) was assigned the value \( n \) at timestamp \( r \). The timestamp associated with an event \( \eta \) is denoted by \( \eta_r \).

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, 19, 32, 49]. In this paper, we consider behaviors induced when the database guarantees only a very weak form of consistency and isolation that allows transactions to (a) witness partial updates of committed actions and (b) 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 those records.

To capture these behaviors, we use a visibility relation between events, \( \text{vis} \), 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 \( \text{vis} \subseteq \Sigma \times \Sigma \) between database states, which is constrained as follows:

\[
\text{vis}' \subseteq \text{vis} \Rightarrow \text{str}' \subseteq \text{str} \quad \forall \eta' \in \text{str}' \forall \text{vis}' \in \text{vis} \Rightarrow (\eta \in \text{str}' \land \text{vis}' = \text{vis} \Rightarrow \text{cnt}' = \text{cnt})
\]

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

Figure 6 presents the operational semantics of our language, which is defined by a small-step reduction relation, \( \Rightarrow \subseteq \Sigma \times \Sigma \times \Sigma \times \Sigma \), between tuples of data-store states (\( \Sigma \)) and a set of currently executing transaction instances (\( \Gamma' \subseteq \text{ctx} \times \text{Var} \rightarrow R_{\text{id}} \times F \)). 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, \( P_{\text{txn}} \). At every step, a new transaction instance can be added to the set of currently running transactions via \( \text{txn}\text{-}\text{invoke} \). Alternatively, a currently running transaction instance can be processed via \( \text{txn}\text{-}\text{step} \). Finally, if the body of a transaction has been completely processed, its return expression is evaluated via \( \text{txn}\text{-}\text{ret} \); 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 (\( \Rightarrow \)) on database states, local states, and commands. The semantics for control commands are straightforward outside of the (itter) rule, which uses an auxiliary function \( \text{concat}(n,c) \) to sequence \( n \) copies of the command \( c \).

Expression evaluation is defined using the big-step relation \( \llbracket \cdot \rrbracket \subseteq \text{ctx} \times \text{Var} \rightarrow R_{\text{id}} \times F \times \text{Exp} \times \text{Val} \) which, given a store holding the results of previous query commands, determines the final value of the expression. The full definition of \( \llbracket \cdot \rrbracket \) 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 (\( \Sigma' \subseteq \Sigma \)) that is used to select or update the contents of records. Neither rule imposes any restrictions on \( \Sigma' \) other than the consistency constraints defined by (ConstructView). The key component of each rule is how it defines the set of new events (\( e \)) that are added to the database. In the select rule, \( e_1 \) captures the withdrawals that occur on database-wide scans to identify records satisfying the select command’s where clause. In an abuse of notation, we write \( \Delta_i(\phi(f \in \eta)) \llbracket n \rrbracket \) as shorthand for \( \Delta_i(\phi[\text{this.f} \in \eta] \llbracket n \rrbracket) \). \( e_2 \) constructs the appropriate read events of these retrieved records. The (update) rule similarly defines \( \epsilon \), 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 (\( \Sigma' \)) of the global store (\( \Sigma \)). 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 $\Gamma$ in the trace is of the form: $\{t_1 : \text{skip}; m_1, \Delta_1\} \cup \cdots \cup \{t_k : \text{skip}; m_k, \Delta_k\}$. As a shorthand, we refer to the final state in a history $h$ as $h_{\text{final}}$. A serial execution history satisfies two important properties:

1. **Strong Atomicity**: $(\forall \eta, \eta', \eta_{\text{cont}} < \eta_{\text{cont}} \Rightarrow \text{vis}(\eta, \eta')) \land \forall \eta, \eta', \eta'' \cdot \text{st}(\eta, \eta') \land (\text{vis}(\eta, \eta'') \Rightarrow \text{vis}(\eta', \eta''))$

2. **Strong Isolation**: $(\forall \eta, \eta', \eta'' \cdot \text{st}(\eta, \eta') \land \text{vis}(\eta'', \eta') \Rightarrow \text{vis}(\eta'', \eta))$

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 $\eta''$ generated by any other transaction that is visible to an event $\eta'$ generated by $T$ to be visible to any event $\eta$ 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. $\text{atomic}(\eta, \eta') \equiv \forall \eta'' \cdot (\text{vis}(\eta, \eta'') \Rightarrow \forall \eta'' \cdot (\text{vis}(\eta, \eta'') \equiv \text{vis}(\eta', \eta'')) \land (\text{vis}(\eta'', \eta) \Rightarrow \text{vis}(\eta'', \eta'))$.

Given a program $P$, we define a database access pair $(\chi)$ as a quadruple $(c_1, \bar{f}_1, c_2, \bar{f}_2)$ where $c_1$ and $c_2$ are database commands from a transaction in $P$, and $\bar{f}_1$ (resp. $\bar{f}_2$) 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 $f_1 \in \bar{f}_1$ which induces a serializability anomaly with another event generated by $c_2$ accessing field $f_2 \in \bar{f}_2$. An example of an anomalous access pair for the program from in Section 2, is $(S1, \{\text{st\_name}\}, S2, \{\text{em\_addr}\})$ and $(U1, \{\text{st\_name}\}, U2, \{\text{em\_addr}\})$; 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, 39] can be...
adapted for this purpose. Section 6 provides more details about how this works in Atropos.

4 Refactoring of Database Programs

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.

4.1 Database Containment

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

The containment relation between a table (e.g. COURSE) and a set of tables (e.g. STUDENT and COURSE_ST_CNT_LOG) is defined using a set of mappings called value correspondences [47]. 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 \( \theta : R \rightarrow R' \), 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 \( \alpha : \mathbb{V} \rightarrow \mathbb{V} \), is used to aggregate a set of values. We say that a table \( X \) is contained by a set of tables \( X' \) under a set of value correspondences, \( (\text{COURSE, STUDENT, co_avail, st_co_avail, } \theta_1, \text{ any}) \) and \((\text{COURSE, COURSE_ST_CNT_LOG, co_st_cnt, co_st_cnt_log, } \theta_2, \text{ sum})\), where \( \theta_1(1) = \{100, 200\}, \theta_2(2) = \{300\}\), \( \theta_2(1) = \{(1, 11), (1, 33)\}\) and \( \theta_2(2) = \{(2, 22)\}\). The aggregator function any : \( \mathbb{V} \rightarrow \mathbb{V} \) returns a non-deterministically chosen value from a set of values.

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 \( h'_\text{fin} = (\Sigma', \Gamma') \)) is a refinement of an execution \( h \) (where \( h_\text{fin} = (\Sigma, \Gamma) \)) if and only if \( \Gamma' \) and \( \Gamma \) have the same collection of finalized transaction instances and there is a set of value correspondences \( \Sigma \) under which \( \Sigma \) is contained in \( \Sigma' \).

Lastly, we define a refactored program \( P' \) to be a refinement of the original program \( P \) 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 \( \leftrightarrow \subseteq \mathbb{V} \times P \times \mathbb{V} \times P \), 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 \( \rho \)); (2) adding a new field to an existing schema \( \rho \), 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 \( \nu \)).

The refactorings represented by (intro \( \nu \)) introduce a new value correspondence \( \nu \), and modify the body and return expressions of a programs transactions via a rewrite function, \( \llbracket \cdot \rrbracket_\nu \). A particular instantiation of \( \llbracket \cdot \rrbracket_\nu \) 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 \( \llbracket \cdot \rrbracket_\nu \) 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...
4.2.1 The redirect rule. Our first refactoring rule is parameterized over the choice of schemas and fields and uses the aggregator any. Given data store states $\Sigma$ and $\Sigma'$, the record correspondence is defined as: $[\hat{\theta}(r) = \{r' \mid r' \in R'_\text{id} \land \forall f \in R_{\text{id}}: \Sigma(r.f) \downarrow n \Rightarrow \Sigma'(r'.\hat{\theta}(f)) \downarrow n\}$. In essence, 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

$$\hat{\theta}_i(\text{COURSE}.\text{co_id}) = \text{STUDENT}.\text{st_co_id}$$

and

$$\hat{\theta}_i(\text{COURSE}.\text{co_id}) = \text{COURSE}_\text{CO}_\text{ST}_\text{CNT}_\text{LOG}.\text{co_id}$$

Defining the record correspondence this way ensures that if a record $r$ is selected in $\Sigma$, the corresponding set of records in $\Sigma'$ can be determined by identifying the values that were used to select $r$, without depending on any particular instance of the tables.

The definition of redirect for this rule is straightforward:

redirect($\phi$, $[\hat{\theta}]$) = $\bigwedge_{f \in \phi_{\text{id}}} \hat{\theta}(f) = \phi[f]_{\exp}$

The one wrinkle in this definition of redirect is that it is only defined when the where clause $\phi$ is well-formed, i.e. $\phi$ only consists of conjunctions of equality constraints on primary key fields. The expression used in such a constraint is denoted by $\phi[f]_{\exp}$. As an example, the where clause of command (S2) in Figure 9 (left) is well-formed, where $\phi[\text{em_id}]_{\exp} = x.\text{st_e_id}$. However, the where clause in (S2') after the refactoring step is not well-formed, since it does not constrain the primary key of the schema. This restriction ensures only select clauses accessing a single record of the original table will be rewritten. Expressions using variables containing the results of those queries are rewritten by substituting the source field name with the target field name, e.g. $\llbracket \text{at}^i(x.f) \rrbracket_\nu = \text{at}^i(x.f')$.

4.2.2 The logger rule. Unfortunately, instantiating intro $\nu$ 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 $\alpha = \text{sum}$ 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 ($\text{Log} \ R$) 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 $\text{Log} \ R.\text{log_id}$, which allows a set of records in $\text{Log} \ R$ to

---

1 A complete formalization of all three refactoring rules, their correctness criteria, and proofs of soundness is presented in the supplementary materials of this submission.
getSt(id):
1: x:=select * from STUDENT where st_id = id // S1
2: y:=select st_em_addr from STUDENT where st_id = id // S2
3: z:=select co_avail from COURSE where co_id = x. st_co_id // S3
4: return (y, em_addr)

intro (EMAIL, STUDENT, en_addr, st_en_addr, [d], any)

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

intro (EMAIL, STUDENT, em_addr, st_en_addr, [d], any)

update STUDENT set st_em_addr = email where em_id = x. st_em_id // U2
update EMAIL set em_addr = email where em_id = x. st_em_id // U1
update STUDENT set st_name = name where st_id = id // U1
update STUDENT set st_em_id = x. st_em_id // U2

Figure 9. A single program refactoring step, where \( \hat{\delta}(\text{EMAIL.em_addr}) = \text{STUDENT.st_em_addr} \)

represent each record in \( R \); and (iii) the schema \( \log R \) has a single field corresponding to the original field \( R.f \), denoted by \( \log R.f' \).

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 \( \text{COURSE.co}_\text{ST}_\text{CNT}_\text{LOG} \) from Section 2 is an example of a logging schema for the source schema and field \( \text{COURSE.co}_\text{ST}_\text{CNT} \).

Under these restrictions, we can define an implementation of \( \tilde{\delta} \) for the logger rule using \( \text{sum} \) as an aggregator.

This refactoring also uses a lifted function \( \hat{\theta} \) for its value correspondence, which allows \( \tilde{\delta} \) to reuse our earlier definition of redirect. We define \( \tilde{\delta} \) on accesses to \( f \) to use program-level aggregators, e.g., \( \tilde{\delta}(\text{at}(x.f)) = \text{sum}(x.f') \).

Finally, the rewritten \( \text{UPDATE} \) commands simply need to log any updates to the field \( f \), so its original value can be recovered in the transformed program, e.g.,

\[
\text{UPDATE } R \text{ SET } f = e + \text{at}(x.f) \quad \tilde{\delta}(v) = \text{UPDATE } R' \text{ SET } f' = e, \text{WHERE } \text{redirect}(\phi, v, \theta) \land R'.\log_id = \text{uuid}.
\]

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

**Theorem 4.2.** (Soundness) Any sequence of refactorings using the rewrite rules described in this section satisfy the correctness properties R1-R3.

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:

\[
\begin{align*}
\text{(U3, \{st_co_id, st_reg\}), U4, \{co_avail\}} & \quad (\chi_1) \\
\text{(S5, \{co_st_cnt\}), U4, \{co_st_cnt\}} & \quad (\chi_2)
\end{align*}
\]

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

The repair procedure next performs a preprocessing 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 preprocessing, the algorithm iterates over all detected anomalous access pairs \( \chi \) and attempts to repair them one by one using \text{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.

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 \text{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 \text{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 \(\chi_1\). 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 \(\chi_1\), but fails to eliminate \(\chi_2\). 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 \(\rho\) 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 (\(\chi_1\) and \(\chi_2\)) are eliminated in the final version of the transaction.

Figure 10. The repair algorithm

Figure 11. Repair steps of transaction regSt

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, 36]. 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.

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, including TPC-C, SEATS and SmallBank [13, 18, 25, 39, 42]. Table 1 presents the results for each benchmark. The first four columns 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.

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 $T'$ 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 [25, 28, 48], we identified three invariants to be preserved by each transaction$^2$: Interestingly, we were able to detect violations of all three invariants in the original program under EC, while the repaired program violated only one. 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 georeplicated 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; see supplementary materials for more details. Each node runs MongoDB (v4.2.9), 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 (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

---

$^2$Detailed descriptions of each invariant can be found in the supplementary materials of our submission.
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.

8 Related Work

Wang et al. [47] describe a synthesis procedure for generating programs consistent with a database refactoring, as determined by a verification procedure that establishes database program equivalence [46]. 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 [22] 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, 21, 27, 34], 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, 48] and static [13, 36, 39] techniques have been developed to discover these violations in various domains and settings.

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 datatypes (CRDTs) have been proposed. CRDTs are abstract datatypes (e.g., sets, counters) equipped with commutative operations that 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 [26, 42], 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.
9 Conclusions

This paper presents ATROPOS, a database refactoring tool intended to repair serializability violations. We have formalized the refactoring procedure and demonstrated experimental results that schema refactoring is a viable strategy for concurrency bug repair in modern database applications.

References
