Robustness against Read Committed for Transaction Templates

Brecht Vandevoort[*]
Hasselt University and Transnational University of Limburg
Hasselt, Belgium
brecht.vandevoort@uhasselt.be

Bas Ketsman
Vrije Universiteit Brussel
Brussels, Belgium
bas.ketsman@vub.be

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

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

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.

PVLDB Reference Format:
Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. Robustness against Read Committed for Transaction Templates. PVLDB, 14(11): 2141 - 2153, 2021.
doi:10.14778/3476249.3476268

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, 21]: 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 [46] 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. [22]).

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, 21, 22] 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.

[*]PhD Fellow of the Research Foundation – Flanders (FWO)

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476268

Our approach is centered on a novel characterization of robustness against RC in the spirit of [21, 29] 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 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).

The sufficiency of our test for robustness, and the modification techniques we introduce to make programs robust, are practically applicable to programs that fit our model of a template. Programs that contain reads based on a predicate, rather than lookups on unchanging attributes such as a primary key, will need further techniques. Also, the necessity we prove for our decision procedure is only valid within our definition of RC isolation. In practice, it is possible for a set of programs running on a particular platform to always generate serializable executions even if they do not meet our test, in the case that the platform's implementation of RC doesn't allow all the possible interleavings which are covered by our definition of RC.

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 [21] 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 [29]. 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 SmallBank and TPC-Ckv (based on TPC-C) showing that we can identify robust subsets that are larger than those identified by previous methods. Still, neither SmallBank nor TPC-Ckv 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, and compared to other robustness techniques for RC [5], especially under higher contention.[1]

Due to space constraints, proofs have been deferred to the online available full version of this paper [47].

2 MOTIVATING EXAMPLE

The SmallBank [3] schema consists of the tables Account(<u>Name</u>, CustomerID), Savings(<u>CustomerID</u>, Balance), and Checking(<u>CustomerID</u>, 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).

[1]In the absence of contention, the three techniques – all sharing a common MVCC code base in the DBMS we use for experimentation – essentially perform the same instructions and no improvements can be expected.

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. 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 R-, W- and U-operations always access exactly one tuple. A 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, X and Z can not be interpreted to be the same object.

Detecting more robust subsets. Figure 3 gives an overview of the maximal robust subsets that are detected using our methods for the SmallBank and TPC-Ckv benchmarks (TPC-Ckv is discussed in Section 7 and the templates are given in Figure 6). 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: 'Only R & W' 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 'Only R & W' 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-Ckv larger robust subsets are detected.

Finally, 'Attr conflicts' 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 R[X : Warehouse{W, Inf}] and U[X : Warehouse{W, YTD}{YTD}] coming from templates NewOrder and Payment, respectively, in the TPC-Ckv benchmark as given in Figure 6. An instantiation of these template mapping X in both operations to the same tuple 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-Ckv 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 as 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 explain in Section 8 how robustness on attribute-level conflicts implies robustness on systems whose concurrency control subsystem works at the granularity of tuples.

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).

Earlier work on robustness against RC [5] based on counterflow dependencies did not consider atomic updates or attribute-level conflicts, but can be extended to these settings. The robust subsets that are detected by these extension are given in Figure 4. A comparison with Figure 3 reveals that although larger subsets are detected, our analysis still detects more and even larger robust subsets for both benchmarks, under both 'Atomic Updates' and 'Attr conflicts'.

3 RELATED WORK

Static robustness checking on the application level. Previous work on static robustness testing [5, 22] 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.*

Other work studies robustness within a framework for uniformly specifying different isolation levels in a declarative way [10, 13, 14]. A key assumption here is *atomic visibility* requiring that either all or none of the updates of each transaction are visible to other transactions. *This aims at higher isolation levels and cannot be used for 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,

```
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:

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

DepositChecking:

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

TransactSavings:

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

Amalgamate:

R[X_1 : Account$\{N, C\}$]
R[X_2 : Account$\{N, C\}$]
U[Y_1 : Savings$\{C, B\}\{B\}$]
U[Z_1 : Checking$\{C, B\}\{B\}$]
U[Z_2 : Checking$\{C, B\}\{B\}$]

WriteCheck:

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

Figure 2: Transaction templates for SmallBank.

	SmallBank	TPC-Ckv
Only R & W	{Bal}	{OS, SL}
Atomic Updates	{Am,DC,TS}, {Bal,DC}, {Bal,TS}	{Del,Pay,SL}, {NO, SL}, {Pay, OS, SL}
Attr conflicts	{Am,DC,TS}, {Bal,DC}, {Bal,TS}	{Del,Pay,NO,SL}, {Pay, OS, SL}

Figure 3: Robust subsets by analysis setting.

	SmallBank	TPC-Ckv
Only R & W	{Bal}	{OS, SL}
Atomic Updates	{Am,DC,TS}, {Bal}	{Del,Pay,SL}, {NO}, {OS,SL}
Attr conflicts	{Am,DC,TS}, {Bal}	{Del,Pay,SL}, {Del,Pay,NO} {OS,SL}

Figure 4: Detection of robust subsets based on counterflow dependencies [5] extended to updates and attribute level conflicts as introduced in this paper.

Fekete et al. [23] 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.*

Transaction chopping splits transactions into smaller pieces to obtain performance benefits and is correct if, for every serializable execution of the chopping, there exists an equivalent serializable execution of the original transactions [42]. Cerone et al. [14, 15] studied chopping under various isolation levels. *Transaction chopping has no direct relationship with robustness testing against RC.*

Making transactions robust. When a workload is not robust against an isolation level, robustness can be achieved by modifying the transaction programs [2–5, 22], using an external lock manager [2, 5, 6], allocating some programs to higher isolation levels [4, 21], 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, 22]: 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 [21]. Alomari [2] considered a refinement that adds an additional write to each transaction running under S2PL.

For RC, Alomari and Fekete [5] consider lock materialization to avoid counterflow dependencies using two approaches: (1) in-database, by adding a write on a newly introduced tuple at the start of each transaction; and, (2) introducing an external lock manager outside of the database that application programs need to access. In contrast, *we employ a code modification technique based on promotion as for SI changing certain read operations to updates. We provide a comparison in Section 9.3.*

Many approaches to increase transaction throughput have been proposed: improved or novel pessimistic (cf., e.g., [27, 37, 39, 45, 48]) or optimistic (cf., e.g., [11, 12, 17, 18, 25, 26, 28, 30–32, 34, 40, 41, 49, 50]) algorithms, as well as approaches based on coordination avoidance (cf., e.g., [19, 20, 33, 36, 38, 43, 44]). *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 $R \in$ Rels, Attr(R) is the finite set of associated attribute names. For every relation $R \in$ 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 $R, S \in$ 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 $R \in$ 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 type(t) $= R$. A *database* **D** over schema Rels assigns to every relation name $R \in$ 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 ReadSet(o) $\subseteq$ Attr(R) and WriteSet(o) $\subseteq$ Attr(R) containing, respectively, the set of attributes that o reads from and writes to. When o is a R-operation then WriteSet(o) $= \emptyset$. Similarly, when o is a W-operation then 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 $\mathtt{W}_i[\mathtt{t}]$, $\mathtt{R}_i[\mathtt{t}]$, and $\mathtt{U}_i[\mathtt{t}]$ 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, 21]). 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.* [21]). 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 operation different from a in s (recall that a write operation is either a W[x] or a U[x]). 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 $T \in \mathcal{T}$.

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)* WriteSet$(a_j) \cap$ WriteSet$(b_i) \neq \emptyset$; or,
- *(wr-conflict)* WriteSet$(a_j) \cap$ ReadSet$(b_i) \neq \emptyset$; or,
- *(rw-conflict)* ReadSet$(a_j) \cap$ 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 $b_i \to_s a_j$ 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, $b_i \to_s a_j$ iff $b_i \to_{s'} a_j$.

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

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 $b_i \to_s a_j$. The following is immediate from [35]:

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 [21], generalized to operations over attributes of tuples and extended with U-operations that combine R- and 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.

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 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 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 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 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.*

[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'.

[3]Recall that a write operation is either a W or a U-operation.

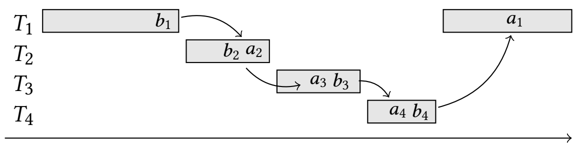


Figure 5: Multiversion split schedule for four transactions.

Robustness. The robustness property [10] (also called *acceptability* in [21, 22]) 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.

Example 5. *Consider transactions T_1 : $\mathtt{R}_1[\mathtt{t}\{\mathtt{a,b,c}\}]\,\mathtt{W}_1[\mathtt{v}\{\mathtt{a}\}]\,\mathtt{C}_1$ and T_2 : $\mathtt{R}_2[\mathtt{v}\{\mathtt{b}\}]\,\mathtt{W}_2[\mathtt{t}\{\mathtt{a,b,d}\}]\,\mathtt{C}_2$. Here, for example, $\mathtt{R}_1[\mathtt{t}\{\mathtt{a,b,c}\}]$ is shorthand for operation $\mathtt{R}_1[\mathtt{t}]$ with read set $\{a, b, c\}$. The two operations on* 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 : $\mathtt{R}_1[\mathtt{t}\{\mathtt{a,b,c}\}]\,\mathtt{R}_2[\mathtt{v}\{\mathtt{b}\}]\,\mathtt{W}_2[\mathtt{t}\{\mathtt{a,b,d}\}]\,\mathtt{C}_2$ $\mathtt{W}_1[\mathtt{v}\{\mathtt{a}\}]\,\mathtt{C}_1$. However, these two transactions are robust against RC at attribute-level granularity.[4] The order of the two operations on tuple* 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 $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \dots, (T_m, b_m, a_1, T_1)$ 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

(1) there is no write operation in $\mathsf{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in any of the transactions $T_2, \dots, T_m$;
(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, $T_{m+1}, \dots, T_n$ are the remaining transactions in $\mathcal{T}$ (those not mentioned in C) in an arbitrary order.

Figure 5 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 5) and placing all other transactions in C in between. The figure does not display the trailing transactions $T_{m+1}, T_{m+2}, \dots$ 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 $\mathsf{prefix}_{b_1}(T_1)$. 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.

[4]This is under the reasonable assumption that a database system can read and/or update all attributes of a tuple in one atomic step.

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 $True$
```

We fix an infinite set of variables **Var** that is disjoint from **Tuples**. Every variable $\mathsf{X} \in$ **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 X : 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 $\mathcal{T}_1 = \{\mathsf{R}[\mathsf{a}_1]\mathsf{R}[\mathsf{s}_1]\mathsf{R}[\mathsf{c}_1], \mathsf{R}[\mathsf{a}_1]\mathsf{R}[\mathsf{a}_2]\mathsf{U}[\mathsf{s}_1]\mathsf{U}[\mathsf{c}_1]\mathsf{U}[\mathsf{c}_2]\}$. 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 $\mathcal{T}_1$:*

```
$T_1$ : $R_1[a_1]R_1[s_1]$                                               $R_1[c_1]C_1$
$T_2$ :                 $R_2[a_1]R_2[a_2]U_2[s_1]U_2[c_1]U_2[c_2]C_2$
```
□

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:

- WriteSet(o_i) $\cap$ WriteSet(o_j) $\neq \emptyset$ (potentially ww-conflicting);
- WriteSet(o_i) $\cap$ ReadSet(o_j) $\neq \emptyset$ (potentially wr-conflicting); or
- ReadSet(o_i) $\cap$ 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 **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. If a counterexample exists, Algorithm 2 needs at most three tuples per type to construct a counterexample. We encode this choice for each variable by assigning the numbers 1 to 3 to specific operations.

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 tuple of each type when $h = 1$ and that the suffix uses the second tuple of each type 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 the tuple assigned to o in τ. There will be two types of edges: (1) inner edges $(\tau, o, i, \text{in}) \longrightarrow (\tau, p, i', \text{out})$ that stay within the same transaction τ and indicate how the chosen tuple version changes (or stays the same) from c_i for o to $c_{i'}$ for p; and (2) outer edges $(\tau, o, i, \text{out}) \longrightarrow (\tau', p, i', \text{in})$ between different occurrences of transaction templates encoding a potentially conflicting quadruple (τ, o, p, τ') and maintaining information on the chosen tuple as well.

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

Algorithm 2: Deciding 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
```

(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.

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 tuples. 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 chosen tuple version 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 12. *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-Ckv 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-Ckv (as can be seen in the row 'Attr conflicts' 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. The latter approach leads to a more general observation with practical value: our algorithm provides a sufficient condition to guarantee serializability for every implementation that can only generate a subset of the schedules allowed by RC.

9 EXPERIMENTS

We discussed the effectiveness of our approach in detecting larger robust subsets in comparison with [5] at the end of Section 2. We focus here on how robustness can improve transaction throughput.

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) [22].[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

[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.

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 6: Abstraction for the TPC-Ckv transaction templates. Attribute names are abbreviated.

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 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 [24].

9.1.3 TPC-Ckv benchmark. The second benchmark is based on the TPC-C benchmark [46]. 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).

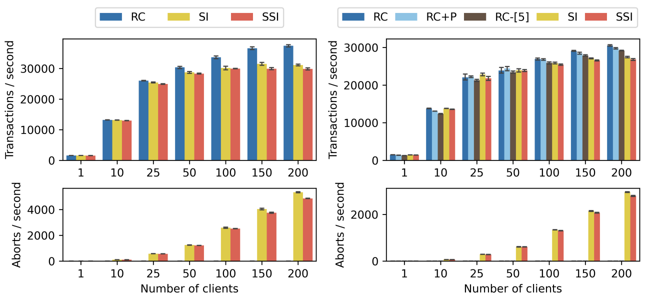
We focus on five different transaction templates:

(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 W.

An abstraction of each transaction template is given in Figure 6. The experiments adhere to the requirements of the official TPC-C benchmark [46], 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 (Figure 6 shows only two orderlines per order to simplify presentation). 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.

9.2 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



(a) Robust subset. (b) Complete benchmark.

Figure 7: 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%.

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 results of this experiment are shown in Figure 7a. 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 8a, Figure 8b and Figure 8c show the throughput for different hotspot probabilities when there are respectively 1000, 100 and 10 accounts in the hotspot. Figure 8d 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 8c, 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-Ckv 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

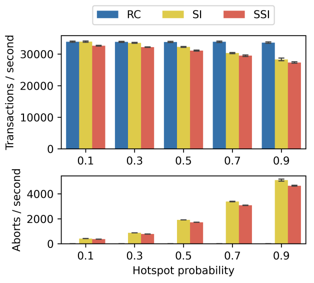
9.3.1 Promotion. When a set of transaction templates is not robust, we propose a template modification technique based on insights from Definition 6: an equivalent set of transaction templates robust against RC can be created by promoting R-operations to U-operations that write back the read value. Such a change does not alter the effect of the transaction template, but the newly introduced write operation will trigger concurrency mechanisms in the database. We emphasize that this is a general technique that can *always* be used to construct an equivalent robust set of templates: Definition 6 requires that operation b_1 is rw-conflicting with a_1 (Condition (3)), but not ww-conflicting with a_1 (Condition (1)), so 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. [22] 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: 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-Ckv to guarantee robustness with a minimal number of promotions.

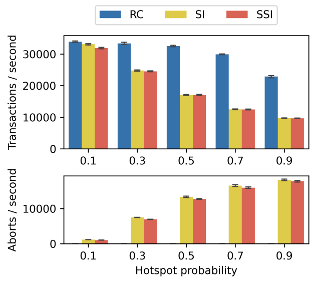
For SmallBank, we can obtain robustness by only promoting all R-operations over the Checking and Savings relations to U-operations leaving all other R-operation intact (cf. Figure 9 and notice that only 2/5 templates are modified and in total only four reads need to be promoted). In our experiments, we refer to this promotion as *RC+P*. Furthermore, this set of promoted R-operations is minimal: if one of the R-operations over the Checking or Savings relations remains,the application of Algorithm 2 reveals that the resulting set of transaction templates is not robust against RC.

For TPC-Ckv, 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+P(Attr)*. To contrast our approach based on attribute-level conflicts with the one based on tuple-level conflicts, we also investigate how to make TPC-Ckv 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+P(Tup)*. 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+P(Attr) to RC+P(Tup), we see that for TPC-Ckv an analysis on the granularity of tuples requires strictly *more* R-operations to be promoted leading to a smaller throughput compared to RC+P(Attr) as the experiments will show.

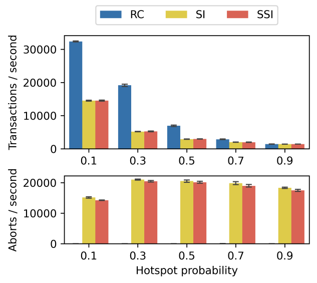
We also compare with the Internal Concurrency Exclusion [5] approach (referring to the latter as *RC-[5]*) for making workloads robust against RC extended to attribute-level conflicts. Each template is changed by adding additional leading W-operations overwriting tuples in a newly introduced Conflict relation such that every pair



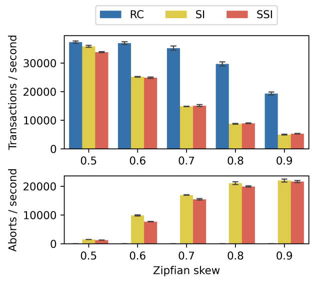
(a) Hotspot of 1000 accounts



(b) Hotspot of 100 accounts



(c) Hotspot of 10 accounts



(d) Zipfian distribution

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

Balance (RC+P):	WriteCheck (RC+P):	Balance (RC-[5]):
R[X : Account{N,C}]	R[X : Account{N,C}]	W[V : Conflict{N}]
U[Y : Savings{C,B}{B}]	U[Y : Savings{C,B}{B}]	R[X : Account{N,C}]
U[Z : Checking{C,B}{B}]	U[Z : Checking{C,B}{B}]	R[Y : Savings{C,B}]
	U[Z : Checking{C,B}{B}]	R[Z : Checking{C,B}]

Figure 9: Left: *all* modified templates in RC+P. Right: Example of *one* template modification for RC-[5].

of instances that might produce a counterflow edge is guaranteed to write to the same tuple in the Conflict relation. The latter is achieved by selecting specific relations in the original benchmark and adding a tuple to relation Conflict for each tuple in the selected relations. Each template is then changed so that if it accesses one of the tuples of these selected relations, it also writes to the corresponding tuple in relation Conflict. For SmallBank, it suffices to select relation Account. Figure 9(right) illustrates the required changes for template Balance. We stress that this change needs to be done for *every* template. There are even two W-operations needed at the beginning of Amalgamate, as it considers two different customers. For TPC-Ckv, the selected relations are Stock (thereby requiring additional writes in templates NewOrder and StockLevel) and Customer (thereby requiring additional writes in templates NewOrder, Delivery, OrderStatus and Payment). We refer to [47] for the concrete templates used for RC-[5].

We also include the performance of the unmodified templates under RC, SI and SSI as a baseline. Recall that both SmallBank and TPC-Ckv are not robust against RC, and SmallBank is not robust against SI. So, to be fair, the performance of the promoted workloads should be compared to SSI (for SmallBank) and SI (for TPC-Ckv).

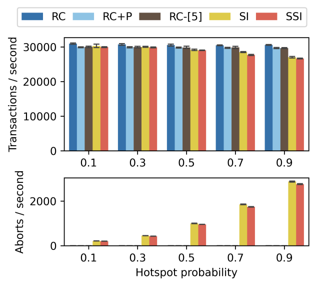
9.3.2 SmallBank. Figure 7b compares the throughput for different numbers of concurrent clients. When contention is lower due to fewer clients, the throughput of RC+P is comparable to RC, SI and SSI. When the number of clients increases, RC+P still outperforms SI and SSI (and also RC-[5]), although the performance gain is less as compared to Figure 7a.

Similarly to Figure 8, 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 in RC+P. Figures 10a, 10b and 10c show the throughput for different hotspot sizes and probabilities. Figure 10d shows the throughput when a Zipfian distribution is used instead. The experiments show that RC+P outperforms SI, SSI, and RC-[5] when the hotspot is smaller, the hotspot probability increases, or skew increases. The improvements over SI and SSI result from the high number of aborts under SI and SSI. The improvement over RC-[5] (which can be an order of magnitude depending on the setting) can be explained by noting that RC+P allows for more concurrency than RC-[5]. Indeed, for RC-[5], two instances of templates that access the same customer can never be concurrent, as they both initially write to the same tuple of type Conflict. For RC+P, some concurrency is still possible in this case: i.e., an execution of DepositChecking can interleave with an execution of TransactSavings over the same customer, as the former only updates the tuple of type Checking, whereas the latter only updates the tuple of type Savings.

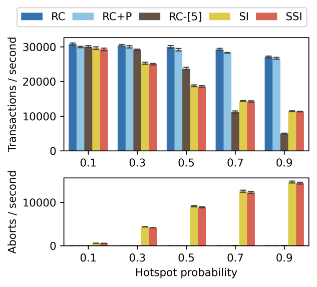
Furthermore, the performance of RC+P is usually comparable to RC. In fact, when the hotspot is small (Figure 10c), RC+P is even able to outperform the original templates under RC due to a reduced number of deadlocks: the abort rate for RC+P increases to around 20 aborts per second under a hotspot probability of 0.9, whereas RC increases to around 45 aborts per second in this setting.

9.3.3 TPC-Ckv. Figure 11a and Figure 12a show the throughput and abort rate depending on the Zipfian skew over the Customer relation, over a dataset consisting of respectively 25 and 10 warehouses. Noticeably, changing the skew over the Customer relation does not result in a significant change in 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. We further investigate the influence of the number of warehouses (Figure 12b), as well as a Zipfian skew over the Warehouse relation (Figure 11b). Figure 11c investigates the influence of different levels of contention on performance by testing different numbers of concurrent clients.

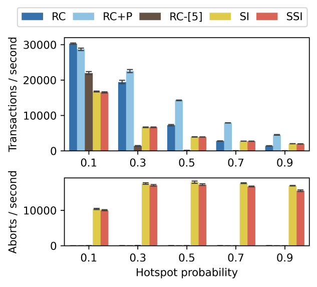
When comparing both promotion strategies, we conclude that RC+P(Attr) always outperforms RC+P(Tup), especially when the number of warehouses is lowered (Figure 12b), or when a larger Zipfian skew is used over the Warehouse relation (Figure 11b). In these cases, it should also be noted that RC+P(Attr) clearly dominates the higher isolation levels SI and SSI. For example, when the dataset consists of 5 warehouses in Figure 12b, the throughput of RC+P(Attr) averages around 8300 transactions per second,



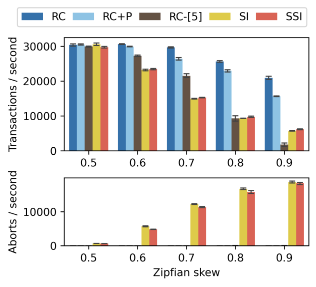
(a) Hotspot of 1000 accounts



(b) Hotspot of 100 accounts

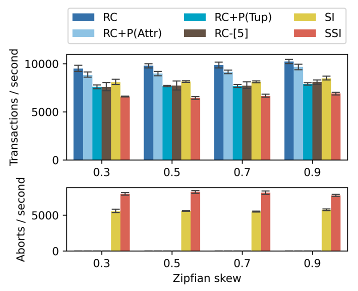


(c) Hotspot of 10 accounts

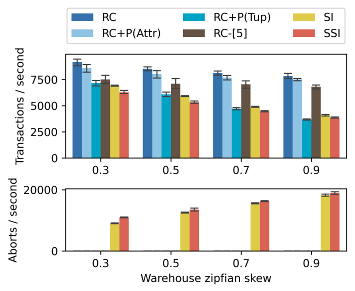


(d) Zipfian distribution

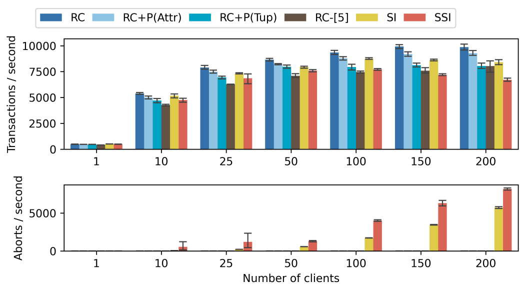
Figure 10: [SmallBank with promotion] Throughput and abort rate with 200 clients and different contention parameters.



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

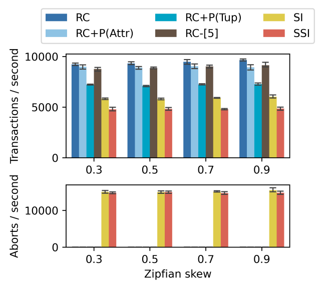


(b) Influence of Zipfian skew over warehouses; fixed Zipfian skew of 0.7 over customers, 200 clients.

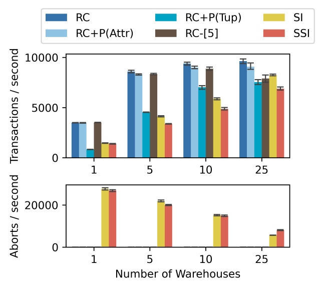


(c) Influence of the number of clients; Zipfian skew of 0.7 over customers, uniform distribution over warehouses.

Figure 11: [TPC-Ckv with promotion] Throughput and abort rate for 25 warehouses and different contention parameters.



(a) Influence of 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 12: [TPC-Ckv + promotion] Throughput and abort rate; 200 clients, uniform distribution over warehouses.

contrasting the average of 4100 transactions per second for SI. Furthermore, the performance loss of RC+P(Attr) compared to RC over the original templates is always relatively small, indicating that our approach allows to achieve serializability guarantees in exchange for a minor performance loss.

When analyzing RC-[5], we see that its throughput relative to the throughput of RC+P(Attr) is highly dependent on the number of warehouses (Figure 12b), as well as the Zipfian skew over the Warehouse relation (Figure 11b). In particular, when the number of warehouses is lowered or the skew increased, the throughput of RC-[5] is similar to that of RC+P(attr). If on the other hand the number of warehouses is larger and a uniform distribution over the Warehouse relation is used, then RC+P(Attr) clearly outperforms RC-[5]. Consider for example the setting with 25 warehouses in Figure 12b. Then, the average throughputs of RC+P(Attr) and RC-[5] are respectively around 9000 and 7800 transactions per second.

9.3.4 Conclusion. Our promotion technique outperforms the isolation levels SI and SSI under higher contention and guarantees serializability under RC while requiring only a minor performance cost compared to RC over the original templates. When comparing to earlier work based on Internal Concurrency Exclusion [5], on SmallBank our approach significantly outperforms RC-[5] when contention increases (an order of magnitude in extreme cases). For TPC-Ckv, the performance gain is similar to that of RC-[5], although we are still able to identify cases where our technique outperforms RC-[5] by more than 15%. Finally, comparing RC+P(Attr) versus RC+P(Tup) for TPC-Ckv shows that promotion based on attribute-level analysis significantly outperforms tuple-level analysis.

ACKNOWLEDGMENTS

The resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation – Flanders (FWO) and the Flemish Government.

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

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

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

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

[31] 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.

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

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

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

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

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

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

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

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

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

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

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

[47] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2021. Robustness against Read Committed for Transaction Templates (full version). (2021). https://arxiv.org/abs/2107.12239.

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

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

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