Allocating Isolation Levels to Transactions in a Multiversion Setting

BRECHT VANDEVOORT, UHasselt, Data Science Institute, ACSL, Belgium
BAS KETSMAN, Vrije Universiteit Brussel, Belgium
FRANK NEVEN, UHasselt, Data Science Institute, ACSL, Belgium

A serializable concurrency control mechanism ensures consistency for OLTP systems at the expense of a reduced transaction throughput. A DBMS therefore usually offers the possibility to allocate lower isolation levels for some transactions when it is safe to do so. However, such trading of consistency for efficiency does not come with any safety guarantees. In this paper, we study the mixed robustness problem which asks whether, for a given set of transactions and a given allocation of isolation levels, every possible interleaved execution of those transactions that is allowed under the provided allocation is always serializable. That is, whether the given allocation is indeed safe. While robustness has already been studied in the literature for the homogeneous setting where all transactions are allocated the same isolation level, the heterogeneous setting that we consider in this paper, despite its practical relevance, has largely been ignored. We focus on multiversion concurrency control and consider the isolation levels that are available in PostgreSQL and Oracle: read committed (RC), snapshot isolation (SI) and serializable snapshot isolation (SSI). We show that the mixed robustness problem can be decided in polynomial time. In addition, we provide a polynomial time algorithm for computing the optimal robust allocation for a given set of transactions, prioritizing lower over higher isolation levels. The present results therefore establish the groundwork to automate isolation level allocation within existing databases supporting multiversion concurrency control.

CCS Concepts: • **Information systems → Database transaction processing**.

Additional Key Words and Phrases: concurrency control, robustness, complexity

ACM Reference Format:
Brecht Vandevoort, Bas Ketsman, and Frank Neven. 2025. Allocating Isolation Levels to Transactions in a Multiversion Setting. 1, 1 (June 2025), 24 pages. https://doi.org/10.1145/3716374

1 INTRODUCTION

The majority of relational database systems offer a range of isolation levels, the highest of which is serializability ensuring what is considered as perfect isolation. This allows users to trade off isolation guarantees for better performance. Executing transactions concurrently at weaker degrees of isolation does carry some risk as it can result in specific anomalies. However, there are situations when a group of transactions can be executed at an isolation level lower than serializability without causing any errors. In this way, we get the higher isolation guarantees of serializability for free in exchange for a lower isolation level, which is typically implementable with a less expensive concurrency control mechanism. This formal property is called *robustness* [13, 19, 20]: a set of transactions $\mathcal{T}$ is called robust against a given isolation level if every possible interleaving of the

Authors' addresses: Brecht Vandevoort, brecht.vandevoort@uhasselt.be, UHasselt, Data Science Institute, ACSL, Diepenbeek, Belgium; Bas Ketsman, bas.ketsman@vub.be, Vrije Universiteit Brussel, Brussels, Belgium; Frank Neven, frank.neven@uhasselt.be, UHasselt, Data Science Institute, ACSL, Diepenbeek, Belgium.

ACM XXXX-XXXX/2025/6-ART
https://doi.org/10.1145/3716374

transactions in $\mathcal{T}$ that is allowed under the specified isolation level is conflict-serializable. There is a famous example that is part of database folklore: the TPC-C benchmark [24] 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. [20]).

The robustness problem received quite a bit of attention in the literature and can be classified in terms of the considered isolation levels: lower isolation levels like (multiversion) Read Committed (RC) [6, 22, 25, 26], Snapshot Isolation (SI) [4, 10, 19, 20], and higher isolation levels [11, 13, 16, 18]. The far majority of this work focused on a *homogeneous* setting where all transactions are allocated the *same* isolation level. So, when a workload is robust against an isolation level, all transactions can be executed under this isolation level and benefit from the speedup offered by the cheaper concurrency control algorithm and the guarantee that the resulting execution will always be conflict-serializable. When a workload is not robust against an isolation level, robustness can still be achieved by modifying the transaction programs [3–6, 20, 25] or using an external lock manager [3, 6, 7]. The downside of these solutions is that they require altering transactions or require drastic changes to the underlying database implementation.

In this paper, we are interested in solutions that refrain from modifying transactions and can be readily used on top of a DBMS without changing any of the database internals. The solution lies within the capabilities of the DBMS itself. Indeed, in practice, an isolation level is not set on the level of the database or even the application but can be specified on the level of an individual transaction. So, a third option for making a transaction workload robust is to allocate problematic transactions to higher isolation levels. That is, by considering *heterogeneous* or *mixed* allocations where individual transactions can be mapped to different isolation levels. Such an approach requires a solution for two research challenges as discussed next: the *robustness problem* and the *allocation problem*. To this end, let $\mathcal{T}$ be a set of transactions, $\mathcal{I}$ a class of isolation levels and $\mathcal{A}$ an allocation (mapping each $T \in \mathcal{T}$ to an isolation level in $\mathcal{I}$). Then define the following problems:

- The **robustness problem for $\mathcal{I}$**: Is every concurrent execution of transactions in $\mathcal{T}$ that is allowed under $\mathcal{A}$, conflict-serializable?
- The **allocation problem for $\mathcal{I}$**: Compute an optimal robust allocation for $\mathcal{T}$ over $\mathcal{I}$ (when it exists).

In order to increase transaction throughput, weaker isolation levels, which are often less strict and permit higher concurrency, are favored over stricter isolation levels which generally limit concurrency.[1] We then say that a robust allocation is *optimal* when no higher isolation level can be exchanged for a weaker one without breaking robustness. A seminal result in this context is that of Fekete [19] who provided polynomial time algorithms for the robustness and the allocation problem for the setting where $\mathcal{I}$ consists of the isolation levels SI and strict two-phase locking (S2PL). More specifically, when $\mathcal{T}$ is not robust against SI, a minimal number of transactions can be found that need to be run under S2PL to make the workload robust.

In the present work, we address the robustness and allocation problem for a wider range of isolation levels: RC, SI, and Serializable Snapshot Isolation (SSI) [14, 23]. These classes are particularly relevant for the following reasons: RC is often configured by default [9]; SI remains the highest possible isolation level in some database systems like Oracle and is well-studied (e.g, [4, 10, 13, 16–21]; and, SSI effectively guarantees serializability. Furthermore, {RC, SI, SSI} is the class of isolation levels available in PostgreSQL, while {RC, SI} are those available in Oracle. We see our results as a significant step towards automating isolation level allocation on top of existing databases. Indeed,

[1]Indeed, Vandevoort et al. [25] have shown that when contention increases, RC outperforms SI w.r.t. transaction throughput.

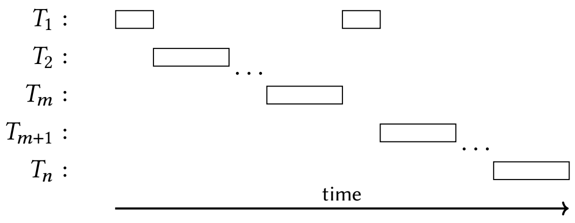


Fig. 1. Abstract representation of a multi-version split schedule where T_1 is the splitted transaction.

we obtain that for {RC, SI, SSI} an optimal robust allocation can always be found in polynomial time. As {RC, SI} does not include a conflict-serializable isolation level, a robust allocation does not always exist. However, the results in this paper show that the existence of a robust allocation for {RC, SI} can be decided in polynomial time, and when a robust allocation exists, an optimal one can be found.

The main technical contribution of this paper is Theorem 3.2 which shows that non-robustness against an allocation for the isolation levels {RC, SI, SSI} can be characterized in terms of the existence of a counterexample schedule of a very specific form that we refer to as a multiversion split schedule. Such split schedules have been used before in the homogeneous setting where all transactions in a workload are assigned to the same isolation level [19, 22, 25]. Generally, a split schedule is of the following form: one transaction is split in two (hence, the name) and some other transactions are placed between these two parts in a serial fashion where both the splitted and the intermediate serial transactions satisfy some additional requirements. All remaining transactions (if any) are appended after the splitted transaction, again, in a serial fashion. We refer to Figure 1 for the general structure of a split schedule. The split schedules used in the cited papers all differ in the additional requirements. When these additional requirements are simple, a direct enumeration of all possible split schedules can be avoided and replaced by a more efficient polynomial time algorithm [22, 25]. In some cases, however, finding a counterexample split schedule is NP-complete [22] or even undecidable [26]. In the present paper, we consider mixed allocations where different transactions can be allocated to different isolation levels. The corresponding split schedule is consequentially more involved as it needs to take interrelationships between multiple isolation levels into account. We show in Theorem 3.4 that a counterexample split schedule can still be efficiently constructed.

The contributions of this paper can be summarized as follows:

(1) We provide a formal framework to reason on robustness in the presence of mixed allocations of isolation levels. In particular, we formally define what it means for a schedule to be allowed under a (mixed) allocation w.r.t. {RC, SI, SSI} (cf., Definition 2.5). Even though these definitions are an abstraction, they are consistent with mixed allocations as they are applied within PostgreSQL and Oracle.
(2) We characterize non-robustness for allocations over {RC, SI, SSI} in terms of the existence of a multiversion split-schedule.
(3) We provide a polynomial time decision procedure for robustness against an allocation over {RC, SI, SSI}.
(4) We show that there is always a unique optimal robust allocation over {RC, SI, SSI} and we provide a polynomial time algorithm for computing it.

(5) We show that is decidable in polynomial time whether there exists a robust allocation over {RC, SI} for a given set of transactions. Furthermore, when a robust allocation exists, an optimal one can be found in polynomial time as well.

Outline. This paper is structured as follows. We introduce the necessary definitions in Section 2. We consider the robustness and allocation problem for {RC, SI, SSI} in Section 3 and Section 4, respectively. We consider robustness and allocation for {RC, SI} in Section 5. We discuss related work in Section 6. We conclude in Section 7.

Novelty Requirement. The present paper is the full version of [27] and supplies all proofs. In particular, full proofs of the following non-trivial results are added: Theorem 3.2, Proposition 3.3, Theorem 3.4, and Proposition 4.1. In addition, this paper includes an optimization involving read-only transactions in our definition of dangerous structures (cf. Section 2.3) that was not present in the conference version. We extended the discussion on dangerous structures in Section 2.3 accordingly and added Example 2.4 to better illustrate the concept. Furthermore, all relevant proofs have been altered to show that our findings presented in [27] still hold in the presence of this optimization.

2 DEFINITIONS

2.1 Transactions and Schedules

We fix an infinite set of objects **Obj**. For an object $\mathtt{t} \in \mathbf{Obj}$, we denote by $\mathtt{R[t]}$ a *read* operation on $\mathtt{t}$ and by $\mathtt{W[t]}$ a *write* operation on $\mathtt{t}$. We also assume a special *commit* operation denoted by $\mathtt{C}$. A *transaction* T over **Obj** is a sequence of read and write operations on objects in **Obj** followed by a commit. In the sequel, we leave the set of objects **Obj** implicit when it is clear from the context and just say transaction rather than transaction over **Obj**.

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. For a transaction T, we use $\mathit{first}(T)$ to refer to the first operation in T.

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 $\mathtt{W}_i\mathtt{[t]}$ and $\mathtt{R}_i\mathtt{[t]}$ to denote a $\mathtt{W[t]}$ and $\mathtt{R[t]}$ 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.* [12, 19]). To avoid ambiguity of notation, we assume that a transaction performs at most one write and one read operation per object. The latter is a common assumption (see, *e.g.* [19]). All our results carry over to the more general setting in which multiple writes and reads per object 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 objects,
- $\leq_s$ encodes the ordering of these operations,
- $\ll_s$ is a *version order* providing for each object $\mathtt{t}$ a total order over all write operations on $\mathtt{t}$ 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 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 object 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 object. The version order $\ll_s$ represents the order in which different versions of an object are installed in the database. For a pair of write operations on the same object, this version order does not necessarily coincide with $\leq_s$. For example, under RC and SI the version order is based on the commit order instead, with the transaction committing first installing the first version of the object, independent of the order of the write operations in $\leq_s$ (cf., Definition 2.3).[2] See Figure 2 for an illustration of a schedule. In this schedule, the read operations on $\mathtt{t}$ in T_1 and T_4 both read the initial version of $\mathtt{t}$ instead of the version written but not yet committed by T_2. Furthermore, the read operation $\mathtt{R}_2[\mathtt{v}]$ in T_2 reads the initial version of $\mathtt{v}$ instead of the version written by T_3, even though T_3 commits before $\mathtt{R}_2[\mathtt{v}]$.

We say that a schedule s is a *single version schedule* if $\ll_s$ is compatible with $\leq_s$ and every read operation always reads the last written version of the object. Formally, for each pair of write operations a and b on the same object, $a \ll_s b$ iff $a <_s b$, and for every read operation a there is no write operation c on the same object 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 [13, 19] 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.

2.2 Conflict-Serializability

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

- *(ww-conflict)* $b_i = \mathtt{W}_i[\mathtt{t}]$ and $a_j = \mathtt{W}_j[\mathtt{t}]$; or,
- *(wr-conflict)* $b_i = \mathtt{W}_i[\mathtt{t}]$ and $a_j = \mathtt{R}_j[\mathtt{t}]$; or,
- *(rw-conflict)* $b_i = \mathtt{R}_i[\mathtt{t}]$ and $a_j = \mathtt{W}_j[\mathtt{t}]$.

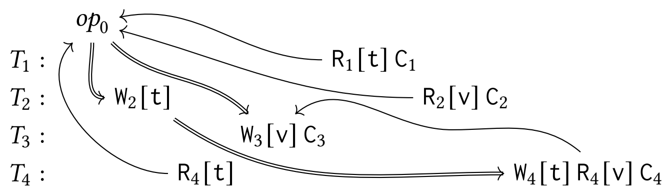
In this case, we also say that b_i and a_j are conflicting operations. Furthermore, commit operations and the special operation op_0 never conflict with any other operation. When b_i and a_j are conflicting operations in $\mathcal{T}$, we say that a_j *depends on* b_i in a schedule s over $\mathcal{T}$,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

- *(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 an object 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. For example, the dependencies $\mathtt{W}_2[\mathtt{t}] \rightarrow \mathtt{W}_4[\mathtt{t}]$, $\mathtt{W}_3[\mathtt{v}] \rightarrow \mathtt{R}_4[\mathtt{v}]$ and $\mathtt{R}_4[\mathtt{t}] \rightarrow \mathtt{W}_2[\mathtt{t}]$ are respectively a ww-dependency, a wr-dependency and a rw-antidependency in schedule s presented in Figure 2.

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 $b_i \rightarrow_{s'} a_j$.

[2]It is interesting to note that for the isolation levels considered in this paper, the definitions indirectly imply that the version order still coincides with $\leq_s$ (cf., Proposition 3.3).



op_0
T_1 :
T_2 :
T_3 :
T_4 :
$\mathtt{R}_1[\mathtt{t}]\,\mathtt{C}_1$
$\mathtt{W}_2[\mathtt{t}]$
$\mathtt{R}_2[\mathtt{v}]\,\mathtt{C}_2$
$\mathtt{W}_3[\mathtt{v}]\,\mathtt{C}_3$
$\mathtt{R}_4[\mathtt{t}]$
$\mathtt{W}_4[\mathtt{t}]\,\mathtt{R}_4[\mathtt{v}]\,\mathtt{C}_4$

Fig. 2. A schedule s with v_s (single lines) and $\ll_s$ (double lines) represented through arrows. The horizontal axis represents the order of operations $\leq_s$.

Definition 2.1. A schedule s is *conflict-serializable* if it is conflict-equivalent to a single version serial schedule.

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

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

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

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

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

Theorem 2.2 (implied by [2]). *A schedule s is conflict-serializable iff $SeG(s)$ is acyclic.*

Figure 3 visualizes the serialization graph $SeG(s)$ for the schedule s in Figure 2. Since $SeG(s)$ is not acyclic, s is not conflict-serializable.

2.3 Isolation Levels

Let $\mathcal{I}$ be a class of isolation levels. An $\mathcal{I}$*-allocation* $\mathcal{A}$ for a set of transactions $\mathcal{T}$ is a function mapping each transaction $T \in \mathcal{T}$ onto an isolation level $\mathcal{A}(T) \in \mathcal{I}$. When $\mathcal{I}$ is not important or clear from the context, we sometimes also say allocation rather than $\mathcal{I}$-allocation. In this paper, we consider the following isolation levels: read committed (RC), snapshot isolation (SI), and serializable snapshot isolation (SSI). In general, with the exception of Section 5, $\mathcal{I} = \{\text{RC}, \text{SI}, \text{SSI}\}$. Before we define what it means for a schedule to consist of transactions adhering to different isolation levels, we introduce some necessary terminology. Some of these notions are illustrated in Example 2.6 below.

Let s be a schedule for a set $\mathcal{T}$ of transactions. Two transactions $T_i, T_j \in \mathcal{T}$ are said to be *concurrent* in s when their execution overlaps. That is, if *first*$(T_i) <_s \mathtt{C}_j$ and *first*$(T_j) <_s \mathtt{C}_i$. We say that a write operation $\mathtt{W}_j[\mathtt{t}]$ in a transaction $T_j \in \mathcal{T}$ *respects the commit order of* s if the version of $\mathtt{t}$ written by

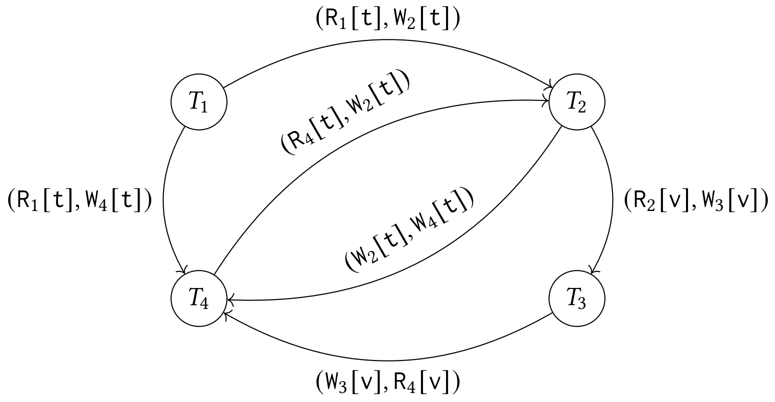

$(\mathtt{R}_1[\mathtt{t}], \mathtt{W}_2[\mathtt{t}])$
T_1
T_2
$(\mathtt{R}_4[\mathtt{t}], \mathtt{W}_2[\mathtt{t}])$
$(\mathtt{R}_1[\mathtt{t}], \mathtt{W}_4[\mathtt{t}])$
$(\mathtt{W}_2[\mathtt{t}], \mathtt{W}_4[\mathtt{t}])$
$(\mathtt{R}_2[\mathtt{v}], \mathtt{W}_3[\mathtt{v}])$
T_4
T_3
$(\mathtt{W}_3[\mathtt{v}], \mathtt{R}_4[\mathtt{v}])$

Fig. 3. Serialization graph $SeG(s)$ for the schedule s presented in Figure 2. Each edge (T_i, T_j) is labelled with the pairs in $\lambda(T_i, T_j)$.

T_j is installed after all versions of t installed by transactions committing before T_j commits, but before all versions of t installed by transactions committing after T_j commits. More formally, $\mathtt{W}_j[\mathtt{t}]$ respects the commit order of s if for every write operation $\mathtt{W}_i[\mathtt{t}]$ in a transaction $T_i \in \mathcal{T}$ different from T_j we have $\mathtt{W}_j[\mathtt{t}] \ll_s \mathtt{W}_i[\mathtt{t}]$ iff $\mathtt{C}_j <_s \mathtt{C}_i$. We next define when a read operation $a \in T$ reads the last committed version relative to a specific operation. For RC this operation is a itself while for SI this operation is *first*(T). Intuitively, these definitions enforce that read operations in transactions allowed under RC act as if they observe a snapshot taken right before the read operation itself, while under SI they observe a snapshot taken right before the first operation of the transaction. A read operation $\mathtt{R}_j[\mathtt{t}]$ in a transaction $T_j \in \mathcal{T}$ is *read-last-committed in s relative to an operation* $a_j \in T_j$ (not necessarily different from $\mathtt{R}_j[\mathtt{t}]$) if the following holds:

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

The first condition says that $\mathtt{R}_j[\mathtt{t}]$ either reads the initial version or a committed version, while the second condition states that $\mathtt{R}_j[\mathtt{t}]$ observes the most recently committed version of t (according to $\ll_s$). A transaction $T_j \in \mathcal{T}$ *exhibits a concurrent write in s* if there is another transaction $T_i \in \mathcal{T}$ and there are two write operations b_i and a_j in s on the same object with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j$ and *first*$(T_j) <_s \mathtt{C}_i$. That is, transaction T_j writes to an object that has been modified earlier by a concurrent transaction T_i.

A transaction $T_j \in \mathcal{T}$ *exhibits a dirty write in s* if there are two write operations b_i and a_j in s with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j <_s \mathtt{C}_i$. That is, transaction T_j writes to an object that has been modified earlier by T_i, but T_i has not yet issued a commit. Notice that by definition a transaction exhibiting a dirty write always exhibits a concurrent write. Transaction T_4 in Figure 2 exhibits a concurrent write, since it writes to t, which has been modified earlier by a concurrent transaction T_2. However, T_4 does not exhibit a dirty write, since T_2 has already committed before T_4 writes to t.

Definition 2.3. Let s be a schedule over a set of transactions $\mathcal{T}$. A transaction $T_i \in \mathcal{T}$ is *allowed under isolation level read committed (RC) in s* if:

- each write operation in T_i respects the commit order of s;
- each read operation $b_i \in T_i$ is read-last-committed in s relative to b_i; and
- T_i does not exhibit dirty writes in s.

$$\begin{tikzcd}
T_1 : & & \mathtt{W}_1[\mathtt{q}] & & \mathtt{R}_1[\mathtt{t}]\,\mathtt{C}_1 \arrow[dr, dashed] & \\
T_2 : & \mathtt{R}_2[\mathtt{v}] \arrow[drr, dashed] & & & & \mathtt{W}_2[\mathtt{t}]\,\mathtt{C}_2 \\
T_3 : & & & \mathtt{W}_3[\mathtt{v}]\,\mathtt{C}_3 & &
\end{tikzcd}$$

Fig. 4. Example of a dangerous structure $T_1 \longrightarrow T_2 \longrightarrow T_3$ with the required rw-antidependencies represented through dashed arrows.

$$\begin{tikzcd}
T_1 : & & & \mathtt{R}_1[\mathtt{q}] & \mathtt{R}_1[\mathtt{t}]\,\mathtt{C}_1 \arrow[dr, dashed] & \\
T_2 : & \mathtt{R}_2[\mathtt{v}] \arrow[dr, dashed] & & & & \mathtt{W}_2[\mathtt{t}]\,\mathtt{C}_2 \\
T_3 : & & \mathtt{W}_3[\mathtt{v}]\,\mathtt{C}_3 & & &
\end{tikzcd}$$

Fig. 5. Example of a dangerous structure $T_1 \longrightarrow T_2 \longrightarrow T_3$ where T_1 is a read-only transaction. The required rw-antidependencies are represented through dashed arrows.

A transaction $T_i \in \mathcal{T}$ is *allowed under isolation level snapshot isolation (SI) in s* if:

- each write operation in T_i respects the commit order of s;
- each read operation in T_i is read-last-committed in s relative to *first*(T_i); and
- T_i does not exhibit concurrent writes in s.

We then say that the schedule s is allowed under RC (respectively, SI) if every transaction is allowed under RC (respectively, SI) in s. The latter definitions correspond to the ones in the literature (see, e.g., [19, 25]). We emphasize that our definition of RC is based on concrete implementations over multiversion databases, found in e.g. PostgreSQL, and should therefore not be confused with different interpretations of the term Read Committed, such as lock-based implementations [12] or more abstract specifications covering a wider range of concrete implementations (see, e.g., [2]). In particular, abstract specifications such as [2] do not require the read-last-committed property, thereby facilitating implementations in distributed settings, where read operations are allowed to observe outdated versions. When studying robustness, such a broad specification of RC is not desirable, since it allows for a wide range of schedules that are not conflict-serializable. We furthermore point out that our definitions of RC and SI are not strictly weaker forms of conflict-serializability. That is, a conflict-serializable schedule is not necessarily allowed under RC and SI as well.

While RC and SI are defined on the granularity of a single transaction, SSI enforces a global condition on the schedule as a whole. For this, recall the concept of dangerous structures from [14]: three transactions $T_1, T_2, T_3 \in \mathcal{T}$ (where T_1 and T_3 are not necessarily different) form a *dangerous structure* $T_1 \longrightarrow T_2 \longrightarrow T_3$ in s if:

- there is a rw-antidependency from T_1 to T_2 and from T_2 to T_3 in s;
- T_1 and T_2 are concurrent in s ;
- T_2 and T_3 are concurrent in s ;
- $\mathtt{C}_3 \leq_s \mathtt{C}_1$ and $\mathtt{C}_3 <_s \mathtt{C}_2$; and
- if T_1 is read-only, then $\mathtt{C}_3 <_s$ *first*(T_1).

Note that this definition of dangerous structures slightly extends upon the one in [14], where it is not required for T_3 to commit before T_1 and T_2. In the full version [15] of that paper, it is shown that, if all transactions are allowed under SI, such a structure can only lead to non-serializable schedules if T_3 commits first. Furthermore, Ports and Grittner [23] show that if T_1 is a read-only transaction, this structure can only lead to non-serializable behavior if T_3 commits before T_1 starts.

Actual implementations of SSI (e.g., PostgreSQL [23]) therefore include this optimization when monitoring for dangerous structures to reduce the number of aborts due to false positives. It is interesting to note that presence of a dangerous structure on itself does not necessarily mean that the schedule s is non-conflict-serializable, as our definition does not require a cycle in the serialization graph $SeG(s)$. However, if all transactions are allowed under SI, then every cycle in $SeG(s)$ implies a dangerous structure as part of the cycle [20, 23]. Stated differently, the absence of dangerous structures is a sufficient condition for conflict-serializability when all transactions are allowed under SI.

Example 2.4. Figure 4 provides an example of a dangerous structure $T_1 \rightarrow T_2 \rightarrow T_3$. Notice in particular that T_1 is not a read-only transaction, thereby allowing T_1 and T_3 to be concurrent. Indeed, if we would replace the write operation $\mathtt{W}_1[\mathtt{q}]$ by a read operation $\mathtt{R}_1[\mathtt{q}]$ in Figure 4, the result would no longer be a dangerous structure. An example of a dangerous structure where T_1 is a read-only transaction is given in Figure 5. □

We are now ready to define when a schedule is allowed under a (mixed) allocation of isolation levels.

Definition 2.5. A schedule s over a set of transactions $\mathcal{T}$ *is allowed under an allocation* $\mathcal{A}$ over $\mathcal{T}$ if:

- for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) = \text{RC}$, T_i is allowed under RC in s;
- for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) \in \{\text{SI}, \text{SSI}\}$, T_i is allowed under SI in s; and
- there is no dangerous structure $T_i \rightarrow T_j \rightarrow T_k$ in s formed by three (not necessarily different) transactions $T_i, T_j, T_k \in \{T \in \mathcal{T} \mid \mathcal{A}(T) = \text{SSI}\}$.

We denote the allocation mapping all transactions to RC (respectively, SI) by $\mathcal{A}_{\text{RC}}$ (respectively, $\mathcal{A}_{\text{SI}}$).

We illustrate some of the just introduced notions through an example.

Example 2.6. Consider the schedule s in Figure 2. Transaction T_1 is concurrent with T_2 and T_4, but not with T_3; all other transactions are pairwise concurrent with each other. The second read operation of T_4 is a read-last-committed relative to itself but not relative to the start of T_4. The read operation of T_2 is read-last-committed relative to the start of T_2, but not relative to itself, so an allocation mapping T_2 to RC is not allowed. All other read operations are read-last-committed relative to both themselves and the start of the corresponding transaction. None of the transactions exhibits a dirty write. Only transaction T_4 exhibits a concurrent write (witnessed by the write operation in T_2). Due to this, an allocation mapping T_4 on SI or SSI is not allowed. The transactions $T1 \rightarrow T2 \rightarrow T3$ form a dangerous structure, therefore an allocation mapping all three transactions T_1, T_2, T_3 on SSI is not allowed. All other allocations, that is, mapping T_4 on RC, T_2 on SI or SSI and at least one of T_1, T_2, T_3 on RC or SI, is allowed. □

As mentioned above, the isolation levels RC and SI are defined through a local condition that should hold for every transaction. This formulation on the granularity of a single transaction is precisely what facilitates the definition of what it means for a schedule to be allowed under a mixed allocation. Mixing isolation levels in this way does introduce some subtleties as the next example shows.

Example 2.7. Consider the schedule s in Figure 6 over two concurrent transactions T_1 and T_2, where both T_1 and T_2 write to object $\mathtt{v}$. (1) Let $\mathcal{A}_1 = \mathcal{A}_{\text{SI}}$. Then, clearly s is not allowed under $\mathcal{A}_1$ as T_2 exhibits a concurrent write which is not allowed by SI. (2) The same is the case for allocation $\mathcal{A}_2$ with $\mathcal{A}_2(T_1) = \text{RC}$ and $\mathcal{A}_2(T_2) = \text{SI}$. (3) However, let $\mathcal{A}_3$ be the allocation with $\mathcal{A}_3(T_1) = \text{SI}$

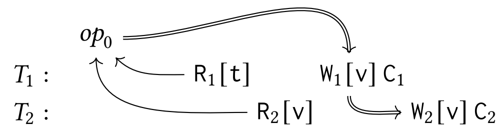


Fig. 6. Schematic representation of schedule s in Example 2.7. The version function v_s (single lines) and version order $\ll_s$ (double lines) are represented through arrows.

and $\mathcal{A}_3(T_2) = \text{RC}$. Then, s is allowed under $\mathcal{A}_3$ as the concurrent write exhibited by T_2 is allowed by RC and T_1 does *not* exhibit a concurrent write. We stress once again that our definitions are in line with those of PostgreSQL.[3] □

2.4 Robustness

We define the robustness property [13] (also called *acceptability* in [19, 20]), which guarantees serializability for all schedules over a given set of transactions for a given allocation.

Definition 2.8 (Robustness). A set of transactions $\mathcal{T}$ is *robust* against an allocation $\mathcal{A}$ for $\mathcal{T}$ if every schedule for $\mathcal{T}$ that is allowed under $\mathcal{A}$ is conflict-serializable.

We refer to $\mathcal{A}$ as a *robust allocation*. The *robustness problem* is then to decide whether a given allocation for a set of transactions $\mathcal{T}$ is a robust allocation. The set of transactions $\{T_1, T_2\}$ presented in Example 2.7 is not robust against allocation $\mathcal{A}_3$, witnessed by schedule s in Figure 6. Indeed, s is allowed under $\mathcal{A}_3$, but not conflict-serializable.

3 DECIDING ROBUSTNESS

In this section, we address the robustness problem as defined in the previous section.

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 *conflicting quadruples* for $\mathcal{T}$. Notice that, conflicting quadruples are not defined w.r.t. a schedule (as is the case for $SeG(s)$ in Section 2.2). Further, for an operation $b \in T$, we denote by $\text{prefix}_b(T)$ the restriction of T to all operations that are before or equal to b according to $\leq_T$. Similarly, we denote by $\text{postfix}_b(T)$ the restriction of T to all operations that are strictly after b according to $\leq_T$.

Definition 3.1 (Multiversion split schedule). Let $\mathcal{T}$ be a set of transactions, $\mathcal{A}$ an allocation for $\mathcal{T}$, 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 conflicting quadruples for $\mathcal{T}$ such that each transaction in $\mathcal{T}$ occurs in at most two different quadruples. A *multiversion split schedule* for $\mathcal{T}$ and $\mathcal{A}$ based on C is a multiversion schedule that has the following form:

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

where

(1) there is no operation in T_1 conflicting with an operation in any of the transactions $T_3, \ldots, T_{m-1}$;
(2) there is no write operation in $\text{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in T_2 or T_m;
(3) if $\mathcal{A}(T_1) \in \{\text{SI}, \text{SSI}\}$, then there is no write operation in $\text{postfix}_{b_1}(T_1)$ ww-conflicting with a write operation in T_2 or T_m;
(4) b_1 is rw-conflicting with a_2;

[3]https://www.postgresql.org/docs/14/transaction-iso.html

(5) b_m is rw-conflicting with a_1 or ($\mathcal{A}(T_1)$ = RC and $b_1 <_{T_1} a_1$);
(6) $\mathcal{A}(T_1) \neq$ SSI or $\mathcal{A}(T_2) \neq$ SSI or $\mathcal{A}(T_m) \neq$ SSI;
(7) if $\mathcal{A}(T_1)$ = SSI and $\mathcal{A}(T_2)$ = SSI, then there is no operation in T_1 wr-conflicting with an operation in T_2; and
(8) if $\mathcal{A}(T_1)$ = SSI and $\mathcal{A}(T_m)$ = SSI, then there is no operation in T_1 rw-conflicting with an operation in T_m.

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

We emphasize that we do not require the sequence of conflicting quadruples C presented in Definition 3.1 to contain all possible conflicting quadruples over the considered set of transactions $\mathcal{T}$. This allows each transaction T in $\mathcal{T}$ to occur in at most two different quadruples in C, even if T is involved in more than two pairs of conflicting operations. The following theorem characterizes non-robustness in terms of the existence of a multiversion split schedule. The proof argument is based on showing that if a multiversion split schedule s for $\mathcal{T}$ and $\mathcal{A}$ based on C satisfies Definition 3.1, then we can construct a version order $\ll_s$ and version function v_s such that s is allowed under $\mathcal{A}$ and not conflict-serializable, thereby witnessing non-robustness. Intuitively, Definition 3.1 (1-3) ensures that the transactions in s do not exhibit concurrent or dirty writes not allowed by $\mathcal{A}$, Definition 3.1 (4-5) enforces (anti)dependencies $b_1 \rightarrow a_2$ and $b_m \rightarrow a_1$ to occur in s, and Definition 3.1 (6-8) ensures that no dangerous structure occurs over transactions adhering to SSI. The opposite direction is more involved, as we show that every schedule s for $\mathcal{T}$ allowed under $\mathcal{A}$ that is not conflict-serializable gives rise to a multiversion split schedule s satisfying Definition 3.1. In particular, we construct the sequence of conflicting quadruples C as in Definition 3.1 based on a chord-free cycle Γ in $SeG(s)$, where the order of transactions in C is chosen such that T_2 commits first in s (among those in Γ).

Theorem 3.2. *For a set of transactions $\mathcal{T}$ and an allocation $\mathcal{A}$ for $\mathcal{T}$, the following are equivalent:*
(1) $\mathcal{T}$ is not robust against $\mathcal{A}$;
(2) there is a multiversion split schedule s for $\mathcal{T}$ and $\mathcal{A}$ based on some sequence of conflicting quadruples C.

Before proving Theorem 3.2, we first present some properties relating dependencies to the order of operations that hold for every schedule s allowed under an allocation $\mathcal{A}$.

Proposition 3.3. *Let $\mathcal{T}$ be a set of transactions and $\mathcal{A}$ an allocation for $\mathcal{T}$. Furthermore, let b_i and a_j be two conflicting operations and T_i and T_j their respective transactions in $\mathcal{T}$. For every schedule s allowed under $\mathcal{A}$, the following are true:*
(1) if $b_i \rightarrow_s a_j$ is a ww-dependency or a wr-dependency in s, then $\mathtt{C}_i <_s a_j$. Furthermore, if $\mathcal{A}(T_j) \in \{SI, SSI\}$, then $\mathtt{C}_i <_s first(T_j)$;
(2) if $b_i \rightarrow_s a_j$ is a rw-antidependency, then $first(T_i) <_s \mathtt{C}_j$. Furthermore, if $\mathcal{A}(T_i) = RC$, then $b_i <_s \mathtt{C}_j$;
(3) if $\mathtt{C}_i <_s first(T_j)$, then $b_i \rightarrow_s a_j$ is an (anti)dependency in s;
(4) if $\mathtt{C}_i <_s a_j$ and $\mathcal{A}(T_j) = RC$, then $b_i \rightarrow_s a_j$ is an (anti)dependency in s;
(5) if b_i is rw-conflicting with a_j and $b_i <_s \mathtt{C}_j$, then $b_i \rightarrow_s a_j$ is a rw-antidependency in s.

Proof. Since s is allowed under $\mathcal{A}$, each write operation in a transaction $T_i \in \mathcal{T}$ respects the commit order of s. As a result, for each pair of write operations $c_i \in T_i$ and $c_j \in T_j$ on the same object $\mathtt{t}$ with T_i and T_j two different transactions occurring in $\mathcal{T}$, we have

$$c_i \ll_s c_j \text{ iff } \mathtt{C}_i <_s \mathtt{C}_j. \qquad (\dagger)$$

Furthermore,

$$\text{there is no transaction } T_i \text{ exhibiting a dirty write in } s. \qquad (\ddagger)$$

Indeed, if $\mathcal{A}(T_i) = \text{RC}$, this is immediate, and if $\mathcal{A}(T_i) \in \{\text{SI}, \text{SSI}\}$, this follows from the fact that a transaction exhibiting a dirty write always exhibits a concurrent write.

(1) We first consider the case where $b_i \rightarrow_s a_j$ is a ww-dependency. By definition and (†), we have $b_i \ll_s a_j$ and therefore $\mathtt{C}_i <_s \mathtt{C}_j$. It now follows that $\mathtt{C}_i <_s a_j$, since $a_j <_s \mathtt{C}_i$ implies that either $b_i <_s a_j <_s \mathtt{C}_i <_s \mathtt{C}_j$ or $a_j <_s b_i <_s \mathtt{C}_i <_s \mathtt{C}_j$. In both cases, there is a transaction exhibiting a dirty write, thereby contradicting (‡). Furthermore, if $\mathcal{A}(T_j) \in \{\text{SI}, \text{SSI}\}$, then $\mathtt{C}_i <_s \mathit{first}(T_j)$ must hold, as otherwise we have $\mathit{first}(T_j) <_s \mathtt{C}_i$ and $b_i <_s \mathtt{C}_i <_s a_j$, thereby witnessing T_j exhibiting a concurrent write in s and therefore contradicting the assumption that T_j is allowed under SI in s.

Next, consider the case where $b_i \rightarrow_s a_j$ is a wr-dependency. By definition, $b_i = v_s(a_j)$ or $b_i \ll_s v_s(a_j)$. Notice that $v_s(a_j)$ must be a write operation different from op_0, since $b_i \ll_s op_0$ cannot hold by definition of $\ll_s$. Let T_k be the transaction in $\mathcal{T}$ with $v_s(a_j) \in T_k$. Then, $\mathtt{C}_i \leq_s \mathtt{C}_k$, since $b_i = v_s(a_j)$ or $b_i \ll_s v_s(a_j)$ and by (†). If $\mathcal{A}(T_j) = \text{RC}$, then a_j is read-last-committed relative to itself, thereby implying that $\mathtt{C}_i \leq_s \mathtt{C}_k <_s a_j$. If $\mathcal{A}(T_j) \in \{\text{SI}, \text{SSI}\}$ instead, then a_j is read-last-committed relative to $\mathit{first}(T_j)$, thereby implying that $\mathtt{C}_i \leq_s \mathtt{C}_k <_s \mathit{first}(T_j) \leq_s a_j$.

(2) Assuming $b_i \rightarrow_s a_j$ is a rw-antidependency, we have $v_s(b_i) \ll_s a_j$. Let c_i be the operation in T_i such that $c_i = b_i$ if $\mathcal{A}(T_i) = \text{RC}$ and $c_i = \mathit{first}(T_i)$ if $\mathcal{A}(T_i) \in \{\text{SI}, \text{SSI}\}$. The desired $c_i <_s \mathtt{C}_j$ is now immediate. Indeed, $v_s(b_i) \ll_s a_j$ and $\mathtt{C}_j <_s c_i$ would contradict b_i being read-last-committed in s relative to c_i.

(3 and 4) The proof is by contraposition. Assume $b_i \rightarrow_s a_j$ is not an (anti)dependency in s. Then, since b_i and a_j are conflicting operations, we have $a_j \rightarrow_s b_i$. We first consider the case where $a_j \rightarrow_s b_i$ is a ww- or a wr-dependency. Then, $\mathtt{C}_j <_s b_i$ follows from (1), thereby implying that $\mathit{first}(T_j) \leq_s a_j <_s \mathtt{C}_i$. Next, assume $a_j \rightarrow_s b_i$ is a rw-antidependency. Then $\mathit{first}(T_j) <_s \mathtt{C}_i$ is immediate by (2). Furthermore, if $\mathcal{A}(T_j) = \text{RC}$, $a_j <_s \mathtt{C}_i$ follows from (2).

(5) Assume b_i is rw-conflicting with a_j. The proof showing by contraposition that the wr-dependency $a_j \rightarrow_s b_i$ implies that $\mathtt{C}_j <_s b_i$ is now immediate by (1). □

Using the properties presented in Proposition 3.3, we now prove Theorem 3.2.

Proof of Theorem 3.2. ($1 \rightarrow 2$) Assume $\mathcal{T}$ is not robust against $\mathcal{A}$. Then, there is a non-conflict-serializable (multiversion) schedule s over $\mathcal{T}$ allowed under $\mathcal{A}$. Let Γ be the cycle in $SeG(s)$ implied by Theorem 2.2. Without loss of generality, we assume Γ is a minimal cycle in $SeG(s)$, consisting of the edges

$$(T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$$

where T_2 is the transaction that commits first (among those in Γ) in s. That is, we enumerate the edges in this minimal cycle Γ in order, starting from the transaction immediately preceding the transaction committing first. More formally, for each T_i in Γ different from T_2, we have $\mathtt{C}_2 <_s \mathtt{C}_i$. Note furthermore that each edge (T_i, b_i, a_j, T_j) in Γ implies that (T_i, b_i, a_j, T_j) is a conflicting quadruple for $\mathcal{T}$. Indeed, by definition of $b_i \rightarrow_s a_j$, operations b_i and a_j are conflicting. Let

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

be the sequence of conflicting quadruples over $\mathcal{T}$ that are implied by the edges in Γ. We argue that the multiversion split schedule s' for $\mathcal{T}$ and $\mathcal{A}$ based on C satisfies the conditions in Definition 3.1. *(Condition 4)* We first show that b_1 is rw-conflicting with a_2. The edge (T_1, b_1, a_2, T_2) in $SeG(s)$ implies that there is a dependency from b_1 to a_2 in s. We conclude from Proposition 3.3(1) that $b_1 \rightarrow_s a_2$ is a rw-antidependency, as otherwise $\mathtt{C}_1 <_s a_2 <_s \mathtt{C}_2$, thereby contradicting the assumption that $\mathtt{C}_2 <_s \mathtt{C}_1$.

(Condition 1) Since we assumed Γ to be a minimal cycle in $SeG(s)$, T_1 cannot have an operation conflicting with an operation in any of the transactions $T_3, \ldots, T_{m-1}$. Indeed, such a pair of conflicting operations would always imply an (anti)dependency in s, and therefore an edge in $SeG(s)$ between two transactions that are not adjacent in Γ.
(Condition 2) We show that there is no write operation in $\text{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in transactions T_2 or T_m. Assume towards a contradiction that there is a write operation $c_i \in T_i$ with $i \in \{2, m\}$ ww-conflicting with an operation $c_1 \in \text{prefix}_{b_1}(T_1)$. Based on Proposition 3.3(1) and 3.3.2 as well as our earlier conclusion that $b_1 \rightarrow_s a_2$ is a rw-antidependency (see Condition 4), we argue by case on $\mathcal{A}(T_1)$ that $c_1 \rightarrow_s c_i$ is a dependency in s, as $c_i \rightarrow_s c_1$ cannot hold. If $\mathcal{A}(T_1) = \text{RC}$, the dependency $c_i \rightarrow_s c_1$ implies $\mathtt{C}_i <_s c_1 \leq_s b_1 <_s \mathtt{C}_2$. Otherwise, if $\mathcal{A}(T_1) \in \{\text{SI}, \text{SSI}\}$, this dependency implies $\mathtt{C}_i <_s \mathit{first}(T_1) <_s \mathtt{C}_2$. Both cases contradict our assumption that T_2 is the first committing transaction among those in Γ.

The dependency $c_1 \rightarrow_s c_i$ implies the edge (T_1, c_1, c_i, T_i) in $SeG(s)$. We therefore conclude that $T_i = T_2$, as we assumed Γ to be a minimal cycle in $SeG(s)$. However, by Proposition 3.3(1), this ww-dependency implies that $\mathtt{C}_1 <_s \mathtt{C}_2$, leading to the desired contradiction.
(Condition 3) Assume $\mathcal{A}(T_1) \in \{\text{SI}, \text{SSI}\}$. We prove that no write operation in $\text{postfix}_{b_1}(T_1)$ ww-conflicting with a write operation in transactions T_2 or T_m exists. Towards a contradiction, assume a write operation $c_i \in T_i$ with $i \in \{2, m\}$ exists, ww-conflicting with an operation $c_1 \in \text{postfix}_{b_1}(T_1)$. The proof is analogous to Condition 1 above: since $c_i \rightarrow_s c_1$ would imply that $\mathtt{C}_i <_s \mathit{first}(T_1) <_s \mathtt{C}_2$, $c_1 \rightarrow_s c_i$ is a dependency in s and, by minimality of Γ, $T_i = T_2$, resulting in the desired contradiction that $\mathtt{C}_1 <_s \mathtt{C}_2$.
(Condition 5) The proof is by case on $\mathcal{A}(T_1)$. First, assume $\mathcal{A}(T_1) \in \{\text{SI}, \text{SSI}\}$. We argue that b_m is rw-conflicting with a_1. Since $b_1 \rightarrow_s a_2$ is a rw-antidependency (see Condition 4 above), it follows from Proposition 3.3(2) that $\mathit{first}(T_1) <_s \mathtt{C}_2$. Assume towards a contradiction that $b_m \rightarrow_s a_1$ is not a rw-antidependency (i.e., a ww-dependency or a wr-dependency). Then, Proposition 3.3(1) implies that $\mathtt{C}_m <_s \mathit{first}(T_1)$, and therefore $\mathtt{C}_m <_s \mathtt{C}_2$, contradicting our assumption that T_2 commits first.

Next, assume $\mathcal{A}(T_1) = \text{RC}$. We argue that b_m is rw-conflicting with a_1 or $b_1 <_{T_1} a_1$. Towards a contradiction, assume b_m is not rw-conflicting with a_1 and $a_1 \leq_{T_1} b_1$. Then, $b_m \rightarrow_s a_1$ must be a wr-dependency or a ww-dependency, and Proposition 3.3(1) implies that $\mathtt{C}_m <_s a_1$. Furthermore, the rw-antidependency $b_1 \rightarrow_s a_2$ (see Condition 4 above), implies that $b_1 <_s \mathtt{C}_2$ (Proposition 3.3(2)). We conclude that $\mathtt{C}_m <_s a_1 \leq_{T_1} b_1 <_s \mathtt{C}_2$, thereby contradicting our assumption that T_2 commits first in s.
(Condition 6) Assume towards a contradiction that $\mathcal{A}(T_1) = \mathcal{A}(T_2) = \mathcal{A}(T_m) = \text{SSI}$. We argue that these three transactions form a dangerous structure $T_m \rightarrow T_1 \rightarrow T_2$ in s, thereby contradicting our assumption that s is allowed under $\mathcal{A}$. Note that we already argued the existence of a rw-antidependency from T_1 to T_2 and from T_m to T_1 (see Condition 4 and Condition 5, respectively). Furthermore, since we assumed T_2 to be the first transaction (among those in Γ) to commit in s, $\mathtt{C}_2 <_s \mathtt{C}_1$ and $\mathtt{C}_2 \leq_s \mathtt{C}_m$ are immediate. It remains to show *(i)* that T_1 and T_2 as well as T_1 and T_m are concurrent in s, and *(ii)* that $\mathtt{C}_2 <_s \mathit{first}(T_m)$ if T_m is read-only. Towards *(i)*, by Proposition 3.3(2), we have $\mathit{first}(T_1) <_s \mathtt{C}_2$ and $\mathit{first}(T_m) <_s \mathtt{C}_1$. Since T_2 commits before T_1, we have $\mathit{first}(T_2) <_s \mathtt{C}_2 <_s \mathtt{C}_1$, thereby showing that T_1 and T_2 are concurrent in s. Lastly, since $\mathit{first}(T_m) <_s \mathtt{C}_1$, transactions T_1 and T_m are concurrent in s as well, as otherwise $\mathtt{C}_m <_s \mathit{first}(T_1) <_s \mathtt{C}_2$, again contradicting our assumption that T_2 commits first. Towards *(ii)*, assume T_m is read-only. Then, the dependency $b_{m-1} \rightarrow_s a_m$ must be a wr-dependency, and by Proposition 3.3(1), we have $\mathtt{C}_{m-1} <_s \mathit{first}(T_m)$. Since $\mathtt{C}_2 \leq_s \mathtt{C}_{m-1}$, we conclude that $\mathtt{C}_2 <_s \mathit{first}(T_m)$.
(Condition 7) Assume towards a contradiction that $\mathcal{A}(T_1) = \mathcal{A}(T_2) = \text{SSI}$ and there is an operation c_1 in T_1 wr-conflicting with an operation c_2 in T_2. Then, $T_2 \neq T_m$, as otherwise $\mathcal{A}(T_1) = \mathcal{A}(T_2) = \mathcal{A}(T_m) = \text{SSI}$ contradicts Condition 6 above. Since Γ is by assumption a minimal cycle, the operations

c_1 and c_2 must lead to a wr-dependency $c_1 \rightarrow_s c_2$ from T_1 to T_2. By Proposition 3.3(1), we have $\mathsf{C}_1 <_s \mathit{first}(T_2) <_s \mathsf{C}_2$, thereby leading to the desired contradiction.
(Condition 8) Assume $\mathcal{A}(T_1) = \mathcal{A}(T_m) = \text{SSI}$ and there is an operation c_1 in T_1 rw-conflicting with an operation c_m in T_m. Analogous to the previous case, we can derive that $T_2 \neq T_m$, and consequently, that the operations c_1 and c_m must lead to a wr-dependency $c_m \rightarrow_s c_1$ from T_m to T_1. By Proposition 3.3(1), we have $\mathsf{C}_m <_s \mathit{first}(T_1)$. Furthermore, we already argued that $b_1 \rightarrow_s a_2$ is a rw-antidependency and therefore, by Proposition 3.3(2), that $\mathit{first}(T_1) <_s \mathsf{C}_2$. As a result, $\mathsf{C}_m <_s \mathsf{C}_2$, again leading to the desired contradiction.

($2 \rightarrow 1$) Let s be a multiversion split schedule for $\mathcal{T}$ and $\mathcal{A}$ based on a sequence of conflicting quadruples

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

It should be noted that Definition 3.1 does not imply a specific version order $\ll_s$ or a specific version function v_s. Indeed, Definition 3.1 only specifies the order of operations $<_s$, as well as which operations should be conflicting (notice that the definition of conflicting operations is not influenced by $\ll_s$ or v_s). To this end, we will extend s with a version order $\ll_s$ and a version function v_s and show that the resulting schedule is a valid counterexample schedule for robustness of $\mathcal{T}$ against $\mathcal{A}$. That is, s is allowed under $\mathcal{A}$ and is not conflict-serializable.

The version order $\ll_s$ is chosen such that for each transaction $T_i \in \mathcal{T}$, each write operation in T_i respects the commit order of s. For each read operation c_i in a transaction $T_i \in \mathcal{T}$, the version function $v_s(c_i)$ is chosen such that c_i is read-last-committed in s relative to c_i if $\mathcal{A}(T_i) = \text{RC}$, and c_i is read-last-committed in s relative to $\mathit{first}(T_i)$ if $\mathcal{A}(T_i) \in \{\text{SI}, \text{SSI}\}$. Notice that such a version order $\ll_s$ and version function v_s always exist.

We next argue that s is allowed under $\mathcal{A}$. By our choice for the version order $\ll_s$ and version function v_s, it only remains to show that (†) there is no transaction $T_i \in \mathcal{T}$ exhibiting a dirty write (if $\mathcal{A}(T_i) = \text{RC}$) or a concurrent write (if $\mathcal{A}(T_i) = \text{SI}$); and (‡) there is no dangerous structure $T_i \rightarrow T_j \rightarrow T_k$ in s formed by three (not necessarily different) transactions $T_i, T_j, T_k \in \{T \in \mathcal{T} \mid \mathcal{A}(T) = \text{SSI}\}$.

Towards (†), let c_i and c_j be two arbitrary ww-conflicting operations from two different transactions T_i and T_j in $\mathcal{T}$. We will show that T_j cannot exhibit a dirty write (respectively, concurrent write) in s if $\mathcal{A}(T_j) = \text{RC}$ (respectively, $\mathcal{A}(T_j) \in \{\text{SI}, \text{SSI}\}$). Notice that c_i and c_j can only witness T_j exhibiting a dirty write and/or a concurrent write if $c_i <_s c_j$, and we will therefore not consider the case where $c_j <_s c_i$. If $i > 1$ and $j > 1$, or if $i = 1$ and $j > m$ then $\mathsf{C}_i <_s \mathit{first}(T_j)$ is immediate by Definition 3.1, so T_j cannot exhibit a concurrent write (and therefore also not a dirty write) witnessed by c_i and c_j. If $i = 1$ and $1 < j \leq m$, then $c_1 \in \text{prefix}_{b_1}(T_1)$ since $c_1 <_s c_j$. As a result, T_j exhibiting a concurrent write (or a dirty write) witnessed by c_1 and c_j would contradict Condition 1 (if $j \in \{3, \ldots, m-1\}$) or Condition 2 (if $j \in \{2, m\}$) of Definition 3.1. If $1 < i \leq m$ and $j = 1$, then $c_1 \in \text{postfix}_{b_1}(T_1)$ since $c_i <_s c_1$. By Definition 3.1, $\mathsf{C}_i <_s c_1$, so T_1 cannot exhibit a dirty write witnessed by c_i and c_1. If furthermore $\mathcal{A}(T_1) \in \{\text{SI}, \text{SSI}\}$, then T_1 exhibiting a concurrent write witnessed by c_i and c_1 would contradict Condition 1 (if $i \in \{3, \ldots, m-1\}$) or Condition 3 (if $i \in \{2, m\}$) of Definition 3.1. Lastly, if $i > m$ and $j = 1$, then by Definition 3.1 we have $c_1 <_s \mathsf{C}_1 <_s c_i$, contradicting our assumption that $c_i <_s c_1$.

For (‡), assume towards a contradiction that there is a dangerous structure $T_i \rightarrow T_j \rightarrow T_k$ in s formed by three (not necessarily different) transactions $T_i, T_j, T_k \in \{T \in \mathcal{T} \mid \mathcal{A}(T) = \text{SSI}\}$. Recall that, by definition of dangerous structure, T_i and T_j as well as T_j and T_k must be concurrent in s; that there must be a rw-antidependency from T_i to T_j and from T_j to T_k in s; that $\mathsf{C}_k \leq_s \mathsf{C}_i$ and $\mathsf{C}_k <_s \mathsf{C}_j$; and that $\mathsf{C}_k <_s \mathit{first}(T_i)$ if T_i is read-only. By Definition 3.1, $T_j = T_1$. Indeed, every pair of concurrent transactions involves T_1, as no other pair of transactions in concurrent in s. Since T_1

cannot contain operations conflicting with operations in transactions different from T_2 and T_m (cf. Condition 1), T_i and T_k must be T_2 and/or T_m. However, $T_2 \rightarrow T_1 \rightarrow T_2$ and $T_m \rightarrow T_1 \rightarrow T_m$ cannot be a dangerous structure in s, as they would contradict Condition 7 and Condition 8, respectively. Analogously, $T_m \rightarrow T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1 \rightarrow T_m$ cannot be a dangerous structure in s, as both would contradict Condition 6.

To conclude the proof, we show that s is not conflict-serializable by arguing that

$$\Gamma = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$$

is a cycle in $SeG(s)$. In particular, for each edge (T_i, b_i, a_j, T_j) in Γ, we will argue that $b_i \rightarrow_s a_j$. First, let $i = 1$ and $j = 2$. By Condition 3 of Definition 3.1, b_1 is rw-conflicting with a_2. By our choice of v_s and $\ll_s$, we have $v_s(b_1) = op_0 \ll_s a_2$. The rw-antidependency $b_1 \rightarrow_s a_2$ is now immediate. Next, let $i > 1$ and $j = i + 1 < m$. From Definition 3.1, we have $\mathsf{C}_i <_s \mathit{first}(T_j)$. It now follows from Proposition 3.3(3) that $b_i \rightarrow_s a_j$. Lastly, let $i = m$ and $j = 1$. According to Condition 4 of Definition 3.1, either b_m is rw-conflicting with a_1, or $\mathcal{A}(T_1) = \text{RC}$ and $b_1 <_{T_1} a_1$. In the former case, $b_m \rightarrow_s a_1$ is immediate by Proposition 3.3(5) since $b_m <_s \mathsf{C}_1$. In the latter case, $a_1 \in \text{postfix}_{b_1}(T_1)$ and $b_m \rightarrow_s a_1$ therefore follows from Proposition 3.3(4). □

Theorem 3.2 is the basis for a PTIME algorithm deciding robustness against a given allocation. The algorithm is presented as Algorithm 1 and makes use of an auxilliary graph structure that we introduce next. For a transaction T_1 and a set of transactions $\mathcal{T}$, define mixed-iso-graph$(T_1, \mathcal{T})$ as the graph containing as nodes all transactions in $\mathcal{T}$ that do not have an operation conflicting with an operation in T_1, and with an edge between transactions T_i and T_j if T_i has an operation conflicting with an operation in T_j.

To verify robustness, Algorithm 1 does not check for the existence of a multiversion split schedule by iterating over all possible sequences C of conflicting quadruples, as this number can be exponential in the size of $\mathcal{T}$. Instead, Algorithm 1 iterates over all possible triples of transactions T_1, T_2 and T_m in $\mathcal{T}$ (where T_1, T_2 and T_m should be interpreted as in Definition 3.1) and verifies whether there exists a path from T_2 to T_m in mixed-iso-graph$(T_1, \mathcal{T} \setminus \{T_1, T_2, T_m\})$ (cf. function *reachable*(T_1, T_2, T_m) in Algorithm 1), thereby witnessing the existence of a corresponding sequence of conflicting quadruples between T_2 and T_m. By definition of mixed-iso-graph$(T_1, \mathcal{T} \setminus \{T_1, T_2, T_m\})$, Definition 3.1 (1) is furthermore satisfied. The remainder of Algorithm 1 verifies whether the remaining conditions in Definition 3.1 hold for at least one choice of $b_1, a_1 \in T_1$, $a_2 \in T_2$ and $b_m \in T_m$. Note in particular that function *ww-conflict-free*(b_1, T_1, T_2, T_m) returning True implies Definition 3.1 (2) and (3). The correspondence between the remaining properties of Definition 3.1 and conditions in Algorithm 1 is straightforward.

THEOREM 3.4. *Algorithm 1 decides whether a set of transactions $\mathcal{T}$ is robust against an allocation $\mathcal{A}$ in time $O(|\mathcal{T}|^3 \cdot (|\mathcal{T}|^3 + k^2\ell^2 + \ell^6))$, with k the total number of operations in $\mathcal{T}$ and ℓ the maximum number of operations in a transaction in $\mathcal{T}$.*

PROOF. Intuitively, Algorithm 1 decides robustness by checking whether a multiversion split schedule s for $\mathcal{T}$ and $\mathcal{A}$ exists. We first argue that the algorithm is correct, followed by a complexity analysis.

Correctness. Assume $\mathcal{T}$ is not robust against $\mathcal{A}$. Then, by Theorem 3.2, there exists a multiversion split schedule s for $\mathcal{T}$ and $\mathcal{A}$ based on some

$$C = (T'_1, b'_1, a'_2, T'_2), (T'_2, b'_2, a'_3, T'_3), \ldots, (T'_{m-1}, b'_{m-1}, a'_m, T'_m), (T'_m, b'_m, a'_1, T'_1).$$

We argue that Algorithm 1 returns False. To this end, assume that variables T_1, T_2, T_m, b_1, a_1, a_2 and b_m in Algorithm 1 are instantiated by respectively T'_1, T'_2, T'_m, b'_1, a'_1, a'_2 and b'_m. The conditions

Algorithm 1: Deciding robustness against an allocation.

```
Input  : Set of transactions $\mathcal{T}$ and allocation $\mathcal{A}$ for $\mathcal{T}$
Output : $True$ iff $\mathcal{T}$ is robust against $\mathcal{A}$
def reachable($T_2, T_m, T_1$):
    if $T_2 = T_m$ then
        return True;
    for $b_2 \in T_2, a_m \in T_m$ do
        if $b_2$ conflicts with $a_m$ then
            return True;
    $G$ := mixed-iso-graph($T_1, \mathcal{T} \setminus \{T_1, T_2, T_m\}$);
    $TC$ := reflexive-transitive-closure of $G$;
    for $(T_3, T_{m-1})$ in $TC$ do
        for $b_2 \in T_2, a_3 \in T_3, b_{m-1} \in T_{m-1}, a_m \in T_m$ do
            if ($b_2$ conflicts with $a_3$ and $b_{m-1}$ conflicts with $a_m$) then
                return True;
    return False;
def ww-conflict-free($b_1, T_1, T_2, T_m$):
    for $c_1 \in T_1$ do
        if $c_1 \in \text{prefix}_{b_1}(T_1)$ or $\mathcal{A}(T_1) \in \{SI, SSI\}$ then
            for $c_2 \in T_2$ do
                if $c_1$ is ww-conflicting with $c_2$ then
                    return False;
            for $c_m \in T_m$ do
                if $c_1$ is ww-conflicting with $c_m$ then
                    return False;
    return True;
def wr-conflict-free($T_i, T_j$):
    for $b_i \in T_i, a_j \in T_j$ do
        if $b_i$ is wr-conflicting with $a_j$ then
            return False;
    return True;
for $T_1 \in \mathcal{T}, T_2 \in \mathcal{T} \setminus \{T_1\}, T_m \in \mathcal{T} \setminus \{T_1\}$ do
    if reachable($T_2, T_m, T_1$) and ($\mathcal{A}(T_1) \neq SSI$ or $\mathcal{A}(T_2) \neq SSI$ or $\mathcal{A}(T_m) \neq SSI$) and
      ($\mathcal{A}(T_1) \neq SSI$ or $\mathcal{A}(T_2) \neq SSI$ or wr-conflict-free($T_1, T_2$)) and
      ($\mathcal{A}(T_1) \neq SSI$ or $\mathcal{A}(T_m) \neq SSI$ or wr-conflict-free($T_m, T_1$)) then
        for $b_1 \in T_1, a_1 \in T_1, a_2 \in T_2, b_m \in T_m$ do
            if ww-conflict-free($b_1, T_1, T_2, T_m$) and $b_m$ conflicts with $a_1$ and
              $b_1$ is rw-conflicting with $a_2$ and ($b_m$ is rw-conflicting with $a_1$ or
              ($\mathcal{A}(T_1) = RC$ and $b_1 <_{T_1} a_1$)) then
                return False;
return True
```

in Algorithm 1 necessary for the algorithm to return False are now implied by Definition 3.1. Note in particular that *reachable*(T'_2, T'_m, T'_1) returns True when variables T_3, T_{m-1}, b_2, a_3, b_{m-1} and a_m

are instantiated by respectively T'_3, T'_{m-1}, b'_2, a'_3 and b'_{m-1}. Indeed, by Definition 3.1, transactions $T'_3, \dots, T_{m-1}$ have no operation conflicting with an operation in T'_1 and C therefore implies a path from T'_3 to T'_{m-1} in mixed-iso-graph$(T'_1, \mathcal{T} \setminus \{T'_1, T'_2, T'_m\})$.

It remains to argue that Algorithm 1 returns True if $\mathcal{T}$ is robust against $\mathcal{A}$. Assume towards a contradiction that Algorithm 1 returns False, witnessed by transactions T_1, T_2 and T_m, and operations $b_1, a_1 \in T_1$, $a_2 \in T_2$ and $b_m \in T_m$. Notice furthermore that there are different options for *reachable*(T_2, T_m, T_1) to return True. For each such option, we construct a sequence of conflicting quadruples C. First, if $T_2 = T_m$, we take

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

Notice in particular that $b_m \in T_2$ since $T_2 = T_m$. Otherwise, if T_2 has an operation b_2 conflicting with an operation a_m in T_m, we take

$$C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_m, T_m), (T_m, b_m, a_1, T_1).$$

Lastly, if *reachable*(T_2, T_m, T_1) returns True because of a path in TC, let T_3, T_{m-1}, $b_2 \in T_2$, $a_3 \in T_3$, $b_{m-1} \in T_{m-1}$ and $a_m \in T_m$ be the transactions and operations witnessing this, and let

$$C' = (T_3, b_3, a_4, T_4), (T_4, b_4, a_5, T_5), \dots, (T_{m-2}, b_{m-2}, a_{m-1}, T_{m-1})$$

be the sequence of conflicting quadruples witnessing the path from transaction T_3 to transaction T_{m-1} in mixed-iso-graph$(T_1, \mathcal{T} \setminus \{T_1, T_2, T_m\})$, where we assume C' to be the empty sequence if $T_3 = T_{m-1}$. We then take

$$C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), C', (T_{m-1}, b_{m-1}, a_m, T_m), (T_m, b_m, a_1, T_1).$$

Because of the conditions in Algorithm 1 that must hold for the algorithm to return False, the multiversion split schedule s for $\mathcal{T}$ and $\mathcal{A}$ based on C is indeed a valid multiversion split schedule (cf. Definition 3.1). According to Theorem 3.2, $\mathcal{T}$ is therefore not robust against $\mathcal{A}$, leading to the desired contradiction.

Complexity. Let k be the total number of operations in $\mathcal{T}$ and ℓ the maximal number of operations in a transaction in $\mathcal{T}$. The outer loop (line 29) of Algorithm 1 iterates over all possible triples of transactions T_1, T_2 and T_m in $\mathcal{T}$, so there are at most $|\mathcal{T}|^3$ iterations.

For each such triple of transactions, a number of conditions is verified, which involve executing functions *reachable*(T_2, T_m, T_1), *wr-conflict-free*(T_1, T_2) and *wr-conflict-free*(T_m, T_1). The first function consists of a number of steps. First, testing whether T_2 and T_m are either the same transaction or have a pair of conflicting operations can be done in time $O(\ell^2)$. Next, To construct G, we must verify for each transaction in $\mathcal{T} \setminus \{T_1, T_2, T_m\}$ whether it has no operation conflicting with an operation in T_1 before adding it as a node in G, and add edges between two nodes if there is a pair of conflicting operations. Both of these steps can be done in time $O(k^2)$. Computing the reflexive-transitive-closure TC over G can be achieved in time $O(|\mathcal{T}|^3)$ by application of the Floyd-Warshall algorithm. Finally, the function *reachable*(T_2, T_m, T_1) verifies whether a pair of transactions T_3 and T_{m-1} exists having operations conflicting with an operation in respectively T_2 and T_m, which can be done in time $O(k^2 \cdot \ell^2)$. Since function *wr-conflict-free* involves testing for a wr-conflict between two transactions, this function runs in time $O(\ell^2)$.

Lastly, the inner loop (line 33) of Algorithm 1 iterates over all operations $b_1 \in \mathcal{T}$, $a_1 \in T_1$, $a_2 \in T_2$ and $b_m \in T_m$ and verifies a number of tests, which includes testing whether the function *ww-conflict-free*(b_1, T_1, T_2, T_m) returns True. Since this function runs in time $O(\ell^2)$, the total time complexity of this inner loop is $O(\ell^6)$. Combining all of the results obtained above, we conclude that Algorithm 1 indeed runs in time $O(|\mathcal{T}|^3 \cdot (|\mathcal{T}|^3 + k^2\ell^2 + \ell^6))$. □

4 THE ALLOCATION PROBLEM

Finding a robust allocation over {RC, SI, SSI} is of course trivial as we can simply assign every transaction to SSI. Such an allocation is undesirable as it enforces the most expensive concurrency control mechanism on all transactions. We are therefore interested in robust allocations that favor RC over SI and SI over SSI.

In the following, we assume a total order[4] RC < SI < SSI over the isolation levels, and introduce the following notions. Let $\mathcal{T}$ be a set of transactions, and let $\mathcal{A}$ and $\mathcal{A}'$ be allocations over $\mathcal{T}$. We denote by $\mathcal{A} \leq \mathcal{A}'$ when $\mathcal{A}(T) \leq \mathcal{A}'(T)$ for all $T \in \mathcal{T}$. Furthermore, $\mathcal{A} < \mathcal{A}'$ when $\mathcal{A} \leq \mathcal{A}'$ and there is a $T \in \mathcal{T}$ with $\mathcal{A}(T) < \mathcal{A}'(T)$.

We say that a robust allocation $\mathcal{A}$ is *optimal* when there is no robust allocation $\mathcal{A}'$ with $\mathcal{A}' < \mathcal{A}$. For an isolation level I, we denote by $\mathcal{A}[T \mapsto I]$ the allocation where T is assigned I and every other transaction $T' \in \mathcal{T}$ is assigned $\mathcal{A}(T')$. For two isolation levels $\mathcal{I}$ and $\mathcal{I}'$ with $\mathcal{I} < \mathcal{I}'$ (respectively $\mathcal{I}' < \mathcal{I}$) we say that $\mathcal{I}$ is a lower (respectively higher) isolation level than $\mathcal{I}'$.

The following proposition obtains some useful properties of robust allocations. Specifically, it says that robustness propagates upwards. That is, if a schedule is robust under an allocation $\mathcal{A}$, it remains robust when assigning a higher isolation level to any of its transactions T. Furthermore, if there exists a robust allocation $\mathcal{A}'$ mapping T to a lower isolation level than $\mathcal{A}(T)$, then $\mathcal{A}(T)$ can be safely updated to that lower isolation as well. That is, s is also robust under $\mathcal{A}[T \mapsto \mathcal{A}'(T)]$.

Proposition 4.1. *Let $\mathcal{T}$ be a set of transactions. Let $\mathcal{A}$ and $\mathcal{A}'$ be allocations for $\mathcal{T}$.*

(1) If $\mathcal{A} \leq \mathcal{A}'$ and $\mathcal{T}$ is robust against $\mathcal{A}$, then $\mathcal{T}$ is robust against $\mathcal{A}'$.
(2) If $\mathcal{T}$ is robust against $\mathcal{A}$ and $\mathcal{A}'$, then $\mathcal{T}$ is robust against $\mathcal{A}'[T \mapsto \mathcal{A}(T)]$ for every $T \in \mathcal{T}$.

Proof. *(1)* The proof is by contraposition. Assume $\mathcal{A} \leq \mathcal{A}'$ and $\mathcal{T}$ is not robust against $\mathcal{A}'$. We argue that $\mathcal{T}$ is not robust against $\mathcal{A}$. According to Theorem 3.2, there is a multiversion split schedule s for $\mathcal{T}$ and $\mathcal{A}'$ based on some sequence of conflicting quadruples C. Note that since $\mathcal{A} \leq \mathcal{A}'$, we have

- $\mathcal{A}'(T) = \text{RC} \Rightarrow \mathcal{A}(T) = \text{RC}$;
- $\mathcal{A}(T) \in \{\text{SI}, \text{SSI}\} \Rightarrow \mathcal{A}'(T) \in \{\text{SI}, \text{SSI}\}$; and
- $\mathcal{A}(T) = \text{SSI} \Rightarrow \mathcal{A}'(T) = \text{SSI}$

for all $T \in \mathcal{T}$. We can therefore conclude that, if the conditions in Definition 3.1 hold for $\mathcal{A}'$, then they must hold for $\mathcal{A}$ as well:

- Condition 3 in Definition 3.1 is immediate if $\mathcal{A}(T_1) = \text{RC}$ and follows from the second bullet above if $\mathcal{A}(T_1) \in \{\text{SI}, \text{SSI}\}$;
- Condition 5 in Definition 3.1 follows from the first bullet above;
- Conditions 6, 7 and 8 in Definition 3.1 follow from the third bullet above; and
- the remaining conditions in Definition 3.1 are trivial, as they do not refer to $\mathcal{A}$.

Hence, s is a multiversion split schedule for $\mathcal{T}$ and $\mathcal{A}$ based on C. By Theorem 3.2, $\mathcal{T}$ is not robust against $\mathcal{A}$.

(2) The proof is by contraposition. Assume $\mathcal{T}$ is not robust against $\mathcal{A}'' = \mathcal{A}'[T \mapsto \mathcal{A}(T)]$ for some $T \in \mathcal{T}$. We argue by case on T that $\mathcal{T}$ is not robust against $\mathcal{A}$ or $\mathcal{A}'$. According to Theorem 3.2, there is a multiversion split schedule s for $\mathcal{T}$ and $\mathcal{A}''$ based on some sequence of conflicting quadruples $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$. Without loss of generalization, we assume that the cycle in $SG(s)$ implied by C is a minimal cycle. In particular, we assume that

$$\text{there is a dependency from } T_2 \text{ to } T_1 \text{ or from } T_1 \text{ to } T_m \text{ in } s \text{ only if } T_2 = T_m. \qquad (\dagger)$$

[4]This order only represents the preference between isolation levels (i.e., RC over SI and SI over SSI), *not* an inclusion relation between isolation levels. For example, not every schedule allowed under $\mathcal{A}_{\text{SI}}$ is allowed under $\mathcal{A}_{\text{RC}}$ (cf. Example 5.2).

Indeed, if the cycle in $SG(s)$ implied by C is not minimal, then we could argue that a smaller subset of $\mathcal{T}$ is already not robust, thereby implying the existence of a multiversion split schedule for this subset.

If $T \notin \{T_1, T_2, T_m\}$, the schedule s is a valid multiversion split schedule for $\mathcal{T}$ and $\mathcal{A}'$ as well, since the conditions in Definition 3.1 only consider the allocated isolation level for transactions T_1, T_2 and T_m, and since $\mathcal{A}''(T) = \mathcal{A}'(T)$ for $T \in \{T_1, T_2, T_m\}$. By Theorem 3.2, $\mathcal{T}$ is therefore not robust against $\mathcal{A}'$.

Otherwise, if $T \in \{T_1, T_2, T_m\}$, assume towards a contradiction that $\mathcal{T}$ is robust against both $\mathcal{A}$ and $\mathcal{A}'$. Let $\mathcal{B}$, $\mathcal{B}'$ and $\mathcal{B}''$ be the allocations for $\mathcal{T}$ obtained by taking $\mathcal{B}(T_i) = \mathcal{A}(T_i)$, $\mathcal{B}'(T_i) = \mathcal{A}'(T_i)$ and $\mathcal{B}''(T_i) = \mathcal{A}''(T_i)$ if $T_i \in \{T_1, T_2, T_m\}$ and $\mathcal{B}(T_i) = \mathcal{B}'(T_i) = \mathcal{B}''(T_i) = \text{SSI}$ otherwise. By construction, $\mathcal{A} \leq \mathcal{B}$, $\mathcal{A}' \leq \mathcal{B}'$ and $\mathcal{A}'' \leq \mathcal{B}''$. By *(1)* we conclude that $\mathcal{T}$ is robust against $\mathcal{B}$ and $\mathcal{B}'$ as well. Notice furthermore that $\mathcal{B}'' = \mathcal{B}'[T \mapsto \mathcal{B}(T)]$ and that the schedule s is a valid multiversion split schedule for $\mathcal{T}$ and $\mathcal{B}''$ as well, since $\mathcal{B}''$ agrees with $\mathcal{A}''$ on T_1, T_2 and T_m.

Since $\mathcal{T}$ is robust against $\mathcal{B}$ and $\mathcal{B}'$ but not against $\mathcal{B}''$, it follows from *(1)* that $\mathcal{B} \nleq \mathcal{B}''$ and $\mathcal{B}' \nleq \mathcal{B}''$. By construction of $\mathcal{B}$, $\mathcal{B}'$ and $\mathcal{B}''$, in particular the fact that $\mathcal{B}'$ and $\mathcal{B}''$ only differ on T, we therefore conclude that $\mathcal{B}(T) < \mathcal{B}'(T)$ (as otherwise $\mathcal{B}' \leq \mathcal{B}''$). Furthermore, there exists a transaction $T' \in \{T_1, T_2, T_m\}$ with $T' \neq T$ and $\mathcal{B}'(T') < \mathcal{B}(T')$ (as otherwise $\mathcal{B} \leq \mathcal{B}''$). Since $\mathcal{B}(T) < \mathcal{B}'(T)$, we have $\mathcal{B}(T) \neq \text{SSI}$, and analogously $\mathcal{B}'(T') < \mathcal{B}(T')$ implies $\mathcal{B}'(T') \neq \text{SSI}$. In the remainder of the proof, we argue that for each choice of $T \in \{T_1, T_2, T_m\}$, $\mathcal{T}$ is not robust against either $\mathcal{B}$ or $\mathcal{B}'$, thereby leading to the desired contradiction.

If $T = T_1$, we have $\mathcal{B}''(T_1) = \mathcal{B}(T_1)$ and $\mathcal{B}(T_1) \neq \text{SSI}$. As a result, the schedule s is a valid multiversion split schedule for $\mathcal{T}$ and $\mathcal{B}$ as well, as it satisfies all conditions in Definition 3.1. By Theorem 3.2, $\mathcal{T}$ is therefore not robust against $\mathcal{B}$.

If $T = T_2$, we have $\mathcal{B}''(T_1) = \mathcal{B}'(T_1)$ and $T' \in \{T_1, T_m\}$, thereby implying that $\mathcal{B}'(T_1) \neq \text{SSI}$ or $\mathcal{B}'(T_m) \neq \text{SSI}$. We argue that s is a valid multiversion split schedule for $\mathcal{T}$ and $\mathcal{B}'$, thereby implying that $\mathcal{T}$ is not robust against $\mathcal{B}'$. Notice that Condition 7 of Definition 3.1 is the only one that is not immediate by the observations above. Towards a contradiction, assume it does not hold. That is, $\mathcal{B}'(T_1) = \text{SSI}$, $\mathcal{B}'(T_2) = \text{SSI}$ and there is an operation in T_1 wr-conflicting with an operation in T_2. By construction of s, this pair of conflicting operations implies a rw-antidependency from T_2 to T_1. According to (†), we have $T_2 = T_m$. But then we have $\mathcal{B}'(T_2) = \text{SSI}$ and $\mathcal{B}'(T_m) \neq \text{SSI}$, leading to the desired contradiction.

The case where $T = T_m$ is analogous to the previous case. We have $\mathcal{B}''(T_1) = \mathcal{B}'(T_1)$ and $T' \in \{T_1, T_2\}$, thereby implying that $\mathcal{B}'(T_1) \neq \text{SSI}$ or $\mathcal{B}'(T_2) \neq \text{SSI}$. Again, we argue that s is a valid multiversion split schedule for $\mathcal{T}$ and $\mathcal{B}'$, thereby implying that $\mathcal{T}$ is not robust against $\mathcal{B}'$. In this case, Condition 8 of Definition 3.1 is the only one that is not immediate by the observations above. Towards a contradiction, assume it does not hold. That is, $\mathcal{B}'(T_1) = \text{SSI}$, $\mathcal{B}'(T_m) = \text{SSI}$ and there is an operation in T_1 rw-conflicting with an operation in T_m. By construction of s, and since $\mathcal{A}''(T_1) = \mathcal{B}''(T_1) = \mathcal{B}'(T_1) = \text{SSI}$, this pair of conflicting operations implies a rw-antidependency from T_1 to T_m in s. According to (†), we have $T_2 = T_m$. The observation that $\mathcal{B}'(T_2) \neq \text{SSI}$ whereas $\mathcal{B}'(T_m) = \text{SSI}$ therefore leads to the desired contradiction again. □

We can now prove the following proposition.

Proposition 4.2. *There is a unique optimal allocation for every set of transactions $\mathcal{T}$.*

Proof. Suppose towards a contradiction that there are two different optimal robust allocations $\mathcal{A}$ and $\mathcal{A}'$. As $\mathcal{A}$ and $\mathcal{A}'$ are different, there exists a transaction $T \in \mathcal{T}$ such that $\mathcal{A}(T) \neq \mathcal{A}'(T)$. W.l.o.g., we assume $\mathcal{A}(T) < \mathcal{A}'(T)$. By Proposition 4.1(2), $\mathcal{T}$ is robust against $\mathcal{A}'[T \mapsto \mathcal{A}(T)]$.

```
Algorithm 2: Computing the optimal robust allocation.
   Input   : Set of transactions $\mathcal{T}$
   Output: Optimal robust allocation $\mathcal{A}$ for $\mathcal{T}$
1 $\mathcal{A} := \mathcal{A}_{\mathrm{SSI}}$;
2 for $T \in \mathcal{T}$ do
3     if $\mathcal{T}$ is robust against $\mathcal{A}[T \mapsto RC]$ then
4         $\mathcal{A} := \mathcal{A}[T \mapsto \mathrm{RC}]$;
5     else if $\mathcal{T}$ is robust against $\mathcal{A}[T \mapsto SI]$ then
6         $\mathcal{A} := \mathcal{A}[T \mapsto \mathrm{SI}]$;
7 return $\mathcal{A}$
```

But then $\mathcal{A}'[T \mapsto \mathcal{A}(T)] < \mathcal{A}'$, which means that $\mathcal{A}'$ is not optimal and leads to the desired contradiction. □

The following theorem shows that the unique optimal allocation can be computed in polynomial time. The corresponding algorithm is given as Algorithm 2.

Theorem 4.3. *An optimal robust allocation can be computed in time polynomial in the size of $\mathcal{T}$ for every set of transactions $\mathcal{T}$.*

Proof. By assumption $\mathcal{T}$ is robust against the allocation $\mathcal{A}_{\mathrm{SSI}}$ that maps all transactions to SSI. Algorithm 2 then improves this allocation towards the optimal one by iterating over each transaction in some fixed order, assigning the minimal isolation level to this transaction that still guarantees robustness. The correctness follows by repeated application of Proposition 4.1(2). It follows in particular from Proposition 4.1(2) that for every robust allocation $\mathcal{A}$ for $\mathcal{T}$ (including $\mathcal{A}_{\mathrm{SSI}}$) there is a sequence of allocations $\mathcal{A}_1 < \mathcal{A}_2 < \ldots < \mathcal{A}_k < \mathcal{A}$ with $\mathcal{A}_1$ denoting the unique optimal allocation for $\mathcal{T}$ and $\mathcal{A}_i = \mathcal{A}_{i+1}[T \rightarrow \mathcal{A}_i(T)]$ for every $i \in [1, k]$. The polynomial time complexity follows directly from Theorem 3.4. □

5 RESTRICTING TO RC AND SI

As already mentioned in the introduction, Oracle restricts to the isolation levels RC and SI. We investigate in this section how the results of the previous sections can be transferred to this setting. In particular, we ignore SSI and restrict attention to RC and SI.

We start with the following result.

Proposition 5.1. *For a set of transactions $\mathcal{T}$, robustness against $\mathcal{A}_{RC}$ implies robustness against $\mathcal{A}_{SI}$.*

This above result is an immediate consequence of Proposition 4.1(1). We mention that it is also a direct consequence of the characterizations for robustness against $\mathcal{A}_{\mathrm{RC}}$ [25] and $\mathcal{A}_{\mathrm{SI}}$ [19]. Indeed, it can be shown that a counterexample for robustness against $\mathcal{A}_{\mathrm{SI}}$ can always be transformed into a counterexample for robustness against $\mathcal{A}_{\mathrm{RC}}$ as well. We do want to emphasize that Proposition 5.1 is *not* a trivial consequence that immediately follows from the definitions of the isolation levels RC and SI, for the simple reason that it is not the case that every schedule allowed under $\mathcal{A}_{\mathrm{SI}}$ is also allowed under $\mathcal{A}_{\mathrm{RC}}$ as the next example shows.

Example 5.2. We give an example of a schedule s that is allowed under SI but not allowed under RC. To this end, consider the schedule s over transactions $\mathtt{W}_1[\mathtt{t}]\,\mathtt{C}_1$ and $\mathtt{R}_2[\mathtt{v}]\,\mathtt{R}_2[\mathtt{t}]\,\mathtt{C}_2$ with operation

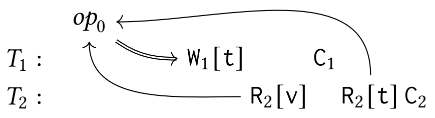

op_0
T_1 :
$\mathtt{W}_1[\mathtt{t}]$
$\mathtt{C}_1$
T_2 :
$\mathtt{R}_2[\mathtt{v}]$
$\mathtt{R}_2[\mathtt{t}]\,\mathtt{C}_2$

Fig. 7. Schematic representation of schedule s in Example 5.2. The version function v_s (single lines) and version order $\ll_s$ (double lines) are represented through arrows.

order $\leq_s$,

$$op_0\,\mathtt{W}_1[\mathtt{t}]\,\mathtt{R}_2[\mathtt{v}]\,\mathtt{C}_1\,\mathtt{R}_2[\mathtt{t}]\,\mathtt{C}_2,$$

version order $op_0 \ll_s \mathtt{W}_1[\mathtt{t}]$, and version function $v_s(\mathtt{R}_2[\mathtt{v}]) = v_s(\mathtt{R}_2[\mathtt{t}]) = op_0$. Figure 7 shows a graphical representation of schedule s. Then, s is allowed under $\mathcal{A}_{\text{SI}}$, but not under $\mathcal{A}_{\text{RC}}$, because $\mathtt{R}_2[\mathtt{t}]$ is not read-last-committed in s relative to itself. □

We formalize when a set of transactions is robustly allocatable against a class of isolation levels:

Definition 5.3. For a class of isolation levels $\mathcal{I}$, a set of transactions $\mathcal{T}$ is *robustly allocatable against* $\mathcal{I}$ if there exists an $\mathcal{I}$-allocation $\mathcal{A}$ such that $\mathcal{T}$ is robust against $\mathcal{A}$.

The only if-direction of the next theorem now immediately follows from Proposition 4.1(1) as $\mathcal{A} \leq \mathcal{A}_{\text{SI}}$ for any $\{\text{RC}, \text{SI}\}$-allocation $\mathcal{A}$ for which a set of transactions is robustly allocatable. The if-direction is trivial, since robustness against $\mathcal{A}_{\text{SI}}$ is an immediate witness for $\mathcal{T}$ being robustly allocatable against $\{\text{RC}, \text{SI}\}$:

Proposition 5.4. *A set of transactions $\mathcal{T}$ is robustly allocatable against $\{RC, SI\}$ iff $\mathcal{T}$ is robust against $\mathcal{A}_{SI}$.*

We now state and prove the main result of this section:

Theorem 5.5. *Let $\mathcal{T}$ be a set of transactions. It can be decided in time polynomial in the size of $\mathcal{T}$ whether $\mathcal{T}$ is robustly allocatable against $\{RC, SI\}$. If $\mathcal{T}$ is robustly allocatable against $\{RC, SI\}$, then an optimal unique allocation can be computed in polymonial time as well.*

Proof. From Proposition 5.4, it suffices to verify whether $\mathcal{T}$ is robust against $\mathcal{A}_{\text{SI}}$, which can be decided in PTIME (Theorem 3.4). Furthermore, an optimal robust $\{\text{RC}, \text{SI}\}$-allocation can be computed by adapting Algorithm 2 to start from $\mathcal{A}_{\text{SI}}$. □

6 RELATED WORK

6.1 Mixing isolation levels.

Adya et al. [2] define isolation levels in terms of phenomena that are forbidden to occur in the serialization graph. Mixed isolation levels are defined in terms of properties of the mixed serialization graph. In particular, a given schedule s is allowed under a mixed allocation if the mixed serialization graph $MSG(s)$ is acyclic. This graph $MSG(s)$ is a subset of $SeG(s)$ where dependency edges $T_i \to T_j$ are only added when relevant for the specified isolation level for T_i and T_j. Adya et al. [2] consider a mixture of READ UNCOMMITTED, READ COMMITTED and serializable transactions and do not consider SI or SSI like we do in this paper.[5] Other work [13, 19] that is discussed further below, consider a limited form of isolation level mixing where one isolation level (say, SI) can be mixed with a conflict-serializable isolation level. *To the best of our knowledge, this paper is the first that jointly considers mixing RC, SI and SSI in the way that it is applied in PostgreSQL.*

[5]A separate graph-based definition of SI is specified in [1], but this definition requires an extension of the serialization graph and incorporating SI in these mixed isolation levels is therefore not trivial.

6.2 Robustness and allocation for transactions.

Fekete [19] is the first work that provides a necessary and sufficient condition for deciding robustness against an isolation level (SI) for a workload of transactions. In particular, that work provides a characterization for optimal allocations when every transaction runs under either SNAPSHOT ISOLATION or strict two-phase locking (S2PL). As a side result, this work presents a characterization for robustness against SNAPSHOT ISOLATION as well. Ketsman et al. [22] provide characterisations for robustness against READ COMMITTED and READ UNCOMMITTED under lock-based semantics. In addition, it is shown that the corresponding decision problems are complete for CONP and LOGSPACE, respectively, which should be contrasted with the polynomial time characterization obtained in [25] for robustness against *multiversion* read committed which is the variant that is considered in this paper. *The present paper is therefore the first to address the robustness and allocation problem for a wider range of isolation levels.*

6.3 Robustness in practice.

The setting in the present paper assumes that the complete set of all transactions in a workload is completely known which is an assumption that can not always be met in practice. We next discuss two complementary approaches that have been previously investigated to adress this.

6.3.1 Transaction templates. In [25] it is assumed that transactions can only be generated through an API consisting of a fixed set of transaction programs. For instance, the TPC-C benchmark [24] consists of five different transaction programs, from which an infinite number of concrete transactions can be instantiated. A finite set of transaction templates, like TPC-C, is robust against an isolation level iff every set of transactions that can be instantiated from these programs, is robust against that isolation level. In [25], a PTIME decision procedure is obtained for robustness against RC for templates without functional constraints and [26] improves that result to NLOGSPACE. The work in [26] also considers robustness against RC in the presence of functional constraints. In addition, in [25] an experimental study was performed showing how an approach based on robustness and making transactions robust through promotion can improve transaction throughput. Interestingly, the characterisations for robustness on the level of templates in [25, 26] is directly based on the corresponding characterisations for robustness on the level of transactions. *In this sense, the results of this paper on the level of transactions can be a stepping stone for corresponding results on the level of transaction templates.*

6.3.2 Transaction Programs. A drawback of the formalisation for transaction templates is that it can not be extended to take updates to key attributes or predicate reads into account. Nevertheless, characterisations for robustness on the level of transactions can still be used to derive sufficient conditions for robustness on the level of arbitrary transaction programs (as written in SQL, say). Previous work on static robustness testing [6, 20] for transaction programs is based on the following key insight: when a *schedule* is not conflict-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). That insight is extended to a workload of *transaction programs* through the construction of 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 then guarantees robustness while the presence of a cycle does not necessarily imply non-robustness. *We observe that the sufficient conditions in these approaches are inspired by characterisations on the level of transactions. In this way, the characterisations presented in* *this paper could pave the way for sufficient conditions on the level of transaction programs as discussed above.*

Other work studies robustness within a framework for uniformly specifying different isolation levels in a declarative way [13, 16–18]. A key assumption here is *atomic visibility* requiring that either all or none of the updates of each transaction are visible to other transactions. These approaches aim at higher isolation levels and cannot be used for RC, as RC does not admit *atomic visibility.* Bernardi and Gotsman [13] furthermore provide a limited form of isolation level mixing. In brief, only one lower isolation level can be chosen (e.g., SI), but a subset of the considered transactions can be marked as serializable. Then, a schedule must adhere to all constraints implied by the lower isolation level and, additionally, a total order over the serializable transactions must exist. *This should be contrasted with our approach, where we allow multiple lower isolation levels to be combined with serializability at the same time.*

6.3.3 Other approaches. Gan et al. [21] present IsoDiff, a tool to detect and resolve potential anomalies caused by executing transactions under READ COMMITTED or SI. IsoDiff derives potential transactions from a database SQL trace and, based on this trace, decides whether cycles with a dangerous structure (for SI) or counterflow edge (for RC) can exist. By including additional timing constraints and correlation constraints, they are able to reduce the number of false positives. A potential pitfall of analyzing a trace is that it may overlook transactions that are rarely executed, thereby incorrectly considering an application to be robust. A subtle difference compared to our work is that the timing constraints proposed as part of IsoDiff assume that a dependency $b_i \rightarrow_s a_j$ always implies that operation b_i occurs before a_j in s, thereby implicitly assuming a single version implementation of RC, rather than the multiversion RC as discussed in this paper. In particular, (multiversion) RC allows for situations where b_i occurs after a_j in s, if $b_i \rightarrow_s a_j$ is a rw-antidependency. Orthogonal to robustness detection, tools such as Elle [8] aim at detecting anomalies that should not occur under a given isolation level. These tools can be used to detect whether a database system implements the declared isolation levels correctly, whereas robustness assumes that the isolation level is implemented correctly to decide whether every possible execution of a given workload is conflict-serializable. *None of the above mentioned works considers robustness in the context of mixed allocations of isolation levels.*

7 CONCLUSION

In this paper, we addressed and solved the robustness and allocation problem for the classes {RC, SI, SSI} and {RC, SI} corresponding to the isolation levels employed in PostgreSQL and Oracle, respectively. As discussed in Section 6.3, these results can be used as a stepping stone for corresponding results on the level of transaction templates and transaction programs, respectively, thereby laying the groundwork for automating isolation level allocation within existing databases that support multiversion concurrency control.

ACKNOWLEDGMENTS

This work is funded by FWO-grant G019921N. We thank Alan Fekete for spotting the omission related to read-only transactions in our original definition of dangerous structures.

REFERENCES

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. MIT, Cambridge, MA, USA.

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

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

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

[5] 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.

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

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

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

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

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

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

[12] 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.

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

[14] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD*. 729–738.

[15] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.* 34, 4 (2009), 20:1–20:42.

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

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

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

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

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

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

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

[23] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *PVLDB* 5, 12 (2012), 1850–1861.

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

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

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

[27] Brecht Vandevoort, Bas Ketsman, and Frank Neven. 2023. Allocating Isolation Levels to Transactions in a Multiversion Setting. In *PODS*. 69–78.