Allocating Isolation Levels to Transactions in a Multiversion Setting

Allocating Isolation Levels to Transactions in a Multiversion Setting

Brecht Vandevoort
UHasselt, Data Science Institute, ACSL
brecht.vandevoort@uhasselt.be

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

Frank Neven
UHasselt, Data Science Institute, ACSL
frank.neven@uhasselt.be

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.

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, $\{\text{RC}, \text{SI}, \text{SSI}\}$ is the class of isolation levels available in Postgres, while $\{\text{RC}, \text{SI}\}$ 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 $\{\text{RC}, \text{SI}, \text{SSI}\}$ an optimal robust allocation can always be found in polynomial time. As $\{\text{RC}, \text{SI}\}$ 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 $\{\text{RC}, \text{SI}\}$ 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 $\{\text{RC}, \text{SI}, \text{SSI}\}$ 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.

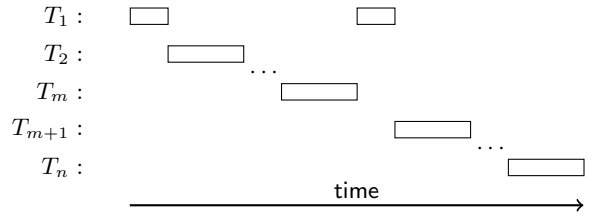

Figure 1: Abstract representation of a multi-version split schedule where T_1 is the splitted transaction.

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. $\{\text{RC}, \text{SI}, \text{SSI}\}$ (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 $\{\text{RC}, \text{SI}, \text{SSI}\}$ in terms of the existence of a multiversion split-schedule.
3. We provide a polynomial time decision procedure for robustness against an allocation over $\{\text{RC}, \text{SI}, \text{SSI}\}$.
4. We show that there is always a unique optimal robust allocation over $\{\text{RC}, \text{SI}, \text{SSI}\}$ 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 $\{\text{RC}, \text{SI}\}$ 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 $\{\text{RC}, \text{SI}, \text{SSI}\}$ in Section 3 and Section 4, respectively. We consider robustness and allocation for $\{\text{RC}, \text{SI}\}$ in Section 5. We discuss related work in Section 6. We conclude in Section 7.

This paper is a shortened version of the PODS 2023 paper [27] where the definition of a multiversion split schedule (Definition 3.1) has been simplified and a more elaborate running example has been added.

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

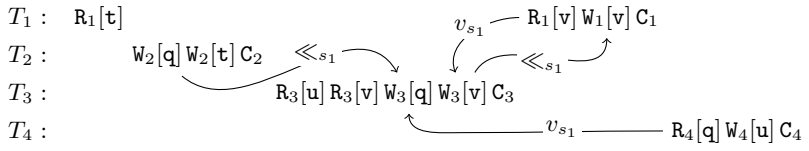

Figure 2: A single version schedule s_1 for $\mathcal{T}_{\text{ex}}$ with v_{s_1} and $\ll_{s_1}$ represented through arrows. The special operation op_0 and all arrows involving op_0 are omitted.

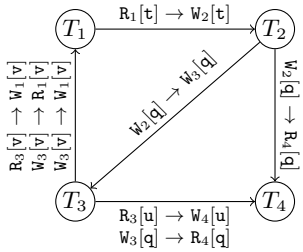

Figure 3: Serialization graph $SeG(s_1)$.

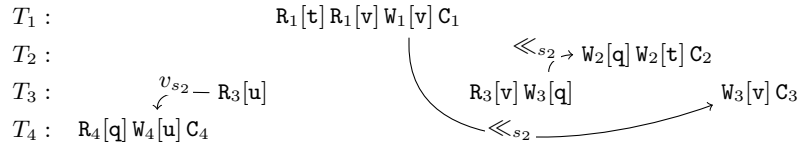

Figure 4: A schedule s_2 for $\mathcal{T}_{\text{ex}}$ with v_{s_2} and $\ll_{s_2}$ represented through arrows. The special operation op_0 and all arrows involving op_0 are omitted.

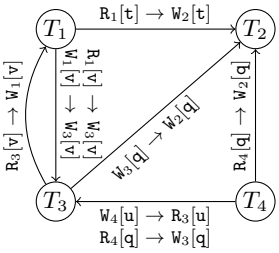

Figure 5: Serialization graph $SeG(s_2)$.

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 $\mathtt{R[t]}$ 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 *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.

As a running example, we define the set of transactions $\mathcal{T}_{\text{ex}} = \{T_1, T_2, T_3, T_4\}$ over four different objects $\mathtt{t}$, $\mathtt{u}$, $\mathtt{v}$, and $\mathtt{q}$ as follows:

- $T_1 = \mathtt{R_1[t]}\,\mathtt{R_1[v]}\,\mathtt{W_1[v]}\,\mathtt{C_1}$;
- $T_2 = \mathtt{W_2[q]}\,\mathtt{W_2[t]}\,\mathtt{C_2}$;
- $T_3 = \mathtt{R_3[u]}\,\mathtt{R_3[v]}\,\mathtt{W_3[q]}\,\mathtt{W_3[v]}\,\mathtt{C_3}$; and
- $T_4 = \mathtt{R_4[q]}\,\mathtt{W_4[u]}\,\mathtt{C_4}$.

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.

Continuing our running example, Figure 2 and Figure 4 illustrate two schedules s_1 and s_2 over $\mathcal{T}_{\text{ex}}$. The version order $\ll$ as well as the version function v are represented as arrows. The special operation op_0 together with its arrows, is omitted in these figures to improve readability. For example, $v_{s_1}(\mathtt{R_1[v]}) = \mathtt{W_3[v]}$ in schedule s_1, since $\mathtt{R_1[v]}$ reads the version of $\mathtt{v}$ written by $\mathtt{W_3[v]}$. In s_2 on the other hand, we

have $v_{s_2}(\mathtt{R}_1[\mathtt{v}]) = op_0$, since $\mathtt{R}_1[\mathtt{v}]$ reads the initial version of $\mathtt{v}$. Furthermore, the read operation $\mathtt{R}_3[\mathtt{v}]$ reads the initial version of $\mathtt{v}$ in schedule s_2 instead of the version written by T_1, even though T_1 commits before $\mathtt{R}_3[\mathtt{v}]$ in s_2.

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 $v_s(a) <_s c <_s a$. For example, s_1 in Figure 2 is a single version schedule, while s_2 in Figure 4 is not as $op_0 = v_{s_2}(\mathtt{R}_3[\mathtt{v}]) <_{s_2} \mathtt{W}_1[\mathtt{v}] <_{s_2} \mathtt{R}_3[\mathtt{v}]$. 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

Two operations a_j and b_i from different transactions T_j and T_i in a set of transactions $\mathcal{T}$ are *conflicting* if they are on the same object $\mathtt{t}$ and at least one of them is a write. In this case, we furthermore say that b_i is *ww-conflicting* (respectively, *wr-conflicting*) with a_j if b_i is a write operation and a_j is a write operation (respectively, a read operation); and b_i is *rw-conflicting* with a_j if b_i is a read operation and a_j is a write operation. 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. An 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 $\mathtt{W}_3[\mathtt{v}] \rightarrow_{s_1} \mathtt{W}_1[\mathtt{v}]$, $\mathtt{W}_3[\mathtt{v}] \rightarrow_{s_1} \mathtt{R}_1[\mathtt{v}]$ and $\mathtt{R}_3[\mathtt{v}] \rightarrow_{s_1} \mathtt{W}_1[\mathtt{v}]$ are respectively a ww-dependency, a wr-dependency and a rw-antidependency in schedule s_1 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 $b_i \rightarrow_s a_j$.

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

Figure 3 and 5 visualize the serialization graphs $SeG(s_1)$ and $SeG(s_2)$ for the schedules s_1 and s_2 in Figure 2 and 4, respectively. For illustration purposes, each edge is labelled with the corresponding dependencies. Since $SeG(s_1)$ and $SeG(s_2)$ are not acyclic, both schedules are 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.

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 $\mathit{first}(T_j) <_s \mathtt{C}_i$. In our running example schedule s_1 in Figure 2, T_1 and T_2 are concurrent, as well as T_1 and T_3. All other pairs of transactions are not concurrent in s_1. 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 $\mathtt{C}_j <_s \mathtt{C}_i$. All write operations in Figure 2 respect the commit order of s_1. In Figure 4, write operations $\mathtt{W}_3[\mathtt{q}]$ and $\mathtt{W}_2[\mathtt{q}]$ do not respect the commit order of s_2 as $\mathtt{C}_2 <_{s_2} \mathtt{C}_3$ but $\mathtt{W}_3[\mathtt{q}] \ll_{s_2} \mathtt{W}_2[\mathtt{q}]$.

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 $\ll_s$). For example, $\mathtt{R}_1[\mathtt{v}]$ in Figure 2 is read-last-committed in s_1 relative to $\mathtt{R}_1[\mathtt{v}]$ but not

to *first*(T_1), whereas in Figure 4 read operation $\mathtt{R}_3[\mathtt{v}]$ is read-last-committed in s_2 relative to *first*(T_3) but not relative to $\mathtt{R}_3[\mathtt{v}]$.

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_1 in Figure 2 exhibits a concurrent write in s_1, since it writes to $\mathtt{v}$, which has been modified earlier by a concurrent transaction T_3. However, T_1 does not exhibit a dirty write in s_1, since T_3 has already committed before T_1 writes to $\mathtt{v}$. In Figure 4, T_2 exhibits a dirty write in s_2 since it writes to $\mathtt{q}$, which has been modified earlier by T_3 and T_3 has not yet committed before $\mathtt{W}_2[\mathtt{q}]$.

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:*

- *write operations in T_i respect 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:*

- *write operations in T_i respect 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. 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;
- $\mathtt{C}_3 \leq_s \mathtt{C}_1$ 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 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 $\mathcal{A}(T_i) = RC$, T_i is allowed under RC;*
- *for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) \in \{SI, SSI\}$, 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 $T_i, T_j, T_k \in \{T \in \mathcal{T} \mid \mathcal{A}(T) = SSI\}$.*

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

In our running example schedule s_1 in Figure 2, the first three transactions form a dangerous structure $T_3 \to T_1 \to T_2$, witnessed by the rw-antidependencies $\mathtt{R}_3[\mathtt{v}] \to_{s_1} \mathtt{W}_1[\mathtt{v}]$ and $\mathtt{R}_1[\mathtt{t}] \to_{s_1} \mathtt{W}_2[\mathtt{t}]$. Therefore, s_1 is not allowed under an allocation mapping all three transactions to SSI. Since $\mathtt{R}_1[\mathtt{v}]$ is read-last-committed relative to itself but not relative to the first operation of T_1, and since T_1 exhibits a concurrent write, T_1 is allowed under RC in s_1, but not allowed under SI. Transactions T_2, T_3 and T_4 are allowed under both RC and SI in s_1. Notice in particular that all read operations in T_3 and T_4 are read-last-committed relative to both themselves and the first operation of the corresponding transaction in s_1. We conclude that s_1 is allowed under all allocations over $\mathcal{T}_{\text{ex}}$ mapping T_1 to RC, and T_2, T_3 and T_4 to either RC, SI or SSI. For schedule s_2 in Figure 4, no allocation $\mathcal{A}$ over $\mathcal{T}_{\text{ex}}$ exists such that s_2 is allowed under $\mathcal{A}$. Indeed, not all write operations in T_2 and T_3 respect the commit order of s_2, and T_2 furthermore exhibits a dirty write, which is not allowed under any isolation level.

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.

	T_1	T_2	T_3	T_4	$\mathcal{T}_{\text{ex}}$ robust against $\mathcal{A}_i$?
$\mathcal{A}_1$	RC	RC	SSI	SSI	no (cf. s_1 in Figure 2)
$\mathcal{A}_2$	SSI	RC	SSI	SSI	yes
$\mathcal{A}_3$	SI	SI	SSI	SSI	yes
$\mathcal{A}_4$	SI	RC	SSI	SSI	yes
$\mathcal{A}_5$	SI	RC	SI	SSI	no (cf. s_3 in Figure 6)
$\mathcal{A}_6$	SI	RC	SSI	SI	no (cf. s_3 in Figure 6)

Table 1: Example allocations for $\mathcal{T}_{\text{ex}}$.

DEFINITION 2.5 (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.

Table 1 presents some allocations over our running example $\mathcal{T}_{\text{ex}}$. For each allocation that $\mathcal{T}_{\text{ex}}$ is not robust against, a counterexample schedule is given. For example, $\mathcal{T}$ is not robust against $\mathcal{A}_1$ because schedule s_1 in Figure 2 is allowed under $\mathcal{A}_1$ and not conflict-serializable.

3. DECIDING ROBUSTNESS

In the next definition, for an operation $b \in T$, we denote by $\mathsf{prefix}_b(T)$ the restriction of T to all operations that are before or equal to b according to $\leq_T$. Similarly, we denote by $\mathsf{postfix}_b(T)$ the restriction of T to all operations that are strictly after b according to $\leq_T$.

DEFINITION 3.1 (Multiversion split schedule). *Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions and $\mathcal{A}$ an allocation for $\mathcal{T}$. A* multiversion split schedule *s for $\mathcal{T}$ and $\mathcal{A}$ is a multiversion schedule allowed under $\mathcal{A}$ 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 $b_1 \in T_1$ and $m \in [2, n]$. Furthermore, for each pair of transactions $T_i, T_j \in \mathcal{T}$ with $i, j \in [1, m]$ and either $j = i+1$ or $i = m$ and $j = 1$, there are two operations $b_i \in T_i$ and $a_j \in T_j$ such that $b_i \rightarrow_s a_j$.

Schedule s_1 in Figure 3 is a multiversion split schedule for our running example set of transactions $\mathcal{T}_{\text{ex}}$ and allocation $\mathcal{A}_1$. Notice in particular the cyclic chain of dependencies implied by the definition. Because of this, such a schedule is never conflict-serializable. The existence of a multiversion split schedule for a set of transactions $\mathcal{T}$ and an allocation $\mathcal{A}$ for $\mathcal{T}$ therefore implies that $\mathcal{T}$ is not robust against $\mathcal{A}$. The next Theorem states that the opposite direction is also true, thereby characterizing non-robustness in terms of the existence of a multiversion split schedule.

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 $\mathcal{A}$.*

Theorem 3.2 forms the basis for a PTIME algorithm deciding robustness against a given allocation, presented as Algorithm 1 in [27]. We emphasize that a naive approach enumerating all multiversion split schedules for $\mathcal{T}$ and $\mathcal{A}$ does not work, as this number can still be exponential in the number of transactions in $\mathcal{T}$. Instead, the algorithm relies on a number of conditions on $\mathcal{A}$ as well as the the operations in $\mathcal{T}$ and makes use of an auxiliary graph structure to efficiently check whether a multiversion split schedule exists that witnesses the desired cyclic chain of dependencies and is allowed under $\mathcal{A}$. We refer to [27] for the details of the algorithm.

THEOREM 3.3 ([27]). *Algorithm 1 in [27] 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 $\{\text{RC}, \text{SI}, \text{SSI}\}$ 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[2] 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 $T \in \mathcal{T}$.*

Applying these results on the allocations in Table 1, $\mathcal{A}_4$ being a robust allocation implies that $\mathcal{A}_2$ and $\mathcal{A}_3$ are robust allocations as well. Furthermore, the robust allocations $\mathcal{A}_2$ and $\mathcal{A}_3$ can be combined into the robust allocation $\mathcal{A}_4$.

We can now prove the following proposition.

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

[2]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).

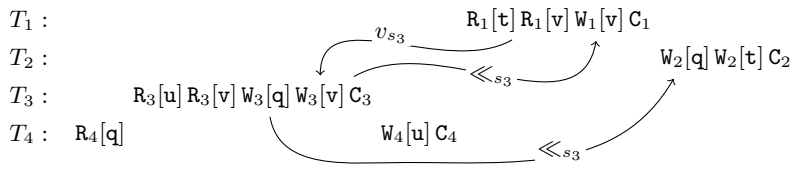

Figure 6: A schedule s_3 for $\mathcal{T}_{\text{ex}}$ with v_{s_3} and $\ll_{s_3}$ represented through arrows. The special operation op_0 and all arrows involving op_0 are omitted.

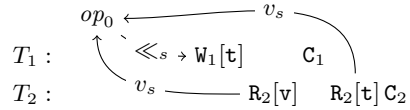

Figure 7: Schematic representation of schedule s in Example 5.2.

Algorithm 1: 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 $\mathcal{A}$

For our running example $\mathcal{T}_{\text{ex}}$, the optimal robust allocation is $\mathcal{A}_4$ in Table 1. Indeed, trying to lower the isolation level for one of the transactions even further always leads to nonrobust allocations (cf., $\mathcal{A}_1$, $\mathcal{A}_5$ and $\mathcal{A}_6$)

The following theorem shows that the unique optimal allocation can be computed in polynomial time. The corresponding algorithm is given as Algorithm 1.

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

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.

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

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

version order $op_0 \ll_s \mathtt{W_1[t]}$, *and version function* $v_s(\mathtt{R_2[v]}) = v_s(\mathtt{R_2[t]}) = op_0$. *Figure 7 shows a graphical representation of schedule s. Then, s is allowed under $\mathcal{A}_{SI}$, but not under $\mathcal{A}_{RC}$, because* $\mathtt{R_2[t]}$ *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 $\{\text{RC}, \text{SI}\}$-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 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.*

6. RELATED WORK

Mixing isolation levels. Adya et al. [2] define isolation levels in terms of phenomena that are forbidden to occur in the serialization graph. they consider a mixture of READ UNCOMMITTED, READ COMMITTED and serializable transactions and do not consider SI or SSI like we do in this paper. Note that a separate graph-based definition of SI is specified in [1], requiring an extension of the serialization graph. Incorporating SI in the mixed setting in [2] is therefore not trivial. Other work [13, 19] 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.*

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 pro-

vides a characterization for optimal allocations when every transaction runs under either SI or strict two-phase locking (S2PL). As a side result, this work presents a characterization for robustness against SI 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.*

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. In [27] we provide an elaborate discussion of different approaches that have been previously investigated to address this [6, 8, 13, 16–18, 20, 21, 25, 26].

7. CONCLUSION

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

Acknowledgments

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

8. REFERENCES

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

[2] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[3] M. Alomari. Serializable executions with snapshot isolation and two-phase locking: Revisited. In *AICCSA*, pages 1–8, 2013.

[4] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *ICDE*, pages 576–585, 2008.

[5] M. Alomari, M. J. Cahill, A. D. Fekete, and U. Röhm. Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels? In *DASFAA*, volume 4947, pages 267–281, 2008.

[6] M. Alomari and A. Fekete. Serializable use of read committed isolation level. In *AICCSA*, pages 1–8, 2015.

[7] M. Alomari, A. D. Fekete, and U. Röhm. A robust technique to ensure serializable executions with snapshot isolation DBMS. In *ICDE*, pages 341–352, 2009.

[8] P. Alvaro and K. Kingsbury. Elle: Inferring isolation anomalies from experimental observations. *PVLDB*, 14(3):268–280, 2020.

[9] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

[10] S. M. Beillahi, A. Bouajjani, and C. Enea. Checking robustness against snapshot isolation. In *CAV*, pages 286–304, 2019.

[11] S. M. Beillahi, A. Bouajjani, and C. Enea. Robustness against transactional causal consistency. In *CONCUR*, pages 1–18, 2019.

[12] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

[13] G. Bernardi and A. Gotsman. Robustness against consistency models with atomic visibility. In *CONCUR*, pages 7:1–7:15, 2016.

[14] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD*, pages 729–738, 2008.

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

[16] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*, pages 58–71, 2015.

[17] A. Cerone and A. Gotsman. Analysing snapshot isolation. *J.ACM*, 65(2):1–41, 2018.

[18] A. Cerone, A. Gotsman, and H. Yang. Algebraic Laws for Weak Consistency. In *CONCUR*, pages 26:1–26:18, 2017.

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

[20] A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. E. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[21] Y. Gan, X. Ren, D. Ripberger, S. Blanas, and Y. Wang. Isodiff: Debugging anomalies caused by weak isolation. *PVLDB*, 13(11):2773–2786, 2020.

[22] B. Ketsman, C. Koch, F. Neven, and B. Vandevoort. Deciding robustness for lower SQL isolation levels. In *PODS*, pages 315–330, 2020.

[23] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *PVLDB*, 5(12):1850–1861, 2012.

[24] TPC-C. On-line transaction processing benchmark. `http://www.tpc.org/tpcc/`.

[25] B. Vandevoort, B. Ketsman, C. Koch, and F. Neven. Robustness against read committed for transaction templates. *PVLDB*, 14(11):2141–2153, 2021.

[26] B. Vandevoort, B. Ketsman, C. Koch, and F. Neven. Robustness against read committed for transaction templates with functional constraints. In *ICDT*, pages 16:1–16:17, 2022.

[27] B. Vandevoort, B. Ketsman, and F. Neven. Allocating isolation levels to transactions in a multiversion setting. In *PODS*, pages 69–78, 2023.