Allocating Isolation Levels to Transactions in a Multiversion Setting

DOI: 10.1145/3584372.3588672
Handle: http://hdl.handle.net/1942/42231

Allocating Isolation Levels to Transactions in a Multiversion Setting

Brecht Vandevoort
UHasselt, Data Science Institute, ACSL
Diepenbeek, Belgium

Bas Ketsman
Vrije Universiteit Brussel
Brussels, Belgium

Frank Neven
UHasselt, Data Science Institute, ACSL
Diepenbeek, Belgium

ABSTRACT

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 Postgres 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**.

KEYWORDS

concurrency control, robustness, complexity

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 transactions in $\mathcal{T}$ that is allowed under the specified isolation level is serializable. There is a famous example that is part of database folklore: the TPC-C benchmark [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 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 Postgres, 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, 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 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 3.3 that a counterexample split schedule can still be efficiently constructed.

The contributions of this paper can be summarized as follows:

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

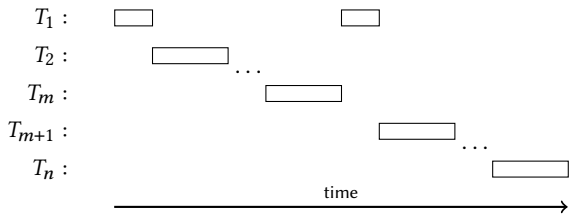

Figure 1: Abstract representation of a multi-version split schedule where T_1 is the splitted transaction.

(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 2.4). Even though these definitions are an abstraction, they are consistent with mixed allocations as they are applied within Postgress 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.

2 DEFINITIONS

2.1 Transactions and Schedules

We fix an infinite set of objects **Obj**. For an object $\mathtt{t} \in$ **Obj**, we denote by `R[t]` a *read* operation on `t` and by `W[t]` a *write* operation on `t`. We also assume a special *commit* operation denoted by `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 $first(T)$ 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 $\mathtt{W}[\mathtt{t}]$ and $\mathtt{R}[\mathtt{t}]$ 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. 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 $b_i \rightarrow_{s'} a_j$.

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 $b_i \rightarrow_s a_j$. Since we are usually not only interested in the existence of dependencies between operations, but also in the operations themselves, we assume the existence of a labeling function λ mapping each edge to a set of pairs of operations. Formally, $(b_i, a_j) \in \lambda(T_i, T_j)$ iff there is an operation $a_j \in T_j$ that depends on an operation $b_i \in T_i$. For ease of notation, we choose to represent $SeG(s)$ as a set of quadruples (T_i, b_i, a_j, T_j) denoting all possible pairs of these transactions T_i and T_j with all possible choices of operations with $b_i \rightarrow_s a_j$. Henceforth, we refer to these quadruples simply as edges. Notice that edges cannot contain commit operations.

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

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

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

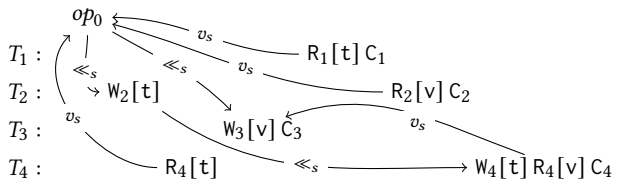

Figure 2: A schedule s with v_s and $\ll_s$ represented through arrows.

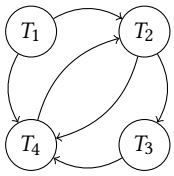

Figure 3: Serialization graph $SeG(s)$ for the schedule s presented in Figure 2.

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 2.5 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 $\mathit{first}(T_i) <_s \mathtt{C}_j$ and $\mathit{first}(T_j) <_s \mathtt{C}_i$. 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 T_j is installed after all versions of $\mathtt{t}$ installed by transactions committing before T_j commits, but before all versions of $\mathtt{t}$ installed by transactions committing after T_j commits. More formally, 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 $\mathit{first}(T)$. 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 $\mathtt{t}$ (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 $\mathit{first}(T_j) <_s \mathtt{C}_i$. 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 $\mathtt{t}$, 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 $\mathtt{t}$.

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.

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 $\mathit{first}(T_i)$; 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. Postgres, 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* $T_1 \to T_2 \to T_3$ 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 ; and,
- $\mathtt{C}_3 \leq_s \mathtt{C}_1$ and $\mathtt{C}_3 <_s \mathtt{C}_2$.

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 such a structure can only lead to non-serializable schedules if T_3 commits first, and actual implementations of SSI (e.g., Postgres [23]) therefore include this optimization when monitoring for dangerous structures to reduce the number of aborts due to false positives.

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

Definition 2.4. 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;
- for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) \in \{\text{SI}, \text{SSI}\}$, T_i is allowed under SI; and
- there is no dangerous structure $T_i \to T_j \to T_k$ 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 2.5. 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 $T1 \to T2 \to T3$ 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

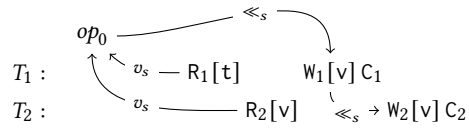

Figure 4: Schematic representation of schedule s in Example 2.6.

to be allowed under a mixed allocation. Mixing isolation levels in this way does introduce some subtleties as the next example shows.

Example 2.6. Consider the schedule s in Figure 4 over two concurrent transactions T_1 and T_2, where both T_1 and T_2 write to object 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 $\mathcal{A}_3(T_1) = \text{SI}$ 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 Postgres.[2] □

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 2.7 (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.

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

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

multiversion schedule that has the following form:

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

where

(1) there is no 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 $\mathsf{prefix}_{b_1}(T_1)$ 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 $\mathsf{postfix}_{b_1}(T_1)$ ww-conflicting with a write operation in T_2 or T_m;
(4) b_1 is rw-conflicting with a_2;
(5) b_m is rw-conflicting with a_1 or ($\mathcal{A}(T_1) = \text{RC}$ and $b_1 <_{T_1} a_1$);
(6) $\mathcal{A}(T_1) \neq \text{SSI}$ or $\mathcal{A}(T_2) \neq \text{SSI}$ or $\mathcal{A}(T_m) \neq \text{SSI}$;
(7) if $\mathcal{A}(T_1) = \text{SSI}$ and $\mathcal{A}(T_2) = \text{SSI}$, then there is no operation in T_1 wr-conflicting with an operation in T_2; and
(8) if $\mathcal{A}(T_1) = \text{SSI}$ and $\mathcal{A}(T_m) = \text{SSI}$, then there is no operation in T_1 rw-conflicting with an operation in T_m.

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

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 $b_1 \to a_2$ and $b_m \to a_1$ 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 C.

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

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[3] 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 $T \in \mathcal{T}$.

We can now prove the following proposition.

[3]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).

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*

Algorithm 2: Computing the optimal robust allocation.

Input : Set of transactions $\mathcal{T}$
Output: Optimal robust allocation $\mathcal{A}$ for $\mathcal{T}$

$\mathcal{A} := \mathcal{A}_{\text{SSI}}$;
for $T \in \mathcal{T}$ **do**
  if $\mathcal{T}$ *is robust against* $\mathcal{A}[T \mapsto RC]$ **then**
    $\mathcal{A} := \mathcal{A}[T \mapsto \text{RC}]$;
  else if $\mathcal{T}$ *is robust against* $\mathcal{A}[T \mapsto SI]$ **then**
    $\mathcal{A} := \mathcal{A}[T \mapsto \text{SI}]$;
return $\mathcal{A}$

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 $\mathcal{A}'[T \mapsto \mathcal{A}(T)]$. 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 $\mathcal{A}_{\text{SSI}}$ that maps all transactions to SSI. Algorithm 2 then refines this allocation by assigning the minimal isolation level to each transaction leading to an optimal robust allocation. 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 $\mathcal{A}_{\text{SSI}}$) 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 $\mathcal{A}_i = \mathcal{A}_{i+1}[T \to \mathcal{A}_i(T)]$ for every $i \in [1, k]$. The polynomial time complexity follows directly from Theorem 3.3. □

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 5.4. We mention that it is also a direct consequence of the characterizations for robustness against $\mathcal{A}_{\text{RC}}$ [25] and $\mathcal{A}_{\text{SI}}$ [19]. Indeed, it can be shown that a counterexample for robustness against $\mathcal{A}_{\text{SI}}$ can always be transformed into a counterexample for robustness against $\mathcal{A}_{\text{RC}}$ 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 $\mathcal{A}_{\text{SI}}$ is also allowed under $\mathcal{A}_{\text{RC}}$ as the next example shows.

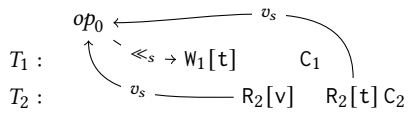

Figure 5: Schematic representation of schedule s in Example 5.2.

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 $W_1[t]\,C_1$ and $R_2[v]\,R_2[t]\,C_2$ with operation order $\leq_s$,

$$op_0\,W_1[t]\,R_2[v]\,C_1\,R_2[t]\,C_2,$$

version order $op_0 \ll_s W_1[t]$, and version function $v_s(R_2[v]) = v_s(R_2[t]) = op_0$. Figure 5 shows a graphical representation of schedule s. Then, s is allowed under $\mathcal{A}_{\text{SI}}$, but not under $\mathcal{A}_{\text{RC}}$, because $R_2[t]$ 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 {RC, 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 {RC, 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 proof 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 3.3). Furthermore, an optimal robust {RC, 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.[4] 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 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 Postgres.*

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.*

[4] 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.

6.3.2 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 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 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 Postgres 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.

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.