Alone Together: Compositional Reasoning and Inference for Weak Isolation

GOWTHAM KAKI, Purdue University, USA
KARTIK NAGAR, Purdue University, USA
MAHSA NAJAFZADEH, Purdue University, USA
SURESH JAGANNATHAN, Purdue University, USA

Serializability is a well-understood correctness criterion that simplifies reasoning about the behavior of concurrent transactions by ensuring they are *isolated* from each other while they execute. However, enforcing serializable isolation comes at a steep cost in performance because it necessarily restricts opportunities to exploit concurrency even when such opportunities would not violate application-specific invariants. As a result, database systems in practice support, and often encourage, developers to implement transactions using weaker alternatives. These alternatives break the strong isolation guarantees offered by serializablity to permit greater concurrency. Unfortunately, the semantics of weak isolation is poorly understood, and usually explained only informally in terms of low-level implementation artifacts. Consequently, verifying high-level correctness properties in such environments remains a challenging problem.

To address this issue, we present a novel program logic that enables compositional reasoning about the behavior of concurrently executing weakly-isolated transactions. Recognizing that the proof burden necessary to use this logic may dissuade application developers, we also describe an inference procedure based on this foundation that ascertains the weakest isolation level that still guarantees the safety of high-level consistency invariants associated with such transactions. The key to effective inference is the observation that weakly-isolated transactions can be viewed as functional (monadic) computations over an abstract database state, allowing us to treat their operations as state transformers over the database. This interpretation enables automated verification using off-the-shelf SMT solvers.

Our development is parametric over a transaction's specific isolation semantics, allowing it to be applicable over a range of weak isolation mechanisms. Case studies and experiments on real-world applications (written in an embedded DSL in OCaml) demonstrate the utility of our approach, and provide strong evidence that automated verification of weakly-isolated transactions can be placed on the same formal footing as their strongly-isolated serializable counterparts.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Information systems → Integrity checking**; *Relational database model*;

Additional Key Words and Phrases: Transactions, Weak Isolation, Concurrency, Rely-Guarantee, Verification

ACM Reference Format:
Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2018. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *Proc. ACM Program. Lang.* 2, POPL, Article 27 (January 2018), 34 pages. https://doi.org/10.1145/3158115

Authors' addresses: Gowtham Kaki, Purdue University, USA, gkaki@purdue.edu; Kartik Nagar, Purdue University, USA, nagark@purdue.edu; Mahsa Najafzadeh, Purdue University, USA, mnajafza@purdue.edu; Suresh Jagannathan, Purdue University, USA, suresh@cs.purdue.edu.

2475-1421/2018/1-ART27
https://doi.org/10.1145/3158115

1 INTRODUCTION

Database transactions allow users to group operations on multiple objects into a single logical unit, equipped with a set of four key properties - atomicity, consistency, isolation, and durability (ACID). Concurrency control mechanisms provide specific instantiations of these properties to yield different ACID variants that characterize how and when the effects of concurrently executing transactions become visible to one another. *Serializability* is a particularly well-studied instantiation that imposes strong atomicity and isolation constraints on transaction execution, ensuring that any permissible concurrent schedule yields results equivalent to a serial one in which there is no interleaving of actions from different transactions.

The guarantees provided by serializability do not come for free, however - pessimistic concurrency control methods require databases to use expensive mechanisms such as two-phase locking that incur overhead to deal with deadlocks, rollbacks, and re-execution [Eswaran et al. 1976; Garcia-Molina et al. 2008]. Similar criticisms apply to optimistic multi-version concurrency control methods that must deal with timestamp and version management [Bernstein and Goodman 1983]. These issues are exacerbated when the database is replicated, requiring additional coordination mechanisms [Bailis et al. 2013a; Bernstein and Das 2013; Davidson et al. 1985; Gilbert and Lynch 2002].

Because serializable transactions favor correctness over performance, there has been long-standing interest [Gray et al. 1976] in the database community to consider weaker variants that try to recover performance, even at the expense of simplicity and ease of reasoning. These instantiations permit a transaction to witness various effects of newly committed, or even concurrently running, transactions while it executes, thus weakening serializability's strong isolation guarantees. The ANSI SQL 92 standard defines three such weak isolation levels which are now implemented in many relational and NoSQL databases. Not surprisingly, weakly-isolated transactions have been found to significantly outperform serializable transactions on benchmark suites, both on single-node databases and multi-node replicated stores [Bailis et al. 2013a, 2014; Shasha and Bonnet 2003], leading to their overwhelming adoption. A 2013 study [Bailis et al. 2013b] of 18 popular ACID and "NewSQL" databases found that only three of them offer serializability by default, and half, including Oracle 11g, do not offer it at all. A 2015 study [Bailis et al. 2015] of a large corpus of database applications finds no evidence that applications manifestly change the default isolation level offered by the database. Taken together, these studies make clear that weakly-isolated transactions are quite prevalent in practice, and serializable transactions are often eschewed.

Unfortunately, weak isolation admits behaviors that are difficult to comprehend [Berenson et al. 1995]. To quantify weak isolation anomalies, Fekete *et al.* [Fekete et al. 2009] devised and experimented with a microbenchmark suite that executes transactions under *Read Committed* weak isolation level - default level for 8 of the 18 databases studied in [Bailis et al. 2013b], and found that 25 out of every 1000 rows in the database violate at least one integrity constraint. Bailis *et al.* [Bailis et al. 2015] rely on Rails' *uniqueness validation* to maintain uniqueness of records while serving Linkbench's [Armstrong et al. 2013] insertion workload (6400 records distributed over 1000 keys; 64 concurrent clients), and report discovering more than 10 duplicate records. Rails relies on database transactions to validate uniqueness during insertions, which is sensible if transactions are serializable, but incorrect under the weak isolation level used in the experiments. The same study has found that 13% of all invariants among 67 open source Ruby-on-Rails applications are at risk of being violated due to weak isolation. Indeed, incidents of safety violations due to executing applications in a weakly-isolated environment have been reported on web services in production [SciMed Bug 2016; Starbucks Bug 2016], including in safety-critical applications such as bitcoin exchanges [Bitcoin Bug 2016; Poloniex Bug 2016]. While enforcing serializability for all transactions would be sufficient to avoid these errors and anomalies, it would likely be an overly conservative strategy; indeed, 75% of the invariants studied in [Bailis et al. 2015] were shown to be preserved under some form of weak isolation. When to use weak isolation, and in what form, is therefore a prominent question facing all database programmers.[1]

A major problem with weak isolation as currently specified is that its semantics in the context of user programs is not easily understood. The original proposal [Gray et al. 1976] defines multiple "degrees" of weak isolation in terms of implementation details such as the nature and duration of locks held in each case. The ANSI SQL 92 standard defines four levels of isolation (including serializability) in terms of various undesirable *phenomena* (*e.g., dirty reads* - reading data written by an uncommitted transaction) each is required to prevent. While this is an improvement, this style of definition still requires programmers to be prescient about the possible ways various undesirable phenomena might manifest in their applications, and in each case determine if the phenomenon can be allowed without violating application invariants. This is understandably hard, especially in the absence of any formal underpinning to define weak isolation semantics. Adya [Adya 1999] presents the first formal definitions of some well-known isolation levels in the context of a sequentially consistent (SC) database. However, there has been little progress relating Adya's system model to a formal operational semantics or a proof system that can facilitate rigorous correctness arguments. Consequently, reasoning about weak isolation remains an error prone endeavor, with major database vendors [MySQL 2016; Oracle 2016; PostgreSQL 2016] continuing to document their isolation levels primarily in terms of the undesirable phenomena a particular isolation level may induce, placing the burden on the programmer to determine application correctness.

Recent results on reasoning about application invariants in the presence of weak consistency [Balegas et al. 2015; Burckhardt et al. 2014; Gotsman et al. 2016; Li et al. 2014, 2012] address broadly related concerns. Weak consistency is a phenomenon that manifests on replicated data stores, where atomic operations are concurrently executed against different replicas, resulting in an execution order inconsistent with any sequential order. In contrast, weak isolation is a property of concurrent transactions interfering with one another, resulting in an execution order that is not serializable. Unlike weak consistency, weak isolation can manifest even in an unreplicated setting, as evident from the support for weakly-isolated transactions on conventional (unreplicated) databases as mentioned above.

In this paper, we propose a program logic for weakly-isolated transactions along with automated verification support to allow developers to verify the soundness of their applications, without having to resort to low-level operational reasoning as they are forced to do currently. We develop a set of syntax-directed compositional proof rules that enable the construction of correctness proofs for transactional programs in the presence of a weakly-isolated concurrency control mechanism. Realizing that the proof burden imposed by these rules may discourage applications programmers from using them, we also present an inference procedure that automatically verifies the weakest isolation level for a transaction while ensuring its invariants are maintained. The key to inference is a novel formulation of database state (represented as sets of tuples) as a monad, and in which database computations are interpreted as state transformers over these sets. This interpretation leads to an encoding of database computations amenable for verification by off-the-shelf SMT solvers. The paper makes the following contributions:

(1) We analyze properties of weak isolation in the context of a DSL embedded in OCaml that treats SQL-based relational database operations (e.g., inserts, selects, deletes, updates, etc.) as computations over an abstract database state.

[1]This position has been echoed by database researchers who lament the lack of a better understanding of this problem; see e.g., http://www.bailis.org/blog/understanding-weak-isolation-is-a-serious-problem.

(2) We develop an operational semantics and a compositional rely-guarantee style proof system for this language capable of relating high-level application invariants to database state, parameterized by a weak isolation semantics that selectively exposes the visibility of these operations to other transactions.
(3) We devise an inference algorithm capable of discovering the weakest isolation level that is sound with respect to a transaction's high-level consistency requirements. The algorithm interprets database operations as state transformers expressed in a language amenable for translation into a decidable fragment of first-order logic, and is thus suitable for automated verification using off-the-shelf SMT solvers.
(4) We present details of an implementation along with an evaluation study on real database benchmarks that justify our approach, and demonstrate the utility of our inference mechanism.

Our results provide the first formalization of weakly-isolated transactions, along with an expressive and compositional proof automation framework capable of verifying the safety of high-level consistency conditions attached to these transactions. Collectively, these contributions allow weakly-isolated transactions to enjoy the same rigorous reasoning capabilities as their strongly-isolated (serializable) counterparts.

The remainder of the paper is organized as follows. The next section provides motivation and background on serializable and weakly-isolated transactions. §3 presents an operational semantics for a core language that supports weakly-isolated transactions, parameterized over different isolation notions. §4 formalizes the proof system that we use to reason about program invariants, and establishes the soundness of these rules with respect to the semantics. §5 describes the inference algorithm, and the state transformer encoding. We describe our implementation in §6, and provide case studies and benchmark results in §7. Related work is given in §8, and §9 concludes.

2 MOTIVATION

We present our ideas in the context of a DSL embedded in OCaml that manages an abstract database state that can be manipulated via a well-defined `SQL` interface. Arbitrary database computations can be built around this interface, which can then be run as transactions using the `atomically_do` combinator provided by the DSL.

Fig. 1 shows a simplified version of the TPC-C `new_order` transaction written in this language. TPC-C is a widely-used and well-studied Online Transaction Processing (OLTP) benchmark that models an order-processing system for a wholesale parts supply business. The business logic is captured in 5 database transactions that operate on 9 tables; `new_order` is one such transaction that uses `District`, `Order`, `New_order`, `Stock`, and `Order_line` tables. The transaction acts on the behalf of a customer, whose id is `c_id`, to place a new order for a given set of items (`item_reqs`), to be served by a warehouse under the district identified by `d_id`. Fig. 2 illustrates the relationship among these different tables.

The transaction manages order placement by invoking appropriate SQL functionality, captured by various calls to functions defined by the `SQL` module. All `SQL` operators supported by the module take a table name (a nullary constructor) as their first argument. The higher-order `SQL.select1` function accepts a boolean function that describes the selection criteria, and returns any record that meets the criteria (it models the SQL query `SELECT ... LIMIT 1`). `SQL.update` also accepts a boolean function (its 3^{rd} argument) to select the records to be updated. Its 2^{nd} argument is a function that maps each selected record to a new (updated) record. `SQL.insert` inserts a given record into the specified table in the database.

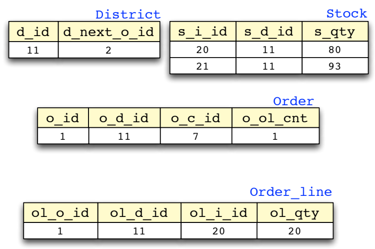
```
let new_order (d_id, c_id, item_reqs) = atomically_do @@ fun () ->
  let dist = SQL.select1 District (fun d -> d.d_id = d_id) in
  let o_id = dist.d_next_o_id in
  begin
    SQL.update(* UPDATE *) District
              (* SET *)(fun d -> {d with d_next_o_id = d.d_next_o_id + 1})
              (* WHERE *)(fun d -> d.d_id = d_id );
    SQL.insert(* INSERT INTO *) Order (* VALUES *){o_id=o_id;
              o_d_id=d_id; o_c_id=c_id; o_ol_cnt=S.size item_reqs; };
    foreach item_reqs @@ fun item_req ->
      let stk = SQL.select1(* SELECT * FROM *) Stock
                (* WHERE *)(fun s -> s.s_i_id = item_req.ol_i_id &&
                                     s.s_d_id = d_id)(* LIMIT 1 *) in
      let s_qty' = if stk.s_qty >= item_req.ol_qty + 10
                   then stk.s_qty - item_req.ol_qty
                   else stk.s_qty - item_req.ol_qty + 91 in
      SQL.update Stock (fun s -> {s with s_qty = s_qty'})
                       (fun s -> s.s_i_id = item_req.ol_i_id);
      SQL.insert Order_line {ol_o_id=o_id; ol_d_id=d_id;
                             ol_i_id=item_req.ol_i_id; ol_qty=item_req.ol_qty}
  end
```

Fig. 1. TPC-C new_order transaction

The new_order transaction inserts a new Order record, whose id is the sequence number of the next order under the given district (d_id). The sequence number is stored in the corresponding District record, and updated each time a new order is added to the system. Since each order may request multiple items (item_reqs), an Order_line record is created for each requested item to relate the order with the item. Each item has a corresponding record in the Stock table, which keeps track of the quantity of the item left in stock (s_qty). The quantity is updated by the transaction to reflect the processing of new orders (if the stock quantity falls below 10, it is automatically replenished by 91).

TPC-C defines multiple invariants, called *consistency conditions*, over the state of the application in the database. One such consistency condition is the requirement that for a given order o, the *order-line-count* field (o.o_ol_cnt) should reflect the number of order lines under the order; this is the number of Order_line records whose ol_o_id field is the same as o.o_id. In a sequential execution, it is easy to see how this condition is preserved. A new Order record is added with its o_id distinct from existing order ids, and its o_ol_cnt is set to be equal to the size of the item_reqs set. The foreach loop runs once for each item_req, adding a new Order_line record for each requested item, with its ol_o_id field set to o_id. Thus, at the end of the loop, the number of Order_line records in the database (i.e., the number of records whose ol_o_id field is equal to o_id) is guaranteed to be equal to the size of the item_reqs set, which in turn is equal to the Order record's o_ol_cnt field; these constraints ensure that the transaction's consistency condition is preserved.

Because the aforementioned reasoning is reasonably simple to perform manually, verifying the soundness of TPC-C's consistency conditions would appear to be feasible. Serializability aids the tractability of verification by preventing any interference among concurrently executing transactions while the new_order transaction executes, essentially yielding serial behaviors.



District

d_id	d_next_o_id
11	2

Stock

s_i_id	s_d_id	s_qty
20	11	80
21	11	93

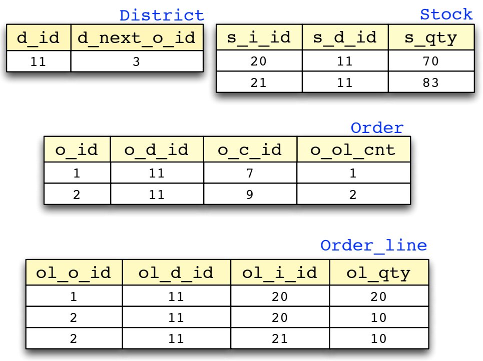
Order

o_id	o_d_id	o_c_id	o_ol_cnt
1	11	7	1

Order_line

ol_o_id	ol_d_id	ol_i_id	ol_qty
1	11	20	20

(a) A valid TPC-C database. The only existing order belongs to the district with d_id=11. Its id (o_id) is one less than the district's d_next_o_id, and its order count (o_ol_cnt) is equal to the number of order line records whose ol_o_id is equal to the order's id.



District

d_id	d_next_o_id
11	3

Stock

s_i_id	s_d_id	s_qty
20	11	70
21	11	83

Order

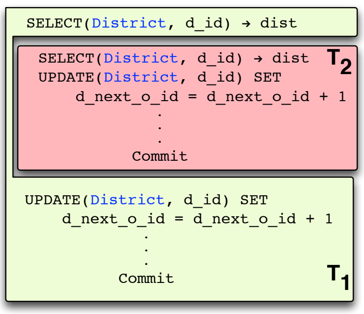
o_id	o_d_id	o_c_id	o_ol_cnt
1	11	7	1
2	11	9	2

Order_line

ol_o_id	ol_d_id	ol_i_id	ol_qty
1	11	20	20
2	11	20	10
2	11	21	10

(b) The database in Fig. 2a after correctly executing a new_order transaction. A new order record is added whose o_id is equal to the d_next_o_id from Fig. 2a. The district's d_next_o_id is incremented. The order's o_ol_cnt is 2, reflecting the actual number of order line records whose ol_o_id is equal to the order's id (2).

Fig. 2. Database schema of TPC-C's order management system. The naming convention indicates primary keys and foreign keys. For e.g., ol_id is the primary key column of the order line table, whereas ol_o_id is a foreign key that refers to the o_id column of the order table.



```
SELECT(District, d_id) → dist
    SELECT(District, d_id) → dist
    UPDATE(District, d_id) SET
          d_next_o_id = d_next_o_id + 1
                  .
                  .
                  .
                Commit
UPDATE(District, d_id) SET
      d_next_o_id = d_next_o_id + 1
                .
                .
                .
              Commit
```

Fig. 3. An RC execution involving two instances (T_1 and T_2) of the new_order transaction depicted in Fig. 1. Both instances read the d_id District record concurrently, because neither transaction is committed when the reads are executed. The subsequent operations are effectively sequentialized, since T_2 commits before T_1. Nonetheless, both transactions read the same value for d_next_o_id resulting in them adding Order records with the same ids, which in turn triggers a violation of TPC-C's consistency condition.

Under weak isolation[2], however, interferences of various kinds are permitted, leading to executions superficially similar to executions permitted by concurrent (racy) programs [Gammie et al. 2015; Hawblitzel et al. 2015]. To illustrate, consider the behavior of the new_order transaction when executed under a *Read Committed* (RC) isolation level, the default isolation level in 8 of the 18 databases studied in [Bailis et al. 2013b]. An executing RC transaction is isolated from *dirty writes*, i.e., writes of uncommitted transactions, but is allowed to witness the writes of concurrent transactions as soon as they are committed. Thus, with two concurrent instances of the new_order transaction (call them T_1 and T_2), both concurrently placing new orders for different customers under the same district (d_id), RC isolation allows the execution shown in Fig. 3.

The figure depicts an execution as a series of SQL operations. In the execution, the new_order instance T_1 (green) reads the d_next_o_id field of the district record for d_id, but before it increments the field, another new_order instance T_2 (red) begins its execution and commits. Note that T_2 reads the same d_next_o_id value as T_1, and inserts new Order and Order_line records with their o_id and ol_o_id fields (resp.) equal

[2]Weak isolation does not violate atomicity as long as the witnessed effects are those of committed transactions

to d_next_o_id. T_2 also increments the d_next_o_id field, which T_1 has already acccessed. This is allowed because reads typically do not obtain a mutually exclusive lock on most databases. After T_2's commit, T_1 resumes execution and adds new Order and Order_line fields with the same order id as T_1. Thus, at the end of the execution, Order_line records inserted by T_1 and T_2 all bear the same order id. There are also two Order records with the same district id (d_id) and order id, none of whose o_ol_cnt reflects the actual number of Order_line records inserted with that order id. This clearly violates TPC-C's consistency condition.

This example does not exhibit any of the anomalies that *characterize* RC isolation [Berenson et al. 1995][3]. For instance, there are no *lost writes* since both concurrent transactions' writes are present in the final state of the database. Program analyses that aim to determine appropriate isolation by checking for possible manifestations of RC-induced anomalies would fail to identify grounds for promoting the isolation level of new_order to something stronger. Yet, if we take the semantics of the application into account, it is quite clear that RC is not an appropriate isolation level for new_order.

While reasoning in terms of anomalies is cumbersome and inadequate, reasoning about weak isolation in terms of traces [Adya 1999; Cerone et al. 2015] on memory read and write actions can complicate high-level reasoning. A possible alternative would be to utilize concurrent program verification methods where the implementation details of weak isolation are interleaved within the program, yielding a (more-or-less) conventional concurrent program. But, considering the size and complexity of real-world transaction systems, this strategy is unlikely to scale.

In this paper, we adopt a different approach that *lifts* isolation semantics (*not* their implementations) to the application layer, providing a principled framework to simultaneously reason about application invariants and isolation properties. To illustrate this idea informally, consider how we might verify that new_order is sound when executed under *Snapshot Isolation* (SI), a stronger isolation level than RC. Snapshot isolation allows transactions to be executed against a private snapshot of the database, thus admitting concurrency, but it also requires that there not be any write-write conflicts (i.e., such a conflict occurs if concurrently executing transactions modify the same record) among concurrent transactions when they commit. Write-write conflicts can be eliminated in various ways, e.g., through conflict detection followed by a rollback, or through exclusive locks, or a combination of both. For instance, one possible implementation of SI, close to the one used by PostgreSQL [PostgreSQL 2016], executes a transaction against its private snapshot of the database, but obtains exclusive locks on the actual records in the database before performing writes. A write is performed only if the record that is to be written has not already been updated by a concurrent transaction. Conflicts are resolved by abort and roll back.

As this discussion hints, implementations of SI on real databases such as PostgreSQL are highly complicated, often running into thousands of lines of code. Nonetheless, the semantics of SI, in terms of how it effects transitions on the database state, can be captured in a fairly simple model. First, effects induced by one transaction (call it T) are not visible to another concurrently executing one during T's execution. Thus, from T's perspective, the global state does not change during its execution. More formally, for every operation performed by T, the global state T witnesses before (Δ) and after (Δ') executing the operation is the same ($\Delta' = \Delta$). After T finishes execution, it commits its changes to the actual database, which may have already incorporated the effects of concurrent transactions. In executions where T successfully commits, concurrent transactions are guaranteed to not be in write-write conflict with T. Thus, if Δ is the global state that T witnessed when it finished execution (the snapshot state), and Δ' is the state to which T commits, then the difference

[3]Berenson *et al.* characterize isolation levels in terms of the anomalies they *admit*. For example, RC is characterized by *lost writes* because it admits the anomaly.

between Δ and Δ' should not result in a write-write conflict with T. To concretize this notion, let the database state be a map from database locations to values, and let δ denote a transaction-local log that maps the locations being written to their updated values. The absence of write-write conflicts between T and the diff between Δ and Δ' can be expressed as: $\forall x \in dom(\delta), \Delta'(x) = \Delta(x)$. In other words, the semantics of SI can be captured as an axiomatization over transitions of the database state ($\Delta \longrightarrow \Delta'$) during a transaction's (T) lifetime:

- While T executes, $\Delta' = \Delta$.
- After T finishes execution, but before it commits its local state δ, $\forall(x \in dom(\delta)).\ \Delta'(x) = \Delta(x)$.

This simple characterization of SI isolation allows us to verify the consistency conditions associated with the `new_order` transaction. First, since the database does not change ($\Delta' = \Delta$) during execution of the transaction's body, we can reason about `new_order` as though it executed in complete isolation until its commit point, leading to a verification process similar to what would have been applied when reasoning sequentially. When `new_order` finishes execution, however, but before it commits, the SI axiomatization shown above requires us to consider global state transitions $\Delta \longrightarrow \Delta'$ that do not include changes to the records (δ) written by `new_order`, i.e., $\forall(x \in dom(\delta)).\ \Delta'(x) = \Delta(x)$. The axiomatization precludes any execution in which there are concurrent updates to shared table fields (e.g., `d_next_o_id` on the same `District` table), but does not prohibit interferences that write to different tables, or write to different records in the same table. We need to reason about the safety of such interferences with respect to `new_order`'s consistency invariants to verify `new_order`.

We approach the verification problem by first observing that a relational database is a significantly simpler abstraction than shared memory. Its primary data structure is a table, with no primitive support for pointers, linked data structures, or aliasing. Although a database essentially abstracts a mutable state, this state is managed through a well-defined fixed number of interfaces (SQL statements), each tagged with a logical formula describing what records are accessed and updated.

This observation leads us away from thinking of a collection of database transactions as a simple variant of a concurrent imperative program. Instead, we see value in viewing them as essentially functional computations that manage database state abstractly, mirroring the structure of our DSL. By doing so, we can formulate the semantics of database operations as state transformers that explicitly relate an operation's pre- and post-states, defining the semantics of the corresponding transformer algorithmically, just like classical predicate transformer semantics (e.g., weakest pre-condition or strongest post-condition). In our case, a transformer interprets a SQL statement in the set domain, modeling the database as a set of records, and a SQL statement as a function over this set. Among other things, one benefit of this approach is that low-level loops can now be substituted with higher-order combinators that automatically lift the state transformer of its higher-order argument, i.e., the loop body, to the state transformer of the combined expression, i.e., the loop. We illustrate this intuition on a simple example.

```
foreach item_reqs @@ fun item_req ->
  SQL.update Stock (fun s -> {s with s_qty = k1})
                   (fun s -> s.s_i_id = item_req.ol_i_id);
  SQL.insert Order_line {ol_o_id=k2; ol_d_id=k3;
                         ol_i_id=item_req.ol_i_id; ol_qty=item_req.ol_qty}
```

Fig. 4. Foreach loop from Fig. 1

Fig. 4 shows a (simplified) snippet of code taken from Fig. 1. Some irrelevant expressions have been replaced with constants (k1, k2, and k3). The body of the loop executes a SQL update followed by an insert. Recall that a transaction reads from the global database (Δ), and writes to a transaction-local database (δ) before committing these updates. An update statement filters the records that match the search criteria from Δ and computes the updated records that are to be added to the local database. Thus, the state transformer for the update statement (call it T_U) is the following function on sets[4]:

$$\begin{array}{l} \lambda(\delta,\Delta).\ \delta \cup \Delta \gg\!= (\lambda s.\text{if table}(s) = \texttt{Stock} \ \wedge\ \texttt{s.s\_i\_id} = \texttt{item\_req.ol\_i\_id} \\ \qquad\qquad\qquad\qquad\quad \text{then } \{\langle \texttt{s\_i\_id} = \texttt{s.s\_i\_id};\ \texttt{s\_d\_id} = \texttt{s.s\_d\_id};\ \texttt{s\_qty} = \texttt{k1}\rangle\} \\ \qquad\qquad\qquad\qquad\quad \text{else } \emptyset) \end{array}$$

Here, the set bind operator extracts record elements (s) from the database, checks the precondition of the update action, and if satisfied, constructs a new set containing a single record that is identical to s except that it binds field `s_qty` to value k_1. This new set is added (via set union) to the existing local database state δ.[5]

The transformer ($\mathsf{T}_I(\delta,\Delta)$) for the subsequent `insert` statement can be similarly constructed:

$$\lambda(\delta,\Delta).\ \delta \cup \{\langle \texttt{ol\_o\_id} = \texttt{k2};\ \texttt{ol\_d\_id} = \texttt{k3};\ \texttt{ol\_i\_id} = \texttt{item\_req.ol\_i\_id};\ \texttt{ol\_qty} = \texttt{item\_req.ol\_qty}\rangle\}$$

Observe that both transformers are of the form $\mathsf{T}(\delta,\Delta) = \delta \cup \mathsf{F}(\Delta)$, where F is a function that returns the set of records added to the transaction-local database (δ). Let F_U and F_I be the corresponding functions for T_U and T_I shown above. The state transformation induced by the loop body in Fig. 1 can be expressed as the following composition of F_U and F_I:

$$\lambda(\delta,\Delta).\ \delta \cup \mathsf{F}_U(\Delta) \cup \mathsf{F}_I(\Delta)$$

The transformer for the loop itself can now be computed to be:

$$\lambda(\delta,\Delta).\ \delta \cup \texttt{item\_reqs} \gg\!= (\lambda \texttt{item\_req}.\ \mathsf{F}_U(\Delta) \cup \mathsf{F}_I(\Delta))$$

Observe that the structure of the transformer mirrors the structure of the program itself. In particular, SQL statements become set operations, and the `foreach` combinator becomes set monad's bind ($\gg\!=$) combinator. As we demonstrate, the advantage of inferring such transformers is that we can now make use of a semantics-preserving translation from the domain of sets equipped with $\gg\!=$ to a decidable fragment of first-order logic, allowing us to leverage SMT solvers for automated proofs without having to infer potentially complex thread-local invariants or intermediate assertions. Sec. 5 describes this translation. In the exposition thus far, we assumed Δ remains invariant, which is clearly not the case when we admit concurrency. Necessary concurrency extensions of the state transformer semantics to deal with interference is also covered in Sec. 5. Before presenting the transformer semantics, we first focus our attention in the following two sections on the theoretical foundations for weak isolation, upon which this semantics is based.

3 $\mathcal{T}$: SYNTAX AND SEMANTICS

Fig. 5 shows the syntax and small-step semantics of $\mathcal{T}$, a core language that we use to formalize our intuitions about reasoning under weak isolation. Variables (x), integer and boolean constants (k), records (r) of named constants, sets (s) of such records, arithmetic and boolean expressions ($e_1 \odot e_2$), and record expressions ($\langle \bar{f} = \bar{e}\rangle$) constitute the syntactic class of expressions (e). Commands (c) include `SKIP`, conditional statements, `LET` constructs to bind names, `FOREACH` loops, SQL statements, their sequential composition ($c_1; c_2$), transactions ($\mathsf{TXN}_i\langle\mathbb{I}\rangle\{c\}$) and their parallel composition ($c_1 \,||\, c_2$). Each transaction is assumed to have a unique identifier i, and executes at the top-level; our semantics does not support nested transactions. The $\mathbb{I}$ in the `TXN` block syntax is the transaction's isolation specification, whose purpose is explained below. Certain terms that only appear at run-time are also present in c. These include a txn block tagged with sets (δ and Δ) of records representing local

[4]Bind ($\gg\!=$) has higher precedence than union ($\cup$). Angle braces ($\langle \ldots \rangle$) are used to denote records.
[5]For now, assume that the record being added is not already present in δ.

Syntax

$$x, y \in \texttt{Variables} \quad f \in \texttt{Field Names} \quad i, j \in \mathbb{N} \quad \odot \in \{+, -, \leq, \geq, =\} \quad k \in \mathbb{Z} \cup \mathbb{B} \quad r \in \langle \bar{f} = \bar{k} \rangle$$

$$\begin{array}{llll}
\delta, \Delta, s & \in & \texttt{State} & := \mathcal{P}\left(\langle \bar{f} = \bar{k} \rangle\right) \\
\mathbb{I}_e, \mathbb{I}_c & \in & \texttt{IsolationSpec} & := (\delta, \Delta, \Delta') \rightarrow \mathbb{P} \\
v & \in & \texttt{Values} & := k \mid r \mid s \\
e & \in & \texttt{Expressions} & := v \mid x \mid x.f \mid \langle \bar{f} = \bar{e} \rangle \mid e_1 \odot e_2 \\
c & \in & \texttt{Commands} & := \mathsf{LET}\ x = e\ \mathsf{IN}\ c \mid \mathsf{IF}\ e\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2 \mid c_1; c_2 \mid \mathsf{INSERT}\ x \\
 & & & \quad \mid \mathsf{DELETE}\ \lambda x.e \mid \mathsf{LET}\ y = \mathsf{SELECT}\ \lambda x.e\ \mathsf{IN}\ c \mid \mathsf{UPDATE}\ \lambda x.e_1\ \lambda x.e_2 \\
 & & & \quad \mid \mathsf{FOREACH}\ x\ \mathsf{DO}\ \lambda y.\lambda z.c \mid \mathsf{foreach}\langle s_1 \rangle\ s_2\ \mathsf{do}\ \lambda x.\lambda y.e \\
 & & & \quad \mid \mathsf{TXN}_i\langle \mathbb{I} \rangle\{c\} \mid \mathsf{txn}_i\langle \mathbb{I}, \delta, \Delta \rangle\{c\} \mid c_1 || c_2 \mid \mathsf{SKIP} \\
\mathcal{E} & \in & \texttt{Eval Ctx} & ::= \bullet \mid \bullet || c_2 \mid c_1 || \bullet \mid \bullet; c_2 \mid \mathsf{txn}_i\langle \mathbb{I}, \delta, \Delta \rangle\{\bullet\}
\end{array}$$

Local Reduction $\boxed{\Delta \vdash ([c]_i, \delta) \longrightarrow ([c']_i, \delta')}$

E-Insert

$$\frac{r.\mathsf{id} \notin \mathsf{dom}(\delta \cup \Delta) \qquad r' = \langle r\ \mathtt{with}\ \mathsf{txn} = i;\ \mathsf{del} = \mathsf{false} \rangle}{\Delta \vdash ([\mathsf{INSERT}\ r]_i, \delta) \longrightarrow ([\mathsf{SKIP}]_i, \delta \cup \{r'\})}$$

E-Select

$$\frac{s = \{r \in \Delta \mid \mathsf{eval}([r/x]e) = \mathsf{true}\} \quad c' = [s/y]c}{\Delta \vdash ([\mathsf{LET}\ y = \mathsf{SELECT}\ \lambda x.e\ \mathsf{IN}\ c]_i, \delta) \longrightarrow ([c']_i, \delta)}$$

E-Delete

$$\frac{\mathsf{dom}(\delta) \cap \mathsf{dom}(s) = \emptyset \qquad s = \{r' \mid \exists (r \in \Delta).\ \mathsf{eval}([r/x]e) = \mathsf{true} \wedge r' = \langle r\ \mathtt{with}\ \mathsf{del} = \mathsf{true};\ \mathsf{txn} = i \rangle\}}{\Delta \vdash ([\mathsf{DELETE}\ \lambda x.e]_i, \delta) \longrightarrow ([\mathsf{SKIP}]_i, \delta \cup s)}$$

E-Update

$$\frac{\mathsf{dom}(\delta) \cap \mathsf{dom}(s) = \emptyset \qquad s = \{r' \mid \exists (r \in \Delta).\ \mathsf{eval}([r/x]e_2) = \mathsf{true} \wedge r' = \langle [r/x]e_1\ \mathtt{with}\ \mathsf{id} = r.\mathsf{id};\ \mathsf{txn} = i;\ \mathsf{del} = r.\mathsf{del} \rangle\}}{\Delta \vdash ([\mathsf{UPDATE}\ \lambda x.e_1\ \lambda x.e_2]_i, \delta) \longrightarrow ([\mathsf{SKIP}]_i, \delta \cup s)}$$

$$\begin{array}{ll}
\text{E-Foreach1} & \Delta \vdash ([\mathsf{FOREACH}\ s\ \mathsf{DO}\ \lambda y.\lambda z.c]_i, \delta) \longrightarrow ([\mathsf{foreach}\langle \emptyset \rangle\ s\ \mathsf{do}\ \lambda y.\lambda z.c]_i, \delta) \\
\text{E-Foreach2} & \Delta \vdash ([\mathsf{foreach}\langle s_1 \rangle\ \{r\} \uplus s_2\ \mathsf{do}\ \lambda y.\lambda z.c]_i, \delta) \longrightarrow ([[r/z][s_1/y]c; \\
 & \qquad\qquad \mathsf{foreach}\langle s_1 \cup \{r\} \rangle\ s_2\ \mathsf{do}\ \lambda y.\lambda z.c]_i, \delta) \\
\text{E-Foreach3} & \Delta \vdash ([\mathsf{foreach}\langle s \rangle\ \emptyset\ \mathsf{do}\ \lambda y.\lambda z.c]_i, \delta) \longrightarrow ([\mathsf{SKIP}]_i, \delta)
\end{array}$$

Top-Level Reduction $\boxed{(c, \Delta) \longrightarrow (c', \Delta')}$

E-Txn-Start

$$\frac{}{(\mathsf{TXN}_i\langle \mathbb{I} \rangle\{c\}, \Delta) \longrightarrow (\mathsf{txn}_i\langle \mathbb{I}, \emptyset, \Delta \rangle\{c\}, \Delta)}$$

E-Txn

$$\frac{\mathbb{I}_e\ (\delta, \Delta, \Delta') \quad \Delta \vdash ([c]_i, \delta) \longrightarrow ([c']_i, \delta')}{(\mathsf{txn}_i\langle \mathbb{I}, \delta, \Delta \rangle\{c\}, \Delta') \longrightarrow (\mathsf{txn}_i\langle \mathbb{I}, \delta', \Delta' \rangle\{c'\}, \Delta')}$$

E-Commit

$$\frac{\mathbb{I}_c\ (\delta, \Delta, \Delta')}{(\mathsf{txn}_i\langle \mathbb{I}, \delta, \Delta \rangle\{\mathsf{SKIP}\}, \Delta') \longrightarrow (\mathsf{SKIP}, \delta \rhd \Delta')}$$

Fig. 5. $\mathcal{T}$: Syntax and Small-step semantics and global database state, and a runtime foreach expression that keeps track of the set (s_1) of items already iterated, and the set (s_2) of items yet to be iterated. Note that the surface-level syntax of the `FOREACH` command shown here is slightly different from the one used in previous sections; its higher-order function has two arguments, y and z, which are invoked (during the reduction) with the set of already-iterated items, and the current item, respectively. This form of `FOREACH` lends itself to inductive reasoning that will be useful for verification (Sec. 4). Our language ensures that all effectful actions are encapsulated within database commands, and that all shared state among processes are only manipulated via transactions and its supported operations. In particular, we do not consider programs in which objects resident on e.g., the OCaml heap are concurrently manipulated by OCaml expressions as well as database actions.

We define a small-step operational semantics for this language in terms of an abstract machine that executes a command, and updates either a transaction-local (δ), or global (Δ) database, both of which are modeled as a set of records of a pre-defined type, i.e., they all belong to a single table. The generalization to multiple tables is straightforward, e.g., by having the machine manipulate a set of sets, one for each table. The semantics assumes that records in Δ can be uniquely identified via their `id` field, and enforces this property wherever necessary. Certain hidden fields are treated specially by the operational semantics, and are hidden from the surface language. These include a `txn` field that tracks the identifier of the transaction that last updated the record, and a `del` field that flags deleted records in δ. For a set S of records, we define $\mathsf{dom}(S)$ as the set of unique ids of all records in S. Thus $|\mathsf{dom}(\Delta)| = |\Delta|$. During its execution, a transaction may write to multiple records in Δ. Atomicity dictates that such writes should not be visible in Δ until the transaction commits. We therefore associate each transaction with a local database (δ) that stores such uncommitted records[6]. Uncommitted records include deleted records, whose `del` field is set to `true`. When the transaction commits, its local database is atomically *flushed* to the global database, committing these heretofore uncommitted records. The flush operation ($\rhd$) is defined as follows:

$$\forall r.\ r \in (\delta \rhd \Delta) \iff (r.\mathtt{id} \notin \mathsf{dom}(\delta) \ \wedge\ r \in \Delta) \ \vee\ (r \in \delta \ \wedge\ \neg r.\mathtt{del})$$

Let $\Delta' = \delta \rhd \Delta$. A record r belongs to Δ' iff it belongs to Δ and has not been updated in δ, i.e., $r.\mathtt{id} \notin \mathsf{dom}(\delta)$, or it belongs to δ, i.e., it is either a new record, or an updated version of an old record, provided the update is not a deletion ($\neg r.\mathtt{del}$). Besides the commit, flush also helps a transaction read its own writes. Intuitively, the result of a read operation inside a transaction must be computed on the database resulting from flushing the current local state (δ) to the global state (Δ). The abstract machine of Fig. 5, however, does not let a transaction read its own writes. This simplifies the semantics, without losing any generality, since substituting $\delta \rhd \Delta$ for Δ at select places in the reduction rules effectively allows reads of uncommitted transaction writes to be realized, if so desired.

The small-step semantics is stratified into a transaction-local reduction relation, and a top-level reduction relation. The transaction-local relation ($\Delta \vdash (c, \delta) \longrightarrow (c', \delta')$) defines a small-step reduction for a command inside a transaction, when the database state is Δ; the command c reduces to c', while updating the transaction-local database δ to δ'. The definition assumes a meta-function eval that evaluates closed terms to values. The reduction relation for SQL statements is defined straightforwardly. `INSERT` adds a new record to δ after checking the uniqueness of its id. `DELETE` finds the records in Δ that match the search criteria defined by its boolean function argument, and adds the records to δ after marking them for deletion. `SELECT` bounds the name introduced by `LET` to the set of records from Δ that match the search criteria, and then executes the bound command c. `UPDATE` uses its first function argument to compute the updated version of the records that match the search criteria defined by its second function argument. Updated records are added to δ.

[6]While SQL's `UPDATE` admits writes at the granularity of record fields, most popular databases enforce record-level locking, allowing us to think of "uncommitted writes" as "uncommitted records".

The reduction of `FOREACH` starts by first converting it to its run-time form to keep track of iterated items (s_1), as well as yet-to-be-iterated items (s_2). Iteration involves invoking its function argument with s_1 and the current element x (note: $\uplus$ in $\{x\} \uplus s_2$ denotes a disjoint union). The reduction ends when s_2 becomes empty. The reduction rules for conditionals, `LET` binders, and sequences are standard, and omitted for brevity.

The top-level reduction relation defines the small-step semantics of transactions, and their parallel composition. A transaction comes tagged with an *isolation specification* $\mathbb{I}$, which has two components $\mathbb{I}_e$ and $\mathbb{I}_c$, that dictate the timing and nature of interferences that the transaction can witness, during its execution ($\mathbb{I}_e$), and when it is about to commit ($\mathbb{I}_c$). Formally, $\mathbb{I}_e$ and $\mathbb{I}_c$ are predicates over the (current) transaction-local database state (δ), the state (Δ) of the global database when the transaction last took a step, and the current state (Δ') of the global database. Intuitively, $\Delta' \neq \Delta$ indicates an interference from another concurrent transaction, and the predicates $\mathbb{I}_e$ and $\mathbb{I}_c$ decide if this interference is allowed or not, taking into account the local database state (δ). For instance, as described in §2, an SI transaction on PostgreSQL defines $\mathbb{I}$ as follows:

$$\begin{array}{lcl} \mathbb{I}_e\ (\delta, \Delta, \Delta') & = & \Delta' = \Delta \\ \mathbb{I}_c\ (\delta, \Delta, \Delta') & = & \forall (r \in \delta)(r' \in \Delta).\ r'.\mathtt{id} = r.\mathtt{id} \Rightarrow r' \in \Delta' \end{array}$$

This definition dictates that no change to the global database state can be visible to an SI transaction while it executes ($\mathbb{I}_e$), and there should be no concurrent updates to records written by the transaction by other concurrently executing ones ($\mathbb{I}_c$). To simplify the presentation, we use $\mathbb{I}$ instead of $\mathbb{I}_e$ and $\mathbb{I}_c$ when its destructed form is not required.

The reduction of a $\mathtt{TXN}_i\langle\mathbb{I}\rangle\{c\}$ begins by first converting it to its run-time form $\mathsf{txn}_i\langle\mathbb{I}, \delta, \Delta\rangle\{c\}$, where $\delta = \emptyset$, and Δ is the current (global) database. Rule E-Txn reduces $\mathsf{txn}_i\langle\mathbb{I}, \delta, \Delta\rangle\{c\}$ under a database state (Δ'), only if the transaction-body isolation specification ($\mathbb{I}_e$) allows the interference between Δ and Δ'. Rule E-Commit commits the transaction $\mathsf{txn}_i\langle\mathbb{I}, \delta, \Delta\rangle\{c\}$ by flushing its uncommitted records to the database. This is done only if the interference between Δ and Δ' is allowed at the commit point by the isolation specification ($\mathbb{I}_c$). The distinction between $\mathbb{I}_e$ and $\mathbb{I}_c$ allows us to model the snapshot semantics of realistic isolation levels that isolate a transaction from interference during its execution, but expose interferences at the commit point.

Local Context Independence As mentioned previously, our operational semantics does not let a transaction read its own writes. It also does not let a transaction overwrite its own writes, due to the premise $\mathsf{dom}(\delta) \cap \mathsf{dom}(s) = \emptyset$ on the E-Delete and E-Update rules. We refer to this restriction as *local context independence*. This restriction is easy to relax in the operational semantics and the reasoning framework presented in the next section; our inference procedure described in §5, however, has a non-trivial dependence on this assumption. Nonetheless, we have encountered few instances in practice where enforcing local context independence turns out to be a severe restriction. Indeed, all of the transactions we have considered in our benchmarks (e.g., TPC-C) satisfy this assumption.

3.1 Isolation Specifications

A distinctive characteristic of our development is that it is parameterized on a weak isolation specification $\mathbb{I}$ that can be instantiated with the declarative characterization of an isolation guarantee or a concurrency control mechanism, regardless of the actual implementation used to realize it. This allows us to model a range of isolation properties that are relevant to the theory and practice of transaction processing systems without appealing to specific implementation artifacts like locks, versions, logs, speculation, etc. A few well-known properties are discussed below:

Unique Ids. As the `new_order` example (§2) demonstrates, enforcing global uniqueness of ordered identifiers requires stronger isolation levels than the ones that are default on most databases (e.g., Read Committed). Alternatively, globally unique sequence numbers, regardless of the isolation level, can be requested from a relational database via SQL's `UNIQUE` and `AUTO_INCREMENT` keywords. Our development crucially relies on the uniqueness of record identifiers[7], which are checked locally for uniqueness by the E-Insert rule. The global uniqueness of locally unique identifiers can be captured as an isolation property thus:

$$\mathbb{I}_{id}(\delta, \Delta, \Delta') \quad = \quad \forall (r \in \delta).\ r.\mathsf{id} \notin \mathsf{dom}(\Delta) \Rightarrow r.\mathsf{id} \notin \mathsf{dom}(\Delta')$$

$\mathbb{I}_{id}$ ensures that if the id of a record is globally unique when it is added to a transaction's δ, it remains globally unique until the transaction commits. This would be achieved within our semantic framework by prohibiting the interference from a concurrent transaction that adds the same id. The axiom thus simulates a global counter protected by an exclusive lock without explicitly appealing to an implementation artifact.

Write-Write Conflicts. Databases often employ a combination of concurrency control methods, both optimistic (e.g., speculation and rollback) and pessimistic (e.g., various degrees of locking), to eliminate write-write (*ww*) conflicts among concurrent transactions. We can specify the absence of such conflicts using our tri-state formulation thus:

$$\mathbb{I}_{ww}(\delta, \Delta, \Delta') \quad = \quad \forall (r' \in \delta)(r \in \Delta).\ r.\mathsf{id} = r'.\mathsf{id} \Rightarrow r \in \Delta'$$

That is, given a record $r' \in \delta$, if there exists an $r \in \Delta$ with the same id (i.e., r' is an updated version of r), then r must be present unmodified in Δ'. This prevents a concurrent transaction from changing r, thus simulating the behavior of an exclusive lock or a speculative execution that succeeded (Note: a transaction writing to r always changes r because its `txn` field is updated).

Snapshots Almost all major relational databases implement isolation levels that execute transactions against a static snapshot of the database that can be axiomatized thus:

$$\mathbb{I}_{ss}(\delta, \Delta, \Delta') \quad = \quad \Delta' = \Delta$$

Read-Only Transactions. Certain databases implement special privileges for read-only transactions. Read-only behavior can be enforced on a transaction by including the following proposition as part of its isolation invariant:

$$\mathbb{I}_{ro}(\delta, \Delta, \Delta') \quad = \quad \delta = \emptyset$$

In addition to these properties, various specific isolation levels proposed in the database or distributed systems literature, or implemented by commercial vendors can also be specified within this framework:

Read Committed (RC) and Monotonic Atomic View (MAV). RC isolation allows a transaction to witness writes of committed transactions at any point during the transaction's execution. Although it offers only weak isolation guarantees, it nonetheless prevents witnessing *dirty writes* (i.e., writes performed by uncommitted transactions). Monotonic Atomic View (MAV) [Bailis et al. 2013a] is an extension to RC that guarantees the continuous visibility of a committed transaction's writes once they become visible in the current transaction. That is, a MAV transaction does not witness *disappearing writes*, which can happen on a weakly consistent machine. Due to the SC nature of our abstract machine (there is always a single global database state Δ; not a vector of states indexed by vector clocks), and our choice to never violate atomicity of a transaction's writes, both RC and MAV are already guaranteed by our semantics. Thus, defining $\mathbb{I}_e$ and $\mathbb{I}_c$ to *true* ensures RC and MAV behavior under our semantics.

Repeatable Read (RR) By definition, multiple reads to a transactional variable in a Repeatable Read transaction are required to return the same value. RR is often implemented (for e.g., in [Bailis et al. 2013a; MySQL 2016]) by executing the transaction against a (conceptual) snapshot of the database, but committing its writes to the actual database. This implementation of RR can be

[7]The importance of unique ids is recognized in real-world implementations. For example, MySQL's InnoDB engine automatically adds a 6-byte unique identifier if none exists for a record.

axiomatized as $\mathbb{I}_e = \mathbb{I}_{ss}$ and $\mathbb{I}_c = \mathit{true}$. However, this specification of RR is stronger than the ANSI SQL specification, which requires no more than the invariance of already read records. In particular, ANSI SQL RR allows *phantom reads*, a phenomenon in which a repeated `SELECT` query might return newly inserted records that were not previously returned. This specification is implemented, for e.g., in Microsoft's SQL server, using record-level exclusive read locks, that prevent a record from being modified while it is read by an uncommitted transaction, but which does not prohibit insertion of new records. The ANSI SQL RR specification can be axiomatized in our framework, but it requires a minor extension to our operational semantics to track a transaction's reads. In particular, the records returned by `SELECT` should be added to the local database δ, but without changing their transaction identifiers (`txn` fields), and flush ($\triangleright$) should only flush the records that bear the current transaction's identifier. With this extension, ANSI SQL RR can be axiomatized thus:

$$\begin{array}{lcl} \mathbb{I}_e(\delta, \Delta, \Delta') & \Leftrightarrow & \forall (r \in \delta). r \in \Delta \Rightarrow r \in \Delta' \\ \mathbb{I}_c(\delta, \Delta, \Delta') & \Leftrightarrow & \mathit{true} \end{array}$$

If a record r belongs to both δ and Δ, then it must be a record written by a different transaction and read by the current transaction (since the current transaction's records are not yet present in Δ). By requiring $r \in \Delta'$, $\mathbb{I}_e$ guarantees the invariance of r, thus the repeatability of the read.

Snapshot Isolation (SI) The concept of executing a transaction against a consistent snapshot of the database was first proposed as Snapshot Isolation in [Berenson et al. 1995]. SI doesn't admit write-write conflicts, and the original proposal, which is implemented in Microsoft SQL Server, required the database to roll-back an SI transaction if conflicts are detected during the commit. This behavior can be axiomatized as $\mathbb{I}_e = \mathbb{I}_{ss}$ (execution against a snapshot), and $\mathbb{I}_c = \mathbb{I}_{ww}$ (avoiding write-write conflicts during the commit). Note that the same axiomatization applies to PostgreSQL's RR, although its implementation (described in Sec. 2) differs considerably from the original proposal. Thus, reasoning done for an SI transaction on MS SQL server carries over to PostgreSQL's RR and vice-versa, demonstrating the benefits of reasoning axiomatically about isolation properties.

Serializability (SER) The specification of serializability is straightforward:

$$\begin{array}{lcl} \mathbb{I}_e\ (\delta, \Delta, \Delta') & = & \Delta' = \Delta \\ \mathbb{I}_c\ (\delta, \Delta, \Delta') & = & \Delta' = \Delta \end{array}$$

4 THE REASONING FRAMEWORK

We now describe a proof system that lets us prove the correctness of a $\mathcal{T}$ program c w.r.t its high-level consistency conditions I, on an implementation that satisfies the isolation specifications ($\mathbb{I}$) of its transactions[8]. Our proof system is essentially an adaptation of a rely-guarantee reasoning framework [Jones 1983] to the setting of weakly isolated database transactions. The primary challenge in the formulation deals with how we relate a transaction's isolation specification ($\mathbb{I}$) to its rely relation (R) that describes the transaction's environment, so that interference is considered only insofar as allowed by the isolation level. Another characteristic of the transaction setting that affects the structure of the proof system is atomicity; we do not permit a transaction's writes to be visible until it commits. In the context of rely-guarantee, this means that the transaction's guarantee (G) should capture the aggregate effect of a transaction, and not its individual writes. While shared memory `atomic` blocks also have the same characteristic, the fact that transactions are weakly-isolated introduces non-trivial complexity. Unlike an `atomic` block, the effect of a transaction is *not* a sequential composition of the effects of its statements because each statement can witness a potentially different version of the state.

$\boxed{\mathbb{R} \vdash \{P\}\ [c]_i\ \{Q\}}$ $\boxed{\{I, R\}\ c\ \{G, I\}}$

RG-Select

$$\frac{P(\delta,\Delta)\ \wedge\ x = \{r \mid r \in \Delta \wedge [r/y]e\} \Rightarrow P'(\delta,\Delta) \quad \mathbb{R} \vdash \{P'\}\ [c]_i\ \{Q\} \quad \mathtt{stable}(\mathbb{R}, P')}{\mathbb{R} \vdash \{P\}\ [\mathsf{LET}\ x = \mathsf{SELECT}\ \lambda y.e\ \mathsf{IN}\ c]_i\ \{Q\}}$$

RG-Insert

$$\frac{\begin{array}{c}\mathtt{stable}(\mathbb{R}, P)\\ \forall \delta,\delta',\Delta,i.\ P(\delta,\Delta)\ \wedge\ j \notin \mathrm{dom}(\delta\cup\Delta)\ \wedge\ \delta' = \delta \cup \{\langle x\ \mathtt{with}\ \mathtt{id} = j;\ \mathtt{txn} = i;\ \mathtt{del} = \mathtt{false}\rangle\} \Rightarrow Q(\delta',\Delta)\end{array}}{\mathbb{R} \vdash \{P\}\ [\mathsf{INSERT}\ x]_i\ \{Q\}}$$

RG-Update

$$\frac{\begin{array}{c}\mathtt{stable}(\mathbb{R}, P) \quad \forall \delta,\delta',\Delta.\ P(\delta,\Delta)\ \wedge\ \delta' = \delta \cup \{r' \mid \exists (r \in \Delta).[r/x]e_2\ \wedge\\ r' = \langle [r/x]e_1\ \mathtt{with}\ \mathtt{id} = r.\mathtt{id};\ \mathtt{txn} = i;\ \mathtt{del} = \mathtt{false}\rangle\} \Rightarrow Q(\delta',\Delta)\end{array}}{\mathbb{R} \vdash \{P\}\ [\mathsf{UPDATE}\ \lambda x.e_1\ \lambda x.e_2]_i\ \{Q\}}$$

RG-Delete

$$\frac{\begin{array}{c}\mathtt{stable}(\mathbb{R}, P)\\ \forall \delta,\delta',\Delta.\ P(\delta,\Delta)\ \wedge\ \delta' = \delta \cup \{r' \mid \exists (r \in \Delta).\ [r/x]e\ \wedge\ r' = \langle r\ \mathtt{with}\ \mathtt{txn} = i; \mathtt{del} = \mathtt{true}\rangle\} \Rightarrow Q(\delta',\Delta)\end{array}}{\mathbb{R} \vdash \{P\}\ [\mathsf{DELETE}\ \lambda x.e]_i\ \{Q\}}$$

RG-Foreach

$$\frac{\begin{array}{c}\mathtt{stable}(\mathbb{R}, Q) \quad \mathtt{stable}(\mathbb{R}, \psi) \quad \mathtt{stable}(\mathbb{R}, P)\\ P \Rightarrow [\emptyset/y]\psi \quad \mathbb{R} \vdash \{\psi \wedge z \in x\}\ [c]_i\ \{Q_c\}\\ Q_c \Rightarrow [y \cup \{z\}/y]\psi \quad [x/y]\psi \Rightarrow Q\end{array}}{\mathbb{R} \vdash \{P\}\ [\mathsf{FOREACH}\ x\ \mathsf{DO}\ \lambda y.\lambda z.c]_i\ \{Q\}}$$

RG-Conseq

$$\frac{\begin{array}{c}\{I, R\}\ \mathsf{TXN}_i\langle \mathbb{I}\rangle\{c\}\ \{G, I\}\\ \mathbb{I}' \Rightarrow \mathbb{I} \quad R' \subseteq R \quad G \subseteq G'\\ \mathtt{stable}(R', \mathbb{I}') \quad \forall \Delta,\Delta'.\ I(\Delta) \wedge G'(\Delta,\Delta') \Rightarrow I(\Delta')\end{array}}{\{I, R'\}\ \mathsf{TXN}_i\langle \mathbb{I}'\rangle\{c\}\ \{G', I\}}$$

RG-Txn

$$\frac{\begin{array}{c}\mathtt{stable}(R, \mathbb{I}) \quad \mathtt{stable}(R, I) \quad \mathbb{R}_e = R\backslash\mathbb{I}_e \quad \mathbb{R}_c = R\backslash\mathbb{I}_c \quad P(\delta,\Delta) \Leftrightarrow \delta = \emptyset \wedge I(\Delta)\\ \mathbb{R}_e \vdash \{P\}\ c\ \{Q\} \quad \mathtt{stable}(\mathbb{R}_c, Q) \quad \forall \delta,\Delta.\ Q(\delta,\Delta) \Rightarrow G(\Delta, \delta \rhd \Delta) \quad \forall \Delta,\Delta'.\ I(\Delta) \wedge G(\Delta,\Delta') \Rightarrow I(\Delta')\end{array}}{\{I, R\}\ \mathsf{TXN}_i\langle \mathbb{I}\rangle\{c\}\ \{G, I\}}$$

Fig. 6. $\mathcal{T}$: Rely-Guarantee rules

4.1 The Rely-Guarantee Judgment

Fig. 6 shows an illustrative subset of the rely-guarantee (RG) reasoning rules for $\mathcal{T}$. We define two RG judgments: top-level ($\{I, R\}\ c\ \{G, I\}$), and transaction-local ($\mathbb{R} \vdash \{P\}\ [c]_i\ \{Q\}$). Recall that the standard RG judgment is the quintuple $\{P, R\}\ c\ \{G, Q\}$. Instead of separate P and Q assertions, our top-level judgment uses I as both a pre- and post-condition, because our focus is on verifying that a $\mathcal{T}$ program *preserves* a databases' consistency conditions[9]. A transaction-local RG judgment does not include a guarantee relation because transaction-local effects are not visible outside a transaction. Also, the rely relation ($\mathbb{R}$) of the transaction-local judgment is not the same as the

[8]Note the difference between I and $\mathbb{I}$. The former constitute *proof obligations* for the programmer, whereas the latter describes a transaction's *assumptions* about the operational characteristics of the underlying system.
[9]The terms *consistency condition*, *high-level invariant*, and *integrity constraint* are used interchangeably throughout the paper.

top-level rely relation (R) because it must take into account the transaction's isolation specification ($\mathbb{I}$). Intuitively, $\mathbb{R}$ is R modulo $\mathbb{I}$. Recall that a transaction writes to its local database (δ), which is then flushed when the transaction commits. Thus, the guarantee of a transaction depends on the state of its local database at the commit point. The pre- and post-condition assertions (P and Q) in the local judgment facilitate tracking the changes to the transaction-local state, which eventually helps us prove the validity of the transaction's guarantee. Both P and Q are bi-state assertions; they relate transaction-local database state (δ) to the global database state (Δ). Thus, the transaction-local judgment effectively tracks how transaction-local and global states change in relation to each other.

4.1.1 Stability. A central feature of a rely-guarantee judgment is a stability condition that requires the validity of an assertion ϕ to be unaffected by interference from other concurrently executing transactions, i.e., the rely relation R. In conventional RG, stability is defined as follows, where σ and σ' denote states:

$$\texttt{stable}(R,\phi) \quad\Leftrightarrow\quad \forall\sigma,\sigma'.\ \phi(\sigma)\ \wedge\ R(\sigma,\sigma') \Rightarrow \phi(\sigma')$$

Due to the presence of local and global database states, and the availability of an isolation specification, we use multiple definitions of stability in Fig. 6, but they all convey the same intuition as above. In our setting, we only need to prove the stability of an assertion (ϕ) against those environment steps which lead to a global database state on which the transaction itself can take its next step according to its isolation specification ($\mathbb{I}$).

$$\texttt{stable}(R,\phi) \quad\Leftrightarrow\quad \forall\delta,\Delta,\Delta'.\phi(\delta,\Delta) \wedge R^*(\Delta,\Delta') \wedge \mathbb{I}(\delta,\Delta,\Delta') \Rightarrow \phi(\delta,\Delta')$$

A characteristic of RG reasoning is that stability of an assertion is always proven w.r.t to R, and not R^*, although interference may include multiple environment steps, and R only captures a single step. This is nonetheless sound due to inductive reasoning: if ϕ is preserved by every step of R, then ϕ is preserved by R^*, and vice-versa. However, such reasoning does not extend naturally to isolation-constrained interference because R^* modulo $\mathbb{I}$ is not same as $\mathbb{R}^*$; the former is a transitive relation constrained by $\mathbb{I}$, whereas the latter is the transitive closure of a relation constrained by $\mathbb{I}$. This means, unfortunately, that we cannot directly replace R^* by R in the above condition.

To obtain an equivalent form in our setting, we require an additional condition on the isolation specification, which we call the *stability condition on* $\mathbb{I}$. The condition requires $\mathbb{I}$ to admit the interference of multiple R steps (i.e., $R^*(\Delta,\Delta'')$, for two database states Δ and Δ''), only if it also admits interference of each R step along the way. Formally:

$$\texttt{stable}(R,\mathbb{I}) \quad\Leftrightarrow\quad \forall\delta,\Delta,\Delta',\Delta''.\ \mathbb{I}(\delta,\Delta,\Delta'')\ \wedge\ R(\Delta',\Delta'') \Rightarrow \mathbb{I}(\delta,\Delta,\Delta')\ \wedge\ \mathbb{I}(\delta,\Delta',\Delta'')$$

It can be easily verified that the above stability condition is satisfied by the isolation axioms from Sec. 3.1. For instance, $\mathbb{I}_{ss}$, the snapshot axiom, is stable because if a the state is unmodified between Δ and Δ'', then it is clearly unmodified between Δ and Δ', and also between Δ' and Δ'', where Δ' is an intermediary state. Modifying and restoring the state Δ is not possible because each new commit bears a new transaction id different from the transaction ids (`txn` fields) present in Δ.

The stability condition on $\mathbb{I}$ guarantees that an interference from R^* is admissible only if the interference due to each individual R step is admissible. In other words, it makes isolation-constrained R^* relation equal to the transitive closure of the isolation-constrained R relation. We call R constrained by isolation $\mathbb{I}$ as R modulo $\mathbb{I}$ ($R\backslash\mathbb{I}$; written equivalently as $\mathbb{R}$), which is the following ternary relation:

$$(R\backslash\mathbb{I})(\delta,\Delta,\Delta') \quad\Leftrightarrow\quad R(\Delta,\Delta') \wedge \mathbb{I}(\delta,\Delta,\Delta')$$

It is now enough to prove the stability of an RG assertion ϕ w.r.t $R\backslash\mathbb{I}$:

$$\texttt{stable}((R\backslash\mathbb{I}),\phi) \quad\Leftrightarrow\quad \forall\delta,\Delta,\Delta'.\ \phi(\delta,\Delta) \wedge (R\backslash\mathbb{I})(\delta,\Delta,\Delta') \Rightarrow \phi(\delta,\Delta')$$

This condition often significantly simplifies the form of $R\backslash\mathbb{I}$ irrespective of R. For example, when a transaction is executed against a snapshot of the database (i.e. $\mathbb{I}_{ss}$), $R\backslash\mathbb{I}_{ss}$ will be the identity function, since any non-trivial interference will violate the $\Delta' = \Delta$ condition imposed by $\mathbb{I}_{ss}$.

4.1.2 Rules. RG-TXN is the top-level rule that lets us prove a transaction preserves the high-level invariant I when executed under the required isolation as specified by $\mathbb{I}$. It depends on a transaction-local judgment to verify the body (c) of a transaction with id i. The precondition P of c must follow from the fact that the transaction-local database (δ) is initially empty, and the global database satisfies the high-level invariant I. The rely relation ($\mathbb{R}_e$) is obtained from the global rely relation R and the isolation specification $\mathbb{I}_e$ as explained above. Recall that $\mathbb{I}_e$ constrains the global effects visible to the transaction while it is executing but has not yet committed, and P and Q of the transaction-local RG judgment are binary assertions; they relate local and global database states. The local judgment rules require one or both of them to be stable with respect to the constrained rely relation $\mathbb{R}_e$.

For the guarantee G of a transaction to be valid, it must follow from the post-condition Q of the body, provided that Q is stable w.r.t the commit-time interference captured by $\mathbb{R}_c$. $\mathbb{R}_c$, like $\mathbb{R}_e$, is computed as a rely relation modulo isolation, except that commit-time isolation ($\mathbb{I}_c$) is considered. The validity of G is captured by the following implication:

$$\forall \delta, \Delta.\ Q(\delta, \Delta) \Rightarrow G(\Delta, \delta \triangleright \Delta)$$

In other words, if Q relates the transaction-local database state (δ) to the state of the global database (Δ) before a transaction commits, then G must relate the states of the global database before and after the commit. The act of commit is captured by the flush action ($\delta \triangleright \Delta$). Once we establish the validity of G as a faithful representative of the transaction, we can verify that the transaction preserves the high-level invariant I by checking the stability of I w.r.t G, i.e., $\forall \Delta, \Delta'.\ I(\Delta) \wedge G(\Delta, \Delta') \Rightarrow I(\Delta')$.

The RG-CONSEQ rule lets us safely weaken the guarantee G, and strengthen the rely R of a transaction. Importantly, it also allows its isolation specification $\mathbb{I}$ to be strengthened (both $\mathbb{I}_e$ and $\mathbb{I}_c$). This means that a transaction proven correct under a weaker isolation level is also correct under a stronger level. Parametricity over the isolation specification $\mathbb{I}$, combined with the ability to strengthen $\mathbb{I}$ as needed, admits a flexible proof strategy to prove database programs correct. For example, programmers can declare isolation requirements of their choice through $\mathbb{I}$, and then prove programs correct assuming the guarantees hold. The soundness of strengthening $\mathbb{I}$ ensures that a program can be safely executed on any system that offers isolation guarantees at least as strong as those assumed.

Salient rules of transaction-local RG judgments are shown in Fig. 6. These rules (RG-UPDATE, RG-SELECT, RG-DELETE, and RG-INSERT) reflect the structure of the corresponding reduction rule from Fig. 5. The rule RG-FOREACH defines the RG judgment for a `FOREACH` loop. As is characteristic of loops, the reasoning is pivoted on a loop invariant ψ that needs to be stable w.r.t $\mathbb{R}$. ψ must be implied by P, the pre-condition of `FOREACH`, when no elements have been iterated, i.e, when $y = \emptyset$. The body of the loop can assume the loop invariant, and the fact that z is an element from the set x being iterated, to prove its post-condition Q_c. The operational semantics ensures that z is added to y at the end of the iteration, hence Q_c must imply $[y \cup \{z\}/y]\psi$. When the loop has finished execution, y, the set of iterated items, is the entire set x. Thus $[x/y]\psi$ is true at the end of the loop, from which the post-condition Q must follow. As with the other rules, Q needs to be stable. The rules for conditionals, sequencing etc., are standard, and hence elided.

4.2 Semantics and Soundness

We now formalize the semantics of the RG judgments defined in Fig. 6, and state their soundness guarantees.

*Definition 4.1 (**Interleaved step and multi-step relations**).* Interleaved step relations interleave global and transaction-local reductions with interference as captured by the corresponding rely relations. They are defined thus:

$$\begin{array}{lcll} (c, \Delta) \longrightarrow_R (c', \Delta') & \overset{def}{=} & (c, \Delta) \longrightarrow (c', \Delta') \lor (c' = c \land R(\Delta, \Delta')) & \texttt{global} \\ ([c]_i, \delta, \Delta) \longrightarrow_R ([c']_i, \delta', \Delta') & \overset{def}{=} & \Delta \vdash ([c]_i, \delta) \longrightarrow ([c']_i, \delta') \land \Delta' = \Delta & \texttt{transaction-local} \\ & & \lor (c' = c \land \delta' = \delta \land \mathbb{R}(\delta, \Delta, \Delta')) & \end{array}$$

An interleaved multi-step relation ($\longrightarrow_R^*$) is the reflexive transitive closure of the interleaved step relation.

*Definition 4.2 (**Semantics of RG judgments**).* The semantics of the global and transaction-local RG judgments are defined thus:

$$\begin{array}{lcl} \mathbb{R} \vdash \{P\}\ [c]_i\ \{Q\} & \overset{def}{=} & \forall \delta, \delta', \Delta, \Delta'.\ P(\delta, \Delta) \land ([c]_i, \delta, \Delta) \longrightarrow_R^* ([\mathsf{SKIP}]_i, \delta', \Delta') \Rightarrow Q(\delta', \Delta') \\ \{I, R\}\ c\ \{G, I\} & \overset{def}{=} & \forall \Delta.\ I(\Delta) \Rightarrow (\forall \Delta'.\ (c, \Delta) \longrightarrow_R^* (\mathsf{SKIP}, \Delta') \Rightarrow I(\Delta')) \\ & & \quad \land\ \texttt{txn-guaranteed}(R, G, c, \Delta) \end{array}$$

The `txn-guaranteed` predicate used in the semantics of the global RG judgment is defined below:

$$\texttt{txn-guaranteed}(R, G, c, \Delta) \overset{def}{=} \forall c', c''\Delta', \Delta''.(c, \Delta) \longrightarrow_R^* (c', \Delta') \land (c', \Delta') \longrightarrow (c'', \Delta'') \Rightarrow G(\Delta', \Delta'')$$

Thus, if $\{I, R\}\ c\ \{G, I\}$ is a valid RG judgment, then (a) every interleaved multi-step reduction of c preserves the database integrity constraint (consistency condition) I, and (b) the effect that every transaction in c has on the database state is captured by G. We can now assert the soundness of the RG judgments in Fig. 6 as follows[10]:

Theorem 4.3 (**Soundness**). *The rely-guarantee judgments defined by the rules in Fig. 6 are sound with respect to the semantics of Definition 4.2.*

Proof Sketch. The most important rule is the top-level rule RG-Txn, which proves that a transaction c which begins its execution in global database state satisfying I and encountering interference R while executing under isolation specification $\mathbb{I}$ finishes its execution in a database state also satisfying I, and also guarantees that its commit step satisfies G. The rule uses the transaction-local RG judgment $\mathbb{R}_e \vdash \{P\}\ c\ \{Q\}$. By E-Txn-Start, the local and global database states at the start of a transaction satisfy P, and the only challenge is that environment steps in an execution covered by $\mathbb{R}_e \vdash \{P\}\ c\ \{Q\}$ are in $\mathbb{R}_e$, while the top-level judgment requires environment steps in R. We show that it is enough to consider only those environment steps in $\mathbb{R}_e$. First, we use an inductive argument and stability of $\mathbb{I}_e$ ($\texttt{stable}(R, \mathbb{I}_e)$) to show that any execution in which the transaction completes all its steps must always preserve the isolation specification $\mathbb{I}_e$ after every environment step. Intuitively, this is because once $\mathbb{I}_e$ gets broken after some environment step, it will continue to remain broken and the transaction would not be able to proceed (according to E-Txn). Since $\mathbb{R}_e$ contains exactly those environment steps which preserve $\mathbb{I}_e$, the local-level RG judgment can be soundly used, which guarantees that after the transaction finishes its execution, its local state δ and global state Δ will satisfy the assertion Q. Environment steps between the last step of the transaction and its commit step can modify the global state, and hence we also require Q to be stable against R. Again, we use an inductive argument, the stability of $\mathbb{I}_c$, and the fact that the transaction must execute its commit step to show that all environment steps must preserve $\mathbb{I}_c$, and hence it is enough to require $\texttt{stable}(\mathbb{R}_c, Q)$. Q guarantees that the commit step is in G, and G in turn guarantees that after execution, the global database state will obey the invariant I.

[10]Full proofs for the major theorems and lemmas defined in this paper are available from [Kaki et al. 2018].

$$x, y, \delta, \Delta \in \texttt{Variables} \quad \varphi \in \mathbb{P}^0 \quad \phi \in \mathbb{P}^1$$
$$s \;\; := \;\; x \mid \delta \mid \Delta \mid \{x \,|\, \varphi\} \mid \mathsf{exists}(\Delta, \phi, s) \mid s_1 \gg\!= \lambda x.s_2 \mid \mathsf{if}\ \varphi\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \mid s_1 \cup s_2$$

Fig. 7. Syntax of the set language $\mathcal{S}$

5 INFERENCE

The rely-guarantee framework presented in the previous section facilitates modular proofs for weakly-isolated transactions, but imposes a non-trivial annotation burden. In particular, it requires each statement (c) of the transaction to be annotated with a stable pre- (P) and post-condition (Q), and loops to be annotated with stable inductive invariants (ψ). While weakest pre-condition style predicate transformers can help in inferring intermediate assertions for regular statements, loop invariant inference remains challenging, even for the simple form of loops considered here. As an alternative, we present an inference algorithm based on state transformers that alleviates this burden. The idea is to infer the logical effect that each statement has on the transaction-local database state δ (i.e., how it transforms δ), and compose multiple such effects together to describe the effect of the transaction as a whole. Importantly, this approach generalizes to loops, where the effect of a loop can be inferred as a well-defined function of the effect of its body, thanks to certain pleasant properties enjoyed by the database programs modeled by our core language. Interpreting database semantics as functional transformations on sets (described in terms of their logical effects) enables an inference mechanism that can leverage off-the-shelf SMT solvers for automated verification.

At the core of our approach is a simple language ($\mathcal{S}$) to express set transformations (see Fig. 7). The language admits set expressions that include variables (x), literals of the form $\{x \,|\, \varphi\}$ where φ is a propositional (quantifier-free) formula on x, a restricted form of existential quantification that binds a set Δ satisfying proposition ϕ in a set expression s, a monadic composition of two set expressions (s_1 and s_2) composed using a bind ($\gg\!=$) operation, a conditional set expression where the condition is a propositional formula, and a union of two set expressions. Symbols δ and Δ are also variables in $\mathcal{S}$, but are used to denote local and database states (also represented as sets), respectively. Constant sets can be written using set literal expressions. For example, the set $\{1, 2\}$ can be written as $\{x \,|\, x = 1 \vee x = 2\}$. The language is carefully chosen to be expressive enough to capture the semantics of $\mathcal{T}$ statements (as well as SQL operations more generally), yet simple enough to have a semantics-preserving translation amenable for automated verification.

Fig. 8 shows the syntax-directed state transformer inference rules for $\mathcal{T}$ commands inside a transaction TXN_i. The rules compute, for each command c, a (meta) function F that returns a set of records as an expression in $\mathcal{S}$, given a global database Δ. Intuitively, $\mathsf{F}(\Delta)$ abstracts the set of records added to the local database δ as a result of executing c under Δ (i.e., $\Delta \vdash ([c]_i, \delta) \longrightarrow_R^* ([\mathsf{SKIP}]_i, \delta \cup \mathsf{F}(\Delta)))$[11]. Note that the function F we call state transformer here is actually the *effect* part of the state transformer introduced in Sec. 2, which is a function T of form $\lambda(\delta, \Delta).\ \delta \cup \mathsf{F}(\Delta)$. Nonetheless, for simplicity, we will continue to refer to F as state transformer. Since the execution is subject to isolation-constrained interference, the inference judgment depends on the isolation-constrained rely relation $\mathbb{R}$, which is used to enforce the stability of the state transformer F. Recall that $\mathbb{R}$ is a tri-state rely relation over δ, Δ and Δ', that admits an interference from Δ and Δ' depending on the local database state δ. Thus, the stability of the state transformer F of c with respect to $\mathbb{R}$ needs to take into account the (possible) prior state of the local database δ, which depends on the context (sequence of previous commands) of c, and computed by the corresponding

[11]Recall that the operational semantics treats deletion of records as the addition of the deleted record with its `del` field set to true in the local store.

$$\boxed{\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}}$$

$$\frac{}{\mathsf{F}_{\mathsf{ctxt}} \vdash \mathtt{INSERT}\ x \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \ \llbracket \mathsf{F}_{\mathsf{ctxt}}[\lambda(\Delta).\ \{r \mid r = \{\langle x\ \mathtt{with}\ \mathtt{del} = \mathtt{false};\ \mathtt{txn} = i\rangle\}\}]\rrbracket_{\langle\mathbb{R},I\rangle}}$$

$$\frac{G = \lambda r.\ \mathsf{if}\ [r/x]e_2\ \mathsf{then}\ \{r' \mid r' = \langle [r/x]e_1\ \mathtt{with}\ \mathtt{id} = r.\mathtt{id};\ \mathtt{del} = r.\mathtt{del};\ \mathtt{txn} = i\rangle\}\ \mathsf{else}\ \emptyset}{\mathsf{F}_{\mathsf{ctxt}} \vdash \mathtt{UPDATE}\ \lambda x.e_1\ \lambda x.e_2 \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \ \llbracket \mathsf{F}_{\mathsf{ctxt}}[\lambda(\Delta).\ \Delta \gg= G]\rrbracket_{\langle\mathbb{R},I\rangle}}$$

$$\frac{G = \lambda r.\ \mathsf{if}\ [r/x]e\ \mathsf{then}\ \{r' \mid r' = \langle r\ \mathtt{with}\ \mathtt{del} = \mathtt{true};\ \mathtt{txn} = i\rangle\}\ \mathsf{else}\ \emptyset}{\mathsf{F}_{\mathsf{ctxt}} \vdash \mathtt{DELETE}\ \lambda x.e \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \ \llbracket \mathsf{F}_{\mathsf{ctxt}}[\lambda(\Delta).\ \Delta \gg= G]\rrbracket_{\langle\mathbb{R},I\rangle}}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}}{\mathsf{F}_{\mathsf{ctxt}} \vdash \mathtt{LET}\ x = e\ \mathtt{IN}\ c \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \ \lambda(\Delta).\ [e/x]\,\mathsf{F}(\Delta)}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F} \qquad G = \lambda r.\ \mathsf{if}\ [r/x]e\ \mathsf{then}\ \{r' \mid r' = r\}\ \mathsf{else}\ \emptyset \quad \mathsf{F}' = \llbracket \mathsf{F}_{\mathsf{ctxt}}[\lambda(\Delta).\ \Delta \gg= G]\rrbracket_{\langle\mathbb{R},I\rangle}}{\mathsf{F}_{\mathsf{ctxt}} \vdash \mathtt{LET}\ y = \mathtt{SELECT}\ \lambda x.e\ \mathtt{IN}\ c \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \ \lambda(\Delta).\ [\mathsf{F}'(\Delta)/y]\,\mathsf{F}(\Delta)}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c_1 \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}_1 \quad \mathsf{F}_{\mathsf{ctxt}} \vdash c_2 \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}_2}{\mathsf{F}_{\mathsf{ctxt}} \vdash \mathtt{IF}\ e\ \mathtt{THEN}\ c_1\ \mathtt{ELSE}\ c_2 \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \ \lambda(\Delta).\ \mathsf{if}\ e\ \mathsf{then}\ \mathsf{F}_1(\Delta)\ \mathsf{else}\ \mathsf{F}_2(\Delta)}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c_1 \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}_1 \quad \mathsf{F}_{\mathsf{ctxt}} \cup \mathsf{F}_1 \vdash c_2 \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}_2}{\mathsf{F}_{\mathsf{ctxt}} \vdash c_1; c_2 \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \ \mathsf{F}_1 \cup \mathsf{F}_2}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}}{\mathsf{F}_{\mathsf{ctxt}} \vdash \mathtt{FOREACH}\ x\ \mathtt{DO}\ \lambda y.\lambda z.\ c \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \ \lambda(\Delta).\ x \gg= (\lambda z.\ \mathsf{F}(\Delta))}$$

Fig. 8. $\mathcal{T}$: State transformer semantics.

state transformer $\mathsf{F}_{\mathsf{ctxt}}$. Thus, the semantics of the state transformer can be understood in terms of the RG judgment as following (formalized as Theorem 5.1 in Sec. 5.1):

$$\mathbb{R} \vdash \{\lambda(\delta,\Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta)\}\ [c]_i\ \{\lambda(\delta,\Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)\}$$

In the above RG judgment, let P denote the pre-condition $\lambda(\delta,\Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta)$, and let Q denote the post-condition $\lambda(\delta,\Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)$. The stability condition on the state transformer F can be derived from the stability condition on Q. Observe that for Q to be stable, $\mathsf{F}_{\mathsf{ctxt}}(\Delta') \cup \mathsf{F}(\Delta')$ must be equal to $\mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)$, where Δ and Δ' are related by R (ignore $\mathbb{I}$ for the moment). Assuming that P is stable, $\mathsf{F}_{\mathsf{ctxt}}(\Delta') = \mathsf{F}_{\mathsf{ctxt}}(\Delta)$ is already given, leaving $\mathsf{F}(\Delta') = \mathsf{F}(\Delta)$ to be enforced. Thus, the stability of F in in the context of $\mathsf{F}_{\mathsf{ctxt}}$ (written $\mathsf{F}_{\mathsf{ctxt}}[\mathsf{F}]$) is defined as following:

$$\mathtt{stable}(\mathbb{R}, \mathsf{F}_{\mathsf{ctxt}}[\mathsf{F}]) \quad \Leftrightarrow \quad \forall \Delta, \Delta', \overline{v}.\ \mathbb{R}(\mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta), \Delta, \Delta') \Rightarrow \mathsf{F}(\Delta) = \mathsf{F}(\Delta')$$

where $\overline{v}$ are the variables that occur free in F; this is possible because of how the inference rules are structured. The equality in $\mathcal{S}$ translates to equivalence in first-order logic, as we describe later. In the inference rules, stability is enforced constructively by a meta-function $\llbracket\cdot\rrbracket_{\langle\mathbb{R},I\rangle}$, which accepts a transformer F (in its context $\mathsf{F}_{\mathsf{ctxt}}$) and returns a new transformer that is guaranteed to be stable under $\mathbb{R}$. $\llbracket\cdot\rrbracket_{\langle\mathbb{R},I\rangle}$ achieves the stability guarantee by abstracting away the bound global state (Δ) in an unstable F to an existentially bound Δ' as described below:

$$\begin{array}{lcll} \llbracket \mathsf{F}_{\mathsf{ctxt}}[\mathsf{F}]\rrbracket_{\langle\mathbb{R},I\rangle} & = & \mathsf{F} & \mathtt{if\ stable}(\mathbb{R}, \mathsf{F}_{\mathsf{ctxt}}[\mathsf{F}]). \\ & = & \lambda(\Delta).\ \mathsf{exists}(\Delta', I(\Delta'), \mathsf{F}(\Delta')) & \mathtt{otherwise.\ } \Delta'\ \mathtt{is\ a\ fresh\ name.} \end{array}$$

Observe that when F is not stable, $\llbracket \mathsf{F} \rrbracket_{\langle \mathbb{R}, I \rangle}$ returns a transformer F′ that simply ignores its Δ argument in favor of a generic Δ′, making F′ trivially stable. It is safe to assume $I(\Delta')$ because all verified transactions must preserve the invariant, and hence only valid database states will ever be witnessed. From the perspective of RG reasoning, $\llbracket \cdot \rrbracket_{\langle \mathbb{R}, I \rangle}$ effectively weakens the post-condition of a statement, as done by the RG-ConSeq rule for transaction-bound commands. The weakening semantics chosen by $\llbracket \cdot \rrbracket_{\langle \mathbb{R}, I \rangle}$, while being simple, is nonetheless useful because of the $I(\Delta')$ assumption imposed on an existentially bound Δ′. The example in Fig. 9 demonstrates. Here,

```
let add_interest acc_id pc = atomically_do @@ fun () ->
  let a = SQL.select1 BankAccount (fun acc -> acc.id = acc_id) in
  let y = a.bal + pc*a.bal in
  SQL.update BankAccount (fun acc -> {acc with bal = acc.bal + y})
                         (fun acc -> acc.id = acc_id)
```

Fig. 9. A transaction that deposits an interest to a bank account.

an add_interest transaction adds a positive interest (determined by pc) to the balance of a bank account, which is required to be non-negative ($I(\Delta) \Leftrightarrow \forall(r \in \Delta).\ r.\mathsf{bal} \geq 0$). The transaction starts by issuing a select1 query, whose transformer F is essentially a singleton set containing a record r whose id is acc_id (i.e., $\mathsf{F}(\Delta) = \{r \mid r \in \Delta \wedge r.\mathsf{id} = \mathsf{acc\_id}\}$). However, F is unstable because $\mathsf{F}(\Delta')$ may not be the same set as $\mathsf{F}(\Delta)$ when $\Delta' \neq \Delta$. A record $r \in \Delta$ whose id = acc_id may have its balance updated by a concurrent withdraw or deposit transaction in Δ′, making the record in Δ′ different from the record in Δ. Hence the stability check fails. Fortunately, the weakening operator ($\llbracket \cdot \rrbracket_{\langle \mathbb{R}, I \rangle}$) allows us to weaken the effect to $\mathsf{exists}(\Delta, I(\Delta), \{r \mid r \in \Delta \wedge r.\mathsf{id} = \mathsf{acc\_id}\})$, which effectively asserts that the select1 query returns a record with id = acc_id from *some* database state that satisfies the non-negative balance invariant I. This weakened assertion is nonetheless enough to deduce that $\mathsf{a.bal} \geq 0$, and subsequently prove that $\mathsf{a.bal} + \mathsf{pc} * \mathsf{a.bal} \geq 0$, allowing us to verify the add_interest transaction.

The state transformer rules, like the earlier RG rules, closely follow the corresponding reduction rules in Fig. 5, except that their language of expression is $\mathcal{S}$. For instance, while the reduction rule for UPDATE declaratively specifies the set of updated records, the state transformer rule uses $\mathcal{S}$'s bind operation to *compute* the set. Other SQL rules do likewise. The rules for LET binders, conditionals, and sequences compose the effects inferred for their subcommands. Thus, the effect of a sequence of commands $c_1; c_2$ is the union of effects F_1 and F_2 of c_1 and c_2, respectively, except that F_2 is computed in a context that includes F_1 (we write $\mathsf{F}_1 \cup \mathsf{F}_2$ as a shorthand for $\lambda(\Delta).\ \mathsf{F}_1(\Delta) \cup \mathsf{F}_2(\Delta)$). The inference rule for FOREACH takes advantage of the $\mathcal{S}$'s bind operator to lift the effect inferred for the loop body to the level of the loop. Since records added to δ in each iteration of FOREACH are independent of the previous iteration (recall that we make a local context independence assumption about database programs; Sec. 3), sequential composition of the effects of different iterations is the same as their parallel composition. Since the loop body is executed once per each $z \in x$, the effect of the the loop is a union of effects (F) for all $z \in x$, all applied to the same state (Δ). That is, $\mathsf{F}_{loop}(\Delta) = \bigcup_{z \in x} \mathsf{F}_{body}(\Delta)$. From the definition of the set monad's bind operator, $\mathsf{F}_{loop}(\Delta) = x \gg= (\lambda z.\ F_{body}(\Delta))$, which mirrors the definition of the rule.

5.1 Soundness of Inference

We now formally state the correspondence between the inference rules given above and the RG judgment of §4:

$$
\begin{array}{lcll}
[\![\delta \mid \Delta \mid \ldots]\!]_{\langle\overline{\nu}\rangle} & = & (\top, \lambda(\overline{v}, r).\, r \in \delta) \mid (\top, \lambda(\overline{v}, r).\, r \in \Delta) \mid \ldots & |\overline{v}| = |\overline{\nu}| \\
[\![\{x \mid \varphi\}]\!]_{\langle\overline{\nu}\rangle} & = & (\top, \lambda(\overline{v}, r).\, [r/x]\varphi) & |\overline{v}| = |\overline{\nu}| \\
[\![\text{if } \varphi \text{ then } s_1 \text{ else } s_2]\!]_{\langle\overline{\nu}\rangle} & = & (\phi_1 \wedge \phi_2,\ \lambda(\overline{v}, r).\, \mathtt{if}\ \varphi \ \mathtt{then}\ \mathsf{G}_1(\overline{v}, r) & (\phi_1, \mathsf{G}_1) = [\![s_1]\!]_{\langle\overline{\nu}\rangle} \\
 & & \qquad\qquad\qquad\qquad \mathtt{else}\ \mathsf{G}_2(\overline{v}, r)) & (\phi_2, \mathsf{G}_2) = [\![s_2]\!]_{\langle\overline{\nu}\rangle} \\
[\![s_1 \cup s_2]\!]_{\langle\overline{\nu}\rangle} & = & (\phi_1 \wedge \phi_2, & (\phi_1, \mathsf{G}_1) = [\![s_1]\!]_{\langle\overline{\nu}\rangle} \\
 & & \quad \lambda(\overline{v}, r).\, \mathsf{G}_1(\overline{v}, r) \vee \mathsf{G}_2(\overline{v}, r)) & (\phi_2, \mathsf{G}_2) = [\![s_2]\!]_{\langle\overline{\nu}\rangle} \\
[\![s_1 \gg= \lambda x.s_2]\!]_{\langle\overline{\nu}\rangle} & = & (\phi_1 \wedge \phi_2 \wedge \forall\overline{\nu}.\forall a.\forall b.\ \pi_1(\overline{\nu}) \Leftrightarrow \qquad \mathsf{fresh}(\pi_1) & \mathsf{fresh}(\pi_2) \quad \mathsf{fresh}(g) \\
 & & \qquad\qquad \mathsf{G}_1(\overline{\nu}, a) \wedge \mathsf{G}_2(\overline{\nu}, a, b) \Rightarrow g(\overline{\nu}, b) & (\phi_1, \mathsf{G}_1) = [\![s_1]\!]_{\langle\overline{\nu}\rangle} \\
 & & \qquad \wedge \forall\overline{\nu}.\forall b.\exists a.\ \pi_2(\overline{\nu}) \Leftrightarrow & (\phi_2, \mathsf{G}_2) = [\![[a/x]s_2]\!]_{\langle\overline{\nu}, a\rangle} \\
 & & \qquad\qquad g(\overline{\nu}, b) \Rightarrow \mathsf{G}_1(\overline{\nu}, a) \wedge \mathsf{G}_2(\overline{\nu}, a, b), & \mathsf{fresh}(a) \quad \mathsf{fresh}(b) \\
 & & \lambda(\overline{v}, r).\, \pi_1(\overline{v}) \ \wedge\ \pi_2(\overline{v}) \ \wedge\ g(\overline{v}, r)) & |\overline{v}| = |\overline{\nu}| \\
[\![\mathsf{exists}(\Delta, \phi, s)]\!]_{\langle\overline{\nu}\rangle} & = & (\phi_s \wedge \forall\overline{\nu}.\forall a.\forall b.\ f(\overline{\nu}, a) \wedge f(\overline{\nu}, b) \Rightarrow a = b & \mathsf{fresh}(a) \quad \mathsf{fresh}(b) \\
 & & \quad \wedge \forall\overline{\nu}.\exists a.\ f(\overline{\nu}, a) & \mathsf{fresh}(f) \\
 & & \quad \wedge \forall\overline{\nu}.\forall a.\forall b.\ \pi(\overline{\nu}) \Leftrightarrow f(\overline{\nu}, a) \wedge [a/\Delta]\phi & \mathsf{fresh}(\pi) \quad \mathsf{fresh}(g) \\
 & & \qquad\qquad\qquad \wedge g(\overline{\nu}, b) = \mathsf{G}_s(\overline{\nu}, b), & (\phi_s, \mathsf{G}_s) = [\![[a/\Delta]s]\!]_{\langle\overline{\nu}\rangle} \\
 & & \lambda(\overline{v}, r).\, \pi(\overline{v}) \ \wedge\ g(\overline{v}, r)) & |\overline{v}| = |\overline{\nu}|
\end{array}
$$

Fig. 10. Encoding $\mathcal{S}$ in first-order logic

Theorem 5.1. *For all i,R,I,c,*$\mathsf{F}_{\mathsf{ctxt}}$, F, *if* $\mathtt{stable}(\mathbb{R}, I)$, $\mathtt{stable}(\mathbb{R}, \mathsf{F}_{\mathsf{ctxt}})$ *and* $\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i, \mathbb{R}, I\rangle} \mathsf{F}$, *then:*

$$\mathbb{R} \vdash \{\lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \ \wedge\ I(\Delta)\}\ [c]_i\ \{\lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)\}$$

Proof Sketch. The proof follows by structural induction on c. Let $P = \lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \wedge I(\Delta)$ and $Q = \lambda(\delta, \Delta).\delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)$. The base cases correspond to `INSERT`, `UPDATE` and `DELETE` statements, where the proof is straightforward. The proofs for `SELECT`, sequencing, and conditionals use the inductive hypothesis to infer the RG-judgments present in the premises of their corresponding RG-rules. The interesting case is the `FOREACH` statement, for which we use the loop invariant $\psi(\delta, \Delta) \Leftrightarrow \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup (y \gg= (\lambda z.\ \mathsf{F}(\Delta)))$, (where assuming that c is the body of the loop, $c \Longrightarrow_{\langle i, \mathbb{R}, I\rangle} \mathsf{F}$) to prove all the premises of RG-Foreach. Using the same notation as the rule RG-Foreach, y refers to the records already processed in previous iterations of the loop, while z refers to the record being processed in the current iteration. At the beginning of the loop $[\phi/y]\psi(\delta, \Delta)$ just reduces to $\delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta)$ which is implied by the pre-condition P. From the inductive hypothesis, we can infer that each iteration corresponds to the application of F. Since all iterations are assumed to be independent of each other, and z is bound to a record in x for each iteration, we conclude that at the end of every iteration, the loop invariant $[y \cup \{z\}/y]\psi$ will be satisfied.

5.2 From $\mathcal{S}$ to the First-Order Logic

Theorem 5.1 lets us replace the local judgment of the RG-Txn rule (Fig. 6) by a state transformer inference judgment. The soundness of a transaction's guarantee can now be established w.r.t the effect F of the body. The RG-Txn rule so updated is shown below ($\mathsf{F}_\emptyset = \lambda(\Delta).\ \emptyset$ denotes an empty context):

$$\frac{\begin{array}{c}\mathtt{stable}(R, \mathbb{I}) \quad \mathtt{stable}(R, I) \quad \mathbb{R}_e = R\backslash\mathbb{I}_e \quad \mathbb{R}_c = R\backslash\mathbb{I}_c \quad \mathsf{F}_\emptyset \vdash c \Longrightarrow_{\langle i, \mathbb{R}_e, I\rangle} \mathsf{F} \\ \mathtt{stable}(\mathbb{R}_c, \mathsf{F}_\emptyset[\mathsf{F}]) \quad \forall\Delta.\ G(\Delta, \mathsf{F}(\Delta)) \quad \forall\Delta, \Delta'.\ I(\Delta) \wedge G(\Delta, \Delta') \Rightarrow I(\Delta')\end{array}}{\{I, R\}\ \mathtt{TXN}_i\langle\mathbb{I}\rangle\{c\}\ \{G, I\}}$$

Automating the application of the RG-Txn rule for a transaction requires automating the multiple implication checks in the premise. While R, G, $\mathbb{I}$ and I are formulas in first-order logic (FOL) with a relatively simple structure, F is an expression in the set language $\mathcal{S}$ (Fig. 7) with a possibly complex structure. Fortunately, however, there exists a semantics-preserving translation from $\mathcal{S}$ to a restricted subset of first-order logic (FOL) that lends itself to automatic reasoning.

The algorithm ($[\![\cdot]\!]_{\langle\cdot\rangle}$) shown in Fig. 10 translates an $\mathcal{S}$ expression (s) to FOL. The translation is based on encoding a set of element type T as a unary predicate on T. The predicate is represented as a meta function that accepts an $x : T$ and returns a quantifier-free proposition that evaluates to true ($\top$) if and only if x is present in the set. Alternatively, the translation may also encode the set as a predicate in the logic itself, in which case a quantified proposition constraining the predicate is also generated. For instance, consider the set $\{1, 2\}$. The predicate describing the set can be encoded as the function $\lambda v.v = 1 \vee v = 2$, with no further constraints, or it can be encoded as the function $\lambda v.g(v)$ with an associated constraint, $\phi \in \mathbb{P}^1 = \forall \nu.\ g(\nu) \Leftrightarrow \nu = 1 \vee \nu = 2$, defining the uninterpreted predicate g. The translation adopts one or the other approach, depending on the need. For uniformity, we consider the encoding of a set as pair (ϕ,G), where G is a meta function, and ϕ is a FOL formula constraining any uninterpreted predicates used in G.

Due to the presence of bind ($\gg\!=$) in $\mathcal{S}$, a set expression s may contain free variables introduced by an enclosing binder. For instance, consider the $\mathcal{S}$ expression $s_1 \gg\!= (\lambda x.\{y \mid y = x + 1\})$, where s_1 is an integer set (expression). The subexpression $\{y \mid y = x + 1\}$ (call it s_2) contains x as a free variable. In such cases, the predicate associated with the subexpression should also be indexed by its free variables so that a unique set exists for each instantiation of the free variables. Thus, the predicate (G) associated with the subexpression from the above example should be $\lambda v_1.\lambda v_2.\ v_2 = v_1 + 1$, so that the set G x_1 is different from the set G x_2 for distinct $x_1, x_2 \in s_1$. Intuitively, the bind expression $s_1 \gg\!= (\lambda x.\{y \mid y = x + 1\})$ denotes the set $\bigcup_{x \in s_1} \mathrm{G}\ x$.

The translation algorithm (Fig. 10) takes free variables into account. Given a set expression $s \in \mathcal{S}$, whose (possible) free variables are $\overline{\nu}$ in the order of their introduction (top-most binder first), $[\![s]\!]_{\langle\overline{\nu}\rangle}$ returns the encoding of s as (ϕ, G). The meta-function G is a predicate indexed by the (possible) free variables of s, and thus its arity is $|\overline{\nu}| + 1$. Note that $\overline{\nu}$ is only a sequence of variables introduced by the enclosing binders of s, and not all may actually occur free in s. Nonetheless, its predicate G is always indexed by $|\overline{\nu}|$ free variables for uniformity. The translation encodes database state as an uninterpreted sort. Considering that the state is actually a set of records, we define an uninterpreted relation "$\in$" to relate records and states. Thus, a variable set expression Δ denoting a database state is encoded as the predicate $\lambda(\overline{v}, r).\ r \in \Delta$, where $|\overline{v}| = |\overline{\nu}|$ (predicates are uncurried for simplicity; $\overline{v}$ is a comma-separated sequence; $r \notin \mathcal{S}$ is a special variable). The constraints associated with the encoding of a state are trivial (denoted $\top$). The set literal expression $\{x \mid \varphi\}$ is encoded straightforwardly. The conditional set expression is encoded as an if-then-else predicate in FOL, where the predicates on true and false branches are computed from the set subexpressions s_1 and s_2, respectively. The conjunction of constraints ϕ_1 and ϕ_2, from $[\![s_1]\!]_{\langle\overline{\nu}\rangle}$ and $[\![s_2]\!]_{\langle\overline{\nu}\rangle}$ (resp.), is propagated upwards as the constraint of the conditional expression. A set union expression is encoded similarly.

The first-order encoding of a bind expression describes the semantics of the set monad's bind operator in FOL. Let s_1 be a set, and let f be a function that maps each variable in s_1 to a new set. Then, $s_2 = s_1 \gg\!= f$ if and only if for all $y \in s_2$, there exists an $x \in s_1$ such that $y = f(x)$, and forall $x \in s_1$, $f(x) \in s_2$. The encoding essentially adds new constraints to this effect. The translation first encodes s_1 and s_2 to obtain (ϕ_1, G$_1$) and (ϕ_2, G$_2$), respectively. Since s_2 is under a new binder that binds x, the free variable sequence of s_2 is $\overline{\nu}, x$. In the interest of hygiene, we substitute a fresh a for x, making the sequence $\overline{\nu}, a$. The set s is encoded as a new uninterpreted predicate g indexed by s's free variables ($\overline{\nu}$). Since the set denoted by g is the result of the bind $s_1 \gg\!= \lambda x.s_2$, first-order constraints defining the bind operation (as described above) are generated.

The constraints relate the predicates G_1 and G_2, representing s_1 and s_2 (resp.), to the uninterpreted predicate g that represents s. The constraints are assigned names (π_1 and π_2) to give them an easy handle.

The first-order encoding of the exists(Δ, ϕ, s) expression essentially Skolemizes the existential. Skolemizing is the process of substituting an existentially bound x in $\phi_x \in \mathbb{P}^1$ with $f(\overline{v})$, where f is a fresh uninterpreted function (called the Skolem function), and $\overline{v}$ are the free variables in ϕ_x bound by enclosing universal quantifiers. Due to the decidability restrictions (Sec. 5.3), the only uninterpreted functions we admit in our logic are boolean (i.e., predicates/relations). Consequently, we cannot define the Skolem function f directly. Instead, we define it via an uninterpreted relation, by explicitly asserting the function property:

$$(\forall \overline{v}.\forall a.\forall b.\ f(\overline{v}, a) \wedge f(\overline{v}, b) \Rightarrow a = b)\ \wedge\ (\forall \overline{v}.\exists a.f(\overline{v}, a))$$

We then replace the existentially bound Δ with a new universally bound a in ϕ and s, such that $f(\overline{v}, a)$ holds, before encoding the existentially bound s.

Example Let us reconsider the TPC-C `new_order` transaction from Sec. 2. Recall that the state transformer (T) for the `foreach` loop shown in Fig. 4 is (k1, k2, and k3 are constants):

$$\lambda(\delta, \Delta).\ \delta \cup \texttt{item\_reqs} \gg= (\lambda \texttt{item\_req}.\ \mathsf{F}_U(\Delta) \cup \mathsf{F}_I(\Delta))$$

where:

$$\begin{aligned}
\mathsf{F}_U &= \lambda(\Delta).\ \Delta \gg= (\lambda s.\text{if } \mathsf{table}(s) = \mathsf{Stock}\ \wedge\ s.\mathsf{s\_i\_id} = \mathsf{item\_req.ol\_i\_id} \\
&\qquad\qquad \text{then } \{\langle \mathsf{s\_i\_id} = s.\mathsf{s\_i\_id};\ \mathsf{s\_d\_id} = s.\mathsf{s\_d\_id};\ \mathsf{s\_qty} = \mathsf{k1}\rangle\} \\
&\qquad\qquad \text{else } \emptyset) \\
\mathsf{F}_I &= \lambda(\Delta).\ \{\langle \mathsf{ol\_o\_id} = \mathsf{k2};\ \mathsf{ol\_d\_id} = \mathsf{k3};\ \mathsf{ol\_i\_id} = \mathsf{item\_req.ol\_i\_id}; \\
&\qquad\qquad \mathsf{ol\_qty} = \mathsf{item\_req.ol\_qty}\rangle\}
\end{aligned}$$

For any Δ, $\mathsf{F}_U(\Delta)$ and $\mathsf{F}_I(\Delta)$ are expressions in $\mathcal{S}$, so can be translated to FOL by the encoding algorithm in Fig. 10. Since the iteration variable `item_req` occurs free in these expressions, the appropriate application of the encoding algorithm is $[\![\mathsf{F}_U(\Delta)]\!]_{\langle \texttt{item\_req}\rangle}$ and $[\![\mathsf{F}_I(\Delta)]\!]_{\langle \texttt{item\_req}\rangle}$, which results in (ϕ_U, G_U) and (ϕ_I, G_I), respectively, where ϕ_U, ϕ_I, G_U, G_I are as shown below:

$$\begin{aligned}
\phi_U &= \forall \mathsf{item\_req}.\forall s.\forall s'.\ \pi_1(\mathsf{item\_req}) \Leftrightarrow \\
&\qquad (s \in \Delta)\ \wedge\ (\text{if } \mathsf{table}(s) = \mathsf{Stock} \wedge s.\mathsf{s\_i\_id} = \mathsf{item\_req.ol\_i\_id} \\
&\qquad\qquad \text{then } s' = \langle \mathsf{s\_i\_id} = s.\mathsf{s\_i\_id};\ \mathsf{s\_d\_id} = s.\mathsf{s\_d\_id};\ \mathsf{s\_qty} = \mathsf{k1}\rangle \\
&\qquad\qquad \text{else } \bot) \Rightarrow g_0(\mathsf{item\_req}, s') \\
&\quad \wedge \forall \mathsf{item\_req}.\forall s'.\exists s.\ \pi_2(\mathsf{item\_req}) \Leftrightarrow \\
&\qquad g_0(\mathsf{item\_req}, s') \Rightarrow s \in \Delta\ \wedge\ \text{if } \mathsf{table}(s) = \mathsf{Stock} \wedge s.\mathsf{s\_i\_id} = \mathsf{item\_req.ol\_i\_id} \\
&\qquad\qquad \text{then } s' = \langle \mathsf{s\_i\_id} = s.\mathsf{s\_i\_id};\ \mathsf{s\_d\_id} = s.\mathsf{s\_d\_id};\ \mathsf{s\_qty} = \mathsf{k1}\rangle \\
&\qquad\qquad \text{else } \bot \\
G_U &= \lambda(\mathsf{item\_req}, r).\ \pi_1(\mathsf{item\_req}) \wedge \pi_2(\mathsf{item\_req}) \wedge g_0(\mathsf{item\_req}, r) \\
\phi_I &= \top \\
G_I &= \lambda(\mathsf{item\_req}, r).\ r = \langle \mathsf{ol\_o\_id} = \mathsf{k2};\ \mathsf{ol\_d\_id} = \mathsf{k3};\ \mathsf{ol\_i\_id} = \mathsf{item\_req.ol\_i\_id}; \\
&\qquad \mathsf{ol\_qty} = \mathsf{item\_req.ol\_qty}\rangle
\end{aligned}$$

Since the transformer (T) of the `foreach` loop is not nested does not contain any free iteration variables, the appropriate application of the encoding algorithm is $[\![\mathsf{T}(\delta, \Delta)]\!]_{\langle\emptyset\rangle}$, which results in the $(\phi_I \wedge \phi_U \wedge \phi_1 \wedge \phi_2, G)$, where ϕ_1, ϕ_2, and G are as defined below:

$$\begin{aligned}
\phi_1 &= \forall \mathsf{item\_req}.\forall s.\ \pi_3 \Leftrightarrow \mathsf{item\_req} \in \mathsf{item\_reqs} \wedge G_U(\mathsf{item\_req}, s') \vee G_I(\mathsf{item\_req}, s') \Rightarrow g_1(s) \\
\phi_2 &= \forall s.\exists \mathsf{item\_req}.\ \pi_4 \Leftrightarrow g_1(s) \Rightarrow \mathsf{item\_req} \in \mathsf{item\_reqs} \wedge G_U(\mathsf{item\_req}, s') \vee G_I(\mathsf{item\_req}, s') \\
G &= \lambda(r).\ \pi_3 \wedge \pi_4 \wedge g_1(r)
\end{aligned}$$

5.3 Decidability

Observe that the encoding shown in Fig. 10 maps to a fragment of FOL that satisfies the following syntactic properties:

- All function symbols, modulo those that are drawn from $\mathbb{P}^0$ and $\mathbb{P}^1$, are uninterpreted and boolean.
- All quantification is first-order; second-order objects, such as sets and functions, are never quantified.
- Quantifiers appear only at the prenex position, i.e., at the beginning of a quantified formula.

The simple syntactic structure of the fragment already makes is amenable for automatic reasoning via an off-the-shelf SMT solver, such as Z3. The decidability of this fragment, however, is more subtle and discussed below.

Consider a set expression s with no free variables (i.e., $\overline{v} = \emptyset$, like $\mathsf{T}(\delta, \Delta)$ from the above example). Let $(\phi, G) = [\![s]\!]_{\langle\emptyset\rangle}$. Note that ϕ is a conjunction of (a). ϕ_i's, where each ϕ_i results from encoding a subexpression s_i of s, and (b). a ϕ_s, resulting from encoding s itself (i.e., its top-level expression). From Fig. 10, it is clear that ϕ_s is either $\top$ (for the first four cases), or it is a prenex-quantified formula, where quantification is either $\forall^2$, or $\exists$, or $\forall\exists$. Generalizing this observation, for a set expression s with $|\overline{v}|$ free variables, ϕ_s, if quantified, is a prenex-quantified formula, where quantification assumes one among the forms of $\forall^{|\overline{v}|+2}$, or $\forall^{|\overline{v}|}\exists$, or $\forall^{|\overline{v}|+1}\exists$. In other words, the number of $\forall$ quantifiers preceding an $\exists$ quantifier is utmost one more than the number of free variables ($\overline{v}$) in s. For the convenience of this discussion, let us call $\forall^{|\overline{v}|+1}\exists$ as the prenex signature of ϕ_s.

Next, in Fig. 10, observe that the (ordered) set $\overline{v}$ is extended only in the encoding rule for $\gg=$. Since an occurrence of $\gg=$ adds a quantifier to $|\overline{v}|$, if s is a bind expression nested inside a top-level bind expression (like $\mathsf{F}_U(\Delta)$ from the above example), then the prenex signature of ϕ_s is $\forall^2\exists$. Furthermore, if the subexpressions of s are neither bind nor exists expressions, then none of the ϕ_i's are quantified, and the prenex signature of $\phi = \bigwedge_i \phi_i \wedge \phi_s$ remains $\forall^2\exists$. A similar observation holds when s is an exists expression nested inside a top-level bind expression. Since exists is generated as a result of stabilizing a SQL command transformer, which is always a non-nested bind expression, the subexpression (s') of exists is a non-nested bind expression. s' is however nested inside a top-level bind expression, hence its prenex signature is $\forall^2\exists$. Since exists does not extend $\overline{v}$, the prenex signature of s remains $\forall^2\exists$. When s is an expression other than $\gg=$ or exists, then ϕ_s is not a quantified formula, and its prenex signature is trivially subsumed by $\forall^2\exists$. Thus, for the subset of $\mathcal{S}$, where bind expressions are restricted to one level of nesting, the FOL formulas generated by the encoding have the prenex signature as $\forall^2\exists$.

The fragment of FOL that admits formulas with prenex signatures of the form $\forall^2\exists^*$ is called the Gödel-Kálmar-Schütte (GKS) fragment [Börger et al. 1996], which is known to be decidable. The language of encoding, however, is a combination of GKS with (a). $\mathbb{P}^0$, the theory from which quantifier-free propositions (φ) that encode object language expressions are drawn, and (b). $\mathbb{P}^1$, the theory from which invariants (I) are drawn. Thus, the encoding of the subset of $\mathcal{S}$ described above is decidable if the combination of GKS + $\mathbb{P}^0$ + $\mathbb{P}^1$ is decidable. We write $\mathcal{S}[\mathbb{P}^0, \mathbb{P}^1]$ to highlight the parameterization of $\mathcal{S}$ on $\mathbb{P}^0$ and $\mathbb{P}^1$. The discussion in the previous paragraph points to the existence of non-trivial subsets in $\mathcal{S}[\mathbb{P}^0, \mathbb{P}^1]$ that are decidable:

Theorem 5.2. *There exist $\mathcal{S}'[\mathbb{P}^0, \mathbb{P}^1] \subset \mathcal{S}[\mathbb{P}^0, \mathbb{P}^1]$ such that $\mathcal{S}'$ is decidable if* GKS + $\mathbb{P}^0$ + $\mathbb{P}^1$ *is decidable.*

One interesting example of such an $\mathcal{S}'$ is the subset described above: $\mathcal{S}$ with bind expressions confined to one level of nesting. We denote this subset as $\mathcal{S}^1[\mathbb{P}^0, \mathbb{P}^1]$, for which we assert decidability:

Corollary 5.3. *$\mathcal{S}^1[\mathbb{P}^0, \mathbb{P}^1]$ is decidable if* GKS + $\mathbb{P}^0$ + $\mathbb{P}^1$ *is decidable.*

$\mathcal{S}^1$ is a useful subset of $\mathcal{S}$, for it corresponds to $\mathcal{T}$ programs without nested `foreach` loops. Observe that the `new_order` transaction (Fig. 1) belongs to this subset. Indeed, $\mathcal{S}^1$, while being a

```
type table_name =  District | Order | Order_line | Stock

type district = {d_id: int; d_next_o_id: int}
type order = {o_id: int; o_d_id: int; o_c_id: int; o_ol_cnt: int}
type order_line = {ol_o_id: int; ol_d_id: int; ol_i_id: int; ol_qty: int}
type stock = {s_i_id: int; s_d_id:int; s_qty: int}
```

Fig. 11. OCaml type definitions corresponding to the TPC-C schema from Fig. 2

restricted version of $\mathcal{S}$, is nonetheless expressive enough to cover all the benchmarks we considered in Sec. 7.

A useful instantiation of $\mathcal{S}^1$ is $\mathcal{S}[\mathsf{BV}, \mathsf{GKS} + \mathsf{BV}]$, where BV is the theory of bit-vector arithmetic, which is often used to encode the finite-bit integer arithmetic of real programs. Finite-bit integer arithmetic has a finite axiomatization in GKS. For instance, 32-bit integers can be encoded as 2^{32} distinct constants of an uninterpreted sort T, while integer operations like addition and multiplication can be encoded as uninterpreted functions whose properties are enumerated for the entire domain of T. Thus BV is subsumed by GKS. Since the latter is decidable, the combination is decidable:

Theorem 5.4. *$\mathcal{S}^1[\mathsf{BV}, \mathsf{GKS} + \mathsf{BV}]$ is decidable.*

This instantiation requires I to be drawn from GKS+BV, which is expressive enough to describe common database integrity constraints, such as referential integrity, non-negativeness of all integer values in a column etc. The isolation specifications presented in §3.1 are already simple first-order formulas that can be encoded in GKS. Furthermore, it is also reasonable to expect the guarantee (G) of a transaction to be expressible in the same logic as its inferred F, since F (without the stability check) is essentially a complete characterization of the transaction, while G is only an abstraction. Thus, with $\mathcal{S}^1[\mathsf{BV}, \mathsf{GKS} + \mathsf{BV}]$ as the language of inference, the verification problem for weakly isolated transactions is decidable.

6 IMPLEMENTATION

We have implemented our DSL to define transactions as monadic computations in OCaml (modulo some syntactic sugar), and our automatic reasoning framework as an extra frontend pass (called ACIDifier) in the ocamlc 4.03 compiler[12]. The input to ACIDifier is a program in our DSL that describes the schema of the database as a collection of OCaml type definitions, and transactions as OCaml functions, whose top-level expression is an application of the `atomically_do` combinator. For instance, TPC-C's schema from Fig. 2 can be described via the OCaml type definitions shown in Fig. 11. ACIDifier also requires a specification of the program in the form of a collection of guarantees (G), one per transaction, and an invariant I that is a conjunction of the integrity constraints on the database. An auxiliary DSL that includes the first-order logic (FOL) combinators has been implemented for this purpose. ACIDifier's verification pass follows OCaml's type checking pass, hence the concrete artifact of verification is OCaml's typed AST. The tool is already equipped with an axiomatization of PostgreSQL and MySQL's isolation levels expressed in our FOL DSL. Other data stores can be similarly axiomatized. The concrete result of verification is an assignment of an isolation level of the selected data store to each transaction in the program.

At the top-level, ACIDifier runs a loop that picks an unverified transaction and progressively strengthens its isolation level until it passes verification. If the selected data store provides a serializable isolation level, and if the program is sequentially correct, then the verification is guaranteed to succeed. Within the loop, ACIDifier first computes the various rely relations needed

[12]The source code is available at available at https://github.com/gowthamk/acidifier

for verification (R, $\mathbb{R}_I$, and $\mathbb{R}_c$). It then traverses the AST of a transaction, applying the inference rules to construct a state transformer, checks its stability, and weakens it ($\llbracket \cdot \rrbracket_{\langle \mathbb{R}, I \rangle}$) if it is not stable. The result of traversing the transaction's AST is therefore a state transformer (F) that is stable w.r.t $\mathbb{R}_I$, which is also stabilized against $\mathbb{R}_c$ (using $\llbracket \cdot \rrbracket_{\langle \mathbb{R}, I \rangle}$), and then checked against the transaction's stated guarantee (G). If the check passes, then the guarantee is verified to check if it preserves the invariant I. The successful result from both checks results in the transaction being certified correct under the current choice of its isolation level. Successful verification of all transactions concludes the top-level execution, returning the inferred isolation levels as its output. ACIDifier uses the Z3 SMT solver as its underlying reasoning engine. Each implication check described above is first encoded in FOL, applying the translation described in §5 wherever necessary.

6.1 Pragmatics

Real-World Isolation Levels The axiomatization of the isolation levels presented in §3.1 leaves out certain nuances of their implementations on real data stores, which need to be taken into account for verification to be effective in practice. We take these into account while linking ACIDifier with store-specific semantics (isolation specifications, etc.). As an example, consider how PostgreSQL implements an `UPDATE` operation. `UPDATE` first selects the records that meet the search criteria from the snapshot against which it is executing (the snapshot is established at the beginning of the transaction if the isolation level is SI, or at the beginning of the `UPDATE` statement if the isolation level is RC). The selected records are then visited in the actual database (if they still exist), write locks are obtained, and the update is performed, provided that each matched record still meets `UPDATE`'s search criteria. If a record no longer meets the search criteria (due to a concurrent update), it is excluded from the update, and the write lock is immediately released. Otherwise, the record remains locked until the transaction commits.

Clearly, this sequence of events is not atomic, unlike the assumption made by our formal model because the implementation admits interference between the updates of individual records that meet the search criteria. Nonetheless, through a series of relatively straightforward deductions, we can show that PostgreSQL's `UPDATE` is in fact equivalent (in behavior) to a sequential composition of two atomic operations $c_1; c_2$, where c_1 is effectively a `SELECT` operation with the same search criteria as `UPDATE`, and c_2 is a slight variation of the original `UPDATE` that updates a record only if a record with the same id is present in the set of records returned by `SELECT`:

$$\mathtt{UPDATE}\ (\lambda x.\ e_1)\ (\lambda x.\ e_2) \quad \longrightarrow \quad \mathtt{LET}\ y = \mathtt{SELECT}\ (\lambda x.\ e_1)\ \mathtt{IN}\ \mathtt{UPDATE}\ (\lambda x.\ e_1 \wedge x.\mathtt{id} \in \mathsf{dom}(y))\ (\lambda x.\ e_2)$$

The intuition behind this translation is the observation that all interferences possible during the execution of the `UPDATE` can be accommodated between the time the records are selected from the snapshot, and the time they are actually updated. ACIDifier performs this translation if the selected store is PostgreSQL, allowing it to reason about `UPDATE` operations in a way that is faithful to its semantics on PostgreSQL. ACIDifier also admits similar compensatory logic for certain combinations of isolation levels and operations on MySQL.

Set functions SQL's `SELECT` query admits projections of record fields, and also application of auxiliary functions such as `MAX` and `MIN`, e.g., `SELECT MAX(ol_o_id) FROM Order_line WHERE ...`, etc. We admit such extensions as set functions in our DSL (e.g., `project`, `max`, `min`), and axiomatize their behavior. For instance:

$$\begin{array}{rcll} s_2 & = & \mathtt{project}\, s_1\, (\lambda z.\ e) & \Leftrightarrow \quad \forall y.\ y \in s_2 \Leftrightarrow \exists (x \in s_1).\ y = [x/z]e \\ x & = & \mathtt{max}\, s & \Leftrightarrow \quad x \in s\ \wedge\ \forall (y \in s).\ y \leq x \end{array}$$

There are however certain set functions whose behavior cannot be completely axiomatized in FOL. These include `sum`, `count` etc. For these, we admit imprecise axiomatizations.

Annotation Burden ACIDifier significantly reduces the annotation burden in verifying a weakly isolated transactions by eliminating the need to annotate intermediate assertions and loop

```
type table_name = Student | Course | Enrollment
type student = {s_id: id; s_name: string}
type course = {c_id: id; c_name: string; c_capacity: int}
type enrollment = {e_id: id; e_s_id: id; e_c_id: id}

let enroll_txn sid cid =
  let crse = SQL.select1 [Course] (fun c -> c.c_id = cid) in
  let s_c_enrs = SQL.select [Enrollment] (fun e -> e.e_s_id = sid &&
                                                  e.e_c_id = cid) in
  if crse.c_capacity > 0 && Set.is_empty s_c_enrs then
    (SQL.insert Enrollment {e_id=new_id (); e_s_id=sid; e_c_id=cid};
     SQL.update Course (fun c -> {c with c_capacity = c.c_capacity - 1})
                       (fun c -> c.c_id = cid)) else ()

let deregister_txn sid =
  let s_enrs = SQL.select [Enrollment] (fun e -> e.e_s_id = sid) in
  if Set.is_empty s_enrs then
    SQL.delete Student (fun s -> s.s_id = sid) else ()
```

Fig. 12. Courseware Application

invariants. Guarantees (G) and global invariants (I), however, still need to be provided. Alternatively, a weakly isolated transaction T can be verified against a generic serializability condition, eliminating the need for guarantee annotations. In this mode, ACIDifier first infers the transformer F_{SER} of T without considering any interference, which then becomes its guarantee (G). Doing likewise for every transaction results in a rely relation (R) that includes F_{SER} of every transaction. Verification now proceeds by taking interference into account, and verifying that each transaction still yields the same F as its F_{SER}. The result of this verification is an assignment of (possibly weak) isolation levels to transactions which nonetheless guarantees behavior equivalent to a serializable execution.

7 EVALUATION

In this section, we present our experience in running ACIDifier on two different applications: *Courseware*: a course registration system described by [Gotsman et al. 2016], and TPC-C.

Courseware The Courseware application allows new courses to be added (via an `add_course` transaction), and new students to be registered (via a `register` transaction) into a database. A registered student can enroll (`enroll`) in an existing course, provided that enrollment has not already exceeded the course capacity (`c_capacity`). A course with no enrollments can be canceled (`cancel_course`). Likewise, a student who is not enrolled in any course can be deregistered (`deregister`). Besides `Student` and `Course` tables, there is also an `Enrollment` table to track the many-to-many enrollment relationship between courses and students. The simplified code for the Courseware application with only `enroll` and `deregister` transactions is shown in Fig. 12. The application is required to preserve the following invariants on the database:

(1) I_1: An enrollment record should always refer to an existing student and an existing course.
(2) I_2: The capacity (`c_capacity`) of a course should always be a non-negative quantity.

Both I_1 and I_2 can be violated under weak isolation. I_1 can be violated, for example, when `deregister` runs concurrently with `enroll`, both at RC isolation. While the former transaction removes the student record after checking that no enrollments for that student exists, the latter transaction concurrently adds an enrollment record after checking the student exists. Both can succeed concurrently, resulting in an invalid state. Invariant I_2 can be violated by two `enroll`s, both reading

Table 1. The discovered isolation levels for TPC-C transactions

	new_order	delivery	payment	order_status	stock_level
MySQL	SER	SER	RC	RC	RC
PostgreSQL	SI	SI	RC	RC	RC

c_capacity=1, and both (atomically) decrementing it, resulting in c_capacity=-1. We ran ACIDIFIER on the Courseware application (Fig. 12) after annotating transactions with their respective guarantees, and asserting $I = I_1 \wedge I_2$ as the correctness condition. The guarantees G_e and G_d for enroll and deregister transactions, respectively, are shown below:

$$\begin{array}{lll}
G_e(\Delta, \Delta') & \Leftrightarrow & \Delta'_s = \Delta_s \;\wedge\; \exists \texttt{cid}.\exists \texttt{sid}. \\
 & & \quad \Delta'_c = \Delta_c \gg\!= \lambda c.\ \texttt{if}\ c.\texttt{c\_id} = \texttt{cid} \\
 & & \qquad\qquad \texttt{then exists}(c',\ c'.\texttt{id} = c.\texttt{id} \wedge c'.\texttt{c\_name} = c.\texttt{c\_name} \\
 & & \qquad\qquad\qquad \wedge\, c'.\texttt{c\_capacity} \geq 0,\ \{c'\}) \\
 & & \qquad\qquad \texttt{else}\ \{c\} \\
 & & \quad \wedge\ \Delta_e = \Delta'_e \gg\!= \lambda e.\ \texttt{if}\ e.\texttt{e\_c\_id} = \texttt{cid} \wedge e.\texttt{e\_s\_id} = \texttt{sid then}\ \emptyset\ \texttt{else}\ \{e\} \\
G_d(\Delta, \Delta') & \Leftrightarrow & \Delta'_c = \Delta_c \;\wedge\; \Delta'_e = \Delta_e \;\wedge\; \exists \texttt{sid}.\ \texttt{if}\ \forall(e \in \Delta_e).\ e.\texttt{e\_s\_id} \neq \texttt{sid} \\
 & & \qquad\qquad \texttt{then}\ \Delta'_s = \Delta_s \gg\!= \lambda s.\ \texttt{if}\ s.\texttt{id} = \texttt{sid then}\ \emptyset\ \texttt{else}\ \{s\} \\
 & & \qquad\qquad \texttt{else}\ \Delta'_s = \Delta_s
\end{array}$$

For the sake of this presentation we split Δ into three disjoint sets of records, Δ_s, Δ_c, and Δ_e, standing for Student, Course, and Enrollment tables, respectively. Observing that the set language $\mathcal{S}$ (Sec. 5), besides being useful for automatic verification, also facilitates succinct expression of transaction semantics, we define G_e and G_d in a combination of FOL and $\mathcal{S}$. G_e essentially says that the enroll transaction leaves the Student table unchanged, while it may update the c_capacity field of a Course record to a non-negative value (even when it doesn't update, it is the case that $c'.\texttt{c\_capacity} \geq 0$, because $c' = c$, and $c \in \Delta_c$, and we know that $I_2(\Delta_c)$). G_e also conveys that enroll might insert a new Enrollment record by stating that Δ_e, the Enrollment table in the pre-state, contains all records e from Δ'_e, the table in the post-state, except when $e.\texttt{e\_c\_id}$ and $e.\texttt{e\_s\_id}$ match cid and sid, respectively. The guarantee G_d of deregister asserts that the transaction doesn't write to Course and Enrollment tables. The transaction might however delete a Student record bearing an id=sid (formally, $\Delta'_s = \Delta_s \gg\!= \lambda s.\ \texttt{if}\ s.\texttt{id} = \texttt{sid then}\ \emptyset\ \texttt{else}\ \{s\}$), for some sid for which no corresponding Enrollment records are present in the pre-state (in other words, $\forall(e \in \Delta_e).\ e.\texttt{e\_s\_id} \neq \texttt{sid}$).

With help of the guarantees, such as those described above, ACIDIFIER was able to automatically discover the aforementioned anomalous executions, and was subsequently able to infer that the anomalies can be preempted by promoting the isolation level of enroll and deregister to SER (on both MySQL and PostgreSQL), leaving the isolation levels of remaining transactions at RC. The total time for inference and verification took less than a minute running on a conventional laptop.

TPC-C The simplified schema of the TPC-C benchmark has been described in Sec. 2. In addition to the tables shown in Fig. 2, the TPC-C schema also has Warehouse and New_order tables that are relevant for verification. To verify TPC-C, we examined four of the twelve consistency conditions specified by the standard, which we name I_1 to I_4:

(1) Consistency condition I_1 requires that the sales bottom line of each warehouse equals the sum of the sales bottom lines of all districts served by the warehouse.
(2) Conditions I_2 and I_3 effectively enforce uniqueness of ids assigned to Order and New_order records, respectively, under a district.
(3) Condition I_4 requires that the number of order lines under a district must match the sum of order line counts of all orders under the district.

Similar to the example discussed in Sec. 2, there are a number of ways TPC-C's transactions violate the aforementioned invariants under weak isolation. ACIDifier was able to discover all such violations when verifying the benchmark against $I = \bigwedge_i I_i$, with guarantees of all three transactions provided. The isolation levels were subsequently strengthened as shown in Table. 1. As before, inference and verification took less than a minute.

To sanity-check the results of ACIDifier, we conducted experiments with a high-contention OLTP workload on TPC-C aiming to explore the space of correct isolation levels for different transactions. The workload involves a mix of all five TPC-C transactions executing against a TPC-C database with 10 warehouses. Each warehouse has 10 districts, and each district serves 3000 customers. There are a total of 5 transactions in TPC-C, and given that MySQL and PostgreSQL support 3 isolation levels each, there are a total of $3^5 = 243$ different configurations of isolation levels for TPC-C transactions on MySQL and PostgreSQL. We executed the benchmark with all 243 configurations, and found 171 of them violated at least one of the four invariants we considered. As expected, the isolation levels that ACIDifier infers for the TPC-C transactions do not result in invariant violations, either on MySQL or on PostgreSQL, and were determined to be the weakest safe assignments possible.

8 RELATED WORK

Specifying weak isolation. Adya [Adya 1999] specifies several weak isolation levels in terms of *dependency graphs* between transactions, and the kinds of dependencies that are forbidden in each case. The operational nature of Adya's specifications make them suitable for runtime monitoring and anomaly detection [Cahill et al. 2008; Revilak et al. 2011; Zellag and Kemme 2014], whereas the declarative nature of our specifications make them suitable for formal reasoning about program behavior. [Sivaramakrishnan et al. 2015] specify isolation levels declaratively as trace well-formedness conditions, but their specifications implicitly assume a complete trace with only committed transactions, making it difficult to reason about a program as it builds the trace. [Cerone et al. 2015] specify isolation levels with atomic visibility, but their specifications are also for complete traces. Both the aforementioned specification frameworks use the vocabulary introduced in [Burckhardt et al. 2014]. However, none of them are equipped with a reasoning framework that can use such specifications to verify programs under weak isolation.

Recent work described in [Crooks et al. 2017] also explores the use of a state-based interpretation of isolation as we do, and like our approach, develops specifications of weak isolation that are not tied to implementation-specific artifacts. However, they do not consider verification (manual or automated) of client programs, and it is not immediately apparent if their specification formalism is amenable for use within a verification toolchain. [Warszawski and Bailis 2017] present a dynamic analysis for weak isolation that attempts to discover weak isolation anomalies from SQL log files. Their solution, while capable of identifying database attacks due to the use of incorrect isolation levels, does not consider how to verify application correctness, infer proper isolation levels, or formally reason about the relationship between weak-isolation levels and application invariants.

Reasoning under weak isolation. [Fekete et al. 2005] propose a theory to characterize non-serializable executions that arise under si. They also propose an algorithm that allocates either si or ser isolation levels to transactions while guaranteeing serializability. [Cerone and Gotsman 2016] improve on Adya's si specification and use it to derive a static analysis that determines the safety of substituting si with a weaker variant called *Parallel Snapshot Isolation* [Sovran et al. 2011]. These efforts focus on establishing the equivalence of executions between a pair of isolation levels, without taking application invariants into account. [Bernstein et al. 2000] propose informal semantic conditions to ensure the satisfaction of application invariants under weaker isolation levels. All these techniques are tailor-made for a finite set of well-understood isolation levels (rooted in [Berenson et al. 1995]).

Reasoning under weak consistency. There have been several recent proposals to reason about programs executing under weak consistency [Alvaro et al. 2011; Bailis et al. 2014; Balegas et al. 2015; Gotsman et al. 2016; Li et al. 2014, 2012]. All of them assume a system model that offers a choice between a *coordination-free* weak consistency level (*e.g.*, eventual consistency [Alvaro et al. 2011; Bailis et al. 2014; Balegas et al. 2015; Li et al. 2014, 2012]) or causal consistency [Gotsman et al. 2016; Lesani et al. 2016]). All these efforts involve proving that atomic and fully isolated operations preserve application invariants when executed under these consistency levels. In contrast, our focus in on reasoning in the presence of weakly-isolated transactions under a strongly consistent store. [Gotsman et al. 2016] adapt *Parallel Snapshot Isolation* to a transaction-less setting by interpreting it as a consistency level that serializes writes to objects; a dedicated proof rule is developed to help prove prove program invariants hold under this model. By parameterizing our proof system over a gamut of weak isolation specifications, we avoid the need to define a separate proof rule for each new isolation level we may encounter.

Inference. [Vafeiadis 2010; Vafeiadis, Viktor 2010] describe *action inference*, an inference procedure for computing rely and guarantee relations in the context of RGSep [Vafeiadis, Viktor and Parkinson, Matthew 2007], an integration of rely-guarantee and separation logic [Reynolds 2002] that allows one to precisely reason about local and shared state of a concurrent program. The ideas underlying action inference have been used to prove memory safety, linearizability, shape invariant inference, etc. of fine-grained concurrent data structures. While our motivation is similar (automated inference of intermediate assertions and local invariants), the context of study (transactions vs. shared-memory concurrency), the objects being analyzed (relational database tables vs. concurrent data structures), the properties being verified (integrity constraints over relational tables vs. memory safety, or linearizability of concurrent data structure operations) and the analysis technique used to drive inference (state transformers vs. abstract interpretation) are quite different.

9 CONCLUSIONS

To improve performance, modern database systems employ techniques that weaken the strong isolation guarantees provided by serializable transactions in favor of alternatives that allow a transaction to witness the effects of other concurrently executing transactions that happen commit during its execution. Typically, it is the responsibility of the database programmer to determine if an available weak isolation level would violate a transaction's consistency constraints. Although this has proven to be a difficult and error-prone process, there has heretofore been no attempt to formalize notions of weak isolation with respect to application semantics, or consider how we might verify the correctness of database programs that use weakly-isolated transactions. In this paper, we provide such a formalization. We develop a rely-guarantee proof framework cognizant of weak isolation semantics, and build on this foundation to devise an inference procedure that facilitates automated verification of weakly-isolated transactions, and have applied our ideas on widely-used database systems to justify their utility. Our solution enables database applications to leverage the performance advantages offered by weak isolation, without compromising correctness.

ACKNOWLEDGEMENTS

We thank KC Sivaramakrishnan for numerous helpful discussions about weak isolation, and for thorough analysis of the material presented in this paper. We are grateful to the anonymous reviewers, and our shepherd, Peter Müller, for their careful reading and insightful comments. This material is based upon work supported by the National Science Foundation under Grant No. CCF-SHF 1717741 and the Air Force Research Lab under Grant No. FA8750-17-1-0006.

REFERENCES

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

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

Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13).* ACM, New York, NY, USA, 1185–1196. https://doi.org/10.1145/2463676.2465296

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013a. Highly Available Transactions: Virtues and Limitations. *PVLDB* 7, 3 (2013), 181–192.

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

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15).* ACM, New York, NY, USA, 1327–1342. https://doi.org/10.1145/2723372.2737784

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013b. HAT, Not CAP: Towards Highly Available Transactions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS'13).* USENIX Association, Berkeley, CA, USA, 24–24. http://dl.acm.org/citation.cfm?id=2490483.2490507

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

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

Arthur J. Bernstein, Philip M. Lewis, and Shiyong Lu. 2000. Semantic Conditions for Correctness at Different Isolation Levels. In *Proceedings of the 16th International Conference on Data Engineering (ICDE '00).* IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=846219.847381

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

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

Bitcoin Bug 2016. How I Stole Roughly 100 BTC From an Exchange and How I Could Have Stolen More! https://goo.gl/4SqaP2

Ergon Börger, Erich Grädel, and Yuri Gurevich. 1996. *The Classical Decision Problem.* Springer-Verlag Telos.

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

Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08).* ACM, New York, NY, USA, 729–738. https://doi.org/10.1145/1376616.1376690

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015) (Leibniz International Proceedings in Informatics (LIPIcs)),* Luca Aceto and David de Frutos Escrig (Eds.), Vol. 42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

Andrea Cerone and Alexey Gotsman. 2016. Analysing Snapshot Isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC).*

Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Conference on Principles of Distributed Computing (PODC).* 73–82.

Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. 1985. Consistency in a Partitioned Network: A Survey. *ACM Comput. Surv.* 17, 3 (Sept. 1985), 341–370. https://doi.org/10.1145/5505.5508

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

Alan Fekete, Shirley N. Goldrei, and Jorge Pérez Asenjo. 2009. Quantifying Isolation Anomalies. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 467–478. https://doi.org/10.14778/1687627.1687681

Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.* 30, 2 (June 2005), 492–528. https://doi.org/10.1145/1071610.1071615

Peter Gammie, Antony L. Hosking, and Kai Engelhardt. 2015. Relaxing Safely: Verified On-the-fly Garbage Collection for x86-TSO. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 99–109. https://doi.org/10.1145/2737924.2738006

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.

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

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. 365–394.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Computer Aided Verification: 27th International Conference*. Springer International Publishing, 449–465. https://doi.org/10.1007/978-3-319-21668-3_26

C. B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 596–619. https://doi.org/10.1145/69575.69577

Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2018. Alone Together: Compositional Reasoning and Inference for Weak Isolation. https://arxiv.org/abs/1710.09844.

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

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

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

MySQL 2016. Transaction Isolation Levels. https://dev.mysql.com/doc/refman/5.6/en/innodb-transaction-isolation-levels.html Accessed: 2016-07-1 10:00:00.

Oracle 2016. Data Concurrency and Consistency. https://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm Accessed: 2016-07-1 10:00:00.

Poloniex Bug 2016. BTC Stolen from Poloniex. https://bitcointalk.org/index.php?topic=499580

PostgreSQL 2016. Transaction Isolation. https://www.postgresql.org/docs/9.1/static/transaction-iso.html Accessed: 2016-07-1 10:00:00.

Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil. 2011. Precisely Serializable Snapshot Isolation (PSSI). In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 482–493. https://doi.org/10.1109/ICDE.2011.5767853

J C Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17^{th} Annual IEEE Symposium on Logic in Computer Science*. IEEE Comput. Soc, 55–74.

SciMed Bug 2016. Avoid Race Conditions that Violate Uniqueness Validation - Rails. http://goo.gl/0QhMQj

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

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

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 (SOSP '11)*. ACM, New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

Starbucks Bug 2016. Hacking Starbucks for unlimited coffee. http://sakurity.com/blog/2015/05/21/starbucks.html

Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Proceedings of the 22^{nd} International Conference on Computer Aided Verification (CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

Vafeiadis, Viktor. 2010. RGSep Action Inference. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*. 345–361.

Vafeiadis, Viktor and Parkinson, Matthew. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*. Springer Berlin Heidelberg, Berlin, Heidelberg, 256–271.

Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 5–20. https://doi.org/10.1145/3035918.3064037

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