Robustness against Read Committed for Transaction Templates

Brecht Vandevoort
brecht.vandevoort@uhasselt.be
Hasselt University and Transnational University of Limburg

Bas Ketsman
bas.ketsman@vub.be
Vrije Universiteit Brussel

Christoph Koch
christoph.koch@epfl.ch
École Polytechnique Fédérale de Lausanne

Frank Neven
frank.neven@uhasselt.be
Hasselt University and Transnational University of Limburg

ABSTRACT

The isolation level Multiversion Read Committed (RC), offered by many database systems, is known to trade consistency for increased transaction throughput. Sometimes, transaction workloads can be safely executed under RC obtaining the perfect isolation of serializability at the lower cost of RC. To identify such cases, we introduce an expressive model of transaction programs to better reason about the serializability of transactional workloads. We develop tractable algorithms to decide whether any possible schedule of a workload executed under RC is serializable (referred to as the robustness problem). Our approach yields robust subsets that are larger than those identified by previous methods. We provide experimental evidence that workloads that are robust against RC can be evaluated faster under RC compared to stronger isolation levels. We discuss techniques for making workloads robust against RC by promoting selective read operations to updates. Depending on the scenario, the performance improvements can be considerable. Robustness testing and safely executing transactions under the lower isolation level RC can therefore provide a direct way to increase transaction throughput without changing DBMS internals.

Under submission to VLDB.

1 INTRODUCTION

Relational database systems provide the ability to trade off isolation guarantees for improved performance by offering a variety of isolation levels, the highest being serializability, which guarantees what is considered to be perfect isolation. Executing transactions concurrently under weaker isolation levels is not without risk, as it can introduce certain anomalies. Sometimes, however, a set of transactions can be executed at an isolation level lower than serializability without introducing any anomalies. This is a desirable scenario: a lower isolation level, usually implementable with a cheaper concurrency control algorithm, gives us the stronger isolation guarantees of serializability for free. This formal property is called robustness [10, 22]: a set of transactions $\mathcal{T}$ is called *robust against a given isolation level* if every possible interleaving of the transactions in $\mathcal{T}$ that is allowed under the specified isolation level is serializable.

There is a famous example that is part of database folklore: the TPC-C benchmark [48] is robust against Snapshot Isolation (SI), so there is no need to run a stronger, and more expensive, concurrency control algorithm than SI if the workload is just TPC-C. This has played a role in the incorrect choice of SI as the general concurrency control algorithm for isolation level Serializable in Oracle and PostgreSQL (before version 9.1, cf. [23]).

Robustness is, fundamentally, a static property of workloads, rather than a property detectable online, while a concrete transaction schedule unfolds. It involves the static or offline analysis of *transaction programs* (code) to decide whether all possible interleavings of transactions (that is, instantiations of transaction programs) at runtime are guaranteed to be robust. Robustness received quite a bit of attention in the literature. Most existing work focuses on SI [3, 7, 22, 23] or higher isolation levels [8, 10, 13, 16]. It is particularly interesting to consider robustness against lower level isolation levels like multi-version Read Committed (referred to as RC from now on). Indeed, RC is widely available, often the default in database systems (see, e.g., [4]), and is generally expected to have better throughput than stronger isolation levels. The work by Alomari and Fekete [5] studies robustness against RC and proposes ways to preanalyse (and then modify) the code of a set of applications allowing to run transactions under RC while still guaranteeing that all executions are serializable.

In general, robustness is a hopelessly undecidable property and previous work has therefore only dealt with very simple models of workloads. In this paper, we focus on pushing the frontier of the robustness problem for RC. Robustness for arbitrary database application code would require the full sophistication of state-of-the-art program analysis and theorem provers and would not allow us to distill general guarantees that can lead to simpler analysis algorithms. We take a middle road, proposing a more expressive model of workloads than previously considered, which lets us still craft a complete and tractable decision procedure for robustness. We will show by examples – specifically the TPC-C and SmallBank benchmarks – that our model allows us to significantly expand the reach of robustness testing, yielding guaranteed serializability at the cost of just RC isolation for a much larger class of workloads.

Our approach is centered on a novel characterization of robustness against RC in the spirit of [22, 30] that improves over the sufficient condition presented in [5], and on a formalization of transaction programs, called *transaction templates*, facilitating fine-grained reasoning for robustness against RC. Key aspects of our formalization are the following:

- Conceptually, *transaction templates* are functions with parameters, and can, for instance, be derived from stored procedures inside a database system. Our abstraction generalizes transactions as usually studied in concurrency control research – sequences

Under submission.

of read and write operations – by making the objects worked on variable, determined by input parameters. Such parameters are *typed* to add additional power to the analysis.

- We support *atomic updates* (that is, a read followed by a write of the same database object, to make a relative change to its value) allowing us to identify some workloads as robust that otherwise would not be.
- Furthermore, we model database objects read and written at the granularity of fields, rather than just entire tuples, decoupling conflicts further and allowing to recognize additional cases that would not be recognizable as robust on the tuple level.

There are also a few restrictions to the model. We assume there is a fixed set of read-only attributes that cannot be updated and which are used to select tuples for update. The most typical example of this are primary key values passed to transaction templates as parameters. The inability to update primary keys is not an important restriction in many workloads, where keys, once assigned, never get changed, for regulatory or data integrity reasons. In general, this restriction on updating and query-based selection of the same fields deals with the fact that the static, workload-level analysis of the phantom problem quickly yields undecidability. This makes our results inapplicable in certain scenarios, but these assumptions are necessary to make robustness decidable for such a versatile class of workloads, and it seems an acceptable trade-off to obtain such a result. It can be hoped that future work will push this decidability frontier even further. These choices provide an interesting tradeoff between tractability and the ability to model and decide the robustness of more realistic workloads, as will be argued and illustrated throughout the remainder of the paper (as in Section 2 for the SmallBank benchmark).

In summary, the technical contributions of this paper are the following. (1) We provide a full characterization for robustness against RC for a workload of mere transactions instances (i.e., in the absence of variables). The characterization forms a main building block for the robustness results for transaction templates mentioned in (3) below. Our result is interesting in its own right as there are not many isolation levels for which complete characterizations are known. The seminal paper by Fekete [22] was the first to provide a characterisation for SI. More recently, such characterisations where obtained for RC and Read Uncommitted under a lock based rather than a multiversion semantics [30]. In fact, it was shown that robustness against RC under a lock-based semantics is coNP-complete which should be contrasted with the polynomial time algorithm for multiversion Read Committed obtained in this paper.

(2) We introduce the formalism of transaction templates and formally define how associated sets of workloads are defined. The new formalism takes into account the type of variables in operations, makes atomic updates explicit, and models database objects read and written at the granularity of fields rather than tuples.

(3) We obtain a polynomial time decision procedure for robustness against RC for workloads of transactions defined by transaction templates. This is the first time a sound and complete algorithm for robustness against RC on the level of transaction programs is obtained – that is, an algorithm that does not produce false positives nor false negatives. In this way, we extend the work in [5] that is based on a sufficient condition for robustness in the sense that false positives never occur but false negatives can. We discuss the implications of our algorithm in detail in Section 8.

(4) We assess the effectiveness of our approach by analyzing the SmallBank and TPC-C benchmarks and showing that our approach identifies robust subsets that are larger than those identified by previous methods. Still, neither SmallBank nor TPC-C is robust against RC when taking all transaction templates into account. We consider ways to make transaction templates robust by promoting selective read operations to update operations and assess the effectiveness of this method on both benchmarks. With these (save) adaptations, both full benchmarks become robust for RC.

(5) We experimentally demonstrate, using these two benchmarks and a well-known and unmodified DBMS, that our approach leads to practical performance improvements compared to when executed under SI or serializable SI, especially under higher contention. The performance improvements can be 10, 20, or, in extreme cases, more than 100%, depending on the scenario at hand.

Outline. We provide an extended example illustrating our results in Section 2 and discuss related work in Section 3. We introduce the necessary definitions in Section 4. We obtain a characterization for robustness against RC in Section 5. In Section 6 and 7, we define templates and present our results for deciding robustness for transaction templates. We discuss how to detect robust subsets in Section 8. We experimentally validate our approach in Section 9 and conclude in Section 10.

2 MOTIVATING EXAMPLE

The SmallBank [3] schema consists of tables Account(Name, CustomerID), Savings(CustomerID, Balance), and Checking(CustomerID, Balance) (key attributes are underlined). The Account table associates customer names with IDs. The other tables contain the balance (numeric value) of the savings and checking accounts of customers identified by their ID. The application code interacts with the database via the following transaction programs: Balance(N) returns the total balance (savings and checking) for a customer with name N. DepositChecking(N,V) makes a deposit of amount V in the checking account of the customer with name N (see Figure 1). TransactSavings(N,V) makes a deposit or withdrawal V on the savings account of the customer with name N. Amalgamate(N_1,N_2) transfers all the funds from customer N_1 to customer N_2. Finally, WriteCheck(N,V) writes a check V against the account of the customer with name N, penalizing if overdrawing.

Formalisation of transactions templates. Figure 2 displays the transaction templates for SmallBank. The corresponding SQL code is provided in Figure 11 in the appendix. A transaction template consists of a sequence of read, write, and update operations to a tuple X in a specific relation. For instance, R[X : Account{N, C}] indicates that a read operation is performed to a tuple in relation Account on the attributes Name and CustomerID. We abbreviate the names of attributes by their first letter to save space. The set $\{N, C\}$ is the read set of the read operation. Similarly, W and U refer to write and update operations to tuples of a specific relation. Write operations have an associated write set while update operations contain a read set followed by a write set: e.g., U[Z : DepositChecking{C, B}{B}}] first reads the CustomerID and Balance of tuple Z and then writes to the attribute Balance. All $\mathtt{R}$-, $\mathtt{W}$- and $\mathtt{U}$-operations always access exactly one tuple. A $\mathtt{U}$-operation is an atomic update that first reads the tuple and then writes to it. Templates serve as abstractions of transaction programs and represent an infinite number of possible workloads. For instance, disregarding attribute sets, $\{R[\mathtt{t}]R[\mathtt{v}]R[\mathtt{q}]U[\mathtt{q}], R[\mathtt{t}']R[\mathtt{v}']R[\mathtt{q}']U[\mathtt{q}'], R[\mathtt{t}]U[\mathtt{q}]\}$ is a workload consistent with the SmallBank templates as it contains two instantiations of WriteCheck and one instantiation of DepositChecking; $\{R[\mathtt{t}]R[\mathtt{v}]R[\mathtt{q}]U[\mathtt{q}']\}$ with $\mathtt{q} \neq \mathtt{q}'$ is not a valid workload as the two final operations in WriteCheck should be on the same object as required by the formalization. Typed variables effectively enforce domain constraints as we assume that variables that range over tuples of different relations can never be instantiated by the same value. For instance, in the transaction template for DepositChecking in Figure 2, $\mathtt{X}$ and $\mathtt{Z}$ can not be interpreted to be the same object.

Robust subsets. Figure 3 gives an overview of the maximal robust subsets that are detected using our methods for the SmallBank and TPC-C benchmarks (TPC-C is discussed in Section 7 and the templates are given in Figure 5). Transaction templates are presented in abbreviated form (e.g., Bal refers to Balance). To assess the effect of the different features of our abstraction we consider different settings: 'No updates' is the setting where updates are modeled through a read followed by a write and where read and write sets always specify the whole set of attributes (that is, conflicts are considered on the level of entire tuples). This setting can be seen to correspond to the one of [5] that only reports the set {Balance} as robust against RC.

The setting 'Atomic Updates' is the extension that models updates explicitly as atomic updates and already allows to detect relatively large robust sets compared to the 'No updates' setting. Indeed, for SmallBank {Am,DC,TS} is a robust subset indicating that any schedule using any number of instantiations of just these three templates that satisfies RC is serializable! Also for TPC-C larger robust subsets are detected.

Finally, 'Attr confl' no longer requires read and write sets to specify all attributes (that is, conflicts are specified on the level of attributes). To illustrate its importance, consider the operations $\mathtt{R}[\mathtt{X} : \text{Warehouse}\{\text{W, Inf}\}]$ and $\mathtt{U}[\mathtt{X} : \text{Warehouse}\{\text{W, YTD}\}\{\text{YTD}\}]$ coming from templates NewOrder and Payment, respectively, in the TPC-C benchmark as given in Figure 5. An instantiation of these template mapping $\mathtt{X}$ in both operations to the same tuple $\mathtt{t}$, does not result in a conflict as the read set of the former is disjoint from the write set of the latter. However, considering conflicts on the granularity of tuples, that is, read and write sets refer to all attributes, does result in a conflict. This difference in granularity has a profound effect for TPC-C as can be seen in the last row of Figure 3: a robust subset of four templates (out of five!) is found: {Del,Pay,NO,SL}. For SmallBank there is no improvement for the simple reason that tuple conflicts always imply attribute conflicts for this benchmark as all attribute conflicts are based on the same Balance attributes in Savings and Checking.

We do not claim that all features in our abstraction are novel. The novelty lies in their combination to push the frontier of the robustness problem for RC. Indeed, Figure 3 clearly shows that when taken together in an explicit formalisation, larger sets of transaction workloads can be safely determined to be robust. This is relevant since robust workloads can be executed under RC at increased throughput compared to SI or serializable SI (see Section 9.2).

```
DepositChecking(N,V):
  SELECT CustomerId INTO :X FROM Account WHERE Name=:N;
  UPDATE Checking SET Balance = Balance+:V
    WHERE CustomerId=:X;
  COMMIT;
```

Figure 1: SQL code for DepositChecking.

Balance:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{N, C}\}]$
$\mathtt{R}[\mathtt{Y} : \text{Savings}\{\text{C, B}\}]$
$\mathtt{R}[\mathtt{Z} : \text{Checking}\{\text{C, B}\}]$

DepositChecking:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{N, C}\}]$
$\mathtt{U}[\mathtt{Z} : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$

TransactSavings:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{N, C}\}]$
$\mathtt{U}[\mathtt{Y} : \text{Savings}\{\text{C, B}\}\{\text{B}\}]$

Amalgamate:

$\mathtt{R}[\mathtt{X}_1 : \text{Account}\{\text{N, C}\}]$
$\mathtt{R}[\mathtt{X}_2 : \text{Account}\{\text{N, C}\}]$
$\mathtt{U}[\mathtt{Y}_1 : \text{Savings}\{\text{C, B}\}\{\text{B}\}]$
$\mathtt{U}[\mathtt{Z}_1 : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$
$\mathtt{U}[\mathtt{Z}_2 : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$

WriteCheck:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{N, C}\}]$
$\mathtt{R}[\mathtt{Y} : \text{Savings}\{\text{C, B}\}]$
$\mathtt{R}[\mathtt{Z} : \text{Checking}\{\text{C, B}\}]$
$\mathtt{U}[\mathtt{Z} : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$

Figure 2: Transaction templates for SmallBank.

	SmallBank	TPC-C
No updates	{Bal}	{OS, SL}
Atomic Updates	{Am,DC,TS}, {Bal,DC}, {Bal,TS}	{Del,Pay,SL}, {NO, SL}, {Pay, OS, SL}
Attr confl	{Am,DC,TS}, {Bal,DC}, {Bal,TS}	{Del,Pay,NO,SL}

Figure 3: Robust subsets by analysis setting.

We refer to the appendix for a detailed robustness analysis for each combination of transaction templates.

3 RELATED WORK

3.1 Static robustness checking on the application level

Previous work on static robustness testing [5, 23] for transaction programs is based on the following key insight: when a *schedule* is not serializable, then the dependency graph constructed from that schedule contains a cycle satisfying a condition specific to the isolation level at hand: dangerous structure for SI and the presence of a counterflow edge for RC. This is extended to a workload of *transaction programs* via a so-called static dependency graph, where 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 guarantees robustness, while the presence of a cycle does not necessarily imply non-robustness. *We provide a formal approach to static robustness testing by making underlying assumptions more explicit within the formalism of transaction templates and obtain a decision procedure that is sound and complete for robustness testing against RC, allowing to detect larger subsets of transactions to be robust as exemplified in Section 2.*

Cerone et al. [13] provide a framework for uniformly specifying different isolation levels in a declarative way. A key assumption is *atomic visibility* requiring that either all or none of the updates of each transaction are visible to other transactions. Based on this framework, Bernardi and Gotsman [10] provide sufficient conditions for robustness against these isolation levels. Similar to the work of Fekete et al. [23], they first identify specific properties admitted by cycles in the dependency graphs of schedules that are allowed by the isolation level but not serializable. While analyzing robustness for a given set of program instances, they assume that each program instance is overestimated by three sets of tuples: those that might be read or written to by the program instance, and those that must be written to by the program instance. based on these sets, a static dependency graph is constructed. Analogous to [23], the absence of cycles with the property related to an isolation level in this graph guarantees that the set of program instances is robust against that isolation level. When analyzing robustness for a set of programs instead of specific program instances, a summary dependency graph is constructed, where each program is represented by a node. This graph is similar to static dependy graphs, but has additional information on the edges related to how the programs should be instantiated to create a specific conflict. This additional information reduces the number of workloads that are falsely identified to be non-robust. Continuing on this line of work, Cerone and Gotsman [14] later studied the problem of robustness against PARALLEL SNAPSHOT ISOLATION towards SI (i.e., whether for a given workload every schedule allowed under PARALLEL SNAPSHOT ISOLATION is allowed under SI). *This declarative framework cannot be used to study robustness against RC, as RC does not admit atomic visibility.*

Executing a non-robust workload under a lower isolation level usually increases throughput at the cost of increasing the number of anomalies. To better quantify this tradeoff for a given workload, Fekete et al. [24] presented a probabilistic model that predicts the rate of integrity violations depending on specific workload configurations. *This line of work is orthogonal to robustness, as a robust workload will increase throughput without introducing anomalies.*

3.2 Making transactions robust

When a workload is not robust against an isolation level, robustness can be achieved by modifying the transaction programs [2–5, 23], using an external lock manager [2, 5, 6], allocating some programs to higher isolation levels [4, 22], or even a combination of these techniques [2].

For SI, two code modification techniques to remove dangerous structures from the static dependency graph have been studied [2–4, 6, 23]: materialization and promotion. The materialization technique materializes conflicts between two potentially concurrent transactions by adding a new tuple to the database symbolizing this conflict and a write to this tuple is added to both transactions enforcing them to be non-concurrent. Alternatively, an external lock manager can be used [6]. The promotion technique promotes a read operation by adding an identity write to the same object. On some DBMS's, promotion can be implemented by changing the `SELECT` statement to `SELECT ... FOR UPDATE`. An alternative to code modification techniques is to allocate some transactions to S2PL instead of SI [22]. Alomari [2] considered a refinement that adds an additional write to each transaction running under S2PL.

For RC, Alomari and Fekete [5] consider materialization via two approaches: (1) an external lock manager and (2) by adding a write on a newly introduced tuple at the start of the transaction. Both techniques essentially take an exclusive lock at the start of the transaction that is released on commit. In contrast, *we employ a code modification technique based on promotion as for SI. This does not require additional tuples to be added to the database or locks to be taken at the start of the transaction. Instead, our approach promotes certain read operations to updates.*

3.3 Other approaches

Instead of weakening the isolation level, other approaches to increase transaction throughput without sacrificing ACID guarantees have been studied as well. Transactions can for example be split in smaller pieces to obtain performance benefits. However, this approach poses a new challenge, as not every serializable execution of these chopped transactions is necessarily equivalent to some serializable execution over the original transactions. A chopping of a set of transactions is correct if for every serializable execution of the chopping there exists an equivalent serializable execution of the original transactions. Shasha et al. [44] provide a graph based characterization for this correctness problem. This problem has been studied for different isolation levels such as SI [14] and PARALLEL SNAPSHOT ISOLATION [15] as well. However, in this case a correct chopping does not guarantee serializability. Instead, it verifies whether every execution of the chopped transactions allowed under an isolation level is equivalent to some execution of the original transactions allowed under this isolation level. *Transaction chopping has no direct relationship with robustness testing against RC.*

Another approach is to modify existing algorithms that guarantee serializability. One notable example is a modification of S2PL where a transaction might release some locks before it acquired all locks. Wolfson [49, 50] uses a sufficient condition to determine for a given workload at which point each lock acquired by a transaction might be released without risking anomalies.

When semantic knowledge of the transaction programs is available, it can be used to weaken the serializability requirement. Farrag and Özsu [21] use semantic knowledge of allowed interleavings between transactions to construct a new concurrency control algorithm that guarantees relatively consistent schedules. These relatively consistent schedules always preserve consistency, but do not necessarily guarantee serializability. Lu et al. [34] provide sufficient conditions under which every execution over a set of transactions under a given lock-based isolation level is semantically correct. A schedule is semantically correct if it has the same semantic effect as a serial schedule. As such, semantic correctness does not necessarily guarantee traditional serializability.

Many approaches to increase transaction throughput have been proposed: improved or novel pessimistic (cf., e.g., [28, 39, 41, 47, 51]) or optimistic (cf., e.g., [11, 12, 17, 18, 26, 27, 29, 31–33, 36, 42, 43, 52, 53]) concurrency control algorithms, or based on coordination avoidance (cf., e.g., [19, 20, 35, 38, 40, 45, 46]). *We do not compare to these as our focus lies on a technique that can be applied to standard DBMS's without any modifications to the database internals.*

4 DEFINITIONS

Databases. A *relational schema* is a set Rels of relation names, and for each $R \in \text{Rels}$, $\text{Attr}(R)$ is the finite set of associated attribute names. For every relation $R \in \text{Rels}$, we fix an infinite set $\mathbf{Tuples}_R$ of abstract objects called tuples. We assume that $\mathbf{Tuples}_R \cap \mathbf{Tuples}_S = \emptyset$ for all $R, S \in \text{Rels}$ with $R \neq S$. We then denote by **Tuples** the set $\bigcup_{R \in \text{Rels}} \mathbf{Tuples}_R$ of all possible tuples. By definition, for every $\mathtt{t} \in \mathbf{Tuples}$ there is a unique relation $R \in \text{Rels}$ such that $\mathtt{t} \in \mathbf{Tuples}_R$. In that case, we say that t is of *type R* and denote the latter by $\text{type}(\mathtt{t}) = R$. A *database* **D** over schema Rels assigns to every relation name $R \in \text{Rels}$ a finite set $R^{\mathbf{D}} \subset \mathbf{Tuples}_R$.

Transactions and Schedules. For a tuple $\mathtt{t} \in \mathbf{Tuples}$, we distinguish three operations R[t], W[t], and U[t] on t, denoting that tuple t is read, written, or updated, respectively. We say that the operation is on the tuple t. The operation U[t] is an atomic update and should be viewed as an atomic sequence of a read of t followed by a write to t. We will use the following terminology: a *read operation* is an R[t] or a U[t], and a *write operation* is a W[t] or a U[t]. Furthermore, an R-operation is an R[t], a W-operation is a W[t], and a U-operation is a U[t]. We also assume a special *commit* operation denoted C. To every operation o on a tuple of type R, we associate the set of attributes $\text{ReadSet}(o) \subseteq \text{Attr}(R)$ and $\text{WriteSet}(o) \subseteq \text{Attr}(R)$ containing, respectively, the set of attributes that o reads from and writes to. When o is a R-operation then $\text{WriteSet}(o) = \emptyset$. Similarly, when o is a W-operation then $\text{ReadSet}(o) = \emptyset$.

A *transaction T* is a sequence of read and write operations followed by a commit. Formally, we model a transaction as a linear order $(T, \leq_T)$, where T is the set of (read, write 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.

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 of different transactions, we add this id as a subscript to the operation. That is, we write $\mathtt{W}_i$[t], $\mathtt{R}_i$[t], and $\mathtt{U}_i$[t] to denote a W[t], R[t], and U[t] occurring in transaction T_i; similarly $\mathtt{C}_i$ denotes the commit operation in transaction T_i. This convention is consistent with the literature (see, *e.g.* [9, 22]). To avoid ambiguity of notation, we assume that a transaction performs at most one write, one read, and one update per tuple. The latter is a common assumption (see, *e.g.* [22]). 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 $(O_s, \leq_s, \ll_s, v_s)$ where O_s is the set containing all operations of transactions in $\mathcal{T}$ as well as a special operation op_0 conceptually writing the initial versions of all existing tuples, $\leq_s$ encodes the ordering of these operations, $\ll_s$ is a *version order* providing for each tuple t a total order over all write operations on t occurring in s, and v_s is a *version function* mapping each read operation a in s to either op_0 or to a write[1] operation different from a in s. We require that $op_0 \leq_s a$ for every operation $a \in O_s$, $op_0 \ll_s a$ for every write operation $a \in O_s$, and that $a <_T b$ implies $a <_s b$ for every $T \in \mathcal{T}$ and every $a, b \in T$. We furthermore require that for every read operation a, $v_s(a) <_s a$ and, if $v_s(a) \neq op_0$, then the operation $v_s(a)$ is on the same tuple as a. Intuitively, op_0 indicates the start of the schedule, the order of operations in s is consistent with the order of operations in every transaction $T \in \mathcal{T}$, and the version function maps each read operation a to the operation that wrote the version observed by a. If $v_s(a)$ is op_0, then a observes the initial version of this tuple. The version order $\ll_s$ represents the order in which different versions of a tuple are installed in the database. For a pair of write operations on the same tuple, this version order does not necessarily coincide with $\leq_s$. For example, under RC the version order is based on the commit order instead.

A schedule s is a *single version schedule* if $\ll_s$ coincides with $\leq_s$ and every read operation always reads the last written version of the tuple. Formally, for each pair of write operations a and b on the same tuple, $a \ll_s b$ iff $a <_s b$, and for every read operation a there is no write operation c on the same tuple as a with $v_s(a) <_s c <_s a$. A single version schedule over a set of transactions $\mathcal{T}$ is *single version serial* if its 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 of schedule is consistent with the common assumption [10, 22] that an underlying recovery mechanism will rollback aborted transactions. We only consider isolation levels that only read committed versions. Therefore there will never be cascading aborts.

Conflict Serializability. Let a_j and b_i be two operations on the same tuple from different transactions T_j and T_i in a set of transactions $\mathcal{T}$. We then say that a_j is *conflicting* with b_i if:

- *(ww-conflict)* $\text{WriteSet}(a_j) \cap \text{WriteSet}(b_i) \neq \emptyset$; or,
- *(wr-conflict)* $\text{WriteSet}(a_j) \cap \text{ReadSet}(b_i) \neq \emptyset$; or,
- *(rw-conflict)* $\text{ReadSet}(a_j) \cap \text{WriteSet}(b_i) \neq \emptyset$.

In this case, we also say that a_j and b_i are conflicting operations. Furthermore, commit operations and the special operation op_0 never conflict with any other operation. When a_j and b_i are conflicting operations in $\mathcal{T}$, we say that a_j *depends on* b_i in a schedule s over $\mathcal{T}$, denoted $b_i \rightarrow_s a_j$ if:[2]

- *(ww-dependency)* b_i is ww-conflicting with a_j and $b_i \ll_s a_j$; or,
- *(wr-dependency)* b_i is wr-conflicting with a_j and $b_i = v_s(a_j)$ or $b_i \ll_s v_s(a_j)$; or,
- *(rw-antidependency)* b_i is rw-conflicting with a_j and $v_s(b_i) \ll_s a_j$.

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 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 rw-antidependency from b_i to a_j implies that b_i observes a version installed before the version written by 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 conflicting operations a_j and b_i, $b_i \rightarrow_s a_j$ iff $b_i \rightarrow_{s'} a_j$.

Definition 1. *A schedule s is conflict serializable if it is conflict equivalent to a single version serial schedule.*

[1] Recall that a write operation is either a W[x] or a U[x].

[2] Throughout the paper, we adopt the following convention: a b operation can be understood as a 'before' while an a can be interpreted as an 'after'.

A *conflict graph* $CG(s)$ 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_i has an operation b_i that conflicts with an operation a_j in T_j and $b_i \rightarrow_s a_j$. The following is immediate from [37]:

Theorem 2. *A schedule s is conflict serializable iff the conflict graph for s is acyclic.*

Our formalisation of transactions and conflict serializability is based on [22], generalized to operations over attributes of tuples and extended with $\mathtt{U}$-operations that combine $\mathtt{R}$- and $\mathtt{W}$-operations into one atomic operation. These definitions are closely related to the formalization presented by Adya et al. [1], but we assume a total rather than a partial order over the operations in a schedule.

We do not concern ourselves with predicate reads here, as our workload model, formalized in Section 6, assumes that the selection of tuples is exclusively on attributes that do not get written. (See the remarks on this restriction in Section 1.) Since predicate reads do not influence conflict serializability in our setting, we omit them in our notation to facilitate presentation. This assumption is in line with other work on robustness (e.g. [5, 10, 22]).

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 ww-conflicting operations a_j and b_i in s on the same tuple $\mathtt{t}$ with $a_j \in T_j$, $b_i \in T_i$ and $T_j \neq T_i$ such that

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

That is, transaction T_j writes to an attribute of 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 $\ll_s$ corresponds to the commit order in s if for every pair of write operations $a_j \in T_j$ and $b_i \in T_i$, $b_i \ll_s a_j$ iff $\mathtt{C}_i <_s a_j$. We say that a schedule s is *read-last-committed (RLC)* if $\ll_s$ corresponds to the commit order and for every read operation a_j in s on some tuple $\mathtt{t}$ the following holds:

- $v_s(a_j) = op_0$ or $\mathtt{C}_i <_s a_j$ with $v_s(a_j) \in T_i$; and
- there is no write[3] operation $c_k \in T_k$ on $\mathtt{t}$ with $\mathtt{C}_k <_s a_j$ and $v_s(a_j) \ll_s c_k$.

That is, a_j observes the most recent version of $\mathtt{t}$ (according to the order of commits) that is committed before a_j. Note in particular that a schedule cannot exhibit dirty reads, defined in the traditional way [9], if it is read-last-committed.

Definition 3. *A schedule is allowed under isolation level read committed (RC) if it is read-last-committed and does not exhibit dirty writes.*

Robustness. Next, we define the robustness property [10] (also called *acceptability* in [22, 23]), which guarantees serializability for all schedules of a given set of transactions for a given isolation level.

Definition 4 (Robustness). *A set $\mathcal{T}$ of transactions is robust against RC if every schedule for $\mathcal{T}$ that is allowed under RC is conflict serializable.*

It is beneficial to model operations on the granularity of the attributes that are read or written.

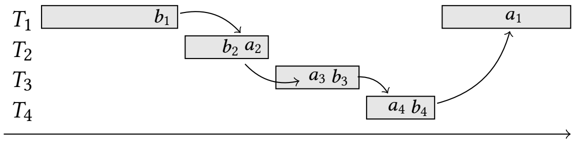


Figure 4: Multiversion split schedule for four transactions.

Example 5. *Consider transactions* $T_1 : \mathtt{R}_1[\mathtt{t}\{\mathtt{a},\mathtt{b},\mathtt{c}\}]\,\mathtt{W}_1[\mathtt{v}\{\mathtt{a}\}]\,\mathtt{C}_1$ *and* $T_2 : \mathtt{R}_2[\mathtt{v}\{\mathtt{b}\}]\,\mathtt{W}_2[\mathtt{t}\{\mathtt{a},\mathtt{b},\mathtt{d}\}]\,\mathtt{C}_2$. *Here, for example,* $\mathtt{R}_1[\mathtt{t}\{\mathtt{a},\mathtt{b},\mathtt{c}\}]$ *is shorthand for operation* $\mathtt{R}_1[\mathtt{t}]$ *with read set* $\{a, b, c\}$. *The two operations on* $\mathtt{v}$ *are in conflict if the concurrency control system of the DBMS works with tuple-level objects, but are not conflicting on the level of attributes. The workload is not robust on the tuple-level, as witnessed by the following schedule that is not (tuple-)conflict equivalent to a serial schedule* $s : \mathtt{R}_1[\mathtt{t}\{\mathtt{a},\mathtt{b},\mathtt{c}\}]\,\mathtt{R}_2[\mathtt{v}\{\mathtt{b}\}]\,\mathtt{W}_2[\mathtt{t}\{\mathtt{a},\mathtt{b},\mathtt{d}\}]\,\mathtt{C}_2\,\mathtt{W}_1[\mathtt{v}\{\mathtt{a}\}]\,\mathtt{C}_1$. *However, these two transactions are robust against RC at attribute-level granularity.*[4] *The order of the two operations on tuple* $\mathtt{t}$ *determines the order of the transactions in a conflict equivalent single version serial schedule. For example, the schedule s is conflict equivalent to the serial schedule $T_1 \cdot T_2$. So, modeling conflicts on the level of attributes allows to identify more workloads as robust.* □

5 ROBUSTNESS FOR TRANSACTIONS

Before introducing our formalisation for transaction templates in the next section, we start by studying the robustness problem for transactions. The results of the present section serve as a building block for our robustness algorithm for transaction templates.

A naive way to decide the robustness property for a set of transactions is to iterate over all possible schedules allowed under RC and verify that none violates conflict serializability. We show in the present section that only schedules with a very particular structure have to be considered which form the basis of a tractable decision procedure. We call these schedules *multiversion split schedules.*

In the next definition, we represent conflicting operations from transactions in a set $\mathcal{T}$ as quadruples (T_i, b_i, a_j, T_j) with b_i and a_j conflicting operations, and T_i and T_j their respective transactions in $\mathcal{T}$. We call these quadruples *conflict quadruples* for $\mathcal{T}$. Further, for an operation $b \in T$, we denote by $\mathsf{prefix}_b(T)$ the restriction of T to all operations that are before or equal to b according to $\leq_T$. Similarly, we denote by $\mathsf{postfix}_b(T)$ the restriction of T to all operations that are strictly after b according to $\leq_T$. Throughout the paper, we interchangeably consider transactions both as linear orders as well as sequences. Therefore, T is then equal to the sequence $\mathsf{prefix}_b(T)$ followed by $\mathsf{postfix}_b(T)$ which we denote by $\mathsf{prefix}_b(T) \cdot \mathsf{postfix}_b(T)$ for every $b \in T$.

Definition 6 (Multiversion split schedule). *Let $\mathcal{T}$ be a set of transactions and $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$ a sequence of conflict quadruples for $\mathcal{T}$ s.t. each transaction in $\mathcal{T}$ occurs in at most two quadruples. A multiversion split schedule for $\mathcal{T}$ based on C is a multiversion schedule that has the form*

$$\mathsf{prefix}_{b_1}(T_1) \cdot T_2 \cdot \ldots \cdot T_m \cdot \mathsf{postfix}_{b_1}(T_1) \cdot T_{m+1} \cdot \ldots \cdot T_n,$$

where

[3] Recall that a write operation is either a $\mathtt{W}$ or a $\mathtt{U}$-operation.

[4] This is under the reasonable assumption that a database can read and/or update all attributes of a tuple in one atomic step, which is for example the case in PostgreSQL and Oracle.

Algorithm 1: Deciding transaction robustness against RC.

```
Input  : Set of transactions $\mathcal{T}$
Output : $True$ iff $\mathcal{T}$ is robust against RC
for $T_1 \in \mathcal{T}$ do
    for $b_1$ a read operation in $T_1$ do
        $G$ := prefix-conflict-free-graph($b_1, T_1, \mathcal{T} \setminus \{T_1\}$);
        $TC$ := reflexive-transitive-closure of $G$;
        for $(T_2, T_m)$ in $TC$ do
            for $a_1 \in T_1$, $a_2 \in T_2$, $b_m \in T_m$ do
                if $a_1$ conflicts with $b_m$ and $b_1$ is
                  rw-conflicting with $a_2$ and ($b_1 <_{T_1} a_1$ or
                  $b_m$ is rw-conflicting with $a_1$ ) then
                    return $False$
return $True$
```

(1) there is no write operation in $\text{prefix}_{b_1}(T_1)$ *ww-conflicting with a write operation in any of the transactions* $T_2, \ldots, T_m$*;*

(2) $b_1 <_{T_1} a_1$ *or* b_m *is rw-conflicting with* a_1*; and,*

(3) b_1 *is rw-conflicting with* a_2*.*

Furthermore, $T_{m+1}, \ldots, T_n$ *are the remaining transactions in* $\mathcal{T}$ *(those not mentioned in* C*) in an arbitrary order.*

Figure 4 depicts a schematic multiversion split schedule. The name stems from the fact that the schedule is obtained by splitting one transaction in two (T_1 at operation b_1 in Figure 4) and placing all other transactions in C in between. The figure does not display the trailing transactions $T_{m+1}, T_{m+2}, \ldots$ and assumes $b_1 <_{T_1} a_1$. Intuitively, Condition (1) guarantees that s is allowed under RC, while Condition (2) and (3) ensure that C corresponds to a cycle in $CG(s)$.

The following theorem characterizes non-robustness in terms of the existence of a multiversion split schedule. The proof shows that for any counterexample schedule allowed under RC, a counterexample schedule can be constructed that is a multiversion split schedule, and that, conversely, any multiversion split schedule s gives rise to a cycle in the conflict-graph $CG(s)$.

Theorem 7. *For a set of transactions* $\mathcal{T}$*, this is equivalent:*

(1) $\mathcal{T}$ *is not robust against RC;*

(2) there is a multiversion split schedule s *for* $\mathcal{T}$ *based on some* C*.*

The above characterization for robustness against RC leads to a polynomial time algorithm that cycles through all possible split schedules. For this, we need to introduce the following notion. For a transaction T_1, an operation $b_1 \in T_1$ and a set of transactions $\mathcal{T}$ with $T_1 \notin \mathcal{T}$, define prefix-conflict-free-graph($b_1, T_1, \mathcal{T}$) as the graph containing as nodes all transactions in $\mathcal{T}$ that do not contain a ww-conflict with an operation in $\text{prefix}_{b_1}(T_1)$. Furthermore, there is an edge between two transactions T_i and T_j if T_i has an operation that conflicts with an operation in T_j.

Theorem 8. *Algorithm 1 decides whether a set of transactions* $\mathcal{T}$ *is robust against RC in time* $O(max\{k.|\mathcal{T}|^3, k^3.\ell\})$*, with* k *the total number of operations in* $\mathcal{T}$ *and* ℓ *the maximum number of operations in a transaction in* $\mathcal{T}$*.*

6 TRANSACTION TEMPLATES

Transaction templates are transactions where operations are defined over typed variables. Types of variables are relation names in Rels and indicate that variables can only be instantiated by tuples from the respective type.

We fix an infinite set of variables **Var** that is disjoint from **Tuples**. Every variable $\mathsf{X} \in \mathbf{Var}$ has an associated relation name in Rels as type that we denote by type(X).

Definition 9. *A transaction template* τ *is a transaction over* **Var**. *In addition, for every operation* o *in* τ *over a variable* X*,* $ReadSet(o) \subseteq Attr(type(X))$ *and* $WriteSet(o) \subseteq Attr(type(X))$*.*

Notice that operations in transaction templates are defined over typed variables whereas they are over **Tuples** in transactions. Indeed, the transaction template for Balance in Figure 2 contains a read operation $o = \mathsf{R}[\mathsf{X} : \text{Account}\{\mathsf{N}, \mathsf{C}\}]$. As explained in Section 2, the notation $\mathsf{X} : \text{Account}\{N, C\}$ is a shorthand for type(X) = Account and ReadSet(o) = $\{N, C\}$.

Recall that we denote variables by capital letters X, Y, Z and tuples by small letters t, v. A variable assignment μ is a mapping from **Var** to **Tuples** such that $\mu(\mathsf{X}) \in \mathbf{Tuples}_{\text{type}(\mathsf{X})}$. By $\mu(\tau)$, we denote the transaction obtained by replacing each variable X in τ with $\mu(\mathsf{X})$. A variable assignment for a database **D** maps every variable to a tuple occurring in a relation in **D**.

A set of transactions $\mathcal{T}$ is *consistent* with a set of transaction templates $\mathcal{P}$ and database **D**, if for every transaction T in $\mathcal{T}$ there is a transaction template $\tau \in \mathcal{P}$ and a variable assignment μ_T for **D** such that $\mu_T(\tau) = T$.

Let $\mathcal{P}$ be a set of transaction templates and **D** be a database. Then, $\mathcal{P}$ is *robust against RC over* ***D*** if for every set of transactions $\mathcal{T}$ that is consistent with $\mathcal{P}$ and **D**, it holds that $\mathcal{T}$ is robust against RC.

Definition 10 (Robustness). *A set of transaction templates* $\mathcal{P}$ *is robust against RC if* $\mathcal{P}$ *is robust against RC for every database* ***D***.

Example 11. *Consider the database* ***D*** *over the SmallBank schema:* $Account^{\mathbf{D}} = \{a_1, a_2\}$*;* $Savings^{\mathbf{D}} = \{s_1, s_2\}$*; and,* $Checking^{\mathbf{D}} = \{c_1, c_2\}$*. For simplicity, we ignore read and write sets. Let* $\mathcal{T}_1 = \{\mathtt{R[a_1]R[s_1]R[c_1]}, \mathtt{R[a_1]R[a_2]U[s_1]U[c_1]U[c_2]}\}$*. Then* $\mathcal{T}_1$ *is consistent with the SmallBank transaction templates and* ***D*** *as witnessed by the transaction templates Balance and Amalgamate, and the variable assignments* $\mu_1 = \{X \to a_1, \mathsf{Y} \to s_1, \mathsf{Z} \to c_1\}$ *and* $\mu_2 = \{X_1 \to a_1, X_2 \to a_2, \mathsf{Y}_1 \to s_1, \mathsf{Y}_2 \to s_2, \mathsf{Z}_2 \to c_2\}$*. The set {Balance, Amalgamate} is not robust against RC, witnessed by* ***D*** *and* $\mathcal{T}_1$*. Indeed, we can construct a multiversion split schedule over* $\mathcal{T}_1$*:*

$$\begin{array}{llll} T_1: & \mathtt{R_1[a_1]R_1[s_1]} & & \mathtt{R_1[c_1]C_1} \\ T_2: & & \mathtt{R_2[a_1]R_2[a_2]U_2[s_1]U_2[c_1]U_2[c_2]C_2} & \end{array}$$

□

7 ROBUSTNESS FOR TEMPLATES

Algorithm 1 cannot be applied directly to test robustness for transaction templates as there are infinitely many sets of transactions $\mathcal{T}$ consistent with a given set of transaction templates $\mathcal{P}$. We use a different approach that resembles Algorithm 1 but that operates directly over transaction templates.

Central to the proposed algorithm (Algorithm 2) is a generalization of conflicting operations: For transaction templates τ_i and τ_j in $\mathcal{P}$, we say that an operation $o_i \in \tau_i$ is *potentially conflicting* with an operation $o_j \in \tau_j$ if o_i and o_j are operations over a variable of the same type, and at least one of the following holds:

- $\text{WriteSet}(o_i) \cap \text{WriteSet}(o_j) \neq \emptyset$ (potentially ww-conflicting);
- $\text{WriteSet}(o_i) \cap \text{ReadSet}(o_j) \neq \emptyset$ (potentially wr-conflicting); or
- $\text{ReadSet}(o_i) \cap \text{WriteSet}(o_j) \neq \emptyset$ (potentially rw-conflicting).

Intuitively, potentially conflicting operations lead to conflicting operations when the variables of these operations are mapped to the same tuple by a variable assignment. Analogously to conflicting quadruples over a set of transactions as in Definition 6, we consider *potentially conflicting quadruples* $(\tau_i, o_i, p_j, \tau_j)$ over a set of transaction templates $\mathcal{P}$ with $\tau_i, \tau_j \in \mathcal{P}$, and $o_i \in \tau_i$ an operation that is potentially conflicting with an operation $p_j \in \tau_j$. A sequence of potentially conflicting quadruples $D = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_m, o_m, p_1, \tau_1)$ over $\mathcal{P}$ (in which multiple occurrences of the same transaction template are allowed) induces a sequence of conflicting quadruples $C = (T_1, b_1, a_2, T_2), \ldots, (T_m, b_m, a_1, T_1)$ by applying a variable mapping μ_i to each τ_i in D. We call such a set of variable mappings simply a *variable mapping* for D, denoted $\bar{\mu}$, and write $\bar{\mu}(D) = C$.

A basic insight is the following: if there is a multiversion split schedule s for some C over a set of transactions $\mathcal{T}$ consistent with $\mathcal{P}$ and a database $\mathbf{D}$ with the properties of Definition 6, then there is a sequence of potentially conflicting quadruples D such that $\bar{\mu}(D) = C$ for some $\bar{\mu}$. The approach followed in Algorithm 2 is then to enumerate sequences D together with mappings $\bar{\mu}$ in search of $\bar{\mu}(D)$ for which the conditions of Definition 6 are satisfied.

First, we show in Lemma 12 that for each D, we only need to consider one mapping $\bar{\mu}$ of a canonical form that partitions the mapped variables into three or four disjoint sets: all variables connected to o_1 (in a way to be made precise next), all variables connected to p_1 (when o_1 and p_1 are themselves connected, the two sets coincide), all variables in τ_1 not in the previous two sets, and all remaining variables in all other templates. Furthermore, at most four different tuples for each variable type are needed. We need the following notion: a variable X in τ_i is *connected* to an operation o in τ_j in D if either $i = j$ and X is the variable of operation o; there is a potentially conflicting quadruple (τ_i, o', p', τ_j) with o' having the same variable as o and p' having variable X; or X is connected to an operation whose variable is connected to o.

We encode the choice of tuples for variables through (total) functions $c : \text{Rels} \to \textbf{Tuples}$ that we call *type mappings* and which map a relation onto a particular tuple of that relation's type. The canonical mapping $\bar{\mu}$ for $D = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_m, o_m, p_1, \tau_1)$ is defined relative to four type mappings c_1, c_2, c_3, and c_4, whose ranges do not matter as long as they are all different. Then $\bar{\mu}$ consists of the following set of m variable mappings μ_i for occurrences τ_i of transaction template in D. For μ_1,

$$\mu_1(\mathsf{X}) = \begin{cases} c_1(\text{type}(\mathsf{X})) & \text{if X is the variable of } o_1, \\ c_2(\text{type}(\mathsf{X})) & \text{if X is connected to } p_1 \text{ and not to } o_1, \\ c_4(\text{type}(\mathsf{X})) & \text{otherwise.} \end{cases}$$

For every $1 < i \leq m$,

$$\mu_i(\mathsf{X}) = \begin{cases} c_1(\text{type}(\mathsf{X})) & \text{if X is connected to } o_1, \\ c_2(\text{type}(\mathsf{X})) & \text{if X is connected to } p_1 \text{ and not to } o_1, \\ c_3(\text{type}(\mathsf{X})) & \text{otherwise.} \end{cases}$$

Lemma 12. *Let $\mathcal{P}$ be a set of transaction templates. The following are equivalent:*

- *$\mathcal{P}$ is not robust against RC;*
- *there is a multiversion split schedule s for some C over a set of transactions $\mathcal{T}$ consistent with $\mathcal{P}$ and a database $\mathbf{D}$, where C is induced by a sequence of potentially conflicting quadruples D over $\mathcal{P}$ and its canonical variable mapping.*

Furthermore, for every sequence of potentially conflicting quadruples D over $\mathcal{P}$ and every variable mapping $\bar{\mu}$ for D, there is a database $\mathbf{D}$ where the transactions in the induced sequence of conflicting quadruples are consistent with.

Example 13. *We provide a more elaborate example justifying the need for exactly four tuples of the same type in a counterexample. Consider the set of transaction templates $\mathcal{P} = \{\tau_1, \tau_2\}$ with*

$\tau_1 : \mathtt{W}_1[\mathtt{Y} : \mathtt{S}\{B\}]\,\mathtt{W}_1[\mathtt{Z} : \mathtt{S}\{A\}]\,\mathtt{W}_1[\mathtt{X} : \mathtt{S}\{A,B\}]\,\mathtt{R}_1[\mathtt{Y} : \mathtt{S}\{A\}]\,\mathtt{W}_1[\mathtt{Z} : \mathtt{S}\{B\}]$,

$\tau_2 : \mathtt{W}_2[\mathtt{X} : \mathtt{S}\{A,B\}]\,\mathtt{W}_2[\mathtt{Y} : \mathtt{S}\{A\}]\,\mathtt{W}_2[\mathtt{Z} : \mathtt{S}\{B\}]$,

and let D be the sequence of potentially conflicting quadruples

$(\tau_1, \mathtt{R}_1[\mathtt{Y} : \mathtt{S}\{A\}], \mathtt{W}_2[\mathtt{Y} : \mathtt{S}\{A\}], \tau_2), (\tau_2, \mathtt{W}_2[\mathtt{Z} : \mathtt{S}\{B\}], \mathtt{W}_1[\mathtt{Z} : \mathtt{S}\{B\}], \tau_1)$.

Then, the multiversion split schedule s_2 based on the sequence of conflict quadruples C induced by D and its canonical variable mapping is as follows (we assume $c_i(S) = \mathtt{t}_i$ for $i \in \{1, 2, 3, 4\}$):

$$\begin{array}{llll} T_1 : & \mathtt{W}_1[\mathtt{t}_1]\,\mathtt{W}_1[\mathtt{t}_2]\,\mathtt{W}_1[\mathtt{t}_4]\,\mathtt{R}_1[\mathtt{t}_1] & & \mathtt{W}_1[\mathtt{t}_2]\,\mathtt{C}_1 \\ T_2 : & & \mathtt{W}_2[\mathtt{t}_3]\,\mathtt{W}_2[\mathtt{t}_1]\,\mathtt{W}_2[\mathtt{t}_2]\,\mathtt{C}_2 & \end{array}$$

There are no dirty writes in s_2, as the write operations on $\mathtt{t}_1$ and $\mathtt{t}_2$ in $\text{prefix}_{\mathtt{R}_1[\mathtt{t}_1]}(T_1)$ write to attributes disjoint from the write operations on $\mathtt{t}_1$ and $\mathtt{t}_2$ in T_2. It is not possible to construct this schedule with less than four tuples, as trying to replace any two tuples t_i and t_j with one tuple leads to a dirty write invalidating the schedule under RC. □

To cycle through all possible sequences D, Algorithm 2 iterates over the possible split transaction templates $\tau_1 \in \mathcal{P}$ and its possible operations $o_1, p_1 \in \tau_1$, and relies on a graph referred to as pt-prefix-conflict-free-graph$(o_1, p_1, h, \tau_1, \mathcal{P})$. Here, $h \in \{1, 2\}$ signals that the prefix and suffix of the split of τ_1 use the same type mapping c_1 when $h = 1$ and that the suffix uses type mapping c_2 when $h = 2$. The graph has as nodes the quadruples (τ, o, i, j) with $\tau \in \mathcal{P}$, $o \in \tau$, $i \in \{1, 2, 3\}$ and $j \in \{\text{in}, \text{out}\}$. Here, $i \in \{1, 2, 3\}$ encodes that o is assigned the type mapping c_i in τ (the type mapping c_4 is not used). There will be two types of edges: (1) inner edges $(\tau, o, i, \text{in}) \to (\tau, p, i', \text{out})$ that stay within the same transaction τ and indicate how the type mapping changes (or stays the same) from c_i for o to $c_{i'}$ for p; and (2) outer edges $(\tau, o, i, \text{out}) \to (\tau', p, i', \text{in})$ between different occurrences of transaction templates encoding a potentially conflicting quadruple (τ, o, p, τ') and maintaining information on type mappings as well.

More formally, a quadruple node (τ, o, i, j) in the graph satisfies the following properties:

(a) $i = 1$ implies that there is no operation $o_1' \in \text{prefix}_{o_1}(\tau_1)$ over the same variable as o_1 in τ_1 s.t. o_1' is potentially ww-conflicting with an operation over the same variable as o in τ.

(b) $i = h$ implies that there is no operation $o_1' \in \text{prefix}_{o_1}(\tau_1)$ over the same variable as p_1 in τ_1 s.t. o_1' is potentially ww-conflicting with an operation over the same variable as o in τ.

Algorithm 2: Deciding transaction template robustness against RC.

```
Input  : Set of transaction templates $\mathcal{P}$
Output: $True$ iff $\mathcal{P}$ is robust against RC
for $\tau_1 \in \mathcal{P}$ do
    for $o_1$ an operation in $\tau_1$, $(p_1, i) \in \tau_1 \times \{1, 2\}$ do
        $G$ := pt-prefix-conflict-free-graph$(o_1, p_1, i, \tau_1, \mathcal{P})$;
        $TC$ := transitive-closure of $G$;
        for $\tau_2, \tau_m$ in $\mathcal{P}$ do
            for $p_2 \in \tau_2$, $o_m \in \tau_m$ do
                if $p_1$ is potentially conflicting with $o_m$ and $o_1$
                 is potentially rw-conflicting with $p_2$ and
                 ($o_1 <_{\tau_1} p_1$ or $o_m$ is potentially
                 rw-conflicting with $p_1$) and
                 $\langle(\tau_2, p_2, 1, in), (\tau_m, o_m, i, out)\rangle$ in $TC$ then
                    return $False$
return $True$
```

Conditions (a) and (b) on the nodes, ensure that condition (1) of Definition 6 is always guaranteed for all possible variable mappings that are consistent with the particular choice of type mapping. Furthermore, two nodes (τ, o, i, j) and (τ', o', i', j') are connected by a directed edge if either

(†) $\tau = \tau'$, j = in, j' = out, and if o and o' are over the same variable in τ, then $i = i'$ (i.e., remain within the same transaction and change the type mapping only when o and o' are not over the same variable); or,

(‡) j = out, j' = in, $i = i'$ and o and o' are potentially conflicting (i.e., the analogy of b and a for consecutive transactions in a split schedule, but here defined for transaction templates).

Theorem 14. *Algorithm 2 Decides whether a set of transaction templates $\mathcal{P}$ is robust against RC in time $O(k^4.\ell)$ with k the total number of operations in $\mathcal{P}$ and ℓ the maximum number of operations in transactions of $\mathcal{P}$.*

8 DETECTING ROBUST SETS

As every subset of a robust set of templates is robust as well, maximal robust subsets of a workload $\mathcal{P}$ can be detected by running Algorithm 2 first on $\mathcal{P}$ itself and if necessary on smaller subsets. Even though there are exponentially many possible subsets, $\mathcal{P}$ is expected to be small and robustness tests can be performed in a static and offline analysis phase.

Algorithm 2 allows for a complete characterization of robustness at attribute-level granularity. We discuss the ramifications of using these results with a DBMS whose concurrency control subsystem works at the granularity of tuples. In this case, an RC implementation isolates more strongly than actually needed to assure serializability on workloads our techniques identify as robust.[5]

There are two ways to employ our decision procedures in this case. The first is to simply coarsen the workload model to the tuple-level by setting, for each operation, the read and write sets to all the attributes of the tuple. In this way, our algorithms give a correct and complete answer at tuple-level granularity. As discussed in Section 2, the row 'Atomic updates' in Figure 3 indicates which sets are robust under this method for SmallBank and TPC-C and, how this improves over considering only reads and writes.

The second approach is to simply work with the attribute-level model and accept that the DBMS is more conservative than necessary. When our algorithm determines a workload to be robust, that workload will still be robust on systems that assure RC 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 RC. However, when our algorithm determines a workload *not* to be robust, they may be too conservative: they might do so by identifying a complete set of counterexample schedules, none of which may actually be allowed under RC at the granularity of tuples. Thus, our attribute-level algorithm technically provides only a *sufficient* rather than a complete condition for robustness on such systems. The second technique nevertheless strictly dominates the first on SmallBank and TPC-C (as can be seen in the row 'Attr confl' in Figure 3), even when the DBMS works with tuple-level objects. It detects all the robust cases of the former approach, plus potentially additional ones that can only be found by attribute-level analysis, but which still are robust on a DBMS with tuple-level concurrency control.

9 EXPERIMENTS

9.1 Experimental Setup

9.1.1 PostgreSQL. We used PostgreSQL 12.4 as a database engine. PostgreSQL uses multiversion concurrency control to implement three different isolation levels: Read Committed (RC), Snapshot isolation (SI), and Serializable Snapshot Isolation (SSI) [23].[6] When reading a tuple, RC reads the last committed version before this read operation, whereas SI and SSI see the last committed version before the start of the transaction. All isolation levels use write locks to avoid dirty writes. If a transaction T_1 wants to update a tuple that has been changed by a concurrent transaction T_2, transaction T_1 will wait for T_2 to commit or abort, thereby releasing the write lock, before proceeding. Notice that in specific cases this can lead to deadlocks, e.g. when multiple concurrent transactions try to update the same set of tuples. Under SI and SSI, T_1 will abort if T_2 successfully committed, according to the first-updater-wins principle. When using SSI, PostgreSQL will furthermore monitor for possible conditions that could lead to unserializable executions, and possibly abort transactions to preserve serializability.

The database system runs on a server with two 2.3 GHz Xeon Gold 6140 CPUs with 18 cores each, 192 GB RAM, and a 200 GB SSD local disk. A separate machine is used to issue the transactional workload to the database system through a low-latency connection. The workload is supplied via a number of concurrently running client processes. Each client sequentially runs transactions from randomly selected transaction templates through this same database connection. When a transaction is aborted, the client immediately retries this transaction with the same parameters, until it eventually

[5] For instance, RC in PostgreSQL acquires locks on the granularity of tuples rather than attributes – see Section 9.1.1 for a more detailed description.

[6] In PostgreSQL 12.4, these three isolation levels are referred to as Read Committed, Repeatable Read, and Serializable, respectively. See https://www.postgresql.org/docs/12/transaction-iso.html for more information.

NewOrder:

$\mathtt{R}[\mathtt{X} : \text{Warehouse}\{\text{W, Inf}\}]$
$\mathtt{U}[\mathtt{Y} : \text{District}\{\text{W, D, Inf, N}\}\{\text{N}\}]$
$\mathtt{R}[\mathtt{Z} : \text{Customer}\{\text{W, D, C, Inf}\}]$
$\mathtt{W}[\mathtt{S} : \text{Order}\{\text{W, D O, C, Sta}\}]$
$\mathtt{U}[\mathtt{T}_1 : \text{Stock}\{\text{W, I, Qua}\}\{\text{Qua}\}]$
$\mathtt{W}[\mathtt{V}_1 : \text{OrderLine}\{\text{W, D, O, OL, I, Del, Qua}\}]$
$\mathtt{U}[\mathtt{T}_2 : \text{Stock}\{\text{W, I, Qua}\}\{\text{Qua}\}]$
$\mathtt{W}[\mathtt{V}_2 : \text{OrderLine}\{\text{W, D, O, OL, I, Del, Qua}\}]$

Delivery:

$\mathtt{U}[\mathtt{S} : \text{Order}\{\text{W, D, O}\}\{\text{Sta}\}]$
$\mathtt{U}[\mathtt{V}_1 : \text{OrderLine}\{\text{W, D, O, OL, Del}\}\{\text{Del}\}]$
$\mathtt{U}[\mathtt{V}_2 : \text{OrderLine}\{\text{W, D, O, OL, Del}\}\{\text{Del}\}]$
$\mathtt{U}[\mathtt{Z} : \text{Customer}\{\text{W, D, C, Bal}\}\{\text{Bal}\}]$

Payment:

$\mathtt{U}[\mathtt{X} : \text{Warehouse}\{\text{W, YTD}\}\{\text{YTD}\}]$
$\mathtt{U}[\mathtt{Y} : \text{District}\{\text{W, D, YTD}\}\{\text{YTD}\}]$
$\mathtt{U}[\mathtt{Z} : \text{Customer}\{\text{W, D, C, Bal}\}\{\text{Bal}\}]$

OrderStatus:

$\mathtt{R}[\mathtt{Z} : \text{Customer}\{\text{W, D, C, Inf, Bal}\}]$
$\mathtt{R}[\mathtt{S} : \text{Order}\{\text{W, D, O, C, Sta}\}]$
$\mathtt{R}[\mathtt{V}_1 : \text{OrderLine}\{\text{W, D, O, OL, I, Del, Qua}\}]$
$\mathtt{R}[\mathtt{V}_2 : \text{OrderLine}\{\text{W, D, O, OL, I, Del, Qua}\}]$

StockLevel:

$\mathtt{R}[\mathtt{T} : \text{Stock}\{\text{W, I, Qua}\}]$

Figure 5: Abstraction for the TPC-C transaction templates. Attribute names are abbreviated.

commits. For 60 seconds, we measure the number of transactions that are committed and the number of aborts. Each experiment is repeated 5 times. The graphs in this section show both the average values, as well as 95% confidence intervals.

9.1.2 SmallBank benchmark (see Section 2). The database is populated with 18000 randomly generated accounts with corresponding checking and savings accounts – as in earlier experiments on the SmallBank benchmark in [3, 5]. Each client uses a uniform distribution when selecting one of the possible templates. To select which accounts to address, we considered two approaches. The first approach fixes a small subset of accounts, referred to as the *hotspot*, and a probability for an account selected for use in a transaction to be from among the hotspot accounts, referred to as the *hotspot probability.* Within the hotspot, each account has an equal probability of being selected. The second approach uses a Zipfian distribution to randomly select accounts [25].

9.1.3 TPC-C benchmark. The second benchmark is based on the TPC-C benchmark [48]. We modified the schema and templates to turn all predicate reads into key-based accesses. The schema consists of six relations:

- Warehouse(<u>WarehouseID</u>, Info, YTD),
- District(<u>WarehouseID</u>, <u>DistrictID</u>, Info, YTD, NextOrderID),
- Customer(<u>WarehouseID</u>, <u>DistrictID</u>, <u>CustID</u>, Info, Balance),
- Order(<u>WarehouseID</u>, <u>DistrictID</u>, <u>OrderID</u>, CustID, Status),
- OrderLine(<u>WarehouseID</u>, <u>DistrictID</u>, <u>OrderID</u>, <u>OrderLineID</u>, ItemID, DeliveryInfo, Quantity), and
- Stock(<u>WarehouseID</u>, <u>ItemID</u>, Quantity).

We focus on five different transaction templates:

(1) NewOrder($W, D, C, I_1, Q_1, I_2, Q_2, \ldots$): creates a new order for the customer identified by (W, D, C). The id for this order is obtained by increasing the NextOrderID attribute of the District tuple identified by (W, D) by one. Each order consists of a number of items $I_1, I_2, \ldots$ with respectively quantities $Q_1, Q_2, \ldots$. For each of these items, a new OrderLine tuple is created and the related stock quantity is decreased.
(2) Payment(W, D, C, A): represents a customer identified by (W, D, C) paying an amount A. This payment is reflected in the database by increasing the balance of this customer by A. This amount is furthermore added to the YearToDate (YTD) income of both the related warehouse and district.
(3) OrderStatus(W, D, C, O): requests information about the current status of the order identified by (W, D, O). This transaction template collects information of the customer identified by (W, D, C) who created the order, the order itself, and the different OrderLine tuples related to this order.
(4) Delivery(W, D, C, O): delivers the order represented by (W, D, O). The status of the order is updated, as well as the DeliveryInfo attribute of each OrderLine tuple related to this order. The total price of the order is deduced from the balance of the customer who made this order, identified by (W, D, C).
(5) StockLevel(W, I): returns the current stock level of item I in warehouse W.

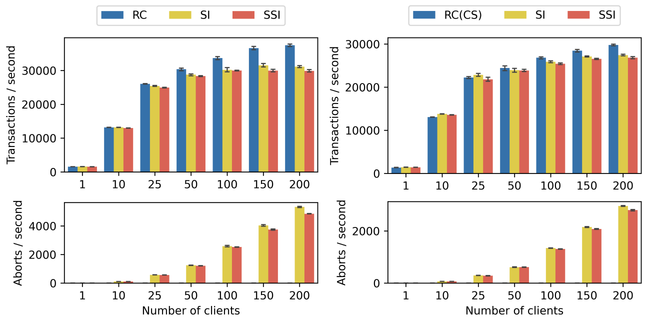
An abstraction of each transaction template is given in Figure 5. The experiments adhere to the requirements of the official TPC-C benchmark [48], with a scaling factor of 25 warehouses. This means that the database is populated with 25 warehouses, where each warehouse is assigned 10 districts and 100000 different stock items. Each district has 3000 customers, and each customer initially has 10 orders. We randomly assign between 5 and 15 orderlines per order.[7] Each client uses a uniform distribution when selecting one of the possible templates. When generating parameters for each transaction, we remain consistent with the TPC-C benchmark. That is, we use a uniform distribution to randomly pick warehouses, districts, items within a warehouse and orders for a customer. Customers within a district are non-uniformly selected based on a Zipfian distribution. We consider one additional setting where warehouses are selected according to a Zipfian distribution.

9.1.4 Overview. We show how robustness can improve transaction throughput. We focus attention in Section 9.2 on a maximal subset of three transaction templates for the SmallBank benchmark that is still robust against RC and compare performance against SI and SSI (even though SmallBank is not robust against SI). Results for TPC-C are similar and not reported. In Section 9.3, we consider experiments where both benchmarks are made robust against RC by promotion of reads to updates.

9.2 Robust workloads

In the experiments below, we show the potential performance benefits of using a lower isolation level over a robust subset of the SmallBank benchmark. The first experiment explores the influence of the number of concurrent clients on both throughput and abort rate. For this experiment, we used a workload of three transaction templates {DepositChecking, TransactSavings and Amalgamate}, since this workload is the largest subset of the Smallbank benchmark that is robust against RC. For this experiment, a hotspot size of 1000 accounts with a hotspot probability of 90% was used. The

[7] Figure 5 shows only two orderlines per order to simplify presentation.



(a) Robust subset of SmallBank benchmark.

(b) Complete SmallBank benchmark with promotion for RC.

Figure 6: Throughput and abort rate per number of concurrent clients for (a subset of) SmallBank. The hotspot consists of 1000 accounts with a hotspot probability of 90%.

results of this experiment are shown in Figure 6a. When the number of clients is low, the different isolation levels result in a similar throughput. However, if the number of concurrent clients increases, RC clearly outperforms both SI and SSI. This is to be expected, since the high number of concurrent clients leads to more concurrent transactions trying to update the same tuple, and consequently more aborts under SI and SSI due to the first-updater-wins principle. It should be noted that under RC, aborts can still occur due to deadlocks, but these aborts are quite rare. In this experiment, the number of aborts under RC never exceeded 0.15 aborts per second.

We next consider different levels of data skew on the dataset. Figure 7a, Figure 7b and Figure 7c show the throughput for different hotspot probabilities when there are respectively 1000, 100 and 10 accounts in the hotspot. Figure 7d shows the throughput for different skew parameters when using a Zipfian distribution. When the data skew increases, RC greatly outperforms the other two isolation levels. However, when contention further increases, the throughput of RC decreases drastically due to transactions waiting for write locks to be released. In Figure 7c, the number of aborts under RC due to detected deadlocks increases to around 33 aborts per second when the hotspot probability is 90%.

Similar findings are obtained when considering maximal subsets of the TPC-C benchmark, for instance, {Payment, OrderStatus and StockLevel}, that are robust against RC.

Conclusion. When a set of transaction templates is robust against RC, choosing this lower isolation level never results in a performance loss. This is to be expected, since SI and SSI require additional overhead when checking for possible serialization failures that require an abort. RC greatly outperforms the other isolation levels for settings with higher contention. Indeed, due to the first-updater-wins principle, SI and SSI need to abort a transaction when two concurrent transactions write to the same object. Higher contention increases this probability, resulting in an increased abort rate.

9.3 Promoted workloads

When a set of transaction templates is not robust, we propose a template modification technique based on insights from Definition 6: an equivalent set of transaction templates robust against RC can be created by promoting R-operations to U-operations that write back the read value. Such a change does not alter the effect of the transaction template, but the newly introduced write operation will trigger concurrency mechanisms in the database. Since Definition 6 requires that operation b_1 is rw-conflicting with a_1 (Condition (3)), but not ww-conflicting with a_1 (Condition (1)), promoting *all* R-operations to U-operations is sufficient to guarantee robustness against RC. The promotion approach is inspired by a technique introduced by Fekete et al. [23] to make a workload robust against SI. However, in contrast to their approach, which introduces additional write operations, we promote an existing R-operation into a U-operation. Fortunately, it is not always necessary to promote all R-operations to obtain robustness against RC as we discuss next.

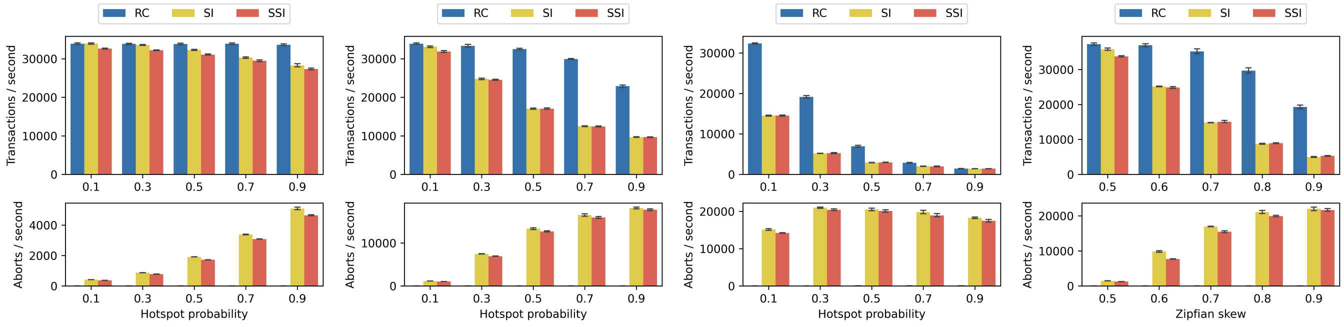
To find a minimal set of R-operations to promote, we can iteratively promote R-operations to U-operations and apply Algorithm 2 to check whether the resulting workload is robust. We applied this technique on both SmallBank and TPC-C to guarantee robustness with a minimal number of promotions.

For SmallBank, we can obtain robustness by only promoting all R-operations over the Checking and Savings relations to U-operations leaving all other R-operation intact. In our experiments, we refer to this as RC(CS). Furthermore, this set of promoted R-operations is minimal: if one of the R-operations over the Checking or Savings relations remains, Algorithm 2 reveals that the resulting set of transaction templates is not robust against RC. For TPC-C, we can obtain robustness by promoting all R-operations over the Customer, Order and OrderLine relations in the OrderStatus template while all other templates remain unchanged. We refer to this promotion as RC(CO). To contrast our approach based on attribute-level conflicts with the one based on tuple-level conflicts, we also investigate how to make TPC-C robust when the read and write sets of operations refer to all attributes in the corresponding relations. Again we applied Algorithm 2 and obtained that all R-operations on tuples over the Warehouse-, Customer- Order- and OrderLine-relations need to be promoted to U-operations, requiring changes in both NewOrder and OrderStatus. We refer to this promotion as RC(WHC). Both promotion strategies are again minimal, since we cannot promote only a strict subset of these R-operations to U-operations without losing robustness. When comparing RC(CO) to RC(WHC), we see that for TPC-C an analysis on the granularity of tuples requires strictly more R-operations to be promoted leading to a smaller throughput compared to RC(CO) as the experiments will show.

In the following experiments, we only use the promoted versions of each template under RC, but keep the unmodified templates when executing under SI and SSI. SmallBank is not robust against SI, so, to be fair, RC should be compared to SSI rather than SI. For completeness sake, we also include experiments w.r.t. SI.

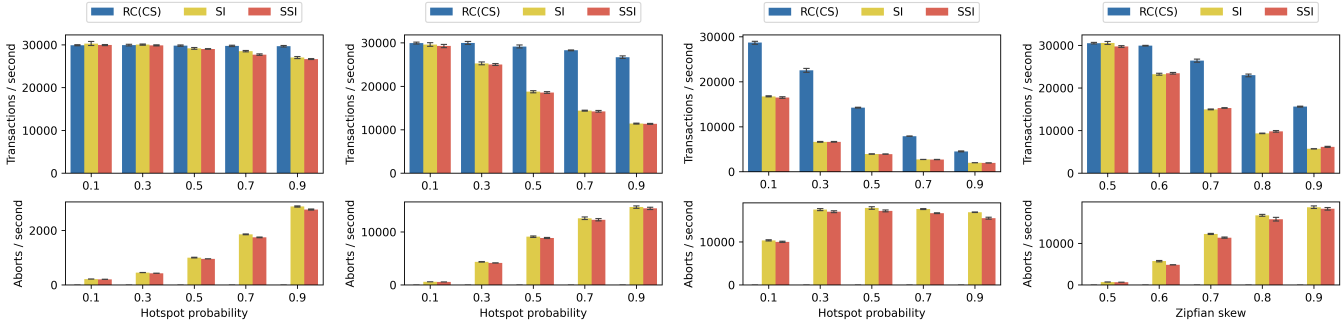
9.3.1 SmallBank. Since promotion under RC leads to taking additional write locks compared to SI and SSI, it is no longer guaranteed that choosing RC will never result in a performance loss. However, experimental analysis will show that even with this additional overhead, RC still outperforms SI and SSI in most scenarios with high contention due to the high number of aborts under SI and SSI.

Figure 6b compares the throughput for different numbers of concurrent clients. When contention is lower due to fewer clients, the throughput of RC is comparable to SI and SSI. When the number



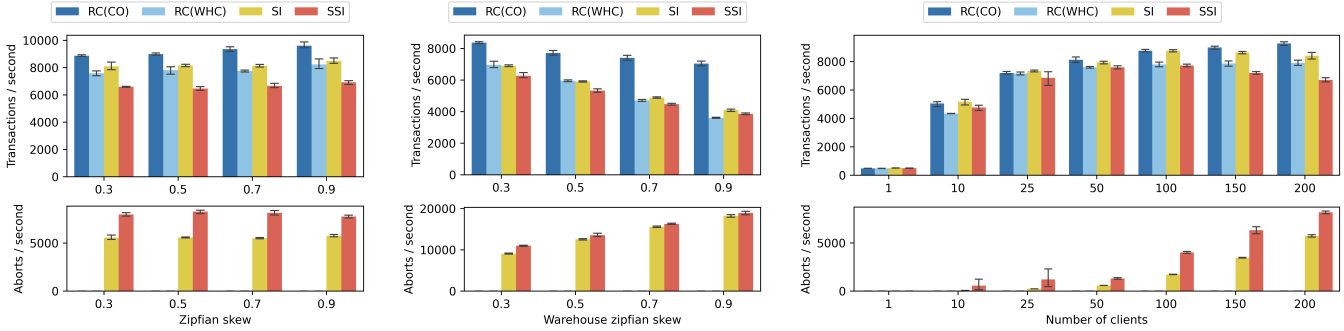
(a) Hotspot of 1000 accounts **(b) Hotspot of 100 accounts** **(c) Hotspot of 10 accounts** **(d) Zipfian distribution**

Figure 7: [Robust subset of SmallBank] Throughput and abort rate with 200 clients and different contention parameters.



(a) Hotspot of 1000 accounts **(b) Hotspot of 100 accounts** **(c) Hotspot of 10 accounts** **(d) Zipfian distribution**

Figure 8: [SmallBank with promotion] Throughput and abort rate with 200 clients and different contention parameters.



(a) Influence of Zipfian skew on customers; 200 concurrent clients, uniform distribution over warehouses.

(b) Influence of Zipfian skew over warehouses; fixed Zipfian skew of 0.7 over customers, 200 clients.

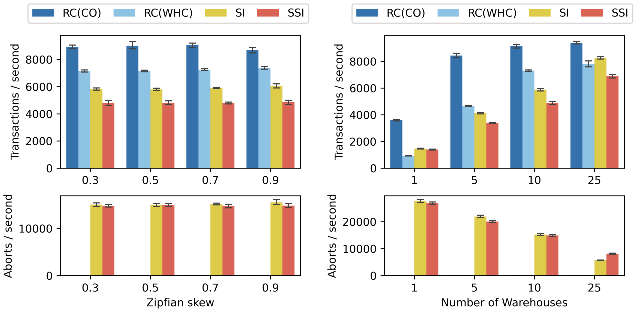
(c) Influence of the number of clients; Zipfian skew of 0.7 over customers, uniform distribution over warehouses.

Figure 9: [TPC-C with promotion] Throughput and abort rate for 25 warehouses and different contention parameters.

of clients increases, RC still outperforms SI and SSI, although the performance gain is less significant compared to Figure 6a.

Similar to Figure 7, we use 200 clients to query the database with different levels of skew, but this time with all transaction templates in the SmallBank benchmark and using promoted operations under RC. Figure 8a, Figure 8b and Figure 8c show the throughput for different hotspot sizes and probabilities. Figure 8d shows the throughput when a Zipfian distribution is used instead. When data skew is low, SI might slightly outperform RC. However, this small performance gain should be contrasted with the fact that SI does not guarantee serializability for this workload.

The different hotspot configurations show that RC(CS) usually outperforms SSI, in particular when the hotspot is smaller or when the hotspot probability is increased. In Figure 8c, the abort rate for RC(CS) increases to around 20 aborts per second under a hotspot probability of 0.9. Under a Zipfian distribution, RC(CS) is comparable to SSI when the Zipfian skew is low, but it clearly outperforms both SI and SSI when the Zipfian skew is increased.



(a) Influence of the Zipfian skew over customers when the dataset consists of 10 warehouses.

(b) Influence of the number of warehouses with a Zipfian skew of 0.7 over the customers.

Figure 10: [TPC-C + promotion] Throughput and abort rate; 200 clients, uniform distribution over warehouses.

9.3.2 TPC-C. For the TPC-C benchmark, RC(CO) slightly outperforms SI and SSI when contention is high due to a larger number of concurrent clients, whereas SI outperforms RC(WHC), as can be seen in Figure 9a. Notice in particular that increasing the skew on customer selection does not reduce throughput. The reason for this is that the throughput bottleneck is not caused by multiple transactions accessing the same customer, but by accessing the same warehouse instead. This is to be expected, since the number of warehouses in our dataset is several magnitudes smaller than the total number of customers. Figure 9b illustrates this by using a Zipfian distribution over the selected warehouses instead of a uniform distribution. An increased skew on the selected warehouses indeed leads to decreased performance for all isolation levels, although RC(CO) keeps outperforming the other isolation levels. A similar result is obtained if we repeat the experiment depicted in Figure 9a with a reduced dataset of only 10 warehouses instead of 25, as can be seen in Figure 10a. Noticeably, RC(WHC) now outperforms SI and SSI, although the throughput is still lower than that of RC(CO). Further investigation on different numbers of warehouses in Figure 10b shows that for all isolation levels, throughput decreases when the number of warehouses decreases, but RC(CO) still outperforms SI and SSI in all scenarios. This decrease in throughput is due to the small number of warehouses being a bottleneck.

Figure 9c further investigates the influence of different levels of contention on performance by testing different numbers of concurrent clients. Notice that when the number of clients is low, SI outperforms both RC(CO) and RC(WHC), since the reduced contention leads to a low number of concurrent writes and, as a result, few aborts. In this scenario, the performance gain for RC compared to SI by avoiding aborts does not outweigh the impact of the additional write locks due to promoted reads.

Conclusion. In contrast to Section 9.2, RC no longer necessarily outperforms the higher isolation levels when some operations are promoted. The introduction of these update operations leads to extra write locks and decreased throughput compared to optimistic concurrency control algorithms such as SI and SSI when contention is low. Under higher contention, RC usually outperforms SI and SSI due to the increased number of aborts, although the relative performance gain highly depends on the amount of skews and the chosen promotion strategy. In particular, we see that for TPC-C we always gain a better performance benefit if we analyse promotion on the granularity of attributes instead of tuples.

10 CONCLUSION

We pushed the frontier of the robustness problem for RC and showed that an explicit formalisation detects larger sets of transaction workloads to be robust. The throughput of a relational database system processing transactions under isolation level Serializable can be improved by an approach based on robustness testing and safely executing transactions under the lower isolation level RC. In the future we plan to cover more expressive transaction programs.

REFERENCES

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

[2] Mohammad Alomari. 2013. Serializable executions with Snapshot Isolation and two-phase locking: Revisited. In *AICCSA*. 1–8.

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

[4] Mohammad Alomari, Michael J. Cahill, Alan D. Fekete, and Uwe Röhm. 2008. Serializable Executions with Snapshot Isolation: Modifying Application Code or Mixing Isolation Levels?. In *DASFAA*, Vol. 4947. 267–281.

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

[6] Mohammad Alomari, Alan D. Fekete, and Uwe Röhm. 2009. A Robust Technique to Ensure Serializable Executions with Snapshot Isolation DBMS. In *ICDE*. 341–352.

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

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

[9] 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.

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

[11] 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.

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

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

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

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

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

[17] 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.

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

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

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

[21] Abdel Aziz Farrag and M. Tamer Özsu. 1989. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Trans. Database Syst.* 14, 4 (1989), 503–525.

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

[23] 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.

[24] Alan D. Fekete, Shirley Goldrei, and Jorge Perez Asenjo. 2009. Quantifying Isolation Anomalies. *Proc. VLDB Endow.* 2, 1 (2009), 467–478.

[25] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*. 243–252.

[26] 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.

[27] 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.

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

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

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

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

[32] 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.

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

[34] Shiyong Lu, Arthur J. Bernstein, and Philip M. Lewis. 2004. Correct Execution of Transactions at Different Isolation Levels. *IEEE Trans. Knowl. Data Eng.* 16, 9 (2004), 1070–1081.

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

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

[37] Christos H. Papadimitriou. 1986. *The Theory of Database Concurrency Control*. Computer Science Press.

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

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

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

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

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

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

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

[49] Ouri Wolfson. 1986. An Algorithm for Early Unlocking of Entities in Database Transactions. *J. Algorithms* 7, 1 (1986), 146–156.

[50] Ouri Wolfson. 1987. The Virtues of Locking by Symbolic Names. *J. Algorithms* 8, 4 (1987), 536–556.

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

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

[53] 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.

APPENDIX

A DETAILED BENCHMARK ANALYSIS

This section provides a detailed overview of the robustness properties for both the SmallBank and TPC-C benchmark. We analyse robustness against RC both on the granularity of attributes and tuples, providing concrete counterexample schedules for all subsets that are not considered robust.

A.1 SmallBank Transaction Templates

Figure 11 contains the SQL code for the SmallBank transaction templates presented in Figure 2. We identified three maximal robust subsets of transaction templates that are robust against RC:

- {DepositChecking, TransactSavings, Amalgamate},
- {Balance, DepositChecking}, and
- {Balance, TransactSavings}.

Figure 12 shows that these are indeed the only robust subsets by providing counterexample multiversion split schedules for sets of templates that are not robust against RC. We only provide counterexamples over minimal subsets that are not robust against RC, as these schedules immediately serve as counterexamples over larger subsets as well. An analysis of SmallBank on the granularity of tuples instead of attributes reveals that the robustness analysis remains unchanged. This is to be expected, since for this benchmark all conflicts on the granularity of tuples coincide with conflicts on the granularity of attributes. Indeed, all conflicting operations access the same attribute Balance in the Checking and Savings relations.

A.2 TPC-C Transaction Templates

For the TPC-C transaction templates given in Figure 5, the maximal subsets robust against RC are:

- {NewOrder, Payment, Delivery, StockLevel}, and
- {Payment, OrderStatus, StockLevel}.

For each minimal subset not robust against RC, a counterexample schedule is given in Figure 13.

When analysing the TPC-C transaction templates on the granularity of tuples instead of attributes, we get the following (smaller) subsets robust against RC:

- {Payment, Delivery, StockLevel},
- {Payment, OrderStatus, StockLevel}, and
- {NewOrder, StockLevel}.

The schedules given in Figure 13 immediately serve as counterexamples on the granularity of tuples, since the schedules in Figure 13 exhibit no dirty writes on the granularity of tuples. Counterexample schedules for the remaining minimal subsets not robust against RC are given in Figure 14.

```
Balance(N):
    SELECT CustomerId INTO :x
      FROM Account
     WHERE Name=:N;

    SELECT Balance INTO :a
      FROM Savings
     WHERE CustomerId=:x;

    SELECT Balance + :a
      FROM Checking
     WHERE CustomerId=:x;
    COMMIT;

Amalgamate(N1,N2):
    SELECT CustomerId INTO :x1
      FROM Account
     WHERE Name=:N1;

    SELECT CustomerId INTO :x2
      FROM Account
     WHERE Name=:N2;

    UPDATE Savings AS new
       SET Balance = 0
      FROM Savings AS old
     WHERE new.CustomerId=:x1
           AND old.CustomerId=new.CustomerId
    RETURNING old.Balance INTO :a;

    UPDATE Checking AS new
       SET Balance = 0
      FROM Checking AS old
     WHERE new.CustomerId=:x1
           AND old.CustomerId=new.CustomerId
    RETURNING old.Balance INTO :b;

    UPDATE Checking
       SET Balance = Balance + :a + :b
     WHERE CustomerId=:x2;

DepositChecking(N,V):
    SELECT CustomerId INTO :x
      FROM Account
     WHERE Name=:N;

    UPDATE Checking
       SET Balance = Balance + :V
     WHERE CustomerId=:x;
    COMMIT;
```

```
TransactSavings(N,V):
    SELECT CustomerId INTO :x
      FROM Account
     WHERE Name=:N;

    UPDATE Savings
       SET Balance = Balance + :V
     WHERE CustomerId=:x;
    COMMIT;

WriteCheck(N,V):
    SELECT CustomerId INTO :x
      FROM Account
     WHERE Name=:N;

    SELECT Balance INTO :a
      FROM Savings
     WHERE CustomerId=:x;

    SELECT Balance INTO :b
      FROM Checking
     WHERE CustomerId=:x;

    IF (:a + :b) < :V THEN
        UPDATE Checking
           SET Balance = Balance - (:V + 1)
         WHERE CustomerId=:x;
    ELSE
        UPDATE Checking
           SET Balance = Balance - :V
         WHERE CustomerId=:x;
    END IF;
    COMMIT;
```

Figure 11: SmallBank SQL Transaction Templates.

T_1 (WriteCheck):	$R_1[x]\,R_1[y]\,R_1[z\{C, B\}]$		$U_1[z\{C, B\}\{B\}]\,C_1$
T_2 (WriteCheck):		$R_2[x]\,R_2[y]\,R_2[z\{C, B\}]\,U_2[z\{C, B\}\{B\}]\,C_2$	

(a) {WriteCheck} is not robust against RC .

T_1 (Balance):	$R_1[x_1]\,R_1[y_1\{C, B\}]$		$R_1[z_1\{C, B\}]\,C_1$
T_2 (Amalgamate):		$R_2[x_1]\,R_2[x_2]\,U_2[y_1\{C, B\}\{B\}]\,U_2[z_1\{C, B\}\{B\}]\,U_2[z_2]\,C_2$	

(b) {Balance, Amalgamate} is not robust against RC .

T_1 (Balance):	$R_1[x]\,R_1[y\{C, B\}]$				$R_1[z\{C, B\}]\,C_1$
T_2 (TransactSavings):		$R_2[x]\,U_2[y\{C, B\}\{B\}]\,C_2$			
T_3 (Balance):			$R_3[x]\,R_3[y\{C, B\}]\,R_3[z\{C, B\}]\,C_3$		
T_4 (DepositChecking):				$R_4[x]\,U_4[z\{C, B\}\{B\}]\,C_4$	

(c) {Balance, DepositChecking, TransactSavings} is not robust against RC .

Figure 12: Counterexamples for robustness against RC for the SmallBank transaction templates. To facilitate readability, we only specify attributes for conflicting operations.

T_1 (OrderStatus):	$R_1[z]\,R_1[s\{W,D,O,C,Sta\}]$		$R_1[v_1\{\alpha\}]\,R_1[v_2\{\alpha\}]\,C_1$
T_2 (NewOrder):		$R_2[x]\,U_2[y]\,R_2[z]\,W_2[s\{W,D,O,C,Sta\}]\,U_2[t_1]\,W_2[v_1\{\alpha\}]\,U_2[t_2]\,W_2[v_2\{\alpha\}]\,C_2$	

(a) {NewOrder, OrderStatus} is not robust against RC. To shorten notation, we use α to denote the set of attributes $\{$W,D,O,OL,I,Del,Qua$\}$

T_1 (OrderStatus):	$R_1[z\{W,D,C,Inf,Bal\}]$		$R_1[s]\,R_1[v_1\{\alpha\}]\,R_1[v_2\{\alpha\}]\,C_1$
T_2 (Delivery):		$U_2[s]\,U_2[v_1\{\beta\}\{Del\}]\,U_2[v_2\{\beta\}\{Del\}]\,U_2[z\{W,D,C,Bal\}\{Bal\}]\,C_2$	

(b) {OrderStatus, Delivery} is not robust against RC. To shorten notation, we use α and β to denote respectively the sets of attributes $\{$W,D,O,OL,I,Del,Qua$\}$ and $\{$W,D,O,OL,Del$\}$

Figure 13: Counterexamples for robustness against RC for the TPC-C transaction templates. To facilitate readability, we only specify attributes for conflicting operations.

T_1 (NewOrder):	$R_1[x]$		$U_1[y]\,R_1[z]\,W_1[s]\,U_1[t_1]\,W_1[v_1]\,U_1[t_2]\,W_1[v_2]\,C_1$
T_2 (Payment):		$U_2[x]\,U_2[y]\,U_2[z]\,C_2$	

(a) {NewOrder, Payment} is not robust against RC.

T_1 (NewOrder):	$R_1[x]\,U_1[y]\,R_1[z]$		$W_1[s]\,U_1[t_1]\,W_1[v_1]\,U_1[t_2]\,W_1[v_2]\,C_1$
T_2 (Delivery):		$U_2[s]\,U_2[v_1]\,U_2[v_2]\,U_2[z]\,C_2$	

(b) {NewOrder, Delivery} is not robust against RC.

Figure 14: Counterexamples for robustness against RC for the TPC-C transaction templates when considering conflicts on the granularity of tuples instead of attributes. We omit attributes in our notation, as they are no longer important to decide conflict serializability.

B PROOFS FOR SECTION 5

B.1 Proof for Theorem 7

(1 → 2) Assume $\mathcal{T}$ is not robust against *RC*. Then there is a schedule s over $\mathcal{T}$ allowed under RC with a cycle C in $CG(s)$. We next construct a multiversion split schedule s' based on a sequence C' of conflict quadruples as defined in Definition 6. Without loss of generality, we assume that C is a minimal cycle in $CG(s)$. Let $T_1, T_2, \dots T_m$ be the transactions in the order that they appear in C, such that T_2 is the transaction (among those in C) that commits first in s. In other words, for every transaction $T_i \in C$ different from T_2, $\mathtt{C}_2 <_s \mathtt{C}_i$. Let

$$C' = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \dots, (T_m, b_m, a_1, T_1)$$

be a sequence of conflict quadruples where for each conflict quadruple $(T_i, b_i, a_{i+1}, T_{i+1})$, we have that a_{i+1} depends on b_i in s, that is, $b_i \rightarrow_s a_{i+1}$. Notice that, since there is an edge from T_i to T_{i+1} in $CG(s)$, we can always find such a pair of operations. We take T_{m+1} to be T_1. We now show that the multiversion split schedule s' based on C' satisfies the conditions in Definition 6.
(Condition 6.3) We assumed that $\mathtt{C}_2 <_s \mathtt{C}_1$. As s is allowed under RC, the existence of a wr- or a ww-dependency from b_1 to a_2 would imply that $\mathtt{C}_1 <_s a_2 <_s \mathtt{C}_2$. Therefore, $b_1 \rightarrow_s a_2$ is an rw-antidependency from b_1 to a_2. As a result, $b_1 <_s \mathtt{C}_2$, and b_1 and a_2 are rw-conflicting.
(Condition 6.1) Next, we prove that there is no ww-conflict between a write operation in $\text{prefix}_{b_1}(T_1)$ and a write operation in any of the transactions $T_2, \dots, T_m$. Towards a contradiction, assume that there is a transaction T_i with a write operation c_i, ww-conflicting with a write operation c_1 in $\text{prefix}_{b_1}(T_1)$. Notice that $c_1 <_s c_i$, as otherwise $c_i <_s \mathtt{C}_i <_s c_1 \leq_s b_1 <_s \mathtt{C}_2$, contradicting our assumption that T_2 commits first. Moreover, T_1 commits before c_i in s, as otherwise c_1 and c_i would imply a dirty write. Since $c_1 \rightarrow_s c_i$ and C is a minimal cycle in $CG(s)$, it immediately follows that $T_i = T_2$. But then T_1 commits before T_2 in s, leading to the desired contradiction.
(Condition 6.2) The last condition to verify is that $b_1 <_{T_1} a_1$ or b_m and a_1 are rw-conflicting. Towards a contradiction, assume that $a_1 \leq_{T_1} b_1$ *and* b_m and a_1 are not rw-conflicting. We argued above that $\text{prefix}_{b_1}(T_1)$ cannot contain a write operation ww-conflicting with a write operation in T_m. Therefore, b_m and a_1 must be wr-conflicting, and $b_m \rightarrow_s a_1$ is a wr-dependency. Since s is allowed under RC, it follows that $b_m <_s \mathtt{C}_m <_s a_1 \leq_s b_1 <_s \mathtt{C}_2$, contradicting our assumption that T_2 commits first in s.

(2 → 1) Let s be a multiversion split schedule for $\mathcal{T}$ based on $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \dots, (T_m, b_m, a_1, T_1)$ consisting of conflicting quadruples. We can assume that s is read-last-committed. Otherwise, choosing an appropriate version order $\ll_s$ and version function v_s. Notice that $\ll_s$ and v_s have no influence on the conflict quadruples in C.

First, we show that schedule s is allowed under RC (c.f. Definition 3). We only need to show that s exhibits no dirty writes. For this, let b_i and a_j be two arbitrary ww-conflicting operations in s in two different transactions T_i and T_j, with $b_i <_s a_j$. If $i > 1$ or $j > m$, it follows from the definition of multiversion split schedule that $b_i <_s \mathtt{C}_i <_s a_j$. For $i = 1$ and $j \leq m$, dirty writes are forbidden by condition (1) of Definition 6.

It remains to show that s is not conflict serializable. To this end, we argue that for each conflicting quadruple (T_i, b_i, a_j, T_j) in C, the operation a_j depends on b_i in s, that is, $b_i \rightarrow_s a_j$, thereby showing that the transactions in C represent a cycle in $CG(s)$. If both T_i and T_j are different from T_1, then $b_i <_s \mathtt{C}_i <_s a_j$ by construction of s. Since s is read-last-committed, it immediately follows that $b_i \rightarrow_s a_j$, independent of whether a_j and b_i are rw-, wr- or ww-conflicting. If $T_i = T_1$, then $b_i = b_1$ and $a_j = a_2$ are rw-conflicting by Definition 6. Since $b_1 <_s a_2$ implies that $op_0 = v_s(b_1) \ll_s a_2$, we obtain an rw-antidependency from b_1 to a_2.

Lastly, if $T_j = T_1$, then $a_j = a_1$ and $b_i = b_m$. By Definition 6, $b_1 <_s a_1$ or b_m and a_1 are rw-conflicting. In the former case, we have that $b_m <_s \mathtt{C}_m <_s a_1$, again implying that $b_m \rightarrow_s a_1$. In the latter case, b_m is a read operation on a tuple $\mathtt{t}$ where $v_s(b_m)$ is either op_0 or the write operation on $\mathtt{t}$ that committed last before b_m. In both cases, $v_s(b_m) \ll_s a_1$, since $b_m <_s \mathtt{C}_1$ and $\ll_s$ coincides with the commit order in $<_s$. The rw-antidependency from b_m to a_1 now follows immediately.

B.2 Proof for Theorem 8

Intuitively, Algorithm 1 applies Theorem 7 and checks whether a multiversion split schedule over $\mathcal{T}$ exists. We first argue that Algorithm 1 is correct, followed by the complexity analysis.

Correctness. Assume $\mathcal{T}$ is not robust against RC. By Theorem 7, a multiversion split schedule s for $\mathcal{T}$ based on some

$$C = (T'_1, b'_1, a'_2, T'_2), (T'_2, b'_2, a'_3, T'_3), \dots, (T'_m, b'_m, a'_1, T'_1)$$

exists. We argue that Algorithm 1 returns False. To this end, assume T_1 and b_1 in Algorithm 1 are instantiated by T'_1 and b'_1, respectively. Then, there is a path from T'_2 to T'_m in prefix-conflict-free-graph($b_1, T_1, \mathcal{T} \setminus \{T_1\}$), witnessed by the conflicts in C. Indeed, by Definition 6, transactions $T'_2, T'_3, \dots, T'_m$ are not ww-conflicting with $\text{prefix}_{b_1}(T_1)$. As a result, (T_2, T_m) is in TC if we instantiate T_2 and T_m by T'_2 and T'_m, respectively. If we take $a_1 = a'_1$, $a_2 = a'_2$ and $b_m = b'_m$, the condition in the if-test of Algorithm 1 is immediate by Definition 6, implying that the algorithm correctly returns False.

It remains to argue that Algorithm 1 returns True when $\mathcal{T}$ is robust against *RC*. Towards a contradiction, assume Algorithm 1 returns False instead, witnessed by transactions $T_1, T_2, T_m \in \mathcal{T}$ and operations $b_1, a_1 \in T_1$, $a_2 \in T_2$ and $b_m \in T_m$. Let

$$C = (T_2, b_2, a_3, T_3), (T_3, b_3, a_4, T_4), \dots, (T_{m-1}, b_{m-1}, a_m, T_m)$$

be the sequence of conflict quadruples witnessing the path from T_2 to T_m in prefix-conflict-free-graph($b_1, T_1, \mathcal{T} \setminus \{T_1\}$) (notice that C can be the empty sequence in the special case that $T_2 = T_m$). Then, the multiversion split schedule s for $\mathcal{T}$ based on

$$C' = (T_1, b_1, a_2, T_2), C, (T_m, b_m, a_1, T_1)$$

is a valid multiversion split schedule. Indeed, the transactions $T_2, \ldots, T_m$ do not contain a ww-conflict with an operation in prefix$_{b_1}(T_1)$ by definition of prefix-conflict-free-graph($b_1, T_1, \mathcal{T}$), and the remaining conditions of Definition 6 are immediate by the if-test in Algorithm 1. According to Theorem 7, this schedule s contradicts our assumption that $\mathcal{T}$ is robust against RC.

Complexity. Let k be the total number of operations in $\mathcal{T}$ and ℓ the maximum number of operations in a transaction in $\mathcal{T}$. The two outer for-loops in Algorithm 1 iterate over all read operations in $\mathcal{T}$, so there are at most k iterations. Each such iteration consists of three steps: constructing the prefix-conflict-free-graph G, computing the reflexive-transitive-closure TC over G, and checking a specific condition over the pairs of transactions in TC.

The construction of G requires us to verify for each transaction in $\mathcal{T} \setminus \{T_1\}$ whether it has an operation that is ww-conflicting with an operation in prefix$_{b_1}(T_1)$. We add each such transaction as a node to G, and add edges to other transactions in G if they have conflicting operations. Both parts can be done in time $O(\ell^2)$. The computation of TC over G can be achieved in time $O(|\mathcal{T}|^3)$ by an application of the Floyd-Warshall algorithm.

The third step checks a specific condition over pairs of transactions in TC. Worst case, TC is the complete graph, and the condition will iterate over all triples of operations (a_1, a_2, b_m), with a_1 an operation in T_1, and a_2 and b_m operations in two other transactions occurring in G. Therefore, this third step can be done in time $O(\ell.k^2)$.

By combining the results above, and since $l \leq k$, we get that Algorithm 1 decides whether $\mathcal{T}$ is robust against RC in time $O(\max\{k.|\mathcal{T}|^3, k^3.\ell\})$.

C PROOFS FOR SECTION 7

C.1 Proof for Lemma 12

First, we observe that for a sequence of potentially conflicting quadruples D for a set $\mathcal{P}$ and a variable mapping $\bar{\mu}$ for D, there always exists a database $\mathbf{D}$ such that the transactions in the sequences of conflicting quadruples C induced by D and $\bar{\mu}$ are consistent with $\mathcal{P}$ and $\mathbf{D}$. Consistency with $\mathcal{P}$ is immediate. As $\mathbf{D}$ we can take the database that contains (in its respective relations) all tuples $\mu_i(\mathsf{X})$ for every variable X in a transaction template τ_i in D with μ_i the variable mapping that $\bar{\mu}$ has assigned to τ_i.

Second, observe that if a variable X in some transaction template occurrence τ_i in a sequence of potentially conflicting quadruples D is connected to an operation o in some (not necessarily different) transaction template occurrence τ_j in D, then every mapping $\bar{\mu}$ for D assigns the same tuple to X in τ_i and the variable of operation o in τ_j.

($2 \Rightarrow 1$) Is a direct result of Definition 10 and Theorem 7.

($1 \Rightarrow 2$) There is a multiversion split schedule s for C over a set $\mathcal{T}$ consistent with $\mathcal{P}$, due to Definition 10 and Theorem 7. From $C = (T_1, b_1, a_2, T_2), \ldots, (T_m, b_m, a_1, \tau_1)$ we can derive a sequence of potentially conflicting quadruples $D = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_m, o_m, p_1, \tau_1)$ and a variable mapping $\bar{\mu}$ for D with variable mappings μ_i for every τ_i, such that $\mu_i(\tau_i) = T_i$, $\mu_i(o_i) = b_i$, and $\mu_i(p_i) = a_i$.

We claim that the canonical variable mapping $\bar{\mu}$ for D induces a sequence of conflicting quadruples C^c and thus a schedule s' for C^c as in Definition 6. Since the transactions in C^c are consistent with $\mathcal{P}$, and we already showed that for every variable mapping for D (including $\bar{\mu}$) there exists a database $\mathbf{D}$ where these transactions are also consistent with, we only have to show that schedule s has properties $(1-3)$ of Definition 6. In the below argument, we write μ'_i to denote the variable mapping for transaction template occurrence τ_i in D implied by $\bar{\mu}$.

Condition (1) requires most explanation. Therefore, towards a contradiction, let us assume that Condition (1) is not true for s'. Then there is a write operation in the prefix of $\mu'_1(\tau_1)$ (say with variable X) that is ww-conflicting with a write operation in another transaction $\mu'_j(\tau_j)$ in s', say with variable Y in the respective operation. The definition of $\bar{\mu}$ for D implies that $\mu'_1(\mathsf{X}) \in \{c_1, c_2\}$ and $\mu'_j(\mathsf{Y}) \in \{c_1, c_2, c_3\}$. More precisely, by the assumption $\mu'_1(\mathsf{X}) = \mu'_j(\tau_j)$, we have that $\mu'_j(\mathsf{Y}) \in \{c_1, c_2\}$ implying (again by definition of $\bar{\mu}$) that Y is connected to either o_1 or p_1 in τ_1. That latter means that also $\mu_1(\mathsf{X}) = \mu_j(\mathsf{Y})$ and thus that there is a write operation in the prefix of $\mu_1(\tau)$ that is ww-conflicting with a write operation in transaction $\mu_j(\tau_j)$ in s, which contradicts that s is a multiversion split schedule.

Condition (2) and Condition (3) are based on the type of operations, which are fixed in D and thus shared between C and C^c. Particularly, for Condition (2) we have that $\mu_1(o_1) <_{\mu_1(\tau_1)} \mu_1(p_1)$ or $\mu_m(o_m)$ is rw-conflicting with $\mu_1(p_1)$, due to s being a multiversion split schedule, from which follows that $o_1 <_{\tau_1} p_1$ or o_m is potentially rw-conflicting with p_1. Since the variable of o_m in τ_m is connected to p_1 in τ_1 it follows that $\mu'_1(o_1) <_{\mu'_1(\tau_1)} \mu'_1(p_1)$ or $\mu'_m(o_m)$ is rw-conflicting with $\mu'_1(p_1)$, thus that Condition (2) is indeed true for s' as well. Condition (3) follows similarly, as $\mu_1(o_1)$ is rw-conflicting with $\mu_2(p_2)$ due to s being a multiversion split schedule, implying that o_1 is potentially rw-conflicting with p_2. Since the variable of p_2 is also connected to o_1 in D, we have that $\mu'_1(p_1)$ is rw-conflicting with $\mu'_2(p_2)$ and thus that Condition (3) is true in s', which concludes the proof.

C.2 Proof for Theorem 14

The correctness of Algorithm 2 follows immediately from the following lemma:

Lemma 15. *Let $\mathcal{P}$ be a set of transaction templates. Then, $\mathcal{P}$ is not robust against RC iff for some transaction template $\tau_1 \in \mathcal{P}$, $o_1, p_1 \in \tau_1$ and $i \in \{1, 2\}$, a path in pt-prefix-conflict-free-graph$(o_1, p_1, i, \tau_1, \mathcal{P})$ from a node $(\tau_2, p_2, 1, in)$ to a node (τ_m, o_m, i, out) exists with the following properties:*

- *p_1 is potentially conflicting with o_m;*
- *o_1 is potentially rw-conflicting with p_2; and*
- *$o_1 <_{\tau_1} p_1$ or o_m is potentially rw-conflicting with p_1.*

Proof. *(if)* Let $P = (\tau_2, p_2, \ell_2, in), (\tau_2, o_2, k_2, out), (\tau_3, p_3, \ell_3, in), \dots, (\tau_m, o_m, k_m, out)$ be the path in pt-prefix-conflict-free-graph$(o_1, p_1, i, \tau_1, \mathcal{P})$, with $\ell_2 = 1$ and $k_m = i$. From this path P, we derive the sequence of potentially conflicting quadruples $D = (\tau_1, o_1, p_2, \tau_2), \dots, (\tau_m, o_m, p_1, \tau_1)$. Note that for each such quadruple $(\tau_j, o_j, p_k, \tau_k)$ in D, the operations o_j and p_k are indeed potentially conflicting: if $j = 1$ or $j = m$, this is immediate by the additional conditions stated in Lemma 15. Otherwise, this follows from the fact that there can only be an edge from $(\tau_j, o_j, f_{o_j}, out)$ to $(\tau_k, p_k, f_{p_k}, in)$ if o_j and p_k are potentially conflicting.

For each template τ_j in D, we next define a variable assignment μ_j using four disjoint tuple mappings c_1, c_2, c_3 and c_4, thereby creating a variable mapping $\bar{\mu}$ for D. By construction of pt-prefix-conflict-free-graph$(o_1, p_1, i, \tau_1, \mathcal{P})$, this $\bar{\mu}$ will actually coincide with the canonical mapping for D. We first define μ_1:

$$
\begin{array}{ll}
\mu_1'(\mathsf{X}) = c_1(\mathrm{type}(\mathsf{X})) & \text{if X is the variable occurring in } o_1, \\
\mu_1'(\mathsf{X}) = c_2(\mathrm{type}(\mathsf{X})) & \text{if the variables occurring in } o_1 \text{ and } p_1 \text{ are different and not connected, and X is the variable occurring in } p_1, \\
\mu_1'(\mathsf{X}) = c_4(\mathrm{type}(\mathsf{X})) & \text{otherwise.}
\end{array}
$$

For each τ_j different from τ_1, the variable assignment μ_j is constructed as follows:

$$
\begin{array}{ll}
\mu_j(\mathsf{X}) = c_k(\mathrm{type}(\mathsf{X})) & \text{if X is the variable occurring in } o_j \text{ and } (\tau_j, o_j, k, out) \text{ is a node in } P, \\
\mu_j(\mathsf{X}) = c_\ell(\mathrm{type}(\mathsf{X})) & \text{if X is the variable occurring in } p_j \text{ and } (\tau_j, p_j, \ell, in) \text{ is a node in } P, \\
\mu_j(\mathsf{X}) = c_3(\mathrm{type}(\mathsf{X})) & \text{otherwise.}
\end{array}
$$

This variable assignment μ_j is well defined for each variable X, even if X is the variable occurring in both o_j and p_j. In this case, there can only be an edge from (τ_j, p_j, ℓ, in) to (τ_j, o_j, k, out) if $k = \ell$. Notice furthermore that c_3 is never used in μ_1, and c_4 is never used in a μ_j different from μ_1, as $k, \ell \in \{1, 2, 3\}$ by construction of P.

Let s be the multiversion split schedule based on $C = \bar{\mu}(D) = (T_1, b_1, a_2, T_2), \dots, (T_m, b_m, a_1, T_1)$. We argue that s satisfies all properties of Definition 6, thereby proving that $\mathcal{P}$ is not robust against RC. Towards a contradiction, assume Condition (1) is not true. That is, there is an operation $b_1' \in \mathrm{prefix}_{b_1}(T_1)$ ww-conflicting with an operation $b_j' \in T_j$. By construction of C, the operations $o_1' \in \tau_1$ over a variable X and $o_j' \in \tau_j$ over a variable Y corresponding to b_1' and b_j' are potentially ww-conflicting. The variable assignments μ_1 and μ_j applied type mapping c_k with $k \in \{1, 2\}$ on both X and Y, as all four type mappings are disjoint and these are the only two type mappings occurring in both μ_1 and μ_j. Since we applied c_1 or c_2, the variable Y is occurring in τ_j in either o_j or p_j (or both). But then the corresponding node (τ_j, o_j, k, out) or (τ_j, p_j, k, in) cannot occur by construction of pt-prefix-conflict-free-graph$(o_1, p_1, i, \tau_1, \mathcal{P})$, leading to the desired contradiction. Condition (2) and Condition (3) are immediate by the properties on P specified in Lemma 15.

(only if) Assume $\mathcal{P}$ is not robust against RC. According to Lemma 12, there exists a multiversion split schedule s for some C over a set of transactions $\mathcal{T}$ consistent with $\mathcal{P}$ and a database $\mathbf{D}$, where C is induced by a sequence of potentially conflicting quadruples $D = (\tau_1, o_1, p_2, \tau_2) \dots (\tau_m, o_m, p_1, \tau_m)$ over $\mathcal{P}$ and it canonical variable mapping $\bar{\mu}$. We introduce a function f mapping each operation in D onto the corresponding type mapping used in the construction of s. More formally, for each operation $o_j \in \tau_j$ over a variable X appearing in D, we have $f(o_j) = i$ such that $\mu_j(X) = c_i(\mathrm{type}(X))$, with μ_j the corresponding variable mapping in $\bar{\mu}$. We now argue that the sequence of nodes $P = (\tau_2, p_2, f(p_2), in), (\tau_2, o_2, f(o_2), out), \dots, (\tau_m, p_m, f(p_m), in), (\tau_m, o_m, f(o_m), out)$ is a valid path in pt-prefix-conflict-free-graph$(o_1, p_1, i, \tau_1, \mathcal{P})$, where $i = 1$ if p_1 and o_1 are over the same variable in τ_1, and $i = 2$ if not.

We first argue that each node $(\tau_j, o_j, f(o_j), k)$ with $k \in \{in, out\}$ on this path P is indeed a node in pt-prefix-conflict-free-graph$(o_1, p_1, i, \tau_1, \mathcal{P})$. To this end, notice that $f(o_j) \in \{1, 2, 3\}$, as only c_1, c_2 and c_3 are used for operations occurring in D. If $f(o_j) = 1$, the node appears in the graph as long as there is no $o_j' \in \tau_j$ over the same variable as o_j and potentially ww-conflicting with an operation in $\mathrm{prefix}_{o_1}(\tau_1)$ over the same variable as o_1. Analogously, if $f(o_j) = 2$ and the operations o_1 and p_1 are not over the same variable in τ_1, then the node appears in the graph if there is no $o_j' \in \tau_j$ over the same variable as o_j and potentially ww-conflicting with an operation in $\mathrm{prefix}_{o_1}(\tau_1)$ over the same variable as p_1. In both cases, the node not appearing in the graph would imply that the schedule s is not a valid multiversion split schedule, as Condition (1) in Definition 6 would be violated. If $f(o_j) = 3$, then the node always occurs in the graph.

We now argue that there is indeed an edge between each consecutive pair of nodes in P. For a pair $(\tau_j, p_j, f(p_j), in), (\tau_j, o_j, f(o_j), out)$, this follows trivially. For a pair $(\tau_j, o_j, f(o_j), out), (\tau_{j+1}, p_{j+1}, f(p_{j+1}), in)$, notice that $f(o_j) = f(p_{j+1})$, as otherwise $(\mu_j(\tau_j), o_j, p_{j+1}, \mu_{j+1}(\tau_{j+1}))$ would not be a conflict quadruple in C, where μ_j and μ_{j+1} are the corresponding variable mappings in $\bar{\mu}$.

To conclude, we show that this path P satisfies all required conditions. Since s is a multiversion split schedule, these conditions are immediate by Definition 6. □

It remains to argue that Algorithm 2 indeed runs in time $O(k^4.\ell)$, with k the total number of operations in $\mathcal{P}$ and ℓ the maximum number of operations in a transaction template in $\mathcal{P}$. The two outer loops of Algorithm 2 iterate over each pair of operations in the same template τ_1 implying that the total number of iterations is $O(k.\ell)$. During each such iteration, the graph G is constructed, containing at most $6k$ nodes. The transitive closure TC over G can therefore be computed in time $O(k^3)$ by an immediate application of the Floyd-Warshall algorithm. The last step of each iteration of the outer loops is to verify for each pair of operations p_2 and o_m in $\mathcal{P}$ whether a specific condition holds. As a result, this check can be verified in time $O(k^2)$. By combining these results, we conclude that Algorithm 2 indeed decides whether $\mathcal{P}$ is robust against RC in time $O(k^4.\ell)$.