Detecting Robustness against MVRC for Transaction Programs with Predicate Reads

Brecht Vandevoort
UHasselt, Data Science Institute, ACSL
Belgium

Bas Ketsman
Vrije Universiteit Brussel
Belgium

Christoph Koch
École Polytechnique Fédérale de Lausanne
Switzerland

Frank Neven
UHasselt, Data Science Institute, ACSL
Belgium

ABSTRACT

The transactional robustness problem revolves around deciding whether, for a given workload, a lower isolation level than Serializable is sufficient to guarantee serializability. The paper presents a new characterization for robustness against isolation level (multi-version) Read Committed. It supports transaction programs with control structures (loops and conditionals) and inserts, deletes, and predicate reads – scenarios that trigger the phantom problem, which is known to be hard to analyze in this context. The characterization is graph-theoretic and not unlike previous decision mechanisms known from the concurrency control literature that database researchers and practicians are comfortable with. We show experimentally that our characterization pushes the frontier in allowing to recognize more and more complex workloads as robust than before.

1 INTRODUCTION

The gold standard for desirable transactional semantics is serializability, and much research and technology development has gone into creating systems that provide the greatest possible transaction throughput. Nevertheless, in practice, a hierarchy of alternative isolation levels of different strengths is available, allowing users to trade off semantic guarantees for better performance. One example is the isolation level (multi-version) Read Committed (MVRC), which does not guarantee serializability but which can be implemented more efficiently than isolation level Serializable. The central question that we address in this paper is: *When is it safe to run a transactional workload under MVRC?*

Recently, a number of researchers have studied the so-called transactional robustness problem [2, 3, 5, 6, 8, 11, 14, 19, 20, 44, 45], which revolves around deciding whether, for a given workload, a lower isolation level than Serializable is sufficient to guarantee serializability. Specifically, a set of transactions is called robust against a given isolation level if every possible interleaving of the transactions under consideration that is allowed under the specified isolation level is serializable. That there is a real chance that nontrivially robust workloads do exist is probably best demonstrated by the fact that the well-known benchmark TPC-C is robust for Snapshot Isolation [20].

Robustness is a static property of workloads involving an offline analysis. A workload (the set of transaction programs at the application level) is analyzed by its developers during development time, and the insight into its robustness for a given low isolation level is later used to consistently deploy it with a database server using a specific isolation level weaker than serializable. Robustness is a hard problem and undecidability is reached quite quickly [45]. For exact characterizations of robustness, the possibility of phantom problem anomalies makes the problem very difficult, and, typically, research on the robustness problem has excluded insertions, deletions, and predicate reads [19, 26, 44, 45], in addition to assuming that transaction programs are linear sequences of reads and writes without any control structures.

To allow for the inclusion of predicate reads, sound robustness tests based on sufficient conditions have been developed [3, 8, 11, 12, 20]. Such conditions are based on the following observation. When a *schedule* is not serializable, then the serialization graph constructed from that schedule contains a cycle satisfying a property specific to the isolation level under consideration: *dangerous structure* [20] for SNAPSHOT ISOLATION and the presence of a counterflow edge for MVRC [3]. This approach is extended to a workload of *transaction programs* via a so-called static dependency graph summarizing all possible serialization graphs for all possible executions allowed under the isolation level at hand. In this static dependency graph, each program is represented by a node, and there is a conflict edge from one program to another if there can be a schedule that gives rise to that conflict. The absence of a cycle satisfying the condition specific to that isolation level then guarantees robustness, while the presence of a cycle does not necessarily imply non-robustness. Indeed, every counterexample cycle in a serialization graph is witnessed by a cycle in this static dependency graph, but the converse is not necessarily true. A major obstacle preventing direct application to practical workloads is that the construction of the static dependency graph is a manual step that should be performed by a database specialist. This is a difficult problem as the decision to place an edge requires reasoning over all possible schedules. In this paper, we build further upon the just mentioned line of work by (i) identifying a more specific condition that holds for all cycles found in the serialization graph of a schedule allowed under MVRC, thereby allowing to identify more workloads as robust against MVRC, and (ii), by providing a more formal approach to construct these static dependency graphs, thereby facilitating automatic construction for a given set of transaction programs.

In this paper, we study the robustness problem for MVRC and obtain a sound robustness detection algorithm that improves over the state-of-the-art in that it (i) can detect larger sets of transaction programs to be robust; (ii) incorporates operations like insert, delete and predicate reads that, to the best of our knowledge, have not been considered before thereby, allowing to verify robustness for a wider range of workloads, including for example TPC-C; and, (iii) can readily be implemented and applied in practice as the static dependency graph (called summary

Series ISSN: 2367-2005

10.48786/edbt.2023.48

graph in this work) can be automatically constructed based on a formalization of transaction programs, called BTP. The precise formalisation facilitates the applicability to any kind of transaction programs consisting of operations for which the following information can be derived (when applicable): type of operation, set of observed and modified attributes, set of attributes used in a predicate read, and implied foreign key constraints. In other words, our techniques require only this information, and do not need to keep and analyze intermediate representations of the transaction program code.

Outline and contributions. To make the paper more readable, we introduce the main ideas behind our formalisation and the algorithm by means of a running example in Section 2 before introducing the necessary definitions in Section 3. In Section 4, we show that when a schedule allowed under MVRC is not serializable, then it must contain a cycle satisfying a certain condition (Theorem 4.2). This improves over the graph-based condition presented in [3]. In Section 5, we introduce the formalism of basic transaction programs (BTPs) incorporating inserts, deletes, predicate reads and control structure. In Section 6, we provide algorithms for constructing the summary graph (Algorithm 1) and testing robustness (Algorithm 2) based on the sufficient condition obtained in Section 4. We show through experiments in Section 7 on two well known transaction benchmarks, TPC-C and Smallbank, that our approach detects strictly more sets of programs as robust compared to earlier work [3]. We furthermore introduce a new synthetic benchmark where the number of programs is parameterized. Based on this benchmark, we show that our algorithm scales to larger sets of programs as well and can test for robustness in a matter of seconds. We discuss related work in Section 8 and conclude in Section 9. Missing proofs can be found in [46].

2 RUNNING EXAMPLE

To illustrate our approach, we introduce a running example based on an auction service. The database schema consists of three relations: Buyer(id, calls), Bids(buyerId, bid), and Log(id, buyerId, bid), where the primary key for each relation is underlined and buyerId in Bids and Log is a foreign key referencing Buyer(id). The relation Buyer lists all potential buyers, Bids keeps track of the current bid for each potential buyer, and Log keeps a register of all bids. Each buyer can interact with the auction service through API calls. For logging purposes, the attribute Buyer(calls) counts the total number of calls made by the buyer. The API interacts with the database via two transaction programs: FindBids(B, T) and PlaceBid(B, V) whose SQL code is given in Figure 1. FindBids returns all current bids above threshold T, whereas PlaceBids increases the bid of buyer B to value V (if V is higher than the current bid, otherwise the current bid remains unchanged) and inserts this newly placed bid as a new tuple in Log. Both programs increment the number of calls for B.

Basic Transaction Programs. We introduce the formalism of *basic transaction programs* (BTP) to overestimate the set of schedules that can arise when executing transaction programs as given in Figure 1. A BTP is a sequence of statements that only retains the information necessary to detect robustness against MVRC: the type of statement (insert, key-based selection/update/delete, or predicate-based selection/update/delete), the relation that is referred to, and the attributes that are read from, written to, and that are used in predicates. In particular, BTPs ignore the concrete predicate selection condition.

```
FindBids(:B, :T):
  UPDATE Buyer --q1
  SET calls = calls + 1
  WHERE id = :B;

  SELECT bid --q2
  FROM Bids
  WHERE bid >= :T;

  COMMIT;
```

```
PlaceBid(:B, :V):
  UPDATE Buyer --q3
  SET calls = calls + 1
  WHERE id = :B;

  SELECT bid into :C --q4
  FROM Bids
  WHERE buyerId = :B;

  IF :C < :V: --q5
    UPDATE Bids
    SET bid = :V
    WHERE buyerId = :B;
  ENDIF;

  :logId = uniqueLogId();

  INSERT INTO Log --q6
  VALUES(:logId, :B, :V);

  COMMIT;
```

Auction schema
Buyer(id,calls)
Bids(buyerId, bid)
Log(id,buyerId,bid)
Foreign keys
f_1: Bids(BuyerId) → Buyer(id)
f_2: Log(BuyerId) → Buyer(id)

	BTP
FindBids	$q_1; q_2$
PlaceBid	$q_3; q_4; (q_5 \mid \varepsilon); q_6$

Figure 1: Auction schema, SQL code and BTP formalization for FindBids(B, T) and PlaceBid(B, V)

q	type(q)	rel(q)	PReadSet(q)	ReadSet(q)	WriteSet(q)
FindBids					
q_1	key upd	Buyer	⊥	{calls}	{calls}
q_2	pred sel	Bids	{bid}	{bid}	⊥
PlaceBid					
q_3	key upd	Buyer	⊥	{calls}	{calls}
q_4	key sel	Bids	⊥	{bid}	⊥
q_5	key upd	Bids	⊥	{}	{bid}
q_6	ins	Log	⊥	⊥	{id, buyerId, bid}

Figure 2: Query details for BTPs FindBids and PlaceBid.

Formally, a BTP is a sequence of statements $q_1; \ldots; q_k$. For example, FindBids is modeled by $q_1; q_2$, where q_1 and q_2 are two statements reflecting the corresponding SQL statements in Figure 1. Each statement q_i is supplemented with additional information as detailed in Figure 2. There, type(q_i) refers to the type of statement: an insert, a key-based or predicate-based selection, update or delete; rel(q_i) is the relation under consideration; ReadSet(q_i) are the attributes read by q_i; WriteSet(q_i) those written by q_i; and, PReadSet(q_i) the attributes used for predicates in the WHERE part of the query. We use ⊥ to indicate that a specific function is not applicable to a statement. For example, q_1 in FindBids is a key-based update over relation Buyer, since the corresponding SQL query selects exactly one tuple based on the primary key attribute Buyer(id). This statement reads and then overwrites the value for attribute Buyer(calls), and therefore ReadSet(q_1) = WriteSet(q_1) = {calls}. Since this statement is not predicate-based, we have PReadSet(q_1) = ⊥. Statement q_2 is a predicate-based selection over relation Bids. The predicate `id = :B` in the corresponding SQL statement only uses the attribute Bids(bid), and therefore PReadSet(q_2) = {bid}. Therefore, ReadSet(q_2) = {bid}.

BTPs incorporate conditional branching and loops as well. Indeed, PlaceBid is modeled by $q_3; q_4; (q_5 \mid \varepsilon); q_6$ supplemented with additional information as depicted in Figure 2. Here, $(q_5 \mid \varepsilon)$ denotes the branching corresponding to the IF-statement in the SQL program: either q_5 is executed (if the condition in the SQL program evaluates to true), or nothing is executed (if the condition evaluates to false). We note that an ELSE-clauses can be modeled by replacing ε by a corresponding statement. Analogously, BTPs allow loop(P) to express iteration, where P is an arbitrary sequence of statements. Intuitively, loop(P) specifies that P can be repeated for an arbitrary yet finite number of iterations. We refer to Section 5 for a formal definition of BTPs.

A set of transaction programs $\mathcal{P}$ induces an infinite set of possible schedules where each transaction in the schedule is an instantiation of a program in $\mathcal{P}$ as informally explained next by means of an example. We refer to Section 5 for a formal treatment. Consider the schedule s over transactions T_1, T_2 and T_3 presented in Figure 3. Here, T_1 and T_2 are instantiations of PlaceBid and T_3 is an instantiation of FindBids (when considered as a BTP). Furthermore, $\mathtt{t}_1$ and $\mathtt{t}_2$ are tuples of relation Buyer, $\mathtt{u}_1$, $\mathtt{u}_2$ and $\mathtt{u}_3$ are tuples of Bids, and $\mathtt{l}_1$ and $\mathtt{l}_2$ are tuples of Log. The operation $\mathtt{R}_1[\mathtt{t}_1]$ (respectively $\mathtt{W}_1[\mathtt{t}_1]$) indicates that transaction T_1 reads (respectively writes to) tuple $\mathtt{t}_1$, and operation $\mathtt{I}_1[\mathtt{l}_1]$ indicates that T_1 inserts a new tuple $\mathtt{l}_1$ into the database. The operation $\mathtt{PR}_3[\text{Bids}]$ in T_3 is a predicate read that evaluates a predicate over all tuples in relation Bids.

Figure 3 further illustrates how each statement in a BTP leads to one or more operations over tuples. For example, the key-based update q_3 in PlaceBid results in two operations $\mathtt{R}_1[\mathtt{t}_1]$ and $\mathtt{W}_1[\mathtt{t}_1]$. Notice in particular that these two operations are over the same tuple $\mathtt{t}_1$ of relation Buyer = rel(q_3), where the first operation reads the value for attribute Buyer(calls) and the second operation overwrites the value for this attribute, as indicated by ReadSet(q_3) and WriteSet(q_3). The predicate-based selection statement q_2 of FindBids results in a larger number of operations in T_3. First, the predicate read $\mathtt{PR}_3[\text{Bids}]$ evaluates a predicate over all tuples in Bids = rel(q_2), where only attribute Bids(bid) is used in the predicate, indicated by PReadSet(q_2). This predicate intuitively corresponds to the WHERE clause of the corresponding SQL statement, but in our formalism, we will only specify the attributes needed in the predicate rather than the predicate itself. Then, T_3 reads three tuples of relation Bids. For each such tuple, only the value of attribute Bids(Bid) is read, as specified by ReadSet(q_2). Also notice how T_1 is an instantiation of PlaceBid where the if-condition evaluates to false, whereas for T_2 it evaluates to true, witnessed by the presence of q_5 in T_2 and its absence in T_1.

Foreign Keys. Schedules should respect foreign keys. Two instantiations of PlaceBid that access the same tuple $\mathtt{t}_1$ of relation Bids also need to access the same Buyer $\mathtt{u}_1$ as Bids(buyerId) is a foreign key referencing Buyer(Id). Such information can be used to rule out inadmissible schedules (that could otherwise inadvertently cause a set of transaction programs to not be robust). For example, the schedule s' obtained from s by substituting $\mathtt{t}_1$ with $\mathtt{t}_2$ in T_1 violates the foreign key constraint and is therefore not admissible. We refer to Section 5 for a more formal treatment of how we handle foreign keys in BTPs.

mvrc, Dependencies and Conflict Serializability. When a database is operating under isolation level Multiversion Read Committed (mvrc), each read operation reads the most recently committed version of a tuple, and write operations cannot overwrite uncommitted changes. For example, under the assumption that s in Figure 3 is allowed under mvrc, $\mathtt{R}_2[\mathtt{t}_1]$ will observe the version of $\mathtt{t}_1$ written by $\mathtt{W}_1[\mathtt{t}_1]$, as T_1 committed before $\mathtt{R}_2[\mathtt{t}_1]$. Read operation $\mathtt{R}_3[\mathtt{u}_1]$ on the other hand will not see the changes made by $\mathtt{W}_2[\mathtt{u}_1]$, as the commit of T_2 occurs after $\mathtt{R}_3[\mathtt{u}_1]$.

We say that two operations occurring in two different transactions are conflicting if they are over the same tuple, access a common attribute of this tuple, and at least one of these two operations overwrites the value for this common attribute. These conflicts introduce dependencies between operations. For example, $\mathtt{W}_1[\mathtt{t}_1]$ in T_1 and $\mathtt{R}_2[\mathtt{t}_1]$ in T_2 are conflicting, as the former modifies the value for attribute Buyer(calls) and the latter reads this value. We therefore say that there is a wr-dependency from $\mathtt{W}_1[\mathtt{t}_1]$ to $\mathtt{R}_2[\mathtt{t}_1]$, denoted by $\mathtt{W}_1[\mathtt{t}_1] \to_s \mathtt{R}_2[\mathtt{t}_1]$. Similarly, since we assume that s is allowed under mvrc, $\mathtt{R}_3[\mathtt{u}_1]$ observes a version of $\mathtt{u}_1$ before the changes made by $\mathtt{W}_2[\mathtt{u}_1]$. We therefore say that there is an rw-antidependency from $\mathtt{R}_3[\mathtt{u}_1]$ to $\mathtt{W}_2[\mathtt{u}_1]$, denoted by $\mathtt{R}_3[\mathtt{u}_1] \to_s \mathtt{W}_2[\mathtt{u}_1]$. The serialization graph $SeG(s)$ contains transactions as nodes and edges correspond to dependencies. It is well-known that a schedule is conflict serializable if there is no cycle in $SeG(s)$. A more formal definition of dependencies, conflict serializability and mvrc can be found in Section 3.

A dependency from a transaction T_i to a transaction T_j is counterflow if T_j commits before T_i (that is, the direction of the dependency is opposite to the commit order). In our running example, the dependency $\mathtt{R}_3[\mathtt{u}_1] \to_s \mathtt{W}_2[\mathtt{u}_1]$ is a counterflow dependency, as T_3 commits after T_2. Alomari and Fekete [3] showed that if a schedule is allowed under mvrc, then every cycle in the serialization graph contains at least one counterflow dependency. We refer to cycles containing at least one counterflow dependency as a type-I cycle. In Theorem 4.2, we refine this condition and show that every such cycle must either contain an adjacent-counterflow pair or an ordered-counterflow pair, as well as a non-counterflow dependency, and refer to the latter as a type-II cycle (formal definitions are given in Section 4). As every type-II cycle is a type-I cycle but not vice-versa, this refinement will allow us to identify larger sets of programs to be robust against mvrc. In Section 7, we will show that our approach indeed leads to practical improvements for all considered benchmarks.

Linear Transaction Programs. We refer to BTPs without branching and loops as linear transaction programs (LTP). For each BTP an equivalent set of LTPs can be derived by unfolding all branching statements and loops. FindBids is also an LTP and PlaceBid can be unfolded into two LTPs PlaceBid$_1$:= $q_3; q_4; q_5; q_6$ and PlaceBid$_2$:= $q_3; q_4; q_6$. Loop unfolding gives rise to an infinite number of LTPs. However, we will show that for detecting robustness against mvrc it suffices to limit loop unfoldings to at most two iterations.

Detecting Robustness against mvrc. A set $\mathcal{P}$ of LTPs is robust against mvrc if every allowed schedule is serializable. We therefore lift the just mentioned condition from serialization graphs to summary graphs. The summary graph $SuG(\mathcal{P})$ summarizes all serialization graphs for all possible schedules allowed under mvrc over transactions instantiated from programs in $\mathcal{P}$. Here, nodes in $SuG(\mathcal{P})$ are programs in $\mathcal{P}$ and if a schedule allowed under mvrc exists with a dependency $b_i \to a_j$, then an edge is added from P_i to P_j where b_i is an operation in transaction T_i instantiated from a program $P_i \in \mathcal{P}$ and a_j is an operation in transaction T_j instantiated from $P_j \in \mathcal{P}$. That edge is annotated with statements P_i and P_j and is dashed when the dependency is counterflow. The summary graph for the three LTPs FindBids, PlaceBid$_1$ and PlaceBid$_2$ is visualized in Figure 4. If we consider for example the dependency $\mathtt{W}_1[\mathtt{t}_1] \to_s \mathtt{R}_2[\mathtt{t}_1]$, we see that $SuG(\mathcal{P})$ has a corresponding edge from PlaceBid$_2$ to PlaceBid$_1$, labeled with q_3 and q_3. Analogously, the counterflow dependency $\mathtt{R}_3[\mathtt{u}_1] \to_s \mathtt{W}_2[\mathtt{u}_1]$ is witnessed by the counterflow edge from FindBids to PlaceBid$_1$ in $SuG(\mathcal{P})$. We present a formal algorithm constructing the graph $SuG(\mathcal{P})$ for a given set of LTPs in Section 6.2.

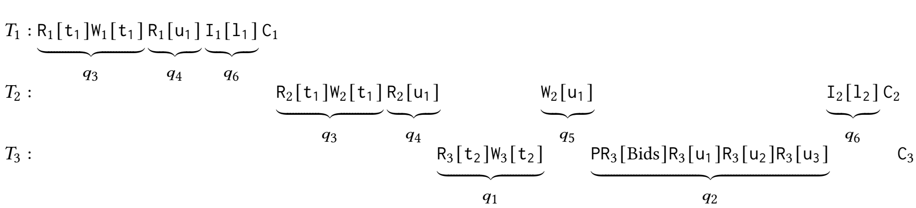
T_1 : $\underbrace{R_1[t_1]W_1[t_1]}_{q_3}\underbrace{R_1[u_1]}_{q_4}\underbrace{I_1[l_1]}_{q_6}C_1$
T_2 : $\underbrace{R_2[t_1]W_2[t_1]}_{q_3}\underbrace{R_2[u_1]}_{q_4}$ $\underbrace{W_2[u_1]}_{q_5}$ $\underbrace{I_2[l_2]}_{q_6}C_2$
T_3 : $\underbrace{R_3[t_2]W_3[t_2]}_{q_1}$ $\underbrace{PR_3[Bids]R_3[u_1]R_3[u_2]R_3[u_3]}_{q_2}$ C_3

Figure 3: Example schedule s where T_1 and T_2 are instantiations of PlaceBid and T_3 is an instantiation of FindBids.

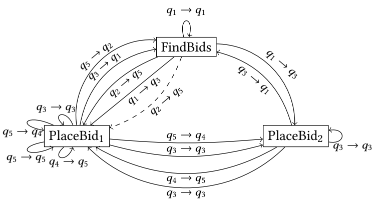

Figure 4: Summary graph containing a type-I but no type-II cycles.

Let s be an arbitrary schedule allowed under mvrc where transactions are instantiations of $\mathcal{P}$. As each dependency in the serialization graph $SeG(s)$ is witnessed by an edge in the summary graph $SuG(\mathcal{P})$, it immediately follows that each cycle in $SeG(s)$ is witnessed by a cycle in $SuG(\mathcal{P})$. So, when $SuG(\mathcal{P})$ does not contain a type-II cycle, we can safely conclude that $\mathcal{P}$ is robust against mvrc. Indeed, the absence of such cycles indicates (by Theorem 4.2) that no schedule allowed under mvrc exists with a cycle in its serialization graph, implying that every such schedule is serializable. The presence of a type-II cycle does not necessarily imply non-robustness as there might not be a single schedule in which the corresponding cycle is realized. However, in that case, the conservative approach is to attest non-robustness to avoid false positives. Algorithm 2 follows this conservative approach and determines $\mathcal{P}$ to be robust iff $SuG(\mathcal{P})$ does not contain a type-II cycle.

We show in Section 6 the summary graph in Figure 4 does not contain a type-II cycle. The set {FindBids, PlaceBid} is therefore identified by Algorithm 2 as robust against mvrc. The SQL programs presented in Figure 1 can thus be safely executed under isolation level mvrc, without risking non-serializable behavior. This improves over earlier work, as the summary graph does contain a type-I cycle (e.g., between FindBids and PlaceBid$_1$), and, hence, the method of [3] can not identify {FindBids, PlaceBid} as robust.

3 DEFINITIONS

Our formalization of transactions and conflict serializability is closely related to the formalization presented by Adya et al. [1]. We extend upon the definitions presented in [44] and include three additional types of operations: predicate reads, inserts and deletes.

3.1 Databases

A *relational schema* is a pair (Rels, FKeys), where Rels is a set of relation names and FKeys is a set of foreign keys. Then, Attr(R) denotes the finite set of attribute names. We fix an infinite set $I(R)$ of abstract objects called tuples, for each $R \in$ Rels. We assume that $I(R) \cap I(S) = \emptyset$ for all $R, S \in$ Rels with $R \neq S$. When $t \in I(R)$, we say that t is of *type R* and denote the latter by rel(t) = R. We often refer to tuples t without mentioning their type, in which case the definition implies there is a unique relation $R \in$ Rels with $\mathtt{t} \in I(R)$.

We associate to t an infinite set $V(\mathtt{t})$ that conceptually represents the different versions that are created when t is changed over time. We require that $V(\mathtt{t}) \cap V(\mathtt{u}) = \emptyset$ for all tuples $\mathtt{t} \neq \mathtt{u}$. Each set $V(\mathtt{t})$ contains two special versions that we refer to as the *unborn* and *dead* version. We refer to all other versions as *visible* versions. Intuitively, the unborn version represents the state of t before it is inserted in the database, the dead version represents the state after the tuple is deleted, and the visible versions are the versions of t that can be read by transactions. For a foreign key $f \in$ FKeys, $dom(f) \in$ Rels and $range(f) \in$ Rels denote the associated domain and range of f, and f itself is a mapping associating each tuple $\mathtt{t} \in I(dom(f))$ to a tuple in $f(\mathtt{t}) \in I(range(f))$.

3.2 Operations over Tuples and Relations

For a tuple t, we distinguish four operations R[t], W[t], I[t] and D[t], denoting that t is read, written, inserted or deleted, respectively, and say that the operation is on the tuple t. We also assume a special *commit* operation denoted by C. We will use the following terminology: a *read operation* is an R[t], and a *write operation* is a W[t], an I[t] or a D[t]. Furthermore, an R-operation is an R[t], a W-operation is a W[t], an I-operation is an I[t], and a D-operation is a D[t]. To every operation o on a tuple of type R, we associate a set of attributes Attr(o) ⊆ Attr(R) to denote the attributes that o reads from or writes to. Furthermore, when o is an I-operation or a D-operation then Attr(o) = Attr(R).

For a relation $R \in$ Rels, a predicate read PR[R] is an operation that evaluates a predicate over each tuple of type R, and Attr(PR[R]) ⊆ Attr(R) contains the set of attributes over which the predicate is evaluated.

3.3 Transactions and Schedules

For $i, j \in \mathbb{N}$ with $i \leq j$, denote by $[i, j]$ the set $\{i, \ldots, j\}$.

A *transaction T* is a sequence of read and write operations on tuples, as well as predicate read operations on relations in Rels, followed by a special commit operation denoted by C. Formally, we model a transaction as a linear order $(T, \leq_T)$, where T is the set of (read, write, predicate read and commit) operations occurring in the transaction and $\leq_T$ encodes the ordering of the operations. As usual, we use $<_T$ to denote the strict ordering. Throughout the paper, we interchangeably consider transactions both as linear orders as well as sequences.

Let a and b be two operations in a transaction T with $a \leq_T b$. An *atomic chunk* (a, b) represents a sequence of operations that cannot be interleaved by other concurrent transactions. Formally, an atomic chunk is a pair (a, b) that denotes the restriction of T to all operations o with $a \leq_T o \leq_T b$. In this paper, we only consider chunks encapsulating specific sequences of operations:

- *key-based update*: R[t]W[t] with rel(t) = R;
- *predicate-based selection*: PR[R]R[t$_1$] . . . R[t$_n$] for an arbitrary number of tuples t$_i$ with rel(t$_i$) = R;
- *predicate-based update*: PR[R]R[t$_1$]W[t$_1$] . . . R[t$_n$]W[t$_n$] for an arbitrary number of tuples t$_i$ with rel(t$_i$) = R; and
- *predicate-based deletion*: PR[R]D[t$_1$] . . . D[t$_n$] for an arbitrary number of tuples t$_i$ with rel(t$_i$) = R.

We refer to Section 5.4 for a discussion on the assumptions we make on a DBMS (including chunks). We denote by $Chunks(T)$ the set of atomic chunks associated to T. For instance, the transactions in Figure 3 have the following chunks: $Chunks(T_1)$ = {(R$_1$[t$_1$], W$_1$[t$_1$])}, $Chunks(T_2)$ = {(R$_2$[t$_1$], W$_2$[t$_1$])}, and $Chunks(T_3)$ = {(R$_3$[t$_2$], W$_3$[t$_2$]), (PR$_3$[Bids], R$_3$[u$_3$])}.

When considering a set $\mathcal{T}$ of transactions, we assume that every transaction in the set has a unique id i and write T_i to make this id explicit. Similarly, to distinguish the operations from different transactions, we add this id as an index to the operation. That is, we write $\mathtt{W}_i[\mathtt{t}]$, $\mathtt{R}_i[\mathtt{t}]$, $\mathtt{I}_i[\mathtt{t}]$ and $\mathtt{D}_i[\mathtt{t}]$ to denote respectively a write operation, read operation, insert or delete on tuple $\mathtt{t}$ occurring in transaction T_i; similarly, $\mathtt{PR}_i[R]$ denotes a predicate read on relation R in transaction T_i and $\mathtt{C}_i$ denotes the commit operation in transaction T_i. This convention is consistent with the literature (see, *e.g.* [7, 19]). To avoid ambiguity of notation, we assume that a transaction performs at most one read operation and at most one write operation per tuple. The latter is a common assumption (see, *e.g.* [19]). All our results carry over to the more general setting in which multiple writes and reads per tuple are allowed.

A *(multiversion) schedule* s over a set $\mathcal{T}$ of transactions is a tuple $(O_s, \leq_s, \mathit{init}_s, v_s^w, v_s^r, \mathit{Vset}_s, \ll_s)$ where *(i)* O_s is the set containing all operations of transactions in $\mathcal{T}$; *(ii)* $\leq_s$ encodes the ordering of these operations; *(iii)* init_s is the *initial version set* associating each tuple $\mathtt{t}$ to a version $\mathit{init}_s(\mathtt{t}) \in V(\mathtt{t})$ which is either the unborn or any visible version of $\mathtt{t}$; *(iv)* v_s^w is a *write version function* mapping each write operation over a tuple $\mathtt{t}$ in O_s to the version in $V(\mathtt{t})$ that this operation created; *(v)* v_s^r is a *read version function* mapping each read operation over a tuple $\mathtt{t}$ in O_s to the version in $V(\mathtt{t})$ that this operation observed; *(vi)* Vset_s is a function mapping each predicate read operation $a \in O_s$ to a *version set* containing the version of each tuple that is observed by a, or, more formally, for each tuple $\mathtt{t} \in I(R)$ a version in $V(\mathtt{t})$ where a is over a relation R; *(vi)* $\ll_s$ is a *version order* providing for each tuple $\mathtt{t}$ a total order over all the versions in $V(\mathtt{t})$ with the unborn and dead version of $\mathtt{t}$ being respectively the first and last version according to $\ll_s$ for $\mathtt{t}$.

We furthermore require that

- the order of operations in s is consistent with the order of operations in every transaction $T \in \mathcal{T}$. That is, $a <_T b$ implies $a <_s b$ for every $T \in \mathcal{T}$ and every $a, b \in T$;
- atomic chunks are not interleaved by operations of other transactions. That is, for every $T_i \in \mathcal{T}$ and for each atomic chunk $(a_i, b_i) \in \mathit{Chunks}(T_i)$, there is no operation c with $a_i <_s c <_s b_i$ and $c \notin T_i$;
- each write operation creates a version that is newer (according to $\ll_s$) than the initial version and that is different from versions created by other write operations. Furthermore, $\mathtt{D}$-operations always create the dead version for a tuple. Formally, for each write operation $a \in O_s$ over a tuple $\mathtt{t}$, we have $\mathit{init}_s(\mathtt{t}) \ll_s v_s^w(a)$ and there is no other write operation $b \in O_s$ over $\mathtt{t}$ with $v_s^w(a) = v_s^w(b)$. If a is a $\mathtt{D}$-operation, then $v_s^w(a)$ is the dead version;
- read and predicate read operations always observe visible versions of tuples that are already installed. That is, for each read and predicate read operation $a \in O_s$, the read version v of a tuple t (being either $v_s^r(a)$ or as defined by $\mathit{Vset}_s(a)$) is visible and either equals $\mathit{init}_s(\mathtt{t})$ or there is a write operation $b \in O_s$ over $\mathtt{t}$ with $b <_s a$ and $v = v_s^w(b)$.
- an operation creates the first visible version of a tuple if and only if it is an $\mathtt{I}$-operation. Formally, for each write operation $a \in O_s$ over a tuple $\mathtt{t}$, a is an $\mathtt{I}$-operation if and only if there is no other write operation $b \in O_s$ over $\mathtt{t}$ with $v_s^w(b) \ll_s v_s^w(a)$ and $\mathit{init}_s(\mathtt{t})$ is the unborn version.

Notice that it follows immediately from these requirements that there can be at most one $\mathtt{I}$-operation and at most one $\mathtt{D}$-operation in O_s over each tuple.

A schedule s is a *single version schedule* if versions are installed in the order that they are written and every (predicate) read operation always observes the most recent version of all relevant tuples. Formally, *(i)* for each pair of write operations a and b on the same tuple, $v_s^w(a) \ll_s v_s^w(b)$ iff $a <_s b$; *(ii)* for every read operation a there is no write operation c on the same tuple as a with $c <_s a$ and $v_s^r(a) \ll_s v_s^w(c)$; and, *(iii)* for every predicate read operation a over a relation R and tuple $\mathtt{t}$ of type R there is no write operation c on $\mathtt{t}$ with $c <_s a$ and $\mathtt{t}_i \ll_s v_s^w(c)$, with $\mathtt{t}_i$ the version of $\mathtt{t}$ in $\mathit{Vset}_s(a)$.

A serial schedule over a set of transactions $\mathcal{T}$ is a single version schedule in which operations from transactions are not interleaved with operations from other transactions. That is, for every $a, b, c \in O_s$ with $a <_s b <_s c$ and $a, c \in T$ implies $b \in T$ for every $T \in \mathcal{T}$.

The absence of aborts in our definition is consistent with the common assumption [8, 19] that an underlying recovery mechanism will roll back transactions that interfere with aborted transactions.

3.4 Conflict Serializability

Let a_j and b_i be two operations from different transactions T_j and T_i in a set of transactions $\mathcal{T}$. We say that a_j *depends on* b_i (or that there is a dependency from b_i to a_j) in a schedule s over $\mathcal{T}$, denoted $b_i \rightarrow_s a_j$ if one of the following holds:

- *(ww-dependency)* b_i and a_j are write operations on the same tuple with $\mathrm{Attr}(b_i) \cap \mathrm{Attr}(a_j) \neq \emptyset$ and $v_s^w(b_i) \ll_s v_s^w(a_j)$;
- *(wr-dependency)* b_i is a write operation and a_j is a read operation on the same tuple with $\mathrm{Attr}(b_i) \cap \mathrm{Attr}(a_j) \neq \emptyset$ and either $v_s^w(b_i) = v_s^r(a_j)$ or $v_s^w(b_i) \ll_s v_s^r(a_j)$;
- *(rw-antidependency)* b_i is a read operation and a_j is a write operation on the same tuple with $\mathrm{Attr}(b_i) \cap \mathrm{Attr}(a_j) \neq \emptyset$ and $v_s^r(b_i) \ll_s v_s^w(a_j)$;
- *(predicate wr-dependency)* b_i is a write operation on a tuple of type R, a_j is a predicate read on relation R, b_i is over a tuple $\mathtt{t}$ and $v_s^w(b_i) = \mathtt{t}_i$ or $v_s^w(b_i) \ll_s \mathtt{t}_i$ with $\mathtt{t}_i$ the version of $\mathtt{t}$ in $\mathit{Vset}_s(a_j)$, and if b_i is not an $\mathtt{I}$ or $\mathtt{D}$ operation, then $\mathrm{Attr}(b_i) \cap \mathrm{Attr}(a_j) \neq \emptyset$; or,
- *(predicate rw-antidependency)* b_i is a predicate read on a relation R, a_j is a write operation on a tuple of type R, a_j is over a tuple $\mathtt{t}$ and $\mathtt{t}_i \ll_s v_s^w(a_j)$ with $\mathtt{t}_i$ the version of $\mathtt{t}$ in $\mathit{Vset}_s(b_i)$, and if a_j is not an $\mathtt{I}$ or $\mathtt{D}$ operation, then $\mathrm{Attr}(b_i) \cap \mathrm{Attr}(a_j) \neq \emptyset$.

Intuitively, a ww-dependency from b_i to a_j implies that a_j writes a version of a tuple that is installed after the version written by b_i. A (predicate) wr-dependency from b_i to a_j implies that b_i either writes the version observed by a_j, or it writes a version that is installed before the version observed by a_j. A (predicate) rw-antidependency from b_i to a_j implies that b_i observes a version installed before the version written by a_j.

Notice that dependencies essentially lift the well-known notion of conflicting operations (*i.e.*, two operations from different transactions over a same tuple with at least one being a write operation) to multi-version schedules. Indeed, ignoring predicate reads, if a_j depends on b_i then a_j and b_i are conflicting; for a single-version schedule s, an operation a_j depends on b_i if and only if a_j and b_i are conflicting with $b_i <_s a_j$.

Two schedules s and s' are *conflict equivalent* if they are over the same set $\mathcal{T}$ of transactions and for every pair of operations a_j and b_i from different transactions, $b_i \rightarrow_s a_j$ iff $b_i \rightarrow_{s'} a_j$.

These dependencies intuitively imply a specific order on pairs of transactions in conflict equivalent serial schedules. That is, when an operation $a_j \in T_j$ depends on an operation $b_i \in T_i$ in a schedule s, then in every serial schedule s' conflict equivalent to s, transaction T_i should occur before transaction T_j.

Definition 3.1. A schedule s is *conflict serializable* if it is conflict equivalent to a serial schedule.

A *serialization graph* $SeG(s)$ for schedule s over a set of transactions $\mathcal{T}$ is the graph whose nodes are the transactions in $\mathcal{T}$ and where there is an edge from T_i to T_j if T_j has an operation a_j that depends on an operator b_i in T_i, thus with $b_i \rightarrow_s a_j$. Since we are usually not only interested in the existence of dependencies between operations, but also in the operations themselves, we assume the existence of a labeling function λ mapping each edge to a set of pairs of operations. Formally, $(b_i, a_j) \in \lambda(T_i, T_j)$ iff there is an operation $a_j \in T_j$ that depends on an operation $b_i \in T_i$. For ease of notation, we choose to represent $SeG(s)$ as a set of quadruples (T_i, b_i, a_j, T_j) denoting all possible pairs of these transactions T_i and T_j with all possible choices of operations with $b_i \rightarrow_s a_j$. Henceforth, we refer to these quadruples simply as edges. Notice that edges cannot contain commit operations.

A *cycle* Γ in $SeG(s)$ is a non-empty sequence of edges

$$(T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_n, b_n, a_1, T_1)$$

in $SeG(s)$, in which every transaction is mentioned exactly twice. Note that cycles are by definition simple. Here, transaction T_1 starts and concludes the cycle. For a transaction T_i in Γ, we denote by $\Gamma[T_i]$ the cycle obtained from Γ by letting T_i start and conclude the cycle while otherwise respecting the order of transactions in Γ. That is, $\Gamma[T_i]$ is the sequence

$$(T_i, b_i, a_{i+1}, T_{i+1}), \cdots, (T_n, b_n, a_1, T_1),$$
$$(T_1, b_1, a_2, T_2), \cdots, (T_{i-1}, b_{i-1}, a_i, T_i).$$

Theorem 3.2 (implied by [1]). *A schedule s is conflict serializable iff $SeG(s)$ is acyclic.*

3.5 Multiversion Read Committed

Let s be a schedule for a set $\mathcal{T}$ of transactions. Then, s *exhibits a dirty write* iff there are two write operations a_j and b_i in s on the same tuple $\mathtt{t}$, $a_j \in T_j$, $b_i \in T_i$ and $T_j \neq T_i$ such that

$$b_i <_s a_j <_s \mathtt{C}_i.$$

That is, transaction T_j writes to a tuple that has been modified earlier by T_i, but T_i has not yet issued a commit.

For a schedule s, the version order $\ll_s$ is consistent with the commit order in s if for every pair of write operations $a_j \in T_j$ and $b_i \in T_i$, we have $v_s^w(b_i) \ll_s v_s^w(a_j)$ iff $\mathtt{C}_i <_s \mathtt{C}_j$. We say that a schedule s is *read-last-committed (RLC)* if the following conditions hold:

- $\ll_s$ is consistent with the commit order;
- for every read operation a_j in s on some tuple $\mathtt{t}$:
  - $v_s^r(a_j) = init_s(\mathtt{t})$ or $\mathtt{C}_i <_s a_j$ with $v_s^r(a_j) = v_s^w(b_i)$ for some write operation $b_i \in T_i$, and
  - there is no write operation $c_k \in T_k$ on $\mathtt{t}$ with $\mathtt{C}_k <_s a_j$ and $v_s^r(a_j) \ll_s v_s^w(c_k)$; and
- for every predicate read operation a_j in s on relation R and tuple $\mathtt{t}$ of type R, with $\mathtt{t}_j$ the version of $\mathtt{t}$ in $Vset_s(a_j)$:
  - $\mathtt{t}_j = init_s(\mathtt{t})$ or $\mathtt{C}_i <_s a_j$ with $\mathtt{t}_j = v_s^w(b_i)$ for some write operation $b_i \in T_i$; and
  - there is no write operation $c_k \in T_k$ on $\mathtt{t}$ with $\mathtt{C}_k <_s a_j$ and $\mathtt{t}_j \ll_s v_s^w(c_k)$.

That is, each (predicate) read operation a_j observes for each relevant tuple the version that was committed most recently (according to the order of commits) before a_j.

Definition 3.3. A schedule is *allowed under isolation level* multiversion read committed (mvrc) if it is read-last-committed and does not exhibit dirty writes.

4 SERIALIZATION GRAPHS UNDER MVRC

Towards a sufficient condition for robustness against mvrc (c.f. Section 6), we present a condition that holds for all cycles in a serialization graph $SeG(s)$ when s is allowed under mvrc.

Let a_j and b_i be two operations occurring in a schedule s with $a_j \in T_j$ and $b_i \in T_i$ such that $b_i \rightarrow_s a_j$. We say that this dependency is a *counterflow dependency* if $\mathtt{C}_j <_s \mathtt{C}_i$ [3]. That is, the direction of the dependency is opposite to the commit order. The following Lemma is a generalization of a result in [3] to include dependencies based on predicate reads:

Lemma 4.1. *In a schedule allowed under mvrc, only (predicate) rw-antidependencies can be counterflow.*

The following theorem presents a property of cycles that must occur in $SeG(s)$ when a schedule s allowed under mvrc is not serializable. The robustness detection method of Section 6 then tests for the absence of such cycles to establish robustness for transaction programs. The theorem is a refinement of [3], where it was proven that a cycle must contain at least one counterflow dependency. Our refined property allows to detect larger sets of transaction programs to be robust as we show in Section 7.

Theorem 4.2. *Let Γ be a cycle in $SeG(s)$ for some schedule s allowed under mvrc. Then Γ contains at least one non-counterflow dependency and at least one of the following two conditions hold:*

1. *there are two adjacent counterflow dependencies in Γ; or*
2. *there are two adjacent dependencies $b_{i-1} \rightarrow_s a_i$ and $b_i \rightarrow_s a_{i+1}$ in Γ, where $b_i \rightarrow_s a_{i+1}$ is a counterflow dependency and either $b_i <_{T_i} a_i$ in the corresponding transaction T_i or b_{i-1} is an* R- *or* PR-*operation.*

To see why Theorem 4.2 holds, note that not every dependency in Γ can be counterflow, as otherwise the implied order on the commits in Γ leads to a transaction committing before itself. The remaining conditions are based on an analogous analysis.

We refer to a pair of dependencies satisfying condition (1) (resp., condition (2)) as an *adjacent-counterflow pair* (*ordered-counterflow pair*).

Definition 4.3. A cycle Γ in $SeG(s)$ for some schedule s is a *type-II* cycle if it has at least one non-counterflow dependency as well as either an adjacent-counterflow pair or an ordered-counterflow pair, and Γ is a *type-I* cycle if it has at least one counterflow dependency.

Every type-II cycle is a type-I cycle but not vice-versa, and the absence of a type-I cycle implies the absence of a type-II cycle. Theorem 4.2 now implies that if a schedule s is allowed under mvrc, then every cycle in $SeG(s)$ is a type-II cycle (and therefore a type-I cycle as well). Conflict serializability of s therefore coincides with the absence of type-II cycles in $SeG(s)$.

type(q)	WriteSet(q)	ReadSet(q)	PReadSet(q)
ins	Attr(rel(q))	$\bot$	$\bot$
key del	Attr(rel(q))	$\bot$	$\bot$
pred del	Attr(rel(q))	$\bot$	$S : \emptyset \subseteq S$
key sel	$\bot$	$S : \emptyset \subseteq S$	$\bot$
pred sel	$\bot$	$S : \emptyset \subseteq S$	$S : \emptyset \subseteq S$
key upd	$S : \emptyset \subsetneq S$	$S : \emptyset \subseteq S$	$\bot$
pred upd	$S : \emptyset \subsetneq S$	$S : \emptyset \subseteq S$	$S : \emptyset \subseteq S$

Figure 5: Constraints relative to type(q).

5 ROBUSTNESS FOR TRANSACTION PROGRAMS

5.1 Basic Transaction Programs

A basic transaction program (BTP) adheres to the following syntax:[1]

$$P \;\leftarrow\; \text{loop}(P) \;\mid\; (P \mid P) \;\mid\; (P \mid \varepsilon) \;\mid\; P;P \;\mid\; q$$

where q is a statement with the following associated functions:

- rel(q): the relation name the statement is over;
- PReadSet(q): the subset of attributes from Attr(rel(q)) used in selection predicates in q, or symbol $\bot$ (for undefined);
- ReadSet(q): the subset of attributes from Attr(rel(q)) that are observed by q, or symbol $\bot$;
- WriteSet(q): the subset of attributes from Attr(rel(q)) that are modified by q, or symbol $\bot$; and
- type(q) $\in$ {ins, key del, pred del, key sel, pred sel, key upd, pred upd} the type of statement.

Statements q can be of one of the following types: insertion, deletion, selection or update. Apart from insertion, each statement depends on a retrieval of tuples at the start of the statement. That retrieval can be a key-based look-up (always returning exactly one tuple) or can be a predicate-based look-up (returning an arbitrary number of tuples). We refer to those types of statements, respectively, as key-based and predicate-based updates, deletions, and selections. Figure 5 details how type(q) constrains PReadSet(q), ReadSet(q), and WriteSet(q). For instance, when type(q) = ins, then WriteSet(q) are all attributes and ReadSet(q) and PReadSet(q) are undefined. The notation $S : \emptyset \subseteq S$ (resp., $S : \emptyset \subsetneq S$) indicates that the set S under consideration can be empty (resp., can not be empty).

A BTP P can furthermore be annotated by a set of foreign key constraints. Each such constraint is an expression of the form $q_j = f(q_i)$, where q_i and q_j are statements occurring in P and f is a foreign key in FKeys. In addition, we require that rel(q_i) = *dom*(f), rel(q_j) = *range*(f), and q_j must be a key-based statement.

In our running example, the foreign key constraints $q_3 = f_1(q_4)$, $q_3 = f_1(q_5)$ and $q_3 = f_2(q_6)$ are added to the BTP given in Figure 1 where f_1 is the foreign key Bids(buyerId)⟶ Buyer(id) and f_2 is the foreign key Log(buyerId)⟶ Buyer(id). Notice, that there is no foreign key constraint $q_1 = f_1(q_2)$ as q_2 does not refer to buyerId.

5.2 Instantiations and schedules

Robustness for a set $\mathcal{P}$ of BTPs is defined in the next subsection w.r.t. the set of all possible schedules over $\mathcal{P}$ that result from transactions that are instantiations of BTPs in $\mathcal{P}$. We first define instantiations of statements and BTPs.

Intuitively, an instantiation of a BTP P is a transaction consisting of a sequence of chunks, which are instantiations of the statements that it consists of. For a formal treatment, we observe that all operations encapsulated in a chunk c are over the same relation, say rel(c). Similarly, since all operations in a chunk are of the same type (i.e., R, W, D, PR), they agree on the set Attr($\cdot$), and we can thus unambiguously define ReadSet(c) to denote Attr($\mathtt{R[t_i]}$) (in case of selection and update) or $\bot$ (otherwise); WriteSet(c) to denote Attr($\mathtt{W[t_i]}$) (in case of an insert, deletion or update) or $\bot$ (otherwise); and PReadSet(c) denoting Attr($\mathtt{PR}[R]$) (in case there is a predicate read) or $\bot$ (otherwise).

An *instantiation* of a BTP P is a transaction that can be obtained by applying the following rules:

- loop(P): unfold with an arbitrary finite number of instantiations of P.
- $P_1 \mid P_2$: replace with either an instantiation of P_1 or P_2;
- $P_1 \mid \varepsilon$: replace with either an instantiation of P_1 or the empty sequence;
- q, with type(q) $\in$ {ins}: replace by operation $a = \mathtt{I[t]}$ for some tuple $\mathtt{t}$ with rel($\mathtt{t}$) = R and Attr(a) = WriteSet(q);
- q, with type(q) $\in$ {key sel}: replace by operation $a = \mathtt{R[t]}$ for some tuple $\mathtt{t}$ with rel($\mathtt{t}$) = R and Attr(a) = ReadSet(q);
- q, with type(q) $\in$ {key del}: replace by operation $a = \mathtt{D[t]}$ for some tuple $\mathtt{t}$ with rel($\mathtt{t}$) = R and Attr(a) = WriteSet(q);
- q, otherwise: replace by an arbitrary chunk c (as defined in Section 3.3, and with arbitrary tuple instantiations) of type type(q) with rel(c) = rel(q), PReadSet(c) = PReadSet(q), ReadSet(c) = ReadSet(q), and WriteSet(c) = WriteSet(q).

If P is annotated with a foreign key constraint $q_j = f(q_i)$, then we furthermore require for every R-, W-, I- and D-operation over a tuple $\mathtt{t}_i$ instantiated from q_i and for every R-, W-, I- and D-operation over a tuple $\mathtt{t}_j$ instantiated from q_j that $\mathtt{t}_j = f(\mathtt{t}_i)$ (i.e., every instantiation of P must respect the foreign key constraints of P). In our running example, T_1 and T_2 are instantiations of PlaceBid where $f_1(\mathtt{u}_1) = \mathtt{t}_1$, and T_3 is an instantiation of FindBids. Indeed, e.g., for T_1, q_3 is replaced by $\mathtt{R_1[t_1]W_1[t_1]}$, q_4 by $\mathtt{R_1[u_1]}$, q_5 by ε, and q_6 by $\mathtt{I_1[l_1]}$. A set of transactions $\mathcal{T}$ is an instantiation of $\mathcal{P}$ if for every $T \in \mathcal{T}$ there is a $P \in \mathcal{P}$ such that T is an instantiation of P. Now, *schedules*($\mathcal{P}$, MVRC) consists of all schedules s allowed under MVRC for all finite sets of transactions that are instantiations of $\mathcal{P}$.

5.3 Robustness

We are now ready to define robustness on the level of BTPs:

Definition 5.1 (Robustness). A set of BTPs $\mathcal{P}$ is *robust against* MVRC if every schedule in *schedules*($\mathcal{P}$, MVRC) is conflict serializable.

We need to address how robustness for BTPs relates to robustness for the SQL programs they model. To this end, we first establish in the following proposition, that robustness over a set of schedules implies robustness over each subset:

PROPOSITION 5.2. *Let schedules($\mathcal{P}$, MVRC) $\subseteq$ schedules($\mathcal{P}'$, MVRC) for $\mathcal{P}$, $\mathcal{P}'$ sets of BTPs. If $\mathcal{P}'$ is robust against MVRC, then $\mathcal{P}$ is robust against MVRC as well.*

The running example in Section 2 already provides an idea on how to translate a set of SQL-programs $\mathcal{P}_{\text{SQL}}$ into the corresponding set $\mathcal{P}$ of BTPs (Appendix A of [46] provides a general construction). From this construction, it follows that, as BTPs abstract

[1] Appendix A in [46] provides an overview of the SQL transactions that inspired the definition of BTP.

away from the concrete conditions used for instance in WHERE-clauses, that $\mathit{schedules}(\mathcal{P}_{\text{SQL}}, \textsc{mvrc}) \subseteq \mathit{schedules}(\mathcal{P}, \textsc{mvrc})$. Therefore, when $\mathcal{P}$ is robust against MVRC, so is $\mathcal{P}_{\text{SQL}}$ and the results in this paper can be directly applied to the considered SQL fragment.

5.4 Assumptions on the DBMS

Our definitions as well as our formalism of program instantiations impose requirements on how the database management system operates. In this section, we discuss these requirements in more detail and argue why they are reasonable.

For a schedule s to be allowed under MVRC, we deliberately require that every (predicate) read operation in s observes the most recently committed version of all relevant tuples, rather than an arbitrary committed version. Although this assumption rules out distributed settings where such a requirement cannot be guaranteed, this more strict definition of MVRC is often necessary to detect larger fragments that are robust against MVRC (without it, we could deliberately choose to observe older versions to facilitate constructing a non-serializable counterexample). For non-distributed systems, this is a reasonable assumption as returning an outdated version when the most recently committed version is available anyway would make little to no sense.

When instantiating transactions from programs, each predicate-based statement is replaced by a number of operations in one atomic chunk, thereby requiring this set of operations to not be interleaved by operations from other transactions. Without this assumption, a predicate-based selection statement over a relation R, for example, could see an inconsistent view of R. Indeed, the read operations instantiated from this statement could be interleaved by a transaction T_j updating tuples of R, thereby resulting in a statement where the updates of T_j are only partially observed. We emphasize that our assumption does not rule out concurrent execution of statements from different programs, as long as the concurrent execution leads to a schedule equivalent to a schedule where the atomic chunks are respected. In Postgres[2] and Oracle, for example, each SQL statement is evaluated over a snapshot taken just before the statement started and can therefore not be influenced by concurrent updates from other transactions that committed while the statement is being evaluated.

For key-based statements, we assume each tuple is uniquely identified by a (primary) key that cannot be altered by update statements, and each key-based statement accesses exactly one tuple (i.e., if no tuple with the specified key exists, the transaction must abort). All benchmarks considered in Section 7 satisfy these assumptions. Our BTP formalism remains applicable if these assumptions are not guaranteed, but in this case each such statement q should be modeled as a predicate-based statement, where $\mathrm{PReadSet}(q)$ contains the key attributes. Note that this over-approximation allows instantiations of q to access more than one tuple, which cannot occur in practice, but one could easily extend BTPs with an additional type of statement accessing at most one tuple. Our robustness results presented in Section 6 remain applicable under such an extension, merely requiring additional checks in Algorithm 1. Our formalism can also be easily extended to multi-relation statements (e.g. joins).

6 DETECTING ROBUSTNESS

6.1 Linear Transaction Programs

Towards an algorithm to detect robustness against MVRC for arbitrary sets of BTPs, we first introduce linear transaction programs (LTPs): a restriction of BTPs where loops and branching are not allowed. More formally, an LTP adheres to the following syntax:

$$P \quad \leftarrow \quad P; P \quad \mid \quad q$$

where q represents a statement as before.

Obviously, for every set of BTPs $\mathcal{P}$, we can construct a (possibly infinite) set of LTPs $\mathcal{P}'$ such that $\mathit{schedules}(\mathcal{P}, \textsc{mvrc}) = \mathit{schedules}(\mathcal{P}', \textsc{mvrc})$ by considering all possible unfoldings of loops and conditional statements. However, w.r.t. robustness testing, we show in Proposition 6.1 that it suffices to restrict attention to loop unfoldings of size at most two as defined next.

For a BTP P, let $\mathit{Unfold}_{\leq 2}(P)$ denote the set of LTPs obtained by repeated application of the following rules:

- $\mathrm{loop}(P_1)$: replace with zero, one or two repetitions of P_1;
- $P_1 \mid P_2$: replace with either P_1 or P_2;
- $P_1 \mid \varepsilon$: replace with either P_1 or the empty sequence.

By slight abuse of notation, we use $\mathit{Unfold}_{\leq 2}(\mathcal{P})$ for a set of BTPs $\mathcal{P}$ to denote the set of LTPs obtained by applying $\mathit{Unfold}_{\leq 2}(P)$ to each $P \in \mathcal{P}$. More formally:

$$\mathit{Unfold}_{\leq 2}(\mathcal{P}) = \bigcup_{P \in \mathcal{P}} \mathit{Unfold}_{\leq 2}(P).$$

Since each $\mathrm{loop}(P_1)$ is replaced by at most two repetitions of P_1, it immediately follows that $\mathit{Unfold}_{\leq 2}(\mathcal{P})$ is a finite set. In practice, unfolding does not increase the size too much, e.g., for TPC-C the number of transaction programs increases from 5 to 13. By construction, it follows that $\mathit{schedules}(\mathit{Unfold}_{\leq 2}(\mathcal{P}), \textsc{mvrc}) \subseteq \mathit{schedules}(\mathcal{P}, \textsc{mvrc})$.

Proposition 6.1. *Far a set $\mathcal{P}$ of BTPs, the following are equivalent:*

(1) *$\mathcal{P}$ is robust against MVRC;*
(2) *$\mathit{Unfold}_{\leq 2}(\mathcal{P})$ is robust against MVRC.*

To see why two iterations of each loop suffice, note that we are looking for a cycle. Since in each transaction only two operations are important for this cycle (one for the incoming edge, one for the outgoing edge), all other iterations not involving one of these two operations can be removed.

We introduce a *summary graph* $\mathit{SuG}(\mathcal{P})$ summarizing all possible serialization graphs for schedules in $\mathit{schedules}(\mathcal{P}, \textsc{mvrc})$. This summary graph is closely related to the dependency graph used by Alomari and Fekete [3] but differs in two aspects. We add additional information to edges necessary to detect type-II cycles, and, whereas [3] relies on a domain specialist that can predict possible conflicts to construct the graph, we provide a formal construction based on the formalism of LTPs (Algorithm 1).

Formally, $\mathit{SuG}(\mathcal{P})$ is a graph where each program in $\mathcal{P}$ is represented by a node, and potential dependencies between two instantiations of programs in $\mathcal{P}$ are represented by edges. Since we are not only interested in the existence of these dependencies, but also in the type of dependency (counterflow or not) and the two statements that give rise to this dependency, we assume an edge labeling function λ. The function λ maps each edge in $\mathit{SuG}(\mathcal{P})$ from a program P_i to a program P_j to a set of tuples (c, q_i, q_j) where $q_i \in P_i$, $q_j \in P_j$, and $c \in \{\mathit{counterflow}, \mathit{non\text{-}counterflow}\}$. We will often represent these edges as a quintuple (P_i, q_i, c, q_j, P_j).

The summary graph $\mathit{SuG}(\mathcal{P})$ should be constructed in such a way that the following condition holds:

Condition 6.2. *Let $b_i \to_s a_j$ be a dependency occurring between transaction T_i and T_j in a schedule $s \in \mathit{schedules}(\mathcal{P}, \textsc{mvrc})$. Let P_i*

[2] Although the Postgres implementation deviates slightly from this presumption, our findings are still accurate. We refer to [46] for more details.

Algorithm 1: Construction of $SuG(\mathcal{P})$ for a set $\mathcal{P}$ of LTPs.

```
Function ncDepConds($q_i$, $q_j$) : Boolean
    return $WriteSet(q_i) \cap WriteSet(q_j) \neq \emptyset$ or
      $WriteSet(q_i) \cap ReadSet(q_j) \neq \emptyset$ or
      $WriteSet(q_i) \cap PReadSet(q_j) \neq \emptyset$ or
      $ReadSet(q_i) \cap WriteSet(q_j) \neq \emptyset$ or
      $PReadSet(q_i) \cap WriteSet(q_j) \neq \emptyset$;
Function cDepConds($q_i$, $q_j$) : Boolean
    if $PReadSet(q_i) \cap WriteSet(q_j) \neq \emptyset$ then
        return true;
    if $ReadSet(q_i) \cap WriteSet(q_j) \neq \emptyset$ then
        for foreign key constraints $q_k = f(q_i)$ for $P_i$ and
          $q_\ell = f(q_j)$ for $P_j$ do
            if $type(q_k), type(q_\ell) \in \{key\ upd, key\ del, ins\}$
              and $q_k <_{P_i} q_i$ and $q_\ell <_{P_j} q_j$ then
                return false;
        return true;
    return false;
Function constructSuG($\mathcal{P}$) : $SuG(\mathcal{P})$
    $S := \emptyset$;
    for $P_i \in \mathcal{P}, P_j \in \mathcal{P}, q_i \in P_i$, and $q_j \in P_j$ with
      $rel(q_i) = rel(q_j)$ do
        if ncDepTable$[q_i, q_j]$ = true or
          (ncDepTable$[q_i, q_j] = \bot$ and
          ncDepConds($q_i, q_j$)) then
            add $(P_i, q_i, non\text{-}counterflow, q_j, P_j)$ to $S$;
        if cDepTable$[q_i, q_j]$ = true or
          (cDepTable$[q_i, q_j] = \bot$ and cDepConds($q_i, q_j$))
          then
            add $(P_i, q_i, counterflow, q_j, P_j)$ to $S$;
    return $S$;
```

and P_j be the programs in $\mathcal{P}$ from which T_i and T_j were instantiated, and let q_i and q_j be the two statements in respectively P_i and P_j from which operations b_i and a_j were instantiated. Then, $SuG(\mathcal{P})$ must have an edge (P_i, q_i, c, q_j, P_j), where c is counterflow iff $b_i \rightarrow_s a_j$ is a counterflow dependency.

6.2 Constructing the Summary Graph

The algorithm to construct the summary graph $SuG(\mathcal{P})$ for a given set of LTPs $\mathcal{P}$ is given in Algorithm 1. We discuss how the edges in the graph $SuG(\mathcal{P})$ are constructed. To this end, let q_i and q_j be two (not necessarily different) statements in respectively programs P_i and P_j with $\mathrm{rel}(q_i) = \mathrm{rel}(q_j)$. The basic idea underlying the construction of $SuG(\mathcal{P})$ is to add an edge (P_i, q_i, c, q_j, P_j) with $c \in \{non\text{-}counterflow, counterflow\}$ if P_i and P_j could have instantiations that admit a c dependency for operations in the transaction fragments instantiated by q_i and q_j, respectively.

For c = *non-counterflow* the conditions are relatively straightforward and mostly analogous to the definition of dependency, since every type of dependency listed in Section 3.4 can be (and sometimes must be) non-counterflow. More precisely, Table (1a) details when the types of q_i and q_j imply that a *non-counterflow* dependency can be admitted (entry is *true*), may not not be admitted (entry is *false*), or when additional checks need to be performed regarding the intersections of involved read, write and predicate read attributes (entry is $\bot$). Algorithm 1, function ncDepConds(q_i, q_j) gives the precise condition of these additional checks.

For c = *counterflow* the approach is similar. Table (1b) shows if a *counterflow* dependency can be admitted based on the types of q_i and q_j. In case of $\bot$, it is tested if the intersection between the (predicate) read attributes of q_i and write attributes of q_j is non-empty, which is analogous to the condition of a (predicate) rw-antidependency (c.f., Section 3.4) which are the only dependencies that can be counterflow. In this case also a check on the foreign keys of the programs is performed, see cDepConds in Algorithm 1.

We remark that, since the edges added to $SuG(\mathcal{P})$ are based on conditions that are independent of a particular schedule, two statements can at the same time allow a counterflow as well as non-counterflow dependency. The following proposition shows that the construction is sound:

Proposition 6.3. *For a set of LTPs $\mathcal{P}$, the summary graph $SuG(\mathcal{P})$ constructed by Algorithm 1 satisfies Condition 6.2.*

6.3 Detecting Robustness for Linear Transaction Programs

We start by lifting Theorem 4.2 to LTPs:

Theorem 6.4. *A set of LTPs $\mathcal{P}$ is robust against mvrc if there is no cycle Γ in $SuG(\mathcal{P})$ containing at least one non-counterflow edge for which at least one of the following two conditions holds:*

- *there are two adjacent counterflow edges in Γ; or*
- *there are two adjacent edges $(P_{i-1}, q_{i-1}, non\text{-}counterflow, q_i, P_i)$ and $(P_i, q'_i, counterflow, q_{i+1}, P_{i+1})$ in Γ, where either $q'_i <_{P_i} q_i$ in the corresponding program P_i, or $type(q_{i-1}) \in \{key\ sel, pred\ sel, pred\ upd, pred\ del\}$.*

The proof relies on Proposition 6.3 to show how these properties about dependencies between operations as in Theorem 4.2 can be lifted to properties over edges in $SuG(\mathcal{P})$. In particular, Condition 6.2 implies that for every schedule s allowed under mvrc, every cycle in $SeG(s)$ is witnessed by a cycle in $SuG(\mathcal{P})$. It should be noted that the cycle Γ in the theorem above is allowed to visit the same nodes/edges multiple times. Note that such a cycle Γ corresponds to a type-II cycle described in Theorem 4.2 lifted to summary graphs. For convenience, we will therefore refer to these cycles in $SuG(\mathcal{P})$ as type-II cycles as well. Figure 4 does not contain a type-II cycle whereas it clearly contains a type-I cycle.

Based on Theorem 6.4, Algorithm 2 then tests for the absence of type-II cycles as a proxy for robustness against mvrc. Notice that Algorithm 2 is sound but incomplete: it can return false negatives but never a false positive, as formally shown in Proposition 6.5. We demonstrate in Section 7 that it can detect strictly larger sets of BTPs to be robust than the state-of-the-art. Even though the complexity is $O(n^6)$ with n the total number of statements in $Unfold_{\leq 2}(\mathcal{P})$, we show that a proof-of-concept implementation runs in a matter of seconds.

Proposition 6.5. *For a set $\mathcal{P}$ of BTPs, if the algorithm returns true, then $\mathcal{P}$ is robust against mvrc.*

7 EXPERIMENTAL VALIDATION

7.1 Benchmarks

We implemented Algorithm 2 in Python and tested it on three benchmarks whose characteristics are given in Table 2. Appendix E of [46] contains a detailed description of their schema, the

$q_i \setminus q_j$	ins	key sel	pred sel	key upd	pred upd	key del	pred del
ins	false	⊥	true	⊥	true	⊥	true
key sel	false	false	false	⊥	⊥	⊥	⊥
pred sel	true	false	false	⊥	⊥	true	true
key upd	false	⊥	⊥	⊥	⊥	⊥	⊥
pred upd	true	⊥	⊥	⊥	⊥	true	true
key del	false	false	true	false	true	false	true
pred del	true	false	true	⊥	true	true	true

(a) ncDepTable

$q_i \setminus q_j$	ins	key sel	pred sel	key upd	pred upd	key del	pred del
ins	false	false	false	false	false	false	false
key sel	false	false	false	⊥	⊥	⊥	⊥
pred sel	true	false	false	⊥	⊥	true	true
key upd	false	false	false	false	false	false	false
pred upd	true	false	false	⊥	⊥	true	true
key del	false	false	false	false	false	false	false
pred del	true	false	false	⊥	⊥	true	true

(b) cDepTable

Table 1: Condition tables used in Algorithm 1.

Algorithm 2: Testing robustness.

```
input  : a set $\mathcal{P}$ of BTPs
output : true if $SuG(\mathcal{P})$ does not contain a type-II cycle,
         false otherwise
$G \leftarrow$ constructSuG($Unfold_{\leq 2}(\mathcal{P})$);
for $(P_1, q_1, \textit{non-counterflow}, q_2, P_2) \in G$ do
    for $(P_3, q_3, c, q_4, P_4) \in G$ do
        if $P_3$ is reachable from $P_2$ in $G$ then
            for $(P_4, q_4', \textit{counterflow}, q_5, P_5) \in G$ do
                if $P_1$ is reachable from $P_5$ in $G$ and
                  ($c = \textit{counterflow}$ or $q_4' <_{P_4} q_4$ or
                  $type(q_3) \in \{$key sel,
                  pred sel, pred upd, pred del$\}$) then
                    return false;
return true;
```

SQL transaction programs as well as their translation into BTPs and foreign key constraints. Since our experimental validation is based on static program analysis, benchmark configuration parameters influencing database size (e.g. number of warehouses for TPC-C) and how often different transactions occur are irrelevant to our experiments. If robustness is detected, serializability is guaranteed for all such possible configurations.

SmallBank [2]. The schema consists of three relations, where each relation has two attributes. SmallBank models a banking application where customers can interact with their savings and checking accounts through five different transaction programs: Balance, Amalgamate, DepositChecking, TransactSavings and WriteCheck. These programs do not contain insert or delete statements, and there is no branching or iteration. Furthermore, tuples are always accessed through their primary key, implying that there are no predicate reads. In this more limited setting, the machinery developed in [44] can completely *decide* robustness against mvrc (that is, never results in false negatives). A comparison with the results of [44] can thus provide insight on the completeness of Algorithm 2.

TPC-C [43]. This benchmark models a multi-warehouse wholesale operation. The database schema consists of nine different relations, where each relation has between 3 and 21 attributes. Five transaction programs (NewOrder, Delivery, Payment, OrderStatus and StockLevel) model different actions, such as creating and delivering orders, handling customer payments, as well as read-only programs collecting information about orders and stock levels.

Auction. The Auction benchmark is presented in Section 2. In Section 7.3 we describe an alternative version of this benchmark where the total number of transaction programs can be scaled.

7.2 Detecting Robustness against mvrc

Different settings. In this paper, as in [44], we deviate from the literature by considering dependencies between operations on the granularity of individual attributes, as it allows to detect larger sets of transaction programs to be robust. To assess this advantage, we also compare with the setting where dependencies are defined on the level of complete tuples, that is, operations over the same tuple are no longer required to access a common attribute for a dependency to occur. We stress that when our algorithm determines a set of transaction programs to be robust, that set will still be robust on systems that assure mvrc with tuple-level database objects, for the simple reason that every conflict on the granularity of attributes implies a conflict on the granularity of tuples. As a result, every schedule that can be created by these systems is allowed under our definition of mvrc. We consider four different settings: ‘tpl dep’, ‘attr dep’, ‘tpl dep + FK’ and ‘attr dep + FK’. The first two settings ignore foreign key constraints, and the settings ‘tpl dep’ and ‘tpl dep + FK’ consider dependencies on the granularity of tuples rather than that of attributes.

Maximal robust subsets. We test robustness for each possible subset of programs for all three benchmarks to detect maximal robust subsets. Figure 6 summarizes the subsets detected as robust against mvrc by Algorithm 2 for each benchmark and setting. Here, transactions are represented by their abbreviations (e.g., NO stands for NewOrder). Visualizations of these summary graphs can be found in Appendix E of [46].

For both SmallBank and TPC-C, we identify a subset consisting of three (out of five) programs as robust against mvrc for setting ‘attr dep + FK’, and for the Auction benchmark, we are even able to detect the complete benchmark as robust against mvrc. When comparing the different settings, we can make the following observations. Attribute-granularity is required for TPC-C to detect a maximal possible robust subset of size 3 (row ‘attr dep + FK’). On the other hand, attribute-granularity does not provide additional benefit over tuple-granularity for SmallBank and Auction. This is not unexpected, as relations in both benchmarks have only a limited number of attributes each whereas TPC-C contains many more attributes per relation. Furthermore, foreign key constraints are necessary to derive the largest robust subsets for TPC-C and Auction (compare the rows ‘attr dep’ with ‘attr dep + FK’). This underlies the utility of foreign key constraints and the effectiveness of our approach, especially when taking into account that deciding robustness against mvrc w.r.t. foreign key constraints is undecidable [45].

Comparison with [3]. Alomari and Fekete [3] detect robustness through the absence of cycles involving at least one counterflow edge, which we refer to as type-I cycles. A direct comparison would be unfair as that work does not include predicate reads or atomic updates, and does not consider attribute-granularity. Furthermore, no formal method is provided to construct a summary graph. Towards an unbiased comparison, we report in Figure 7 the maximal robust subsets that can be detected via the absence of type-I cycles in the corresponding summary graphs (as constructed through Algorithm 1) for the different settings. When comparing to Figure 6, we see that our technique detects more and larger subsets as robust for all benchmarks. Subsets not detected by [3] are displayed in bold in Figure 6. Notice in particular

	SmallBank	TPC-C	Auction	Auction(n)
relations	3	9	3	3
attributes per relation	2	3–21	2	2
transaction programs	5	5	2	$2n$
nodes / unfolded tr pr	5	13	3	$3n$
edges (counterflow)	56 (12)	396 (83)	17 (1)	$8n + 9n^2$ (n)

Table 2: Benchmark characteristics.

Alg 2	SmallBank	TPC-C	Auction
tpl dep	{Am, DC, TS}, **{Bal, DC}**, **{Bal, TS}**	{OS, SL}, {NO}	{FB}
attr dep	{Am, DC, TS}, **{Bal, DC}**, **{Bal, TS}**	{OS, SL}, {NO}	{FB}
tpl dep + FK	{Am, DC, TS}, **{Bal, DC}**, **{Bal, TS}**	{OS, SL}, {NO}	**{FB, PB}**
attr dep + FK	{Am, DC, TS}, **{Bal, DC}**, **{Bal, TS}**	**{OS, Pay, SL}**, {NO, Pay}	**{FB, PB}**

Figure 6: Robust subsets based on absence of type-II cycles (Algorithm 2). Subsets for which the summary graph contains a type-I cycle, that are thus not detected by [3], are in bold.

Method of [3]	SmallBank	TPC-C	Auction
tpl dep	{Am, DC, TS}, {Bal}	{OS, SL}, {NO}	{FB}
attr dep	{Am, DC, TS}, {Bal}	{OS, SL}, {NO}	{FB}
tpl dep + FK	{Am, DC, TS}, {Bal}	{OS, SL}, {NO}	{PB}, {FB}
attr dep + FK	{Am, DC, TS}, {Bal}	{NO, Pay}, {Pay, SL}, {OS, SL}	{PB}, {FB}

Figure 7: Robust subsets wrt absence of type-I cycles ([3]).

that Algorithm 2 correctly identifies the Auction benchmark as a whole as robust against mvrc, whereas [3] only detects singleton sets as robust against mvrc.

False negatives. Algorithm 2 is based on a sufficient condition and can result in false negatives. Earlier work [44] provided a complete characterization for deciding robustness against mvrc for benchmarks satisfying certain restrictions: tuples can only be accessed through key-based lookup (ruling out predicate-based dependencies) and the value of keys is not allowed to be changed. As discussed earlier, SmallBank can be captured by this restricted formalism and [44] therefore lists the actual robust subsets. Comparing with Figure 6, we can report that Algorithm 2 finds *all* maximal robust subsets and does not report any false negatives. That is, for each subset of SmallBank not detected as robust by Algorithm 2, a counterexample schedule exists that is allowed under mvrc but not conflict serializable. Sometimes, specific details such as predicate conditions can lead to robustness not detected by our algorithm. For the TPC-C benchmark for example, we identified {Delivery} as a false negative. The reason is that, for each district, Delivery first identifies the oldest open order through a predicate read, followed by deleting this tuple from relation NewOrder and handling the order. Because of this, no two instances of Delivery over the same warehouse can be concurrent: if they are, they would select the same oldest open order, and the second one to delete it would have to abort, since the tuple no longer exists.

7.3 Scalability

We reiterate that robustness is static property and involves an offline analysis where a set of transaction programs can be tested at design time. There is no need to perform online robustness testing during transaction processing. Execution times in the order of milliseconds are therefore not required. Previous work [3, 44] has already established the performance benefit of executing transactions under the lower isolation level mvrc over executing them under a higher isolation level such as snapshot isolation or serializable, so we do not repeat such experiments here.

Table 2 describes for each benchmark the size of the summary graph in terms of the number of nodes as well as the number of (counterflow) edges. Since programs with loops and branches are unfolded, the number of nodes can be larger than the number of programs at the application level. For each of the benchmarks, our implementation runs in a fraction of a second. To better illustrate the feasibility of our approach for larger sets of programs (and, consequently, larger summary graphs), we next present a modification of the Auction benchmark, referring to it as Auction(n), where the total number of programs depends on a scaling parameter n, which should be contrasted with the benchmarks presented in Section 7.1 where the number of programs is fixed (5 for SmallBank and TPC-C, and 2 for Auction).

Auction(n) extends upon Auction, as presented in Section 2, by modelling the auction of n different items, where the bids for each item i are stored in a separate relation Bids^i(buyerId, bid), rather than having only one relation Bids.[3] For each item i, Auction(n) has two different programs: FindBids^i and PlaceBid^i. The meaning of these programs as well as the program details remain as discussed in Section 2, with the only difference that they are now over item i and corresponding relation Bids^i. The statement details of the corresponding BTP programs are as presented in Figure 2, with the only exception that rel(q_2), rel(q_4) and rel(q_5) are now the corresponding Bids^i. Notice that Auction(1) corresponds to the Auction benchmark as introduced in Section 2. By construction, the number of BTPs in Auction(n) is $2 \cdot n$, and since each PlaceBid^i is unfolded in two LTPs, the derived set of LTPs has size $3 \cdot n$.

Algorithm 2 detects Auction(n) as robust against mvrc for each n. We emphasize that the summary graph of Auction(n) does not just consist of n connected components, where each such component is equivalent to the graph given in Figure 4. Indeed, since each statement still writes to the relation Buyer, the summary graph will have a non-counterflow edge between each pair of programs, even if they are over different items. The skeleton of the summary graph for Auction(n) is given in [46].

Figure 8 shows the execution time of our implementation as well as the resulting number of edges in the summary graph for Auction(n) for different values of n. For each value of n, the experiment was repeated 10 times, and the graph shows both the average value as well as the 95% confidence interval. These results demonstrate that our approach can be applied to larger sets of programs. We reiterate that execution times of seconds (or even larger) are acceptable, as robustness detection is a form of static program analysis that has no influence on the actual transaction throughput once programs are being executed under mvrc. Furthermore, we stress that the parameter n refers to the number of transaction programs in the benchmark (which is unlikely to be a three figure number in practice), and does not refer to the concurrent online execution of transactions which of course can be several orders of magnitude larger. Our experiments do not cover scalability according to the complexity of transaction programs, as such experiments would require a benchmark where the complexity (number of nested loops and branching) of the programs can be scaled. We are not aware of such a benchmark. To be even more precise, it is the size of the resulting summary graph that influences the required time to analyze the workload (cf. Table 2). While more nested loops and branching leads to more unfolded nodes in the graph, the same

[3] Alternatively, we can still assume that all bids are stored in one relation Bids and each Bids^i acts as a view over this relation, disjoint with all other views.

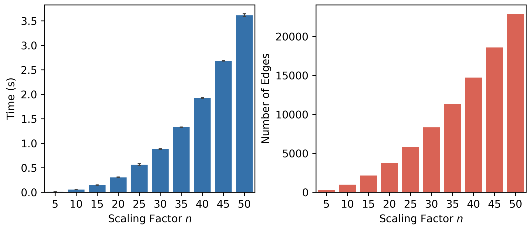

Figure 8: *(left)* Time required to verify robustness against MVRC for Auction(*n*) for different scaling factors. *(right)* Number of edges in the corresponding summary graphs.

increasing in the size of the summary graph can be achieved by simply adding programs as well (which is what we did in our Auction(*n*) benchmark).

8 RELATED WORK

As mentioned in the introduction, previous work on static robustness testing [3, 20] for transaction programs is based on testing for the absence of cycles in a static dependency graph containing some dangerous structure. This paper builds further upon the above ideas but is different in two key aspects: (1) Through the formalism of BTPs, our approach can be readily implemented and does not require a database expert for the construction of the summary graph. The only manual step that is required is to model SQL code in terms of BTPs and foreign key constraints. (2) For the first time inserts, deletes as well as predicate reads are incorporated providing a significant step towards the utilization of robustness testing in practice.

Our earlier work [44] provides a complete algorithm for deciding robustness against MVRC but is restricted to the setting where tuples can only be accessed through key-based lookup and key attributes are not allowed to change. That approach can not be extended to include inserts, deletes, or predicate reads. In fact, we show in [45] that the extension to foreign key constraints already renders the problem undecidable. Undecidability is circumvented in this paper by devising a sound but incomplete algorithm. The work in [26] considers robustness on the level of transactions rather than transaction programs and is based on locking rather than versioning as a concurrency control mechanism.

Gan et al. [21] present IsoDiff, a tool to detect and resolve potential anomalies caused by executing transactions under READ COMMITTED or SNAPSHOT ISOLATION instead of SERIALIZABLE. Similar to our approach, IsoDiff is based on detecting cycles with a specific structure. For READ COMMITTED, IsoDiff searches for type-I cycles, but includes additional timing constraints and correlation constraints to reduce the number of false positives. Contrasting our work, IsoDiff derives potential transactions from a database SQL trace, while we derive potential transactions through our formalism of BTPs. A potential pitfall of analyzing a trace is that it may overlook transactions that are rarely executed, thereby incorrectly considering an application to be robust. The correlation constraints IsoDiff derives from these traces correspond to the foreign key constraints expressed over BTPs. A more subtle difference is that the timing constraints proposed as part of IsoDiff assume that a dependency $b_i \rightarrow_s a_j$ always implies that operation b_i occurs before a_j in s, thereby implicitly assuming a single version implementation of READ COMMITTED, rather than MVRC as discussed in this paper. In particular, MVRC allows for situations where b_i occurs after a_j in s, if $b_i \rightarrow_s a_j$ is a rw-antidependency.

Other work studies robustness within a framework for uniformly specifying different isolation levels in a declarative way [8, 11, 12]. A key assumption here is *atomic visibility* requiring that either all or none of the updates of each transaction are visible to other transactions. This work aims at higher isolation levels and cannot be used for MVRC, as MVRC does not admit *atomic visibility.*

Transaction chopping splits transactions into smaller pieces to obtain performance benefits and is correct if, for every serializable execution of the chopping, there is an equivalent serializable execution of the original transactions [39]. Cerone et al. [12, 13] studied chopping under various isolation levels. Transaction chopping has no direct relationship with robustness testing against MVRC.

Many approaches to increase transaction throughput without sacrificing serializability have been proposed: improved or novel pessimistic (cf., e.g., [24, 34, 36, 42, 47]) or optimistic (cf., e.g., [9, 10, 15, 16, 22, 23, 25, 27–29, 31, 37, 38, 49, 50]) algorithms, as well as approaches based on coordination avoidance (cf., e.g., [17, 18, 30, 32, 33, 35, 40, 41, 48]). Robustness differs from these approaches in that it can be applied to standard DBMS's without any modifications to the database internals. Instead, the robustness property is leveraged to guarantee serializability even though the database system provides a lower isolation level.

Orthogonal to robustness detection, tools such as Elle [4] aim at detecting anomalies that should not occur under a given isolation level. These tools can be used to detect whether a database system implements the declared isolation levels correctly, whereas robustness assumes that the isolation level is implemented correctly to decide whether every possible execution of a given workload is serializable.

Our formalization of transactions and conflict serializability is closely related to the formalization presented by Adya et al. [1], but with some important differences, which we discuss next. We assume a total rather than a partial order over the operations in a schedule, and the different types of write operations are made more explicit by introducing inserts and deletes. In particular, we require that only an insert operation can create the first visible version after the unborn version, and only a delete operation can create the dead version in a schedule. Our definitions consider an atomic update operation as well, which is essentially a read operation followed by a write operation on the same object, and which cannot be interleaved by other operations in a schedule. Atomic chunks take this assumption one step further by allowing arbitrary sequences of operations in a transaction to act as one atomic operation. We furthermore assume that all operations are over concrete (database) tuples rather than abstract objects, and keep track of the specific attribute values that each operation observes or modifies. As illustrated in [44], explicitly taking into account these atomic update operations as well as the attributes that are accessed can greatly increase the effectiveness of robustness detection.

9 CONCLUSIONS

The present paper makes a significant step towards robustness testing in practice: through a formal approach based on BTPs, we provide an algorithm for robustness testing that (1) can be readily implemented; and (2) improves over the state-of-the-art in that it incorporates a larger set of operations (inserts, deletes, predicate reads) and can detect larger sets of transaction programs to be robust against MVRC. In the future, we plan to cover more expressive transaction programs.

ACKNOWLEDGMENTS

This work is funded by FWO-grant G019921N.

REFERENCES

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *ICDE*. 67–78.

[2] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE*. 576–585.

[3] Mohammad Alomari and Alan Fekete. 2015. Serializable use of Read Committed isolation level. In *AICCSA*. 1–8.

[4] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *PVLDB* 14, 3 (2020), 268–280.

[5] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019. Checking Robustness Against Snapshot Isolation. In *CAV*. 286–304.

[6] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019. Robustness Against Transactional Causal Consistency. In *CONCUR*. 1–18.

[7] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*. 1–10.

[8] Giovanni Bernardi and Alexey Gotsman. 2016. Robustness against Consistency Models with Atomic Visibility. In *CONCUR*. 7:1–7:15.

[9] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. 2015. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *SIGMOD*. 1295–1309.

[10] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR*. 9–20.

[11] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *CONCUR*. 58–71.

[12] Andrea Cerone and Alexey Gotsman. 2018. Analysing Snapshot Isolation. *J.ACM* 65, 2 (2018), 1–41.

[13] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2015. Transaction Chopping for Parallel Snapshot Isolation. In *DISC*, Vol. 9363. 388–404.

[14] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2017. Algebraic Laws for Weak Consistency. In *CONCUR*. 26:1–26:18.

[15] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*. 1243–1254.

[16] Bailu Ding, Lucja Kot, Alan J. Demers, and Johannes Gehrke. 2015. Centiman: elastic, high performance optimistic concurrency control by watermarking. In *SoCC*. 262–275.

[17] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *PVLDB* 10, 5 (2017), 613–624.

[18] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *PVLDB* 8, 11 (2015), 1190–1201.

[19] Alan Fekete. 2005. Allocating isolation levels to transactions. In *PODS*. 206–215.

[20] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528.

[21] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *PVLDB* 13, 11 (2020), 2773–2786.

[22] Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, and Aoying Zhou. 2019. Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. *PVLDB* 12, 5 (2019), 584–596.

[23] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *PVLDB* 13, 5 (2020), 629–642.

[24] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. 2009. Improving OLTP Scalability using Speculative Lock Inheritance. *PVLDB* 2, 1 (2009), 479–489.

[25] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*. 603–614.

[26] Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. 2020. Deciding Robustness for Lower SQL Isolation Levels. In *PODS*. 315–330.

[27] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*. 1675–1687.

[28] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.

[29] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD*. 21–35.

[30] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *PVLDB* 13, 11 (2020), 2047–2060.

[31] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*. 677–689.

[32] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *SIGMOD*. 527–542.

[33] Thamir M. Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-oriented, Control-free Concurrency Architecture. In *Middleware*, Paulo Ferreira and Liuba Shrira (Eds.). 13–25.

[34] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *SIGMOD*. 1583–1598.

[35] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-latency, Geo-replicated Transactions. *PVLDB* 12, 11 (2019), 1747–1761.

[36] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2012. Lightweight Locking for Main Memory Database Systems. *PVLDB* 6, 2 (2012), 145–156.

[37] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. 2014. Reducing Database Locking Contention Through Multi-version Concurrency. *PVLDB* 7, 13 (2014), 1331–1342.

[38] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting. In *SIGMOD*. 245–258.

[39] Dennis E. Shasha, François Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.* 20, 3 (1995), 325–363.

[40] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP transactions via learned abort prediction. In *aiDM*. 1:1–1:8.

[41] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*. 1–12.

[42] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-Aware Lock Scheduling for Transactional Databases. *PVLDB* 11, 5 (2018), 648–662.

[43] TPC-C. [n.d.]. On-Line Transaction Processing Benchmark. ([n. d.]). http://www.tpc.org/tpcc/.

[44] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2021. Robustness against Read Committed for Transaction Templates. *PVLDB* 14, 11 (2021), 2141–2153.

[45] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2022. Robustness Against Read Committed for Transaction Templates with Functional Constraints. In *ICDT*, Vol. 220. 16:1–16:17.

[46] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2023. Detecting Robustness against MVRC for Transaction Programs with Predicate Reads (full version). (2023). https://arxiv.org/abs/2302.08789.

[47] Cong Yan and Alvin Cheung. 2016. Leveraging Lock Contention to Improve OLTP Application Performance. *PVLDB* 9, 5 (2016), 444–455.

[48] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting Single-Threaded Model in Multi-Core In-Memory Systems. *TKDE* 28, 10 (2016), 2635–2650.

[49] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*. 1629–1642.

[50] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-Memory Databases. *PVLDB* 9, 6 (2016), 504–515.