Transaction Isolation in Mixed-Level and Mixed-Scope Settings

Stephen J. Hegner
DBMS Research of New Hampshire
PO Box, 2153, New London, NH 03257, USA
`dbmsnh@gmx.com`

Abstract

Modern database-management systems permit the isolation level to be set on a per-transaction basis. In such a *mixed-level* setting, it is important to understand how transactions running at different levels interact. More fundamentally however, these levels are sometimes of different scopes. For example, `READ COMMITTED` and `REPEATABLE READ` are of *local scope*, since the defining properties depend upon only the transaction and its relationship to those running concurrently. On the other hand, `SERIALIZABLE` is of *global scope*; serializability is a property of a schedule of transactions, not of a single transaction. In this work, in addition to formalizing the interaction of transactions at different levels, the meaning of serializability within local scope is also addressed.

1 Introduction

In a modern relational database-management system (RDBMS), there is tradeoff between performance via transaction concurrency and adequate isolation of transactions from the operations of other transactions. It has therefore long been held that a single notion of isolation is not adequate. Rather, the level of isolation should be determined by the needs of the application. This philosophy is integral to SQL, the standard of which [10, Part 2, Sec. 4.36] identifies four distinct levels of isolation for transactions, ordered with increasing isolation as

`READ UNCOMMITTED` < `READ COMMITTED` < `REPEATABLE READ` < `SERIALIZABLE`.

Based upon the names of the isolation levels, as well as upon the semantics as defined in the SQL standard, this classification is very confusing, because it mixes isolation levels of distinct scope. The isolation levels `READ UNCOMMITTED`, `READ COMMITTED`, and `REPEATABLE READ` are *local* in scope; the definitions apply to individual transactions, with the relevant isolation properties of a transaction T completely determined in conjunction with the behavior of those transactions which run concurrently with it. On the other hand, the isolation level `SERIALIZABLE` is *global* in scope; it applies to an entire schedule of transactions. Indeed, it makes no sense to say that an individual transaction is serializable; it only makes sense to say that a set of transactions, organized into a schedule S, is serializable; that is, that the results of running the transactions according to S is equivalent to running them in some schedule S' with no concurrency. This raises the obvious question of what it means, in a mixed-level system, to run some but not all transactions with serializable isolation. A main goal of this paper is to address such questions of isolation involving multiple scopes.

It is the apparent intent of the SQL standard that the `SERIALIZABLE` isolation level serve double duty, with both local and global scope, called a *multiscope* isolation level. On the one hand, it is defined to be a local isolation level, call it `DEGREE 3`,[1] which is slightly stronger than `REPEATABLE READ`. On the other hand, it is also defined in the standard to be a *serializable-generating* isolation level, in the sense that if all transactions are run at that level, then the result must be a serializable schedule. Unfortunately, it has been known for some time that `DEGREE 3`, as defined above, is not serializable generating [2, Sec. A5B], rendering the standard somewhat confusing at best. Nevertheless, the idea of a serializable-generating local isolation level is an important one. In this paper, working within the context of modern MVCC (multi-version concurrency control), a minimal serializable-generating local isolation level called RCX is identified. Interestingly, while it is slightly stronger than `READ COMMITTED`, it is not order comparable to `REPEATABLE READ`. Indeed, it is shown that it is not necessary to require repeatable reads (*i.e.*, to require that the transaction read the same value for a data object x regardless of when the read occurs during its lifetime) in order to achieve serializable-generating behavior; rather, the critical requirement is to prohibit so-called backward read-write dependencies. Thus, the SQL standard imposes a condition for the local component of its multiscope isolation level `SERIALIZABLE` which is not necessary for serialization.

For such a serializable-generating local isolation level, it is natural to ask whether it has any properties related to serializability when run in a mixed-level context, with other transactions running at other levels. The answer is shown to be in the affirmative. RCX, as well as all higher levels of isolation, are *serializable preserving*, in the sense that if transaction T runs at that level, then adding T to the schedule will not result in any new cycles. In particular, if the existing schedule is serializable, then adding T will preserve that property.

The way in which new serializable-generating strategies, including SSI [4], [7] and SSN [15], fit into this picture is also examined. SSI is of particular interest because it is used for implementation of the `SERIALIZABLE` isolation level in PostgreSQL [12]. Both may be termed *preemptive regional* strategies. They look for certain small structures in the conflict graph which are a necessary part of any cycle, aborting one of the participants when such a structure is found. However, in contradistinction to RCX, neither is serializable preserving, or has any similar property, so they have dubious benefit in a mixed-level context.

The paper is organized as follows. In Sec. 2, necessary background material on transactions and serializability is summarized. In Sec. 3, local isolation levels are studied, with a particular focus on how they interact with each other in a mixed-mode setting. In Sec, 4, serialization in multi-scope settings is examined. Finally, Sec. 5 contains conclusions and further directions.

2 Transactions, Schedules, and Serialization

In this section, the basic ideas of transactions, schedules, and serialization are summarized. The focus is to provide a precise and ambiguous notation and terminology to use as a foundation for the ideas presented in Sec. 3 and 4.

[1]The SQL standard gives `SERIALIZABLE` no name to identify its local scope. Since it is sometimes called *Degree 3* isolation in the literature, [9, Sec. 7.6], the moniker `DEGREE 3` is introduced here, purely for clarification. Technically, it is `REPEATABLE READ` which additionally prohibits so-called *phantoms*.

2.1 Data objects and the global schema A database schema $\mathbf{D}$ is defined by a set $\mathsf{DObj}\langle\mathbf{D}\rangle$ of data objects. Each such object has a single value at any point in time, which may be read or written by a transaction.

In this work, such a schema $\mathbf{D}$, called the *global schema*, is fixed. The current instance of the global schema is called the *global database*.

2.2 Time Brackets are used to identify time intervals of the real numbers $\mathbb{R}$, using common conventions. For $a, b \in \mathbb{R}$, $[a, b] = \{c \in \mathbb{R} \mid a \leq c \leq b\}$, $(a, b] = \{c \in \mathbb{R} \mid a < c \leq b\}$, and $[a, b) = \{c \in \mathbb{R} \mid a \leq c < b\}$.

2.3 Transactions A *transaction* T over $\mathsf{DObj}\langle\mathbf{D}\rangle$ is defined by certain *time points* in $\mathbb{R}$, in addition to read and write operations. First, T has a *start time* $t_{\mathsf{Start}}\langle T\rangle$ and an *end time* $t_{\mathsf{End}}\langle T\rangle$, with $t_{\mathsf{Start}}\langle T\rangle < t_{\mathsf{End}}\langle T\rangle$.

The specification of operations on the database follow an *object-level model*, in which it is only known whether a transaction reads and/or writes a given $x \in \mathsf{DObj}\langle\mathbf{D}\rangle$, without knowledge of specific values. The *read set* $\mathsf{ReadSet}\langle T\rangle \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$ of T consists of all data objects which T reads. Similarly, the *write set* $\mathsf{WriteSet}\langle T\rangle \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$ of T consists of all data objects which T writes.

The request time assignment of T provides the time at which read and write operations are requested by the transaction. Formally, the *request time assignment* Req for T assigns to each $x \in \mathsf{ReadSet}\langle T\rangle$ a time $t^{\mathsf{Req}}_{\mathsf{Read}\langle x\rangle}\langle T\rangle \in [t_{\mathsf{Start}}\langle T\rangle, t_{\mathsf{End}}\langle T\rangle)$, and to each $x \in \mathsf{WriteSet}\langle T\rangle$ a time $t^{\mathsf{Req}}_{\mathsf{Write}\langle x\rangle}\langle T\rangle \in (t_{\mathsf{Start}}\langle T\rangle, t_{\mathsf{End}}\langle T\rangle]$. Note that the read time $t^{\mathsf{Req}}_{\mathsf{Read}\langle x\rangle}\langle T\rangle$ may be the same as the start time, but that it must occur strictly before the end time. Similarly, the write time $t^{\mathsf{Req}}_{\mathsf{Write}\langle x\rangle}\langle T\rangle$ may be the same as the end time, but that it must occur strictly after the start time. Furthermore, if $x \in \mathsf{ReadSet}\langle T\rangle \cap \mathsf{WriteSet}\langle T\rangle$, then $t^{\mathsf{Req}}_{\mathsf{Read}\langle x\rangle}\langle T\rangle < t^{\mathsf{Req}}_{\mathsf{Write}\langle x\rangle}\langle T\rangle$; that is a write must occur after a read. It is assumed that each transaction T reads and writes a data object x at most once. The set of all *time points* of T is $\mathsf{TimePoints}\langle T\rangle =$
$$\{t_{\mathsf{Start}}\langle T\rangle, t_{\mathsf{End}}\langle T\rangle\} \cup \{t^{\mathsf{Req}}_{\mathsf{Read}\langle x\rangle}\langle T\rangle \mid x \in \mathsf{ReadSet}\langle T\rangle\} \cup \{t^{\mathsf{Req}}_{\mathsf{Write}\langle x\rangle}\langle T\rangle \mid x \in \mathsf{WriteSet}\langle T\rangle\}.$$
The set of all transactions over $\mathbf{D}$ is denoted $\mathsf{Trans}\langle\mathbf{D}\rangle$.

2.4 Effective time assignments In early systems using single-version concurrency control (SVCC), the request time of a read or write was often the same as the time at which the global database was actually read or written by the transaction. However, for modern systems, this is almost never the case for writes and often not the case for reads either. Rather, there is an *effective time assignment*, whose values depend upon the isolation protocol. In virtually all cases for a system employing MVCC, the effective time for a write is at the end of the transaction, while the effective time for a read depends upon the isolation protocol. With the *read-request-write-end* time assignment, denoted RRWE, all writes occur at the end of the transaction, while reads occur at their request times. Specifically, for $x \in \mathsf{ReadSet}\langle T\rangle$, $t^{\mathsf{RRWE}}_{\mathsf{Read}\langle x\rangle}\langle T\rangle = t^{\mathsf{Req}}_{\mathsf{Read}\langle x\rangle}\langle T\rangle$, while for $x \in \mathsf{WriteSet}\langle T\rangle$, $t^{\mathsf{RRWE}}_{\mathsf{Write}\langle x\rangle}\langle T\rangle = t_{\mathsf{End}}\langle T\rangle$. With the *read-beginning-write-end* time assignment, denoted RBWE, all reads occur at the start of the transaction, and all writes occur at the end. Specifically, for $x \in \mathsf{ReadSet}\langle T\rangle$, $t^{\mathsf{RBWE}}_{\mathsf{Read}\langle x\rangle}\langle T\rangle = t_{\mathsf{Start}}\langle T\rangle$, while for $x \in \mathsf{WriteSet}\langle T\rangle$, $t^{\mathsf{RRWE}}_{\mathsf{Write}\langle x\rangle}\langle T\rangle = t_{\mathsf{End}}\langle T\rangle$. RRWE is typically associated with variants of read-committed isolation, while RBWE is usually associated with variants of snapshot isolation, as elaborated in Sec. 3. $\mathsf{TASetEff} = \{\mathsf{RRWE}, \mathsf{RBWE}\}$ denotes the set consisting of the two effective time assignments.

2.5 Transactions with effective time assignment It is important to be able to run the same transaction at different levels of isolation, which may have associated with them different effective time assignments (see 3.3). Therefore, rather than building a fixed effective time assignment into a transaction, it is more appropriate to associate such a time assignment as a parameter. Formally, a *transaction with effective time assignment*, or a Teff-*transaction* for short, is a pair $\langle T, \tau\rangle$ with $T \in \mathsf{Trans}\langle\mathbf{D}\rangle$ and $\tau \in \mathsf{TASetEff}$. The set of all Teff-transactions over $\mathbf{D}$ is denoted $\mathsf{TransTeff}\langle\mathbf{D}\rangle$.

The set of time points of $\langle T, \tau\rangle$ is exactly the same as the set of time points of T; $\mathsf{TimePoints}\langle\langle T, \tau\rangle\rangle = \mathsf{TimePoints}\langle T\rangle$. Observe that in an effective time assignment (RRWE or RBWE) each write occurs at $t_{\mathsf{End}}\langle T\rangle$, and each read at either its request time $t^{\mathsf{Req}}_{\mathsf{Read}\langle T\rangle}\langle x\rangle$ or else at $t_{\mathsf{Start}}\langle T\rangle$, so the effective time assignment does not add any new time points, beyond those defined by transaction start and end, plus effective times of reads and writes.

2.6 Schedules and temporal relationships between transactions A pair $\{T_1, T_2\} \subseteq \mathsf{Trans}\langle\mathbf{D}\rangle$ is *time compatible* if $\mathsf{TimePoints}\langle T_1\rangle \cap \mathsf{TimePoints}\langle T_2\rangle = \emptyset$. A pair $\{\langle T_1, \tau_1\rangle, \langle T_2, \tau_2\rangle\} \subseteq \mathsf{TransTeff}\langle\mathbf{D}\rangle$ is *time compatible* precisely in the case that $\{T_1, T_2\}$ has that property.

A *schedule* over $\mathsf{Trans}\langle\mathbf{D}\rangle$ is a finite (possibly empty) set $S \subseteq \mathsf{TransTeff}\langle\mathbf{D}\rangle$ for which every distinct pair $\{\langle T_1, \tau_1\rangle, \langle T_2, \tau_2\rangle\} \subseteq S$ is time compatible. Define $\mathsf{TransOf}\langle S\rangle = \{T \mid \langle T, \tau\rangle \in S\}$.

Two distinct transactions $\{T_1, T_2\} \subseteq \mathsf{TransOf}\langle S\rangle$ are *concurrent*, written $T_1 \parallel T_2$, if both $t_{\mathsf{Start}}\langle T_1\rangle < t_{\mathsf{End}}\langle T_2\rangle$ and $t_{\mathsf{Start}}\langle T_2\rangle < t_{\mathsf{End}}\langle T_1\rangle$ hold. If $\{T_1, T_2\}$ is not concurrent, then it is *serial* in S. In that case, if $t_{\mathsf{Start}}\langle T_1\rangle < t_{\mathsf{Start}}\langle T_2\rangle$, write $T_1 <_S T_2$, and if $t_{\mathsf{Start}}\langle T_2\rangle < t_{\mathsf{Start}}\langle T_1\rangle$, write $T_2 <_S T_1$.

The set of all schedules over $\mathsf{Trans}\langle\mathbf{D}\rangle$ is denoted $\mathsf{Sched}\langle\mathbf{D}\rangle$.

2.7 Serializable behavior of schedules Roughly speaking, a schedule S is *serializable* if its transactions may be relocated in time so that no two are concurrent, while preserving the effect of all read and write operations. There are many distinct ways to formalize this idea; in [16, Ch. 3] there are descriptions of no fewer than five major alternatives, many with minor variants. In this work, the notion of *conflict serializability* of a schedule S will be used, owing to its simple characterization in terms of edges in the *conflict graph* of the schedule, also called the *direct serialization graph*, or *DSG*, of S. For comprehensive summaries of conflict serializability, see [11, Sec. 2.6] and [16, Sec.3.8]. In addition, [1] examines the DSG with an eye towards modern isolation protocols. Here, only the essential notions, will be identified.

The DSG associated with a schedule S is denoted $\mathsf{DSG}\langle S\rangle$. In that graph, the vertices are the members of $\mathsf{TransOf}\langle S\rangle$. There are three types of edges (also called *dependencies*). For $\langle T_1, \tau_1\rangle, \langle T_2, \tau_2\rangle \in S$, there is a *read-write edge*, or rw-*edge*, from T_1 to T_2, denoted $T_1 \xrightarrow{\mathsf{rw}} T_2$, if T_1 reads some data object x for which T_2 is the next writer. More precisely, this means that $t^{\tau_1}_{\mathsf{Read}\langle x\rangle}\langle T_1\rangle < t^{\tau_2}_{\mathsf{Write}\langle x\rangle}\langle T_2\rangle$, and for no other $\langle T_3, \tau_3\rangle \in S$ with $x \in \mathsf{WriteSet}\langle T_3\rangle$ is it the case that $t^{\tau_1}_{\mathsf{Read}\langle x\rangle}\langle T_1\rangle < t^{\tau_3}_{\mathsf{Write}\langle x\rangle}\langle T_3\rangle < t^{\tau_2}_{\mathsf{Write}\langle x\rangle}\langle T_2\rangle$.

Similarly, there is a *write-write edge*, or ww-*edge*, from T_1 to T_2, denoted $T_1 \xrightarrow{\mathsf{ww}} T_2$ if T_1 writes some data object x and T_2 is the next writer of x; *i.e.*, $t^{\tau_1}_{\mathsf{Write}\langle x\rangle}\langle T_1\rangle < t^{\tau_2}_{\mathsf{Write}\langle x\rangle}\langle T_2\rangle$, and for no other $\langle T_3, \tau_3\rangle \in S$ with $x \in \mathsf{WriteSet}\langle T_3\rangle$ is it the case that $t^{\tau_1}_{\mathsf{Write}\langle x\rangle}\langle T_1\rangle < t^{\tau_3}_{\mathsf{Write}\langle x\rangle}\langle T_3\rangle < t^{\tau_2}_{\mathsf{Write}\langle x\rangle}\langle T_2\rangle$.

Finally, there is *write-read edge*, or wr-*edge*, from T_1 to T_2, denoted $T_1 \xrightarrow{\mathsf{wr}} T_2$, if T_1 writes some data object x and T_2 subsequently reads the version of x which T_1 wrote; *i.e.*, $t^{\tau_1}_{\mathsf{Write}\langle x\rangle}\langle T_1\rangle < t^{\tau_2}_{\mathsf{Read}\langle x\rangle}\langle T_2\rangle$, and for no other $\langle T_3, \tau_3\rangle \in S$ with $x \in \mathsf{WriteSet}\langle T_3\rangle$ is it the case that $t^{\tau_1}_{\mathsf{Write}\langle x\rangle}\langle T_1\rangle < t^{\tau_3}_{\mathsf{Write}\langle x\rangle}\langle T_3\rangle < t^{\tau_2}_{\mathsf{Read}\langle x\rangle}\langle T_2\rangle$.

Note that effective time assignments are used throughout these definitions.

For $T_1 \xrightarrow{\mathsf{zz}} T_2$, $\mathsf{zz} \in \{\mathsf{rw}, \mathsf{ww}, \mathsf{wr}\}$ is called the *type* of the edge, which is furthermore *outgoing* from T_1 and *incoming* to T_2.

A schedule S is *conflict serializable* if $\mathsf{DSG}\langle S\rangle$ contains no directed cycles. (Cycles in the DSG are always taken to be directed in this work.) If S is conflict serializable, then an *equivalent serial order* is any (irreflexive) total order $\prec$ of $\mathsf{TransOf}\langle S\rangle$ for which $T_1 \prec T_2$ implies that there is no directed path in $\mathsf{DSG}\langle S\rangle$ from T_2 to T_1. Less formally, if there is an edge of the form $T_1 \xrightarrow{\mathsf{zz}} T_2$, with $\mathsf{zz} \in \{\mathsf{rw}, \mathsf{ww}, \mathsf{wr}\}$, then T_1 must precede T_2 in any equivalent serial order. The order $\prec$ is *commit-order preserving* if for every distinct pair $\{T_1, T_2\} \subseteq \mathsf{TransOf}\langle S\rangle$, $t_{\mathsf{End}}\langle T_1\rangle < t_{\mathsf{End}}\langle T_2\rangle$ implies $T_1 \prec T_2$.

2.8 The temporal sense of edges Given a schedule S, an edge $T_1 \xrightarrow{\mathsf{zz}} T_2$ in $\mathsf{DSG}\langle S\rangle$, with $\mathsf{zz} \in \{\mathsf{rw}, \mathsf{ww}, \mathsf{wr}\}$ is called *(temporally) forward* if T_1 commits before T_2 ($t_{\mathsf{End}}\langle T_1\rangle < t_{\mathsf{End}}\langle T_2\rangle$) and *(temporally) backward* if T_2 commits before T_1 ($t_{\mathsf{End}}\langle T_2\rangle < t_{\mathsf{End}}\langle T_1\rangle$). If $T_1 \xrightarrow{\mathsf{zz}} T_2$ is a forward (resp. backward) edge, this may be noted explicitly via $T_1 \xrightarrow{f:\mathsf{zz}} T_2$ (resp. $T_1 \xrightarrow{b:\mathsf{zz}} T_2$). If the type of an edge (rw, ww, or wr) is unimportant, and only its temporal direction is relevant, this may be denoted via $T_1 \xrightarrow{f:\text{-}} T_2$ or $T_1 \xrightarrow{b:\text{-}} T_2$. For $T_1 \xrightarrow{d:\mathsf{zz}} T_2$, with $\mathsf{zz} \in \{\mathsf{rw}, \mathsf{ww}, \mathsf{wr}\}$ and $d \in \{f, b\}$, $d{:}\mathsf{zz}$ is called the *sensed type* of the edge.

Note that a temporally backward edge $T_1 \xrightarrow{b:\mathsf{zz}} T_2$ must always connect concurrent transactions, regardless of the type zz. An edge $T_1 \to T_2$ in which T_2 ends before T_1 begins is never possible in any DSG.

$\mathsf{DSG}\langle S\rangle$ has the *unisense* property if either all edges are temporally forward or else all edges are temporally backward.

2.9 Observation — Consequences of unisense edges *Let S be a schedule over* $\mathsf{Trans}\langle \mathbf{D}\rangle$.

(a) *If* $\mathsf{DSG}\langle S\rangle$ *has the unisense property, then it must be acyclic.*

(b) *If* $\mathsf{DSG}\langle S\rangle$ *is conflict serializable, then there is an equivalent serial order $\prec$ which is commit-order preserving iff* $\mathsf{DSG}\langle S\rangle$ *has the unisense property with all forward edges.*

PROOF: (a) First assume that all edges are forward. Then, for any cycle $T_1 \xrightarrow{f:\text{-}} T_2 \xrightarrow{f:\text{-}} \ldots \xrightarrow{f:\text{-}} T_n \xrightarrow{f:\text{-}} T_1$, it must be the case that $t_{\mathsf{End}}\langle T_1\rangle < t_{\mathsf{End}}\langle T_2\rangle < \ldots t_{\mathsf{End}}\langle T_n\rangle < t_{\mathsf{End}}\langle T_1\rangle$, which is impossible. Thus, no such cycle is possible. The proof for all backward edges is analogous.

(b) This is immediate from the definition of equivalent serial order (see 2.7). □

2.10 Observation — Impossible edges *Let S be a schedule over* $\mathsf{Trans}\langle \mathbf{D}\rangle$.

(a) *In* $\mathsf{DSG}\langle S\rangle$, *edges of sensed type $b{:}\mathsf{ww}$ and $b{:}\mathsf{wr}$ are not possible.*

(b) *In* $\mathsf{DSG}\langle S\rangle$ *with* $\langle T_1,\tau_1\rangle, \langle T_2,\tau_2\rangle \in S$ *distinct, if* T_2 *uses* RBWE *in* S*; i.e., if* $\tau_2 = \mathsf{RBWE}$*, then no edge of the form* $T_1 \xrightarrow{f:\mathsf{wr}} T_2$ *is possible when* $T_1 \parallel T_2$*.*

PROOF: (a) Let $\langle T_1,\tau_1\rangle, \langle T_2,\tau_2\rangle \in S$. For an edge of the form $T_1 \xrightarrow{-:\mathsf{ww}} T_2$ (resp. $T_1 \xrightarrow{-:\mathsf{wr}} T_2$) to exist in $\mathsf{DSG}\langle S\rangle$, it must be the case that $t^{\tau_1}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle < t^{\tau_2}_{\mathsf{Write}\langle T_2\rangle}\langle x\rangle$ (resp. $t^{\tau_1}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle < t^{\tau_2}_{\mathsf{Read}\langle T_2\rangle}\langle x\rangle$) for some $x \in \mathsf{WriteSet}\langle T_1\rangle \cap \mathsf{WriteSet}\langle T_2\rangle$ (resp. $x \in \mathsf{WriteSet}\langle T_1\rangle \cap \mathsf{ReadSet}\langle T_2\rangle$). Since $t^{\tau_1}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle = t_{\mathsf{End}}\langle T_1\rangle$ for both $\tau_1 = \mathsf{RRWE}$ and $\tau_1 = \mathsf{RBWE}$, it follows that $t_{\mathsf{End}}\langle T_1\rangle = t^{\tau_1}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle < t^{\tau_2}_{\mathsf{Write}\langle T_2\rangle}\langle x\rangle = t_{\mathsf{End}}\langle T_2\rangle$ (resp. $t_{\mathsf{End}}\langle T_1\rangle = t^{\tau_1}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle < t^{\tau_2}_{\mathsf{Read}\langle T_2\rangle}\langle x\rangle < t_{\mathsf{End}}\langle T_2\rangle$); *i.e.*, that T_1 commits before T_2, making any such edge forward.

(b) For an edge of the form $T_1 \xrightarrow{f:\mathsf{wr}} T_2$ to exist in $\mathsf{DSG}\langle S\rangle$, there must be an $x \in \mathsf{WriteSet}\langle T_1\rangle \cap \mathsf{ReadSet}\langle T_2\rangle$ with $t^{\tau_1}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle < t^{\tau_2}_{\mathsf{Read}\langle T_2\rangle}\langle x\rangle$. Since $\tau_2 = \mathsf{RBWE}$, $t^{\tau_2}_{\mathsf{Read}\langle T_2\rangle}\langle x\rangle = t_{\mathsf{Start}}\langle T_2\rangle$, which implies that $t_{\mathsf{End}}\langle T_1\rangle = t^{\tau_1}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle < t^{\tau_2}_{\mathsf{Read}\langle T_2\rangle}\langle x\rangle = t_{\mathsf{Start}}\langle T_2\rangle$, and so T_1 and T_2 are not concurrent. □

3 Concurrency-Based Isolation Levels

In this section, local isolation levels, also called *concurrency-based isolation levels*, are examined in detail, with a focus on how transactions run at different levels of isolation relate to each other. Although the study is formal, the properties of fundamental variants such as read committed (RC) and snapshot isolation (SI) are based upon the way that corresponding levels behave in PostgreSQL [13].

3.1 Concurrency-based properties of a transaction Informally, a *concurrency-based property* (also called *local property*) of a Teff-transaction is one which is based only upon the properties of that transaction, and how it relates to those other transactions in a schedule S with which it is concurrent. Three main ways of characterizing such properties are the following.

Locks: Lock-based characterization of isolation was the first to be studied systematically [8], [9, Sec. 7.6]. More modern approaches, including S2PL and SS2PL [3], were developed subsequently; however, these approaches have fallen out of favor with the rise of MVCC.

Anomalies: The main ideas (*dirty read*, *lost update*, *phantom*) are developed in the early lock-based approach of [8], and are also used in the SQL standard [10, Part 2]. They are somewhat tied to the older SVCC, and due to a lack of rigorous definition, are also subject to multiple interpretations [2].

DSG: In this approach, the properties are based upon edges between concurrent transactions in the DSG. It is well suited to the modern MVCC architecture, providing clean, direct characterizations of isolation levels such as snapshot isolation (SI).

In this work, the focus is upon DSG-based characterization, since the study of serialization in Sec. 4 is based upon it, and a systematic investigation does not appear to have been conducted previously.

3.2 Winner and loser transactions Let $S \in \mathsf{Sched}\langle\mathbf{D}\rangle$. As shown in 2.10, edges of (sensed) type b:ww and b:wr are never possible in a $\mathsf{DSG}\langle S\rangle$ when all transactions have effective time assignment RRWE or RBWE. Since those are the only effective time assignments considered, such edge types will not be considered further. Of the four remaining types, f:rw, b:rw, f:ww, and f:wr, any such edge has a *winner* and a *loser*. For all edges of type rw or wr, regardless of sense, the winner is always the first committer. Specifically, in the case of an edge $T_1 \xrightarrow{b:\mathsf{rw}} T_2$, the winner is T_2, while for $T_1 \xrightarrow{f:\mathsf{rw}} T_2$ and $T_1 \xrightarrow{f:\mathsf{wr}} T_2$, the winner is T_1.

For edges of type f:ww, there are two principal variants, *first-committer wins* (FCW) and *first-updater wins* (FUW). With FCW, the winner is the transaction which commits first wins, exactly as for the other three types of edges. With FUW, it is the first transaction which declares a write, according to request, not effective times, which wins. The situation is a bit complex, since there may be a (nonempty) set $X \subseteq \mathsf{DObj}\langle \mathbf{D} \rangle$ which each transaction writes. It is the first writer over all such data objects which wins. Formally, for an edge $T_1 \xrightarrow{f:\mathsf{ww}} T_2$, transaction T_1 wins if $\min(\{t^{\mathsf{Req}}_{\mathsf{Write}\langle T_1 \rangle}\langle x \rangle \mid x \in X\}) < \min(\{t^{\mathsf{Req}}_{\mathsf{Write}\langle T_2 \rangle}\langle x \rangle \mid x \in X\})$; otherwise, T_2 wins.

The choice of FCW or FUW is a system-wide policy, since it must be applied to pairs of transactions. Most existing systems use FUW, although Pyrrho [5] (see also 4.7) is a notable exception.

3.3 General local DSG-based isolation levels A local DSG-based isolation level for a transaction T is defined by three items, the effective time assignment used by T, a set of sensed edge types, and a read-only status. Formally, recall from 2.4 that $\mathsf{TASetEff} = \{\mathsf{RRWE}, \mathsf{RBWE}\}$, and let $\mathsf{CEdges} = \{f{:}\mathsf{rw}, b{:}\mathsf{rw}, f{:}\mathsf{ww}, f{:}\mathsf{wr}\}$, $\mathsf{RWmode} = \{\mathsf{RW}, \mathsf{RO}\}$. Then, define an *isolation-policy triple* to be an ordered triple $\langle \tau, \Delta, \mu \rangle$ with $\tau \in \mathsf{TASetEff}$, $\Delta \subseteq \mathsf{CEdges}$, and $\mu \in \mathsf{RWmode}$. In $\langle \tau, \Delta, \mu \rangle$, τ identifies the effective time assignment used by the transaction, Δ identifies the types of concurrent edges which are forbidden or impossible to loser transactions, and μ indicates whether the transaction is read-write or read-only. The set of all isolation-policy triples over $\mathbf{D}$ is denoted $\mathsf{PolTr}\langle \mathbf{D} \rangle$. A *local DSG-based isolation level* is defined by such a triple.

In general, a loser transaction with a forbidden edge type must abort in order to satisfy the isolation level. It is very important to understand why only loser transactions may forbid edge types. Consider, for example, an edge of the form $T_1 \xrightarrow{b:\mathsf{rw}} T_2$ in $\mathsf{DSG}\langle S \rangle$. According to the conditions spelled out in 3.2, T_2 is the winner and T_1 is the loser because T_2 commits first (backward edge). Now let $x \in \mathsf{ReadSet}\langle T_1 \rangle \cap \mathsf{WriteSet}\langle T_2 \rangle$. At the time at which T_2 commits, it may not be known that T_1 intends to read x; *i.e.*, it may be the case that $t^{\mathsf{Req}}_{\mathsf{Write}\langle T_2 \rangle}\langle x \rangle \leq t_{\mathsf{End}}\langle T_2 \rangle < t^{\mathsf{Req}}_{\mathsf{Read}\langle T_1 \rangle}\langle x \rangle$. Since committed transactions cannot be rolled back, there is no reasonable way that such a policy could be enforced, other than by delaying the commit of T_2. As such delays are not part of the model, it is impossible for the winner to enforce an edge-prohibition policy.

3.4 Named DSG-based isolation levels Using the notion of concurrency-based property of 3.1, eight named isolation levels are summarized in Table 1. Column 2 indicates the effective time assignment used, while columns 3-6 indicate the status of members of CEdges for that policy, with "P" indicating that the edge type is prohibited for the loser transaction, "X" indicating that it is impossible for the loser transaction to have such an edge under the indicated policy, and blank indicating allowed. These policies are discussed in detail, including the meaning of the abbreviations, in 3.5, 3.6, and 3.7.

Policy	Eff Time Assign	Status conc edge type				RW Mode	Used in practice?
		f:rw	b:rw	f:ww	f:wr		
RC	RRWE					RW	Y
RCX	RRWE		P			RW	?
SI	RBWE			P	X	RW	Y
SIX	RBWE		P	P	X	RW	Y
RCRO	RRWE	X		X		RO	Y
RCXRO	RRWE	X	P	X		RO	?
SIRO	RBWE	X		X	X	RO	Y
SIXRO	RBWE	X	P	X	X	RO	Y

Table 1: Concurrency properties of transaction classes

3.5 RRWE-based isolation levels The fundamental RRWE-based isolation level is *read committed* RC. Its representation as a policy triple is $\langle \mathsf{RRWE}, \emptyset, \mathsf{RW}\rangle$. This may be taken as the definition of the name; thus $\mathsf{RC} = \langle \mathsf{RRWE}, \emptyset, \mathsf{RW}\rangle$. In accordance with RRWE, all reads are performed at request time, while writes are performed at the end of the transaction. There are no further restrictions on allowable edges of the DSG. RC is very common isolation level in real systems, usually offered via the `READ COMMITTED` SQL isolation level.

Although not widely used in real systems, an important theoretical variant of RC for this work is *read-committed with excluded backward dependencies*, or $\mathsf{RCX} = \langle \mathsf{RRWE}, \{b{:}\mathsf{rw}\}, \mathsf{RW}\rangle$. It differs from RC only in that backward rw-edges are not allowed, subject, of course, to the general limitation that only a loser transaction may prohibit an edge. As will be seen in 4.6, it is the weakest local isolation level which guarantees serializability of schedules.

3.6 RBWE-based isolation levels The fundamental RBWE-based isolation level is *snapshot isolation* $\mathsf{SI} = \langle \mathsf{RBWE}, \{f{:}\mathsf{ww}, f{:}\mathsf{wr}\}, \mathsf{RW}\rangle$. All effective reads are performed at the beginning of the transaction, while writes are performed at the end. SI is very common isolation level in real systems, often offered using the `REPEATABLE READ`[2] or `SERIALIZABLE` SQL isolation level.

For the reader who has learned that concurrent writes are prohibited under SI, it may seem strange that forward ww-edges are allowed for the winner. To understand this better, consider an edge $T_1 \xrightarrow{f:\mathsf{ww}} T_2$ in the DSG, with $T_1 \parallel T_2$, and suppose that $x \in \mathsf{WriteSet}\langle T_1\rangle \cap \mathsf{WriteSet}\langle T_2\rangle$. If both T_1 and T_2 run with isolation SI, then since only one of them can be the winner (as defined in 3.2), the edge is not allowed. However, suppose that T_1 runs under SI but T_2 runs under RC (or RCX) isolation. If FCW is used for conflict resolution, then since T_1 commits first, it is the winner. Although T_2 is the loser, its isolation level permits concurrent writes. As it writes x after T_1 commits, that write is completely outside of the lifetime of T_1. If conflicts are resolved via FUW, then either T_1 or T_2 may be the winner. However, even if the winner runs under SI, if the loser runs under RC or RCX, then by a similar argument, both transactions will write x. One must be very careful when asserting that concurrent writes are prohibited under SI when characterizing a mixed-level setting. A transaction running under RC plays by different rules

[2]Strictly speaking, SI does not provide `READ COMMITTED` isolation. See [2, Remark 9] for details.

than one running under SI; the SI transaction cannot impose its rules on its RC neighbor.

Note, however, that $T_1 \xrightarrow{f:\mathsf{wr}} T_2$ is impossible when the loser transaction (which must be T_2) runs under SI, since with RBWE reading from a concurrent transaction cannot occur.

The level *snapshot isolation with backward rw exclusion* is $\mathsf{SIX} = \langle \mathsf{RBWE}, \{b{:}\mathsf{rw}, f{:}\mathsf{ww}, f{:}\mathsf{wr}\}, \mathsf{RW}\rangle$. It is the same as SI, save for that backward rw-edges are not allowed. It bears the same relationship to SI as RCX does to RC. It is the sole mode of isolation of Pyrrho, described in 4.7. As a simple example, suppose that, in schedule S, T_1 running under SIX reads x and writes y, so $t_{\mathsf{Start}}\langle T_1\rangle = t^{\mathsf{RBWE}}_{\mathsf{Read}\langle T_1\rangle}\langle x\rangle < t^{\mathsf{RBWE}}_{\mathsf{Write}\langle T_1\rangle}\langle y\rangle = t_{\mathsf{End}}\langle T_1\rangle$, and T_2, also running under SIX, writes x, so $t_{\mathsf{Start}}\langle T_2\rangle < t^{\mathsf{RBWE}}_{\mathsf{Write}\langle T_2\rangle}\langle x\rangle = t_{\mathsf{End}}\langle T_1\rangle$. Then $T_1 \xrightarrow{d:\mathsf{rw}} T_2$ in $\mathsf{DSG}\langle S\rangle$. If $d = f$; *i.e.*, if the edge is forward, then both transactions may commit. However, if $d = b$; *i.e.*, if the edge is backward, then the loser must abort. Under FCW, as is the case in Pyrrho (see 4.7), this loser is always T_1.

3.7 Read-only transactions Since it is possible to define a transaction to be read only in SQL, a read-only mode is also supported in the isolation model presented here. $\mathsf{RCRO} = \langle \mathsf{RRWE}, \{f{:}\mathsf{rw}, f{:}\mathsf{ww}\}, \mathsf{RO}\rangle$ is essentially the same as RC with read-only mode enabled. Similarly, $\mathsf{SIRO} = \langle \mathsf{RBWE}, \{f{:}\mathsf{rw}, f{:}\mathsf{ww}, f{:}\mathsf{wr}\}, \mathsf{RO}\rangle$ is essentially the same as SI, with read-only mode enabled. Analogously, $\mathsf{RCXRO} = \langle \mathsf{RRWE}, \{f{:}\mathsf{rw}, b{:}\mathsf{rw}, f{:}\mathsf{ww}\}, \mathsf{RO}\rangle$ and $\mathsf{SIXRO} = \langle \mathsf{RBWE}, \{f{:}\mathsf{rw}, b{:}\mathsf{rw}, f{:}\mathsf{ww}, f{:}\mathsf{wr}\}, \mathsf{RO}\rangle$.

Observe that an edge of the form $T_1 \xrightarrow{f:\mathsf{rw}} T_2$ is not possible if the loser (which must be T_2) is read only.

3.8 Ordering of policy triples Policy triples admit a natural ordering. For $\mathsf{TASetEff}$, use the order $\mathsf{RRWE} < \mathsf{RBWE}$, and for RWmode, use the order $\mathsf{RW} < \mathsf{RO}$. Then define $\langle \tau_1, \Delta_1, \mu_1\rangle \leq \langle \tau_2, \Delta_2, \mu_2\rangle$ iff $\tau_1 \leq \tau_2$, $\Delta_1 \subseteq \Delta_2$, and $\mu_1 \leq \mu_2$. The idea is that lesser policies in this ordering correspond to lower levels of isolation. The intuition behind the ordering on $\mathsf{TASetEff}$ is that RBWE imposes more constraints than does RRWE. For example, even under RRWE, a transaction T could perform all of its reads at the very beginning; this would be the case if $t^{\mathsf{Req}}_{\mathsf{Read}\langle T\rangle}\langle x\rangle = t_{\mathsf{Start}}\langle T\rangle$ for every $x \in \mathsf{ReadSet}\langle T\rangle$. Similarly, the intuition behind the ordering on RWmode is that prohibiting writes is a stronger condition than allowing them. Finally, it is clear that prohibiting (or rendering impossible) more types of edges results in a more restrictive policy. For the set $\mathsf{CBIso} = \{\mathsf{RC}, \mathsf{RCX}, \mathsf{SI}, \mathsf{SIX}, \mathsf{RCRO}, \mathsf{SIRO}, \mathsf{RCXRO}, \mathsf{SIXRO}\}$, $\mathsf{RC} < \mathsf{RCX} < \mathsf{SIX} < \mathsf{SIXRO}$, $\mathsf{RC} < \mathsf{SI} < \mathsf{SIX}$, $\mathsf{RC} < \mathsf{RCRO} < \mathsf{SIRO}$, $\mathsf{SI} < \mathsf{SIRO}$, and $\mathsf{RCX} < \mathsf{RCXRO}$.

4 Multiscope Serializable Isolation

In this section, the main ideas of multiscope serializable isolation are developed.

4.1 Transactions with isolation A *transaction with isolation* is an ordered pair $\langle T, \iota\rangle$ in which $T \in \mathsf{Trans}\langle \mathbf{D}\rangle$ and ι is a local DSG-based isolation level. The isolation level ι may be represented either as a member of CBIso, or else as a policy triple. Thus, $\langle T, \mathsf{RCX}\rangle$ and $\langle T, \langle \mathsf{RRWE}, \{b{:}\mathsf{rc}\}, \mathsf{RWmode}\rangle\rangle$ have exactly the same meaning. The set of all transactions with isolation over $\mathbf{D}$ is denoted $\mathsf{TransIso}\langle\mathbf{D}\rangle$. A transaction with isolation $\langle T, \iota\rangle$ carries strictly more information than a transaction with effective time assignment $\langle T, \tau\rangle$. For $\iota = \langle\tau, \Delta, \mu\rangle \in \mathsf{PolTr}$, define $\pi_{\mathsf{TASetEff}}\langle\iota\rangle = \tau$; then $\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle = \langle T, \tau\rangle$ is the associated transaction with effective time assignment.

4.2 Schedule augmentation strategies When a transaction is ready to commit, a test must be made to determine whether that commit should be allowed. If so, it is added to the set of committed transactions. If not, it must be rejected. To formalize this, begin by defining $\langle S, \langle T, \iota\rangle\rangle$ with $S \in \mathsf{SchedIso}\langle\mathbf{D}\rangle$ and $\langle T, \iota\rangle \in \mathsf{TransIso}\langle\mathbf{D}\rangle$ to be an *augmentation pair* over $\mathbf{D}$ if adding $\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle$ to S results in a schedule with the property that each transaction in S must either have committed before T, or else run concurrently with T: for every $T' \in \mathsf{TransOf}\langle S\rangle$, one of $t_{\mathsf{End}}\langle T'\rangle < t_{\mathsf{End}}\langle T\rangle$ or $T \parallel T'$ must hold. Think of S as the collection of existing transactions, with $\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle$ a candidate to be added to S. An *(augmentation) test routine* is a function $\alpha : \mathsf{AugPr}\langle\mathbf{D}\rangle \to \{0, 1\}$, with $\langle S, \langle T, \iota\rangle\rangle \mapsto 1$ indicating that $\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle$ should commit and be added to S, and $\langle S, \langle T, \iota\rangle\rangle \mapsto 0$ indicating that it should not.

A central example is $\mathsf{AugTest}_{b:\mathsf{rw}}$, defined on elements by $\langle S, \langle T, \iota\rangle\rangle \mapsto 1$ iff $\mathsf{DSG}\langle S \cup \{\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle\}\rangle$ does not contain an edge of the form $T \xrightarrow{b:\mathsf{rw}} T'$ or $T' \xrightarrow{b:\mathsf{rw}} T$ for a $\langle T', \tau'\rangle \in S$, with T the loser transaction for that edge. Another is $\mathsf{AugTest}_{\mathsf{PolTr}}$, defined on elements by $\langle S, \langle T, \iota\rangle\rangle \mapsto 1$ iff $\mathsf{DSG}\langle S \cup \{\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle\}\rangle$ does not contain any edges involving T which are forbidden by ι. More precisely, if $\iota = \langle\tau, \Delta, \mu\rangle$, then no new edge of a type in Δ is allowed in the case that T is the loser transaction associated with that edge. Finally, for $\kappa \in \mathsf{PolTr}\langle\mathbf{D}\rangle$, $\mathsf{AugTest}_{\geq\kappa}$ is defined on elements by $\langle S, \langle T, \iota\rangle\rangle \mapsto 1$ iff $\iota \geq \kappa$ and $\mathsf{AugTest}_{\mathsf{PolTr}}(\langle T, \iota\rangle) = 1$. Thus, $\mathsf{AugTest}_{\geq\kappa}$ allows $\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle$ to be added to S iff ι provides DSG-based isolation at level κ or greater, and adding $\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle$ to $\mathsf{DSG}\langle S\rangle$ does not result in new edges which are forbidden for T by ι.

These examples are *local* in scope in that the test conditions depend only upon the transaction $\langle T, \iota\rangle$ to be added and certain properties of those transactions in S which run concurrently with it. The reference routine for serialization, which is *global* in scope, is $\mathsf{AugTest}_{\mathsf{DSG}}$, defined on elements by $\langle S, \langle T, \iota\rangle\rangle \mapsto 1$ iff $\mathsf{DSG}\langle S \cup \{\langle T, \iota\rangle\}\rangle$ does not contain any cycles which include T. Other examples which are not local in scope are considered in 4.10 and 4.12.

Processing a sequence of transactions, in order to build a schedule, is formalized as follows. An *ordered schedule* over $\mathbf{D}$ is a sequence $C = \langle\langle T_1, \iota_1\rangle, \langle T_2, \iota_2\rangle, \ldots, \langle T_k, \iota_k\rangle\rangle$ with the properties that $\{\langle T_i, \pi_{\mathsf{TASetEff}}\langle\iota_i\rangle\rangle \mid 1 \leq i \leq k\} \in \mathsf{Sched}\langle\mathbf{D}\rangle$ and for $1 \leq i < j \leq k$, one of $t_{\mathsf{End}}\langle T_i\rangle < t_{\mathsf{End}}\langle T_j\rangle$ or $T_i \parallel T_j$ must hold. The *stepwise commit-based DSG construction* of S using α begins with the empty schedule $\emptyset$, and adds, in the order specified by C, each pair of the form $\langle T_i, \pi_{\mathsf{TASetEff}}\langle\iota_i\rangle\rangle$ which α classifies as acceptable. Formally, $\mathsf{Step}\langle C, \alpha, 0\rangle = \emptyset$;
$\mathsf{Step}\langle C, \alpha, i+1\rangle = \mathsf{Step}\langle C, \alpha, i\rangle \cup \{\langle T_{i+1}, \pi_{\mathsf{TASetEff}}\langle\iota_{i+1}\rangle\rangle\}$
if $\alpha(\langle\mathsf{Step}\langle C, \alpha, i\rangle, \langle T_{i+1}, \pi_{\mathsf{TASetEff}}\langle\iota_{i+1}\rangle\rangle\rangle) = 1$;
$\mathsf{Step}\langle C, \alpha, i+1\rangle = \mathsf{Step}\langle C, \alpha, i\rangle$ otherwise.

4.3 FUW and delayed commit In the formalism of 4.2, if adding $\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle$ to S results in a forbidden edge of the form $T' \xrightarrow{-:-} T$ or $T \xrightarrow{-:-} T'$, then T is not permitted to commit. With FUW, it may be the case that T' has not yet committed when the test is performed. To maximize concurrency, many systems will suspend T until it is known whether or not T' commits. If T'

does not commit, T may continue. Although space limitations preclude formalizing this idea (which involves introducing suspendable transactions with flexible time points), omitting it does not alter the main results developed here. In any case, this issue does not arise with FCW. Indeed, with FCW, all transactions in S will have committed before T.

4.4 Serial properties of augmentation strategies An augmentation test routine α is *serializable generating* (abbreviated SerGen) if for any ordered schedule C of length k, $\mathsf{Step}\langle C, \alpha, k\rangle$ is conflict serializable. Thus, it produces serializable schedules when only transactions which pass its test are allowed. This is the global-scope meaning of serializability, as intended in the `SERIALIZABLE` isolation level of SQL. The routine α is *commit-order* SerGen if it is SerGen and some equivalent serial order is commit-order preserving.

The routine α is *serializable preserving* (abbreviated $\mathsf{SerPres}$) if for any augmentation pair $\langle S', \langle T, \iota\rangle\rangle$ over $\mathbf{D}$ with $\alpha(\langle S', \langle T, \iota\rangle\rangle) = 1$, T does not participate in any cycle of $S' \cup \{\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle\}$. In particular, if S' is conflict serializable, then so too is $S' \cup \{\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle\}$. Observe that $\mathsf{SerPres}$ always implies SerGen.

In contrast to SerGen, the property of $\mathsf{SerPres}$ is local in scope; it does not depend upon properties of the extant schedule S, except those which result from concurrency of its transactions with $\langle T, \tau\rangle$. (Note that S' is universally quantified in the definition of $\mathsf{SerPres}$; it can be the result of running and committing transactions at any level of isolation.) Thus, as elaborated in 5.1, $\mathsf{SerPres}$ is an appropriate semantics for `SERIALIZABLE` when applied to a single transaction, since, on the one hand, it provides SerGen behavior when all transactions run at that level, and, on the other hand, it provides a meaningful contribution to serializability even when other transactions run at different levels of isolation.

$\mathsf{AugTest}_{\mathsf{DSG}}$ is always $\mathsf{SerPres}$ (and hence SerGen). The interesting question is whether there are simpler, local isolation levels which also provide these properties. This is established in the affirmative below.

4.5 Theorem — $\mathsf{AugTest}_{b:\mathsf{rw}}$ is both SerGen and $\mathsf{SerPres}$ *The augmentation test* $\mathsf{AugTest}_{b:\mathsf{rw}}$ *is both serializable generating and serializable preserving.*

Proof: Let $\langle S, \langle T, \iota\rangle\rangle \in \mathsf{AugPr}\langle\mathbf{D}\rangle$. For T to be part of a cycle in $S \cup \{\langle T, \pi_{\mathsf{TASetEff}}\langle\iota\rangle\rangle\}$, it must have at least one outgoing edge. Since it is the last transaction to commit, that outgoing edge must be backward. However, $\mathsf{AugTest}_{b:\mathsf{rw}}$ prohibits edges of type b:rw for the loser transaction, and since T commits last, it must be the loser for any rw-edge (see 3.2). Thus, T cannot have outgoing edges of type b:rw. Since outgoing edges of types b:ww and b:wr are not possible (see 2.10(a)), T cannot have any outgoing edges at all, so it cannot be involved in a cycle of $\langle S, \langle T, \iota\rangle\rangle \in \mathsf{AugPr}\langle\mathbf{D}\rangle$. Hence $\mathsf{AugTest}_{b:\mathsf{rw}}$ is $\mathsf{SerPres}$, and so also SerGen. □

4.6 Corollary — $\mathsf{AugTest}_{\geq\mathsf{RCX}}$ *The augmentation test routines* $\mathsf{AugTest}_{\geq\mathsf{RCX}}$ *and* $\mathsf{AugTest}_{\geq\mathsf{SIX}}$ *are both* $\mathsf{SerPres}$ *(and hence* SerGen*), with* $\mathsf{AugTest}_{\geq\mathsf{RCX}}$ *the weakest such test defined by a policy triple.*

Proof: The proof follows immediately from 4.5 and the fact that RCX and SIX prohibit edges of type b:rw. □

4.7 Serialization in Pyrrho The Pyrrho RDBMS [5], [6] employs SIX for its only isolation level. In view of 4.6, it thus provides a working instance of true `SERIALIZABLE` isolation which is based entirely upon a local property of transactions. A unique feature of Pyrrho is that it uses pure FCW for conflict resolution; transactions are never blocked for any reason.

4.8 Example — Wide cursor stability Some RDBMSs offer a feature called *cursor stability* [9, Sec. 7.6.2] as part the isolation level `READ COMMITTED`. Suppose that transaction T_1, running under RC, reads and then later writes data object x. Suppose further that transaction T_2 also runs under RC and also writes x, and commits between the two operations of T_1. Formally, $t^{\mathsf{RRWE}}_{\mathsf{Read}\langle T_1\rangle}\langle x\rangle < t^{\mathsf{RRWE}}_{\mathsf{Write}\langle T_2\rangle}\langle x\rangle < t_{\mathsf{End}}\langle T_2\rangle < t^{\mathsf{RRWE}}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle < t_{\mathsf{End}}\langle T_1\rangle$. This behavior is not serializable because in the serialization $T_1 \prec T_2$, the final write of x is by T_2, not by T_1 as it should be; and in $T_2 \prec T_1$, T_1 does not read the initial value of x, before T_2 wrote it. *Cursor stability* prevents this this "in between" write by T_2, either by locking x between the read and write of T_1, or else by T_1 rereading x after the commit of T_2. However, it does this only when the read and the write of T_1 are part of the same SQL statement. With serialization, this behavior is not permitted, regardless of the "distance" between $t^{\mathsf{RRWE}}_{\mathsf{Read}\langle T_1\rangle}\langle x\rangle$ and $t^{\mathsf{RRWE}}_{\mathsf{Write}\langle T_1\rangle}\langle x\rangle$, since there is an edge $T_1 \xrightarrow{b:\mathsf{rw}} T_2$ which is not allowed RCX or any stronger isolation level. Thus, RCX effectively provides *wide cursor stability*, which does not require the read and the write to be part of the same statement. If the loser runs under an isolation which is $\mathsf{SerPres}$, cursor stability is automatic.

If preservation of commit order is desired in the serialization, then $\mathsf{AugTest}_{b:\mathsf{rw}}$ is actually optimal in the following sense.

4.9 Theorem — Optimality of $\mathsf{AugTest}_{b:\mathsf{rw}}$ $\mathsf{AugTest}_{b:\mathsf{rw}}$ *is a globally optimal commit-order-preserving* SerGen *augmentation test routine, in the precise sense that any other such routine* $\mathsf{AugTest}'$ *with the property that* $\mathsf{AugTest}'(\langle S, \langle T, \tau\rangle\rangle) = 1$ *but* $\mathsf{AugTest}_{b:\mathsf{rw}}(\langle S, \langle T, \tau\rangle\rangle) = 0$ *for some* $\langle S, \langle T, \tau\rangle\rangle \in \mathsf{AugPr}\langle \mathbf{D}\rangle$ *cannot be commit-order preserving.*

PROOF: The proof follows immediately from 2.9(b), since the presence of any backward edge in the DSG implies that commit order must be violated in any serialization. □

4.10 SSI — a preemptive serializable-preserving strategy The SerGen strategy SSI (serializable SI) [4], [7] is used to implement the `SERIALIZABLE` isolation level of PostgreSQL [12]. Define a *dangerous structure* (DS) to be a path in the DSG of the form $T_2 \xrightarrow{-:-} T_1 \xrightarrow{b:\mathsf{rw}} T_0$ in which T_0 commits first and $T_1 \parallel T_2$. (Note that $T_0 \parallel T_1$ is automatic since the edge is backward.) T_0 and T_2 may be the same transaction, in which case $\{T_0, T_1\}$ forms a cycle by itself. As shown in [7, Thm. 2.1], if all transactions run under SI, then every cycle of the DSG contains a DS. To represent this in terms of an augmentation routine, define $\mathsf{AugTest}_{\mathsf{SSI}}$ on elements by $\langle S, \langle T, \iota\rangle\rangle \mapsto 0$ iff T is the last transaction to commit in a DS of $\mathsf{DSG}\langle S\rangle$.

It is worth noting that it is not necessary to require that all transactions run under SI; RC is sufficient. However, a proof will not be presented here; only the original SI-based SSI will be evaluated. Unfortunately, while serializable generating, $\mathsf{AugTest}_{\mathsf{SSI}}$ is not serializable preserving.

4.11 Serialization properties of SSI AugTest$_{\mathsf{SSI}}$ *is* SerGen *but not* SerPres.

Proof: The proof that AugTest$_{\mathsf{SSI}}$ is SerGen is found in [4], [7]. To show that it is not SerPres, it suffices to present a counterexample. In Fig. 1, a DSG cycle $T_0 \xrightarrow{f:\mathsf{rw}} T_4 \xrightarrow{b:\mathsf{rw}} T_3 \xrightarrow{f:\mathsf{rw}} T_2 \xrightarrow{b:\mathsf{rw}} T_1 \xrightarrow{b:\mathsf{rw}} T_0$. consisting of five transaction is shown. Time increases horizontally,

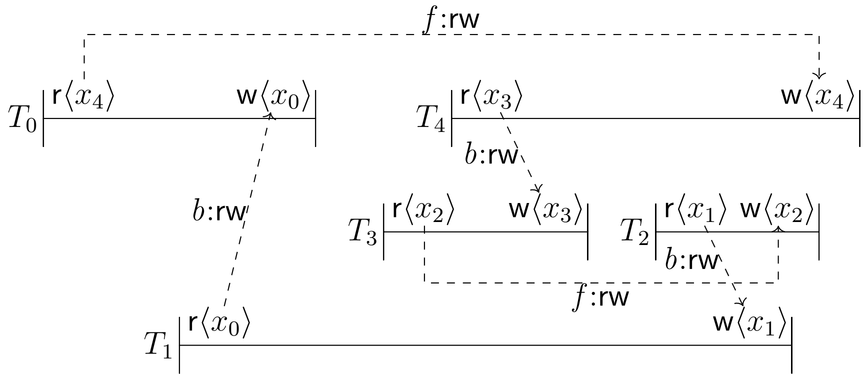

Figure 1: DSG with no DS involving the last transaction to commit

with the beginning and end of each transaction marked by a vertical bar; the commit order is $\langle T_0, T_3, T_1, T_2, T_4\rangle$. The reads and writes of each transaction are depicted by r⟨-⟩ and w⟨-⟩ respectively, Each transaction T_i runs under SSI, so as Teff-transactions, $\langle T_i, \tau_i\rangle = \langle T_i, \mathsf{RBWE}\rangle$. The last transaction to commit, T_4, is not part of any DS. So, letting $S' = \{\langle T_i, \tau_i\rangle \mid 0 \leq i \leq 3\}$, it is immediate that AugTest$_{\mathsf{SSI}}\langle S', \langle T_i, \tau_i\rangle\rangle = 1$. □

4.12 SSN Recently, a preemptive SerGen strategy which relies on a more complex "dangerous structure" than does SSI has been developed [15]. Dubbed *serializable safety net*, or SSN for short, it is of particular relevance to this work in that any local level of isolation which is at least as strong as RC may be serialized, thus reinforcing the observation that local isolation and serialization are orthogonal. As is the case with SSI, SSN is not serializable preserving. While space limitations preclude a full proof, the reader familiar with the construction in [15] can verify easily that the schedule of Fig. 1 provides the necessary counterexample.

5 Conclusions and Further Directions

5.1 Conclusions The semantics of including serializable isolation, global in scope, in a mixed-mode setting with levels of local scope, such as RC and SI, has been investigated. Two alternatives have been identified. In the first, *serializable generating* (SerGen), the serializable level has meaning only when all transactions run at that level. SSI and SSN fall into that category. While highly effective when used exclusively, they revert to a lower level otherwise, with little or no additional benefit. The second is *serializable preserving* (SerPres), which has the property that, regardless of the DSG consisting of all committed transactions, adding a new transaction will never result in a new cycle. Used for `SERIALIZABLE`, it provides a semantics which is both local and global in scope, in line with the original intent of the SQL standard. The augmentation test AugTest$_{\mathsf{DSG}}$ which examines the entire DSG for cycles has this property, although it has large space complexity.[3] Identified in this paper is a far less complex option, in which a standard local complexity level, such as RC or SI, is augmented to disallow all backward rw-dependencies (resulting in RCX or SIX). As elaborated in 5.2, it is proposed that this alternative be explored more thoroughly, as a suitable implementation of SQL `SERIALIZABLE`.

An additional issue arises if SI is used to implement `REPEATABLE READ` in an RDBMS, while RCX is used to implement `SERIALIZABLE`. The unusual (and likely unwanted) situation arises that the two are incomparable as local levels of isolation. Put another way, SI offers higher isolation than RCX in one way — it prohibits concurrent writes, even though it offers lower isolation in another — it permits backward rw-dependencies. This can be remedied by implementing `SERIALIZABLE` as SIX, but it nevertheless shows that complex decisions must be made when enlisting a single isolation level to serve multiple scopes.

5.2 Further directions The following two topics are proposed for further investigations.

Performance measurement for RCX and SIX: Although SIX is used in the Pyrrho system (see 4.7), it has not been compared for performance to alternatives such as SSI (used in PostgreSQL). Since SIX appears to perform well in Pyrrho, it may be the case that although it will have a higher number of false positives (aborted transactions due to concurrency conflicts) than SSI, (since every DS must contain a backward dependency), it may nevertheless be completely satisfactory for many transaction mixes. Advantages of RCX and SIX (over SSI and SSN) include that they are far simpler to implement, and that they provides serializable-preserving isolation. It is thus proposed to study their performance experimentally. In addition, a parallel comparison of FUW and FCW is warranted, given the success of FCW in Pyrrho.

Extension to lock-based approaches: Due to space limitations, the local levels of isolation studied in this paper have been limited to those which are DSG based. However, locked-based levels, such as S2PL and SS2PL, are also of importance, as they are the classical local isolation levels which deliver serializable-preserving behavior. An investigation of how they fit into the framework of this paper is therefore warranted.

References

[1] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In D. B. Lomet and G. Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78, 2000.

[2] 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 *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 1–10, 1995.

[3]It should be noted that one experimental system, called PSSI, has taken exactly the approach of constructing the entire DSG (with all transactions running under SI) to achieve serializable generating behavior, reporting good results [14].

[3] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Trans. Software Eng.*, 17(9):954–960, 1991.

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

[5] M. Crowe. The Pyrrho Database Management System. `https://pyrrhodb.uws.ac.uk/index.htm`. Accessed 2019-03-30.

[6] M. Crowe. Transactions in the Pyrrho database engine. In M. H. Hamza, editor, *IASTED International Conference on Databases and Applications, part of the 23rd Multi-Conference on Applied Informatics, Innsbruck, Austria, February 14-16, 2005*, pages 71–76. IASTED/ACTA Press, 2005.

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

[8] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 365–394. North-Holland, 1976.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[10] J. Melton, editor. *ISO/IEC 9075:2011, Information Technology — Database Languages — SQL.* ANSI, 2011. (The 2011 SQL Standard).

[11] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[12] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endowment*, 5(12):1850–1861, 2012.

[13] PostgreSQL: The World's Most Advanced Open Source Relational Database. `https://www.postgresql.org`. Accessed 2019-03-30.

[14] S. Revilak, P. E. O'Neil, and E. J. O'Neil. Precisely serializable snapshot isolation (PSSI). In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 482–493, 2011.

[15] T. Wang, R. Johnson, A. Fekete, and I. Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.*, 26(4):537–562, 2017.

[16] G. Weikum and G. Vossen. *Transactional Information Systems.* Morgan Kaufmann, 2002.