When View- and Conflict-Robustness Coincide for Multiversion Concurrency Control

Peer-reviewed author version

VANDEVOORT, Brecht; KETSMAN, Bas & NEVEN, Frank (2024) When View- and Conflict-Robustness Coincide for Multiversion Concurrency Control. In: Proceedings of the ACM on Management of Data, 2 (2) (Art N° 91).

DOI: 10.1145/3651592
Handle: http://hdl.handle.net/1942/45427

When View- and Conflict-Robustness Coincide for Multiversion Concurrency Control

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

A DBMS allows trading consistency for efficiency through the allocation of isolation levels that are strictly weaker than serializability. The *robustness problem* 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 *safe*. In the literature, safe is interpreted as conflict-serializable (to which we refer here as conflict-robustness). In this paper, we study the view-robustness problem, interpreting safe as view-serializable. View-serializability is a more permissive notion that allows for a greater number of schedules to be serializable and aligns more closely with the intuitive understanding of what it means for a database to be consistent. However, view-serializability is more complex to analyze (e.g., conflict-serializability can be decided in polynomial time whereas deciding view-serializability is NP-complete). While conflict-robustness implies view-robustness, the converse does not hold in general. In this paper, we provide a sufficient condition for isolation levels guaranteeing that conflict- and view-robustness coincide and show that this condition is satisfied by the isolation levels occurring in Postgres and Oracle: read committed (RC), snapshot isolation (SI) and serializable snapshot isolation (SSI). It hence follows that for these systems, widening from conflict- to view-serializability does not allow for more sets of transactions to become robust. Interestingly, the complexity of deciding serializability within these isolation levels is still quite different. Indeed, deciding conflict-serializability for schedules allowed under RC and SI remains in polynomial time, while we show that deciding view-serializability within these isolation levels remains NP-complete.

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

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

1 INTRODUCTION

The holy grail in concurrency control is serializability. This notion guarantees the highest level of isolation between transactions, ensuring that the results of a transaction remain invisible to other transactions until it is committed. Additionally, serializability offers a simple and intuitive model for programmers enabling them to focus exclusively on the correctness of individual transactions independent of any concurrent transactions. Formally, an interleaving or schedule of transactions is *serializable* when it is equivalent to a serial execution of those transactions. The definition of serializability crucially depends on the chosen notion of equivalence, that is, when two schedules of transactions are considered to be equivalent. The most prevalent notion in the concurrency literature is that of *conflict-equivalence* which requires that the ordering of conflicting operations in both schedules is preserved.[1] A schedule then is *conflict-serializable* when it is conflict-equivalent to a serial schedule.

A more permissive notion of equivalence is that of *view-equivalence* which requires that corresponding read operations in the two schedules see the same values and that the execution of both schedules leaves the database in the same state. *View-serializability* is then based on view-equivalence and aligns more closely with the intuitive understanding of what it means for a

[1]Two operations are conflicting when they access the same object in the database and at least one of them is a write operation.

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

database to be consistent as it focuses on the outcome rather than on the order of specific interactions between transactions. In addition, view-serializability is more broadly applicable than conflict-serializability in that it allows strictly more schedules to be serializable. Given the many advantages and high desirability of serializable schedules, it might seem counterintuitive that in practice only the more stringent notion of conflict-serializability is considered. The main reason, however, is that reasoning about conflict-serializability is much more intuitive than reasoning about view-serializability. Indeed, conflict-serializability can be characterized through the absence of cycles in a so-called conflict-graph which in turn gives rise to a very natural and efficient polynomial time decision algorithm. Moreover, the idea of defining admissible schedules in terms of the absence of cycles in a graph structure naturally extends to the definition of isolation levels where additional requirements are enforced on the type of edges in cycles (see, e.g., [1]). In strong contrast, however, deciding view-serializability is NP-complete [16]. Conflict-serializability can therefore be considered as a good approximation for view-serializability that is easier to implement and that can be the basis of concurrency control algorithms with acceptable performance. Next, we argue that in the context of robustness it does make sense to reconsider view-serializability.

Many relational database systems offer a range of isolation levels, allowing users to trade in isolation guarantees for better performance. However, executing transactions concurrently at weaker degrees of isolation does carry some risk as it can result in specific anomalies. Nevertheless, there are situations when a group of transactions can be executed at an isolation level weaker than serializability without causing any errors. In this way, we get the higher isolation guarantees of serializability for the price of a weaker isolation level, which is typically implementable with a less expensive concurrency control mechanism. There is a famous example that is part of database folklore: the TPC-C benchmark [19] 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. [13]).

The property discussed above is called *(conflict-)robustness*[2] [4, 12, 13]: a set of transactions $\mathcal{T}$ is called *conflict-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 conflict-serializable. The robustness problem received quite a bit of attention in the literature, and we refer to Section 5 for an extensive discussion of prior work. In [20], it was experimentally verified that, under high contention, workloads that are conflict-robust against Read Committed (RC) can be evaluated faster under RC compared to stronger isolation levels. This means that the stronger guarantee of serializability is obtained at the lower cost of evaluating under RC. Unfortunately, not all workloads are conflict-robust against a weaker isolation level. A natural question is therefore whether such workloads are *view-robust* which would imply that they could still be safely executed at the weaker isolation level despite not being conflict-robust. Here, safe refers to view-serializable which, as discussed above, better corresponds to the intuitive understanding of what it means for a database to be consistent. We discuss this next in more detail.

View-robustness against an isolation level is defined in analogy to conflict-robustness where it is now required that every possible interleaving allowed by the isolation level must be *view*-serializable. It readily follows that conflict-robustness implies view-robustness: when every allowed schedule is conflict-equivalent to a serial schedule, it is also view-equivalent to the same serial schedule as conflict-equivalence implies view-equivalence. However, we show that there are isolation levels for

[2]Actually, robustness is the term used in the literature as it was always clear that robustness w.r.t. conflict-serializability was meant. We use the term conflict-robustness in this paper to distinguish it from view-robustness and just say robustness when the distinction does not matter.

which view-robustness does not imply conflict-robustness (cf., Proposition 3.4(2)). That is, there are sets of transactions that are view-robust but not conflict-robust against those isolation levels, and view-robustness can therefore allow more sets of transactions to be safely executed.

In this paper, we study the view-robustness problem for the isolation levels occurring in PostgreSQL and Oracle: RC, SI, and serializable snapshot isolation (SSI) [13, 18]. We point out that the naïve algorithm which determines non-robustness by merely guessing an allowed schedule and testing that it is not view-serializable, is in the complexity class Σ_2^P. It is important to realize that even such high complexity does not necessarily rule out practical applicability. Indeed, detecting robustness is not an online problem that occurs while a concrete transaction schedule unfolds possibly involving thousands of transactions. The approach of [20–22] that we follow in this paper, is targeted at settings where transactions are generated by a handful of transaction *programs*, for instance, made available through an API. The TPC-C benchmark, for example, consists of five different transaction programs, from which an unlimited number of concrete transactions can be instantiated. Robustness then becomes a *static* property that can be tested offline at API design time. When the small set of transaction programs passes the robustness test, the database isolation level can be set to the weaker isolation level for that API without fear of introducing anomalies. To further clarify the difference between transaction programs and mere transactions: a transaction is a sequence of read and write operations to concrete database objects while a transaction program is a parameterized transaction (possibly containing loops and conditions) that can be instantiated to form an unlimited number of concrete transactions. Previous work has shown that algorithms for deciding conflict-robustness for transaction programs use algorithms for deciding conflict-robustness for mere transactions as basic building blocks [20, 21]. It therefore makes sense to first study view-robustness for concrete transactions (as we do in this paper) in an effort to increase the amount of workloads that can be executed safely at weaker isolation levels.

Notwithstanding this inherent potential for practical applicability, we show the (at least to us) surprising result that for the isolation levels of PostgreSQL and Oracle, view-robustness *always* implies conflict-robustness. The latter even extends to the setting of mixed allocations where transactions can be allocated to different isolation levels (as for instance considered in [12, 23]). This means in particular, that for these systems, widening from conflict- to view-serializability does not allow for more sets of transactions to become robust. As a main technical tool, we identify a criterion (Condition C1) for isolation levels in terms of the existence of a counter-example schedule of a specific form that witnesses non-conflict-robustness. We then show that for classes of isolation levels that satisfy this condition, view-robustness always implies conflict-robustness (Theorem 3.7) and prove that the class {RC, SI, SSI} satisfies it (Theorem 3.8).

While view- and conflict-robustness coincide for the class {RC, SI, SSI}, it is interesting to point out that conflict- and view-serializability do not. In fact, the complexity of the corresponding decision problems within these isolation levels is quite different.[3] Indeed, it readily follows that deciding conflict-serializability for schedules allowed under RC and SI remains in polynomial time,[4] while we show that deciding view-serializability within these isolation levels remains NP-complete.

In addition to the practical motivation outlined above to study view-robustness, the present paper can also be related to work done by Yannakakis [25] who showed that view- and conflict-robustness coincide for the class of isolation levels that allows all possible schedules.[5] The present paper can

[3]That is, assuming P≠NP.

[4]For SSI the problem is trivial as any schedule allowed under SSI is both conflict- and view-serializable.

[5]That paper refers to the problems as view- and conflict-safety, and studies safety for more serializability notions.

therefore be seen as an extension of that research line where it is obtained that (*i*) view- and conflict-robustness do not coincide for all classes of isolation levels; and, (*ii*) view- and conflict-robustness do coincide for the isolation levels in PostgreSQL and Oracle.

Outline. This paper is further organized as follows. We introduce the necessary definitions in Section 2. We discuss conflict- and view-robustness in Section 3 and consider the complexity of deciding view-serializability in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6. Missing proofs can be found in [24].

2 DEFINITIONS

2.1 Transactions and schedules

We fix an infinite set of objects **Obj**. A *transaction* T over **Obj** is a sequence of operations $o_1 \cdots o_n$. To every operation o, we associate action$(o) \in \{\mathtt{R}, \mathtt{W}, \mathtt{C}\}$ and object$(o) \in$ **Obj**. We say that o is a read, write or commit operation when action(o) equals R, W, and C, respectively, and that o is an operation on object(o). 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 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.

We introduce some notation to facilitate the exposition of examples. For an object $\mathtt{t} \in$ **Obj**, we denote by $\mathtt{R[t]}$ a *read* operation on t and by $\mathtt{W[t]}$ a *write* operation on t. We denote the special *commit* operation simply by C. 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. To avoid ambiguity of this notation, in the literature it is commonly assumed that a transaction performs at most one write and one read operation per object (see, *e.g.* [3, 12]). *We follow this convention only in examples and emphasize that all our results hold for 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 t a total order over all write operations on t occurring in s, and,
- v_s is a *version function* mapping each read operation a in s to either op_0 or to a write operation 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 $a \in O_s$, $a \ll_s b$ iff $a <_T b$ for every pair of write operations a, b occurring in a transaction $T \in \mathcal{T}$ and writing to the same object, 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, these requirements imply the following: 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. If however these two write operations $a, b \in O_s$ occur in the same transaction $T \in \mathcal{T}$, then we do require that these versions are installed in the order implied by the order of operations in $\mathcal{T}$. That is, $a \ll_s b$ iff $a <_T b$. See Figure 1 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 [4, 12] 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 $\mathtt{W}_2[\mathtt{t}] \rightarrow_{s_1} \mathtt{W}_4[\mathtt{t}]$, $\mathtt{W}_3[\mathtt{v}] \rightarrow_{s_1} \mathtt{R}_4[\mathtt{v}]$ and $\mathtt{R}_4[\mathtt{t}] \rightarrow_{s_1} \mathtt{W}_2[\mathtt{t}]$ are respectively a ww-dependency, a wr-dependency and a rw-antidependency in schedule s_1 presented in Figure 1.

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.*

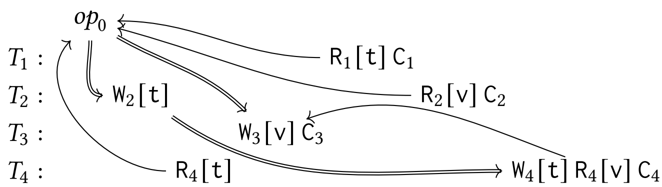

Fig. 1. A schedule s_1 with v_{s_1} (single lines) and $\ll_{s_1}$ (double lines) represented through arrows.

Figure 3 visualizes the serialization graph $SeG(s_1)$ for the schedule s_1 in Figure 1. Since $SeG(s_1)$ is not acyclic, s_1 is not conflict-serializable.

2.3 View-Serializability

For a schedule s and object t occurring in a write operation in a transaction in s, let $last_{\ll_s}(\mathtt{t})$ denote the write operation installing the last version of t in s. That is, there is no write operation b_i on t in s with $last_{\ll_s}(\mathtt{t}) \ll_s b_i$. Two schedules s and s' are *view-equivalent* if they are over the same set $\mathcal{T}$ of transactions and for every read operation b_i in $\mathcal{T}$, $v_s(b_i) = v_{s'}(b_i)$ and $last_{\ll_s}(\mathtt{t}) = last_{\ll_{s'}}(\mathtt{t})$ for each t occurring in a write operation in a transaction in s. In other words, view-equivalence requires the two schedules to observe identical versions for each read operation and to install the same last versions for each object.

Definition 2.3. A schedule s is *view-serializable* if it is view-equivalent to a single-version serial schedule.

THEOREM 2.4 ([16]). *Deciding whether a schedule s is view serializable is* NP*-complete, even if s is restricted to single-version schedules.*

The following Theorem extends a well-known result [16] for single-version schedules towards multiversion schedules:

THEOREM 2.5. *If a schedule s is conflict-serializable, then it is view-serializable.*

The opposite direction does not hold as the next example shows.

Example 2.6. Consider the schedule s_2 over a set of three transactions $\mathcal{T} = \{T_1, T_2, T_3\}$ visualized in Figure 2. This schedule is not conflict-serializable, witnessed by the cycle in $SeG(s_2)$ given in Figure 3. However, s_2 is view-serializable, as it is view equivalent to the single-version serial schedule $s' : T_1 \cdot T_2 \cdot T_3$. Notice in particular that $v_{s_2}(\mathtt{R}_1[\mathtt{t}]) = v_{s'}(\mathtt{R}_1[\mathtt{t}]) = op_0$ and that T_3 installs the last version of objects t and v in both schedules. Notice that view-serializability does not impose any restrictions on the ordering of $\mathtt{W}_1[\mathtt{t}]$ and $\mathtt{W}_2[\mathtt{t}]$ as t is not read by T_3. □

2.4 Isolation levels

Let $\Sigma_\mathcal{T}$ be the set of all possible schedules over a set of transactions $\mathcal{T}$. An *allocation* $\mathcal{A}$ for $\mathcal{T}$ defines a set $\Sigma_\mathcal{A} \subseteq \Sigma_\mathcal{T}$. A schedule s over $\mathcal{T}$ is *allowed under* $\mathcal{A}$ if $s \in \Sigma_\mathcal{A}$. We furthermore assume that $\mathcal{A}$ implies an allocation for each set of transactions $\mathcal{T}' \subseteq \mathcal{T}$, and write $\mathcal{A}[\mathcal{T}']$ to denote the allocation obtained by restricting $\mathcal{A}$ to the transactions in $\mathcal{T}'$. Typically, the set of allowed schedules is defined through the isolation level assigned to each transaction. To this end, let $\mathcal{I}$ be a class of isolation levels. An $\mathcal{I}$*-allocation* $\mathcal{A}$ for a set of transactions $\mathcal{T}$ is an allocation to which we relate a function $f_\mathcal{A}$ mapping each transaction $T \in \mathcal{T}$ onto an isolation level $f_\mathcal{A}(T) \in \mathcal{I}$. As a slight abuse of notation, we will frequently denote $f_\mathcal{A}(T)$ by $\mathcal{A}(T)$. Intuitively, $\Sigma_\mathcal{A}$ then contains

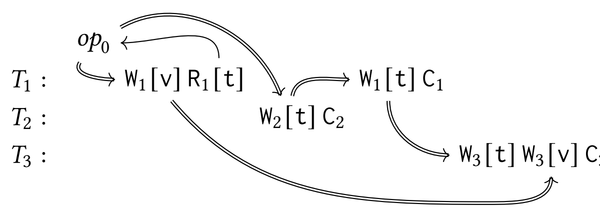

Fig. 2. A schedule s_2 with v_{s_2} (single lines) and $\ll_{s_2}$ (double lines) represented through arrows.

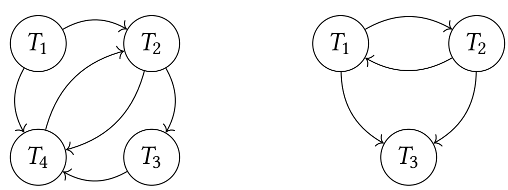

Fig. 3. Serialization graphs $SeG(s_1)$ *(left)* and $SeG(s_2)$ *(right)* for the schedules s_1 and s_2 presented in Figure 1 and Figure 2.

all schedules that can be obtained by executing the transactions in $\mathcal{T}$ under the isolation levels prescribed by $\mathcal{A}$.

In this paper, we consider the following isolation levels: read committed (RC), snapshot isolation (SI), and serializable snapshot isolation (SSI). Before we define when a schedule is allowed under a {RC, SI, SSI}-allocation, we introduce some necessary terminology. Some of these notions are illustrated in Example 2.9 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 $v_s(\mathtt{R}_j[\mathtt{t}]) \in T_i$ with $\mathtt{C}_i <_s a_j$ for some $T_i \in \mathcal{T}$; and
- there is no write operation $\mathtt{W}_k[\mathtt{t}] \in O_s$ 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.

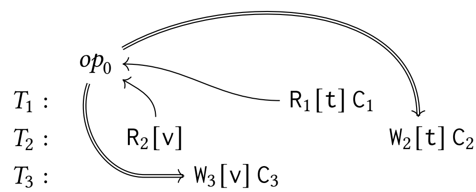

Fig. 4. A schedule s with v_s (single lines) and $\ll_s$ (double lines) represented through arrows. The three transactions form a dangerous structure $T_1 \to T_2 \to T_3$.

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 $b_i <_s a_j <_s \mathsf{C}_i$. 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 1 exhibits a concurrent write, since it writes to t, which has been modified earlier by a concurrent transaction T_2. However, T_4 does not exhibit a dirty write, since T_2 has already committed before T_4 writes to t.

Definition 2.7. 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., [12, 20]). We emphasize that our definition of RC is based on concrete implementations over multiversion databases, found in e.g. PostgreSQL, and should therefore not be confused with different interpretations of the term Read Committed, such as lock-based implementations [3] 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 or view-serializability. That is, a conflict-serializable (respectively, view-serializable) schedule is not necessarily allowed under RC and SI as we discuss further in Section 4.

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 [7]: 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;
- $\mathsf{C}_3 \leq_s \mathsf{C}_1$ and $\mathsf{C}_3 <_s \mathsf{C}_2$; and
- if T_1 only contains read operations, then $\mathsf{C}_3 <_s \mathit{first}(T_1)$.

An example of a dangerous structure is visualized in Figure 4. Note that this definition of dangerous structures slightly extends the one in [7], where it is not required for T_3 to commit before T_1 and T_2. In the full version [8] of that paper, it is shown that such a structure can only lead to non-serializable schedules if T_3 commits first. Furthermore, Ports and Grittner [18] show that if T_1 is a read-only transaction, this structure can only lead to non-serializable behaviour if T_3 commits before T_1 starts. Actual implementations of SSI (e.g., PostgreSQL [18]) 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 $\{\text{RC}, \text{SI}, \text{SSI}\}$-allocation.

Definition 2.8. A schedule s over a set of transactions $\mathcal{T}$ *is allowed under an* $\{RC, SI, SSI\}$*-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 $T_i \rightarrow T_j \rightarrow T_k$ in s formed by three (not necessarily different) transactions $T_i, T_j, T_k \in \{T \in \mathcal{T} \mid \mathcal{A}(T) = \text{SSI}\}$.

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

Example 2.9. Consider the schedule s_1 in Figure 1. Transaction T_1 is concurrent with T_2 and T_4, but not with T_3; all other transactions are pairwise concurrent with each other. The second read operation of T_4 is a read-last-committed relative to itself but not relative to the start of T_4. The read operation of T_2 is read-last-committed relative to the start of T_2, but not relative to itself, so an allocation mapping T_2 to RC is not allowed. All other read operations are read-last-committed relative to both themselves and the start of the corresponding transaction. None of the transactions exhibits a dirty write. Only transaction T_4 exhibits a concurrent write (witnessed by the write operation in T_2). Due to this, an allocation mapping T_4 on SI or SSI is not allowed. The transactions $T1 \rightarrow T2 \rightarrow T3$ form a dangerous structure, therefore an allocation mapping all three transactions T_1, T_2, T_3 on SSI is not allowed. All other allocations, that is, mapping T_4 on RC, T_2 on SI or SSI and at least one of T_1, T_2, T_3 on RC or SI, is allowed. □

3 CONFLICT- AND VIEW-ROBUSTNESS

3.1 Definitions and basic properties

We define the robustness property [4] (also called *acceptability* in [12, 13]), which guarantees serializability for all schedules over a given set of transactions for a given allocation.

Definition 3.1 (Robustness). A set of transactions $\mathcal{T}$ is *view-robust* (respectively, *conflict-robust*) against an allocation $\mathcal{A}$ for $\mathcal{T}$ if every schedule over every $\mathcal{T}' \subseteq \mathcal{T}$ that is allowed under $\mathcal{A}[\mathcal{T}']$ is view-serializable (respectively, conflict-serializable).

It is important to note that the above definition demands serializability for every schedule over *every subset of* $\mathcal{T}$. View- and conflict-robustness correspond to view- and conflict-safety as defined by Yannakakis [25].

We can define a less stringent version as follows:

Definition 3.2. A set of transactions $\mathcal{T}$ is *exact view-robust* (respectively, *exact conflict-robust*) against an allocation $\mathcal{A}$ for $\mathcal{T}$ if every schedule over $\mathcal{T}$ that is allowed under $\mathcal{A}[\mathcal{T}]$ is view-serializable (respectively, conflict-serializable).

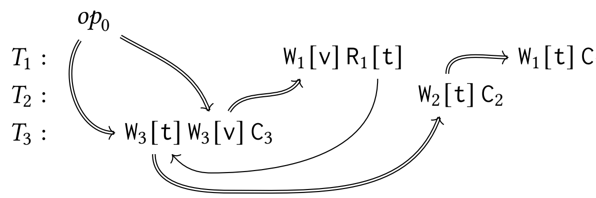

Fig. 5. A schedule s_4 with v_{s_4} (single lines) and $\ll_{s_4}$ (double lines) represented through arrows.

Exact conflict-robustness is what is used in, e.g., [12, 23].[6] However, for conflict-robustness the distinction does not matter as exact conflict-robustness is the same as conflict-robustness. Indeed, a schedule s' over a subset $\mathcal{T}' \subseteq \mathcal{T}$ that is not conflict-serializable can always be extended to a schedule s over $\mathcal{T}$ that is not conflict-serializable by appending the remaining transactions in $\mathcal{T}$ (those not appearing in $\mathcal{T}'$) to s' in a serial fashion.[7] Indeed, the cycle in the serialization graph cannot disappear by adding transactions to the schedule. For view-serializability, this argument does not work. In fact, the obtained schedule s can be view-serializable, even if s' is not view-serializable.

As explained in the introduction, the relevance of robustness lies in its utility as a static property that is tested offline w.r.t. a small set of transaction programs. Here, any instantiation of any subset of such templates should be taken into account. Therefore, view-robustness is the desirable property, not exact view-robustness. We next give an example of a set of transactions that is exact view-robust but not view-robust.

Example 3.3. Reconsider the set of transactions $\mathcal{T} = \{T_1, T_2, T_3\}$ from Example 2.6. Let $\mathcal{A}$ be the {RC, SI, SSI}-allocation for $\mathcal{T}$ with $\mathcal{A}(T_i) = \text{RC}$ for each $T_i \in \mathcal{T}$. The schedule s_2 over $\mathcal{T}$ presented in Figure 2 is allowed under $\mathcal{A}$ but not conflict-serializable, thereby witnessing that $\mathcal{T}$ is not (exact) conflict-robust against $\mathcal{A}$.

However, $\mathcal{T}$ is exact view-robust against $\mathcal{A}$ since every schedule s over $\mathcal{T}$ allowed under $\mathcal{A}$ is view-serializable. Notice in particular that if s is allowed under $\mathcal{A}$, then T_1 and T_3 cannot be concurrent in s, as this would always imply a dirty write. For every schedule s allowed under $\mathcal{A}$, we can therefore identify a view-equivalent single-version serial schedule s' over $\mathcal{T}$, depending on $v_s(\mathtt{R}_1[\mathtt{t}])$:

- if $v_s(\mathtt{R}_1[\mathtt{t}]) = op_0$, then s is view-equivalent to either $T_1 \cdot T_2 \cdot T_3$ or $T_1 \cdot T_3 \cdot T_2$, depending on whether T_2 commits before T_3 in s;
- if $v_s(\mathtt{R}_1[\mathtt{t}]) = \mathtt{W}_2[\mathtt{t}]$, then s is view-equivalent to either $T_2 \cdot T_1 \cdot T_3$ or $T_3 \cdot T_2 \cdot T_1$, depending on whether T_1 commits before T_3 in s; and
- if $v_s(\mathtt{R}_1[\mathtt{t}]) = \mathtt{W}_3[\mathtt{t}]$, then s is view-equivalent to either $T_3 \cdot T_1 \cdot T_2$ or $T_2 \cdot T_3 \cdot T_1$, depending on whether T_1 commits before T_2 in s.

Even though $\mathcal{T}$ is exact view-robust against $\mathcal{A}$, it is not view-robust against $\mathcal{A}$, as we will show next. Let s_3 be the schedule over $\mathcal{T}' = \{T_1, T_2\}$ obtained by removing T_3 from s_2 in Figure 2. Then, s_3 is allowed under $\mathcal{A}[\mathcal{T}']$ but not view-serializable. Indeed, T_1 observes the initial version of object t and installs the last version of t, and should therefore occur both before and after T_2 in a view-equivalent single-version serial schedule, leading to the desired contradiction.

Notice in particular that s_3 cannot be extended to a non-view-serializable schedule over $\mathcal{T}$ by appending or prepending T_3. Indeed, s_2 in Figure 2 is the view-serializable schedule obtained by

[6]In these works exact robustness is just called robustness.

[7]We assume that if s' is allowed under $\mathcal{A}[\mathcal{T}']$ then the extended schedule s is allowed under $\mathcal{A}$ as well. This property holds for the isolation levels considered in this paper.

appending T_3, whereas the schedule s_4 obtained by prepending T_3 is given in Figure 5. As explained above, s_4 is view-equivalent to the serial schedule $T_2 \cdot T_3 \cdot T_1$. The crucial difference between s_3 and s_4 is that by prepending T_3, transaction T_1 no longer observes the initial version of object t but the version written by T_3 instead. This allows T_2 to be situated before T_1 in a view-equivalent schedule, as long as T_3 is situated between T_2 and T_1 in this view-equivalent schedule. □

The next result shows that conflict-robustness always implies view-robustness but not vice-versa.

Proposition 3.4. (1) *For every allocation $\mathcal{A}$ for a set of transactions $\mathcal{T}$, if $\mathcal{T}$ is conflict-robust against $\mathcal{A}$, then $\mathcal{T}$ is view-robust against $\mathcal{A}$.*
(2) *There is a set of transactions $\mathcal{T}$ and an allocation $\mathcal{A}$ for $\mathcal{T}$, such that $\mathcal{T}$ is view-robust against $\mathcal{A}$ but not conflict-robust.*

3.2 Generalized split schedules

Towards identifying classes of isolation levels $\mathcal{I}$ for which view-robustness and conflict-robustness coincide, we first introduce the notion of a *generalized split schedule*.

In the next definition, we use $\mathsf{prefix}_b(T)$ for an operation b in T to denote 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 3.5 (Generalized split schedule). Let $\mathcal{T} = \{T_1, T_2, \ldots T_n\}$ be a set of transactions. A *generalized split schedule* s over $\mathcal{T}$ is a multiversion schedule over $\mathcal{T}$ that has the following form:

$$\mathsf{prefix}_{b_1}(T_1) \cdot T_2 \cdot \ldots \cdot T_n \cdot \mathsf{postfix}_{b_1}(T_1),$$

where

(1) for each pair of transactions $T_i, T_j \in \mathcal{T}$ with either $j = i + 1$ or $i = n$ and $j = 1$, there is a $b_i \in T_i$ and $a_j \in T_j$ such that $b_i \rightarrow_s a_j$;
(2) if for some pair of transactions $T_i, T_j \in \mathcal{T}$ an operation $a_j \in T_j$ depends on an operation $b_i \in T_i$ in s, then $j = i + 1$, or $i = n$ and $j = 1$; and
(3) each write operation occurring in s respects the commit order of s.

The conditions above correspond to the following. Condition (1) says that there is a cyclic chain of dependencies; condition (2) stipulates that this cycle is minimal; and, condition (3) enforces that transactions writing to the same object install versions in the same order as they commit.

The next lemma says that a generalized split schedule functions as a counterexample schedule for conflict-robustness as well as for view-robustness. It forms a basic building block for Theorem 3.7.

Lemma 3.6. *Let s be a generalized split schedule for a set of transactions $\mathcal{T}$. Then, s is not conflict-serializable and not view-serializable.*

3.3 Sufficient condition

We are now ready to formulate a sufficient condition for allocations for which conflict-robustness implies view-robustness:

(C1) An allocation $\mathcal{A}$ for a set of transactions $\mathcal{T}$ satisfies Condition C1 if a generalized split schedule s' over a set of transactions $\mathcal{T}' \subseteq \mathcal{T}$ such that s' is allowed under $\mathcal{A}[\mathcal{T}']$ always exists when $\mathcal{T}$ is not conflict-robust against $\mathcal{A}$.

The next theorem shows that C1 is indeed a sufficient condition.

Theorem 3.7. *Let $\mathcal{A}$ be an allocation for a set of transactions $\mathcal{T}$ for which Condition C1 holds. Then, the following are equivalent:*

(1) $\mathcal{T}$ *is conflict-robust against* $\mathcal{A}$*; and,*
(2) $\mathcal{T}$ *is view-robust against* $\mathcal{A}$*.*

3.4 Robustness against RC, SI, and SSI

In this section, we obtain that for the isolation levels of PostgreSQL and Oracle, widening from conflict- to view-serializability does not allow for more sets of transactions to become robust.

Theorem 3.8. *Let* $\mathcal{A}$ *be an* {*RC, SI, SSI*}*-allocation for a set of transactions* $\mathcal{T}$*. Then, the following are equivalent:*

(1) $\mathcal{T}$ *is conflict-robust against* $\mathcal{A}$*;*
(2) $\mathcal{T}$ *is view-robust against* $\mathcal{A}$*;*

To prove the above theorem, it suffices to show by Theorem 3.7 that Condition C1 holds for {RC, SI, SSI}-allocations. It follows from Vandevoort et al. [23] that for {RC, SI, SSI}-allocations, exact conflict-robustness is characterized in terms of the absence of schedules of a very specific form, referred to as *multiversion split schedules*. In particular, they show that if a set of transactions $\mathcal{T}$ is not exact conflict-robust against an {RC, SI, SSI}-allocation $\mathcal{A}$, then there exists a multiversion split schedule s over $\mathcal{T}$ that is allowed under $\mathcal{A}$ and that is not conflict-serializable. For a set of transactions $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ with $n \geq 2$, a multiversion split schedule 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 either $j = i+1$ or $i = m$ and $j = 1$, there is a $b_i \in T_i$ and $a_j \in T_j$ such that $b_i \rightarrow_s a_j$. Although the structure of a multiversion split schedule is similar to the structure of a generalized split schedule, there are two important differences: (1) a multiversion split schedule allows a tail of serial transactions $T_{m+1} \cdot \ldots \cdot T_n$ to occur after $\mathsf{postfix}_{b_1}(T_1)$; and (2), the cyclic chain of dependencies is not necessarily minimal. Furthermore, Vandevoort et al. [23] only consider exact conflict-robustness, whereas we consider conflict-robustness. Our proof of Theorem 3.8 follows the following steps: (1) show that conflict-robustness and exact conflict-robustness coincides for {RC, SI, SSI}-allocations; (2) show that a counterexample multiversion split schedule allowed under a {RC, SI, SSI}-allocation $\mathcal{A}$ can always be transformed into a generalized split schedule that is allowed under $\mathcal{A}$. Theorem 3.8 then readily follows.

From Theorem 3.3 in [23] and Theorem 3.8, we have the following corollary:

Corollary 3.9. *Let* $\mathcal{A}$ *be an* {*RC, SI, SSI*}*-allocation for a set of transactions* $\mathcal{T}$*. Deciding whether* $\mathcal{T}$ *is view-robust against* $\mathcal{A}$ *is in polynomial time.*

4 DECIDING VIEW-SERIALIZABILITY

It is well-known that view-serializability is NP-hard [16]. The proof is based on a reduction from the polygraph acyclicity problem but the resulting schedules make extensive use of dirty writes which are not permitted under RC or SI. We present a modified reduction that avoids dirty writes and obtain that deciding view-serializability remains NP-hard even when the input is restricted to schedules only consisting of transactions that are allowed under RC, or SI, respectively.

Theorem 4.1. *Deciding view-serializability is NP-hard, even for schedules only consisting of transactions allowed under RC, or SI, respectively.*

We stress that Example 2.6 and Example 2.9 already show that view- and conflict-serializability do not coincide for the class of isolation levels {RC, SI, SSI}.

5 RELATED WORK

5.1 View-serializability

For single-version schedules, Yannakakis [25] formally defines view-serializability. For this problem, NP-hardness follows from a trivial extension based on a result by Papadimitriou [15], proving that deciding final-state serializability is NP-complete. Yannakakis [25] furthermore proves that within this setting of single-version schedules, the class of conflict-serializable schedules is the largest monotonic class contained in the class of view-serializable schedules. That is, a single-version schedule s is conflict-serializable if and only if all its (single-version) subschedules (including s itself) are view-serializable.

Bernstein and Goodman [5] extend concurrency control towards multiversion databases. Their definition of view-equivalence only requires the two schedules to observe identical versions for each read operation, and does not require the two schedules to install the same last versions for each object (as we do in this paper). The rationale is that when each write operation introduces a new version, the final database state will contain all versions and subsequent transactions executed afterwards could access any version that is required. However, in the context of isolation levels, a context not considered in [5], restrictions can be put on which versions can be read by subsequent transactions. For instance, practical systems like PostgreSQL always require that the last committed version should be read (which allows the DBMS to safely remove old versions when all active transactions can only observe newer versions of this object). In the setting of this paper, it therefore makes sense to additionally require that view-equivalent schedules install the same last versions for each object.

Our definition of view-serializability is furthermore different from multiversion serializability studied by Papadimitriou and Kanellakis [17]. In particular, our definition assumes the version function v_s to be fixed while searching for a view-equivalent single-version serial schedule, whereas their setting only assumes the order of operations $\leq_s$ to be fixed. Such an order is then said to be multiversion serializable if a version function exists such that the resulting schedule is view-equivalent to a single-version serial schedule. For this notion of multiversion serializability, they additionally consider a setting where at any point in time only the k most recent versions of each object are stored for some fixed positive integer k. It is interesting to note that the isolation levels considered in this paper do not imply a fixed upper bound on the number of versions for each object that must be stored at any point in time. Indeed, since transactions allowed under SI and SSI essentially take a snapshot of the versions visible at the start of the transaction, concurrent transactions can require a potentially unbounded number of versions to be maintained. Finally, in the context of isolation levels that effectively constrain version functions, as considered in this paper, it is sensible to consider the version function to be fixed when defining serializability.

5.2 Robustness and allocation for transactions

Yannakakis [25] studies S-*safety* of transaction sets with respect to different notions of serializability S. A set of transactions $\mathcal{T}$ is said to be S-*safe* if every (single-version) schedule s over a subset of $\mathcal{T}$ is S-serializable. This notion of S-safety therefore corresponds to S-robustness against allocations allowing *all* single-version schedules. A particularly interesting result by Yannakakis [25] is that view-safety and conflict-safety coincide. That is, for a given set of transactions $\mathcal{T}$, one can construct a single-version schedule that is not view-serializable over a subset of $\mathcal{T}$ iff a single-version schedule over a subset of $\mathcal{T}$ exists that is not conflict-serializable. Our work can therefore be seen as a generalization of this result to specific classes of isolation levels. It is worth to point out that Proposition 3.4(2) already shows that the property does not hold for all classes of isolation levels.

Fekete [12] is the first work that provides a necessary and sufficient condition for deciding exact conflict-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, a characterization for exact conflict-robustness against SNAPSHOT ISOLATION is obtained. Ketsman et al. [14] provide characterisations for exact conflict-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 [20] for exact conflict-robustness against *multiversion* read committed which is the variant that is considered in this paper. Vandevoort et al. [23] consider exact conflict-robustness against allocations over {RC, SI, SSI} for which they provide a polynomial time decision procedure. They show in addition that there always is a unique optimal robust allocation over {RC, SI, SSI} and provide a polynomial time algorithm to compute it. Other work studies exact conflict-robustness within a framework for uniformly specifying different isolation levels in a declarative way [4, 9–11]. 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*. None of these works consider view-robustness.

Finally, we mention that the robustness problem is orthogonal to the problem of deciding whether a schedule is allowed under an isolation level. Biswas and Enea [6] study this problem for a setting where schedules are represented by a partial order over the transactions (referred to as the session order) and a write-read relation, indicating for each read operation the write operation that wrote the observed version. A schedule is allowed under an isolation level if the partial order can be extended to a total order over the transactions consistent with the write-read relation as well as the criteria specific to the isolation level. For this setting, they show that verifying READ COMMITTED, READ ATOMIC and CAUSAL CONSISTENCY can be done in polynomial time, but deciding PREFIX CONSISTENCY and SNAPSHOT ISOLATION are NP-complete. Their setting should be contrasted with ours, where schedules are given as a total order over all operations and with a fixed version function and version order.

6 CONCLUSIONS

We showed that conflict- and view-robustness coincide for the class of isolation levels {RC, SI, SSI}. The main implication is that for systems deploying these isolation levels, widening from conflict- to view-serializability does not allow for more sets of transactions to become robust. In addition, this paper can be seen as an extension of work by Yannakakis [25] who studied robustness (under the name of safety) for various serializability notions and obtained that view- and conflict-robustness coincide when *all* schedules are allowed.

An interesting direction for future work is studying robustness relative to other notions of serializability. A particularly interesting notion is the one of strict view-serializability. This notion extends view-serializability by requiring that the relative order of non-concurrent transactions should be preserved in the equivalent single-version serial schedule. Trivially, strict view-robustness implies view-robustness as every strict view-serializable schedule is view-serializable as well. However, since conflict-serializable schedules are not always strict view-serializable, conflict-robustness does not imply strict view-robustness in general. In fact, it can be shown that this result remains to hold even if we restrict our attention to {RC}-allocations or {SI}-allocations, and the complexity of strict view-robustness is therefore still open for {RC, SI, SSI}-allocations. It would furthermore be interesting to identify relevant subsets where conflict-robustness coincides with strict view-robustness.

ACKNOWLEDGMENTS

This work is partly 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] 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.

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

[5] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (1983), 465–483.

[6] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28.

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

[8] 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.

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

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

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

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

[13] 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.

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

[15] Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J.ACM* 26, 4 (1979), 631–653.

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

[17] Christos H. Papadimitriou and Paris C. Kanellakis. 1984. On Concurrency Control by Multiple Versions. *ACM Trans. Database Syst.* 9, 1 (1984), 89–99.

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

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

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

[21] 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.

[22] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2023. Detecting Robustness against MVRC for Transaction Programs with Predicate Reads. In *EDBT*. 565–577.

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

[24] Brecht Vandevoort, Bas Ketsman, and Frank Neven. 2024. When View- and Conflict-Robustness Coincide for Multiversion Concurrency Control (full version). (2024). https://arxiv.org/abs/2403.17665.

[25] Mihalis Yannakakis. 1984. Serializability by Locking. *J.ACM* 31, 2 (1984), 227–244.

Received December 2023; revised February 2024; accepted March 2024