Concurrency control for database theorists

Bas Ketsman
Vrije Universiteit Brussel, Belgium

Christoph Koch
École Polytechnique Fédérale de Lausanne, Switzerland

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

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

ABSTRACT

The aim of this paper is to serve as a lightweight introduction to concurrency control for database theorists through a uniform presentation of the work on robustness against Multiversion Read Committed and Snapshot Isolation.

1 Introduction

In this paper, we take a simplistic approach and view a transaction as a sequence of reads and writes to database objects. For instance, $T_1 = \mathtt{R}_1[\mathtt{t}]\,\mathtt{W}_1[\mathtt{v}]\,\mathtt{C}_1$ is a transaction that first reads an object t, writes to an object v and then commits. Transactions are considered to be atomic: they are executed completely or not at all, and once committed they can not be rolled back. A transaction workload then consists of a set of transactions. At its core, database concurrency control is a balancing act between two conflicting desires: the wish to increase transaction throughput via concurrent access, that is, interleaving of the execution of transactions, and the desire for data consistency for which concurrent access sometimes needs to be restricted.

The holy standard in concurrency control for guaranteeing data consistency is *serializability.* A concurrent execution of a transaction workload is serializable when it is equivalent to a serial, that is, non-interleaved execution, of the transactions. Serializability guarantees that no data anomalies can occur. There are several concurrency control protocols that guarantee serializability: for instance, *Strict Two-Phase Locking (S2PL)* and *Serializable Snapshot Isolation (SSI).* As these protocols restrict concurrent access and typically have a negative effect on transaction throughput, databases offer a way to trade in data consistency for an increased level of concurrency through the mechanism of isolation levels that are less strict than serializability. Examples of such weaker isolation levels are, for instance, *Multiversion Read Committed (RC)* and *Snapshot Isolation (SI).* These isolation levels are less restrictive but can induce data anomalies and therefore, in general, do not guarantee serializability.

However, there are situations when a group of transactions can be executed at an isolation level lower than serializability without causing any errors. In this way, we get the higher isolation guarantees of serializability for free in exchange for a lower isolation level, which is typically implementable with a less expensive concurrency control mechanism. This formal property is called *robustness* [12, 10]: a set of transactions $\mathcal{T}$ is called robust against a given isolation level if every possible interleaving of the transactions in $\mathcal{T}$ that is allowed under the specified isolation level is serializable. There is a famous example that is part of database folklore: the TPC-C benchmark [16] 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]).

In this paper, we present a gentle introduction to the theory of robustness. In particular, we consider the isolation levels that are offered by systems like Postgres and Oracle: RC, SI and SSI. A main technical tool in the study of robustness is that of a split schedule. It is the canonical form of a counterexample schedule witnessing non-robustness and lies at the basis of polynomial time algorithms for the robustness problem.

A more complete survey and more high level account on robustness can be found in [19]. A much more detailed exposition can be found in Vandevoort's Phd Thesis [17]. For deeper exploration of the theoretical aspects of concurrency control, we refer to the excellent but rather outdated book by Papadimitriou [15]. General references for concurrency control are, e.g., [22, 5, 6], or any recent textbook on database systems.

This paper is further organized as follows. We introduce the necessary terminology on transactions and schedules in Section 2 and discuss serializability in Section 3.2. We define the isolation levels RC, SI, and SSI in Section 4. In Section 5, we consider the robustness problem. We discuss transaction programs and templates in Section 6, and conclude in Section 7.

2 Definitions

2.1 Transactions

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

Formally, we model a transaction as a linear order $(T, \leq_T)$, where T is the set of (read, write and commit) operations occurring in the transaction and $\leq_T$ encodes the ordering of the operations. As usual, we use $<_T$ to denote the strict ordering. For a transaction T, we use *first*(T) to refer to the first operation in T.

When considering a set $\mathcal{T}$ of transactions, we assume that every transaction in the set has a unique id i and write T_i to make this id explicit. Similarly, to distinguish the operations of different transactions, we add this id as a subscript to the operation. That is, we write $\mathtt{W}_i[\mathtt{t}]$ and $\mathtt{R}_i[\mathtt{t}]$ to denote a $\mathtt{W}[\mathtt{t}]$ and $\mathtt{R}[\mathtt{t}]$ occurring in transaction T_i; similarly $\mathtt{C}_i$ denotes the commit operation in transaction T_i. This convention is consistent with the literature (see, *e.g.* [3, 12]). To avoid ambiguity of notation, we assume that a transaction performs at most one write and one read operation per object. The latter is a common assumption (see, *e.g.* [12]). All our results carry over to the more general setting in which multiple writes and reads per object are allowed.

2.2 Schedules

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 a linear ordering of O_s (with $a \leq_s b$ and $b \leq_s a$ meaning $a = b$),
- $\ll_s$ is a *version order* providing for each object $\mathtt{t}$ a total order over all write operations on $\mathtt{t}$ occurring in s, and,
- v_s is a *version function* mapping each read operation a in s to either op_0 or to a write operation in s.

We require that $op_0 \leq_s a$ for every operation $a \in O_s$, $op_0 \ll_s a$ for every write operation $a \in O_s$, and that $a <_T b$ implies $a <_s b$ for every $T \in \mathcal{T}$ and every $a, b \in T$. We furthermore require that for every read operation a, $v_s(a) <_s a$ and, if $v_s(a) \neq op_0$, then the operation $v_s(a)$ is on the same object as a. Intuitively, op_0 indicates the start of the schedule, the order of operations in s is consistent with the order of operations in every transaction $T \in \mathcal{T}$, and the version function maps each read operation a to the operation that wrote the version observed by a. If $v_s(a)$ is op_0, then a observes the initial version of this object. The version order $\ll_s$ represents the order in which different versions of an object are installed in the database. For a pair of write operations on the same object, this version order does not necessarily coincide with $\leq_s$. For example, under RC and SI the version order is based on the commit order instead.

We say that a schedule s is a *single version schedule* if $\ll_s$ agrees 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 [12, 4] 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.

Example 2.1. *As a running example, consider* *the set of transactions $\mathcal{T} = \{T_1, T_2, T_3\}$ with*

$$\begin{aligned} T_1 &= \mathtt{R}_1[\mathtt{t}]\,\mathtt{W}_1[\mathtt{v}]\,\mathtt{C}_1; \\ T_2 &= \mathtt{R}_2[\mathtt{v}]\,\mathtt{W}_2[\mathtt{q}]\,\mathtt{C}_2;\ \textit{and,} \\ T_3 &= \mathtt{R}_3[\mathtt{q}]\,\mathtt{W}_3[\mathtt{t}]\,\mathtt{W}_3[\mathtt{q}]\,\mathtt{C}_3 \end{aligned}$$

Let s_1 be the schedule over $\mathcal{T}$ where the ordering $\leq_{s_1}$ of operations is

$$op_0\,\mathtt{R}_3[\mathtt{q}]\,\mathtt{W}_3[\mathtt{t}]\,\mathtt{R}_1[\mathtt{t}]\,\mathtt{W}_1[\mathtt{v}]\,\mathtt{C}_1\,\mathtt{R}_2[\mathtt{v}]\,\mathtt{W}_2[\mathtt{q}]\,\mathtt{C}_2\,\mathtt{W}_3[\mathtt{q}]\,\mathtt{C}_3.$$

The version order $\ll_{s_1}$ equals

- *$op_0 \ll_{s_1} \mathtt{W}_3[\mathtt{t}]$ for object $\mathtt{t}$,*
- *$op_0 \ll_{s_1} \mathtt{W}_1[\mathtt{v}]$ for object $\mathtt{v}$, and,*
- *$op_0 \ll_{s_1} \mathtt{W}_2[\mathtt{q}] \ll_{s_1} \mathtt{W}_3[\mathtt{q}]$ for object $\mathtt{q}$.*

Furthermore, the version function v_{s_1} is

$$\{\mathtt{R}_3[\mathtt{q}] \to op_0,\ \mathtt{R}_1[\mathtt{t}] \to \mathtt{W}_3[\mathtt{t}],\ \mathtt{R}_2[\mathtt{v}] \to \mathtt{W}_1[\mathtt{v}]\}.$$

Here, the version order $\mathtt{W}_2[\mathtt{q}] \ll_{s_1} \mathtt{W}_3[\mathtt{q}]$ should be interpreted as transaction T_2 installing a version of $\mathtt{q}$ that should precede the version installed by transaction T_3. Furthermore, $v_{s_1}(\mathtt{R}_1[\mathtt{t}]) = \mathtt{W}_3[\mathtt{t}]$ implies that T_3 observes the initial version of $\mathtt{t}$, whereas T_1 observes the version written by T_3. Notice that s_1 is a single version schedule, as $\ll_{s_1}$ coincides with $\leq_{s_1}$ and according to the version function v_{s_1} each read operation observes the most recently written version.

Next, let s_2 be the schedule where the ordering $\leq_{s_2}$ is equal to $\leq_{s_1}$, but where the version order $\ll_{s_2}$ equals $op_0 \ll_{s_2} \mathtt{W}_3[\mathtt{t}]$ for object $\mathtt{t}$, $op_0 \ll_{s_2} \mathtt{W}_1[\mathtt{v}]$ for object $\mathtt{v}$ and $op_0 \ll_{s_2} \mathtt{W}_3[\mathtt{q}] \ll_{s_2} \mathtt{W}_2[\mathtt{q}]$ for object $\mathtt{q}$, and the version function v_{s_2} is

$$\{\mathtt{R}_3[\mathtt{q}] \to op_0,\ \mathtt{R}_1[\mathtt{t}] \to op_0,\ \mathtt{R}_2[\mathtt{v}] \to \mathtt{W}_1[\mathtt{v}]\}.$$

Contrasting s_1, this schedule s_2 is not a single version schedule. Note in particular that $\mathtt{W}_3[\mathtt{q}] \ll_{s_2} \mathtt{W}_2[\mathtt{q}]$, whereas $\mathtt{W}_2[\mathtt{q}] \leq_{s_2} \mathtt{W}_3[\mathtt{q}]$. That is, the version of $\mathtt{q}$ installed by T_3 should precede the version of T_2, even though this version of T_3 is installed later according to $\leq_{s_2}$. We remark that the latter can for example happen under a timestamp based concurrency protocol if the Thomas Write Rule [11] is applied. Furthermore, the read operation $\mathtt{R}_1[\mathtt{t}]$ does not read the most recent version, as it observes the initial version of $\mathtt{t}$ rather than the more recent version written by $\mathtt{W}_3[\mathtt{t}]$. Schematic representations of schedules s_1 and s_2 are given in Figure 1. □

For ease of readability, we will henceforth only use schematic representations of schedules.

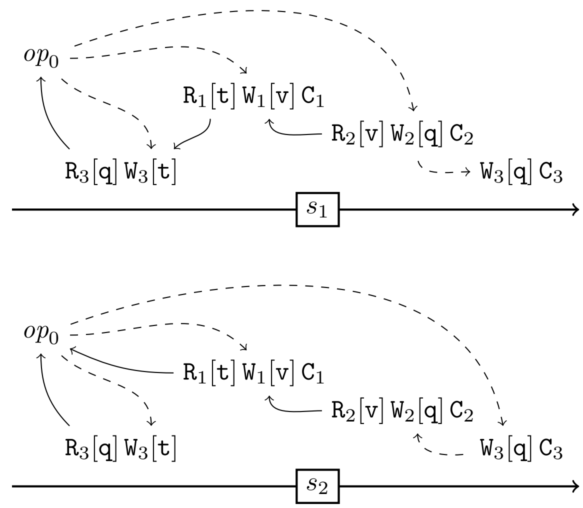


Figure 1: Schedules s_1 and s_2 from Example 2.1 with solid (resp., dashed) arrows representing their version function (resp., version order).

3 Serializability

As explained in the introduction, a schedule is serializable when it is equivalent to a serial schedule. We therefore need to address precisely what equivalence in this context means. Furthermore, the equivalent serial schedule must additionally be single version, as multiversion serial schedules can still exhibit concurrency anomalies.

Example 3.1. *Towards a multiversion serial schedule exhibiting a concurrency issue, consider the set of transactions $\mathcal{T} = \{T_a, T_b\}$ with*

$$\begin{aligned} T_a &= \mathtt{W}_a[\mathtt{t}]\,\mathtt{W}_a[\mathtt{v}]\,\mathtt{C}_a;\ \textit{and,} \\ T_b &= \mathtt{R}_b[\mathtt{t}]\,\mathtt{W}_b[\mathtt{v}]\,\mathtt{C}_b \end{aligned}$$

Let s be the schedule over $\mathcal{T}$ where the ordering $\leq_s$ of operations is

$$op_0\,\mathtt{W}_a[\mathtt{t}]\,\mathtt{W}_a[\mathtt{v}]\,\mathtt{C}_a\,\mathtt{R}_b[\mathtt{t}]\,\mathtt{R}_b[\mathtt{v}]\,\mathtt{C}_b.$$

The version order $\ll_s$ equals

- *$op_0 \ll_s \mathtt{W}_a[\mathtt{t}]$ for object $\mathtt{t}$, and,*
- *$op_0 \ll_s \mathtt{W}_a[\mathtt{v}]$ for object $\mathtt{v}$.*

Furthermore, the version function v_s is

$$\{\mathtt{R}_b[\mathtt{t}] \to op_0,\ \mathtt{R}_b[\mathtt{v}] \to \mathtt{W}_a[\mathtt{v}],\}.$$

Although the schedule s executes T_a before T_b in a serial fashion according to $\leq_s$, the version function v_s implies that T_b observes the original value of $\mathtt{t}$ and the updated value of $\mathtt{v}$. In other words, s

exhibits a concurrency anomaly where T_b observes only a partial update of T_a.

Most of the literature considers conflict serializability even though it is not the most general notion. We define view, final-state, and conflict serializability.

3.1 View and final-state serializability

We start with view equivalence which requires that each read operation reads the result of the same write operation (as defined by the respective version function) in both schedules.

In essence this means that every operation must 'view' the same values in equivalent schedules. We introduce the graph D, to make this explicit.[1] For a schedule s, $D(s)$ has as nodes $O_s \setminus \{op_0\}$ and there is an edge $o \to_D o'$ iff

- $o = \mathtt{R}_i[\mathtt{t}] <_s \mathtt{W}_i[\mathtt{v}] = o'$, that is, o' writes a value that can depend on an earlier read o in the same transaction; or,
- o' reads the value written by o, that is, $o = \mathtt{W}_i[\mathtt{t}]$ and $o' = \mathtt{R}_j[\mathtt{t}]$ with $i \neq j$, and $o = v_s(o')$.

Intuitively, the edge $o \to_D o'$ indicates that o must occur before o' when read dependencies need to be preserved.

The latter leads to the following notion of equivalence. Two schedules s and s' are *view equivalent* if they are over the same set $\mathcal{T}$ of transactions and $D(s) = D(s')$.

We now turn to final-state equivalence which only enforces dependencies for operations that contribute to the final value of at least one object. In other words, dependencies for a write operation to an object that is overwritten without being read, can be discarded. In this context, define $\mathrm{LW}(s) \subseteq O_s$ as those write operations $\mathtt{W}_i[\mathtt{t}]$ that are the *last* in s to *write* $\mathtt{t}$. That is, $\mathtt{W}_i[\mathtt{t}] \in \mathrm{LW}(s)$ iff $\mathtt{W}_i[\mathtt{t}] \in O_s$ and there is no $\mathtt{W}_j[\mathtt{t}] \in O_s$ with $\mathtt{W}_i[\mathtt{t}] <_s \mathtt{W}_j[\mathtt{t}]$. Now define D_1 as the graph obtained from D by removing every connected component (in D) not containing a write operation from $\mathrm{LW}(s)$

Two schedules s and s' are *final-state equivalent* if they are over the same set $\mathcal{T}$ of transactions and $D_1(s) = D_1(s')$.

DEFINITION 3.1. *A schedule s is* final-state serializable *(resp.,* view serializable*) if it is final-state equivalent (resp., view equivalent) to a single version serial schedule.*

THEOREM 3.1. *[15] Deciding whether a schedule s is final-state or view serializable is* coNP*-complete.*

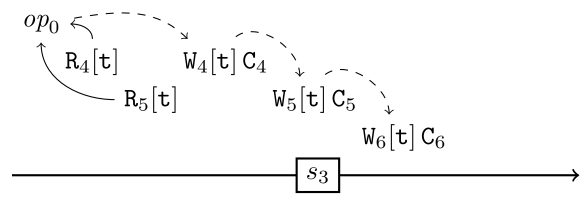
[1]The notation D and D_1 comes from [15].

Notice that schedules that are view serializable are also final-state serializable, but not vice versa, as the next example shows.

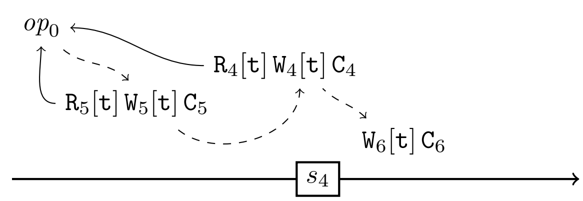
EXAMPLE 3.2. *We consider the set of transactions $\mathcal{T} = \{T_4, T_5, T_6\}$ with*

$$\begin{aligned} T_4 &= \mathtt{R}_4[\mathtt{t}]\,\mathtt{W}_4[\mathtt{t}]\,\mathtt{C}_4; \\ T_5 &= \mathtt{R}_5[\mathtt{t}]\,\mathtt{W}_5[\mathtt{t}]\,\mathtt{C}_5;\ \textit{and,} \\ T_6 &= \mathtt{W}_6[\mathtt{t}]\,\mathtt{C}_6, \end{aligned}$$

and the single-version schedule s_3 over $\mathcal{T}$.



Schedule s_3 is final-state serializable because graph $D_1(s_3) = (O_{s_3}, \emptyset) = D_1(s_4)$ with s_4 being the following single-version serial schedule over $\mathcal{T}$.



We note that s_3 is not view-serializable because in every single-version serial schedule s over $\mathcal{T}$, $D(s)$ must have at least one of the following edges: $\mathtt{W}_5[\mathtt{t}] \to \mathtt{R}_4[\mathtt{t}]$, $\mathtt{W}_4[\mathtt{t}] \to \mathtt{R}_5[\mathtt{t}]$, $\mathtt{W}_6[\mathtt{q}] \to \mathtt{R}_4[\mathtt{q}]$ or $\mathtt{W}_6[\mathtt{t}] \to \mathtt{R}_5[\mathtt{t}]$. $D(s_3)$ has no such edge. □

3.2 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 $b_i \to_s a_j$ if:[2]

[2]Throughout the paper, we adopt the following convention: a b operation can be understood as a 'before' while an a can be interpreted as an 'after'.

- *(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.

EXAMPLE 3.3. *Consider schedule s_2 as defined in Example 2.1. In this schedule, the dependency $\mathtt{W}_3[\mathtt{q}] \rightarrow_{s_2} \mathtt{W}_2[\mathtt{q}]$ is a ww-dependency since $\mathtt{W}_3[\mathtt{q}] \ll_{s_2} \mathtt{W}_2[\mathtt{q}]$. Schedule s_2 furthermore has a wr-dependency from $\mathtt{W}_1[\mathtt{v}]$ to $\mathtt{R}_2[\mathtt{v}]$, as $v_{s_2}(\mathtt{R}_2[\mathtt{v}]) = \mathtt{W}_1[\mathtt{v}]$. The dependency $\mathtt{R}_1[\mathtt{t}] \rightarrow_{s_2} \mathtt{W}_3[\mathtt{t}]$ is a rw-antidependency, witnessed by $v_{s_2}(\mathtt{R}_1[\mathtt{t}]) = op_0 \ll_{s_2} \mathtt{W}_3[\mathtt{t}]$.* □

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

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

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

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

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

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

THEOREM 3.2 (IMPLIED BY [1]). *A schedule s is conflict serializable iff $SeG(s)$ is acyclic.*

The previous Theorem essentially extends the well known characterization of conflict serializability for single version schedules based on acyclicity of conflict graphs (see, e.g., [15]) towards multiversion schedules. In brief, the conflict graph $CG(s)$ for a single version 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 is conflicting with an operation b_i in T_i and $a_i <_s b_j$. Note in particular that $CG(s)$ is defined solely in terms of conflicting operations and $<_s$, whereas $SeG(s)$ takes into account $\ll_s$ and v_s as well. It can be proven that, if s is a single version schedule, $CG(s)$ and $SeG(s)$ are identical.

COROLLARY 3.1. *Deciding whether a schedule s is conflict serializable is in* PTIME.

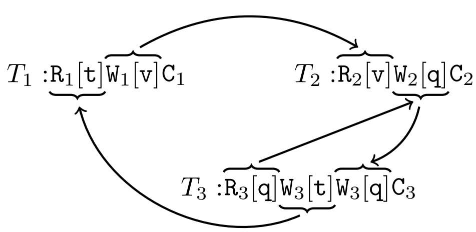
EXAMPLE 3.4. *The serialization graphs for schedules s_1 and s_2 in Example 2.1 are given in Figure 2. Since $SeG(s_1)$ contains cycles, we conclude that s_1 is not conflict serializable. The serialization graph $SeG(s_2)$ on the other hand is acyclic, thereby implying that s_2 is conflict serializable. Indeed, s_2 is conflict equivalent to the single version serial schedule $T_1 \cdot T_3 \cdot T_2$.* □

Notice that conflict serializability implies view serializability but not vice versa.

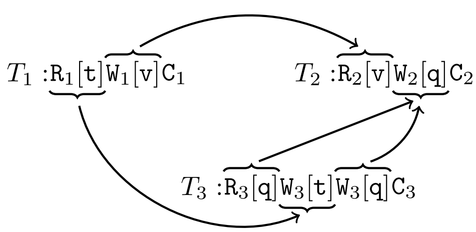
EXAMPLE 3.5. *We consider the set of transactions $\mathcal{T} = \{T_4, T_6, T_7\}$ with*

$$\begin{aligned} T_4 &= \mathtt{R}_4[\mathtt{t}]\,\mathtt{W}_4[\mathtt{t}]\,\mathtt{C}_4; \\ T_6 &= \mathtt{W}_6[\mathtt{t}]\,\mathtt{C}_6; \text{ and,} \\ T_7 &= \mathtt{W}_7[\mathtt{t}]\,\mathtt{C}_7. \end{aligned}$$

We consider the following single-version schedule s_5 over $\mathcal{T}$.

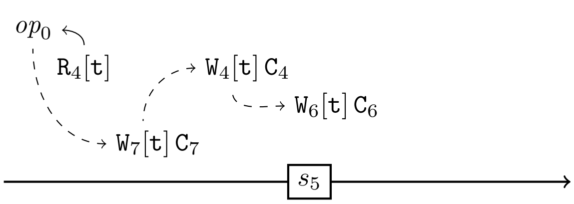


(a) $SeG(s_1)$

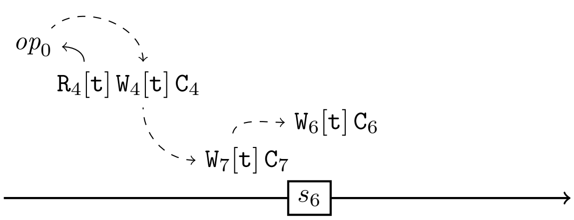


(b) $SeG(s_2)$

Figure 2: Serialization graphs for schedules s_1 and s_2 as defined in Example 2.1.



Schedule s_5 is clearly not conflict-serializable since $\{T_4 \to T_7, T_7 \to T_4\} \subseteq SeG(s_5)$. But s_5 is view-equivalent with the single-version serial schedule s_6.



Indeed, $D(s_5) = D(s_6) = \{\mathtt{R}_4[\mathtt{t}] \to_D \mathtt{W}_4[\mathtt{t}]\}$. □

The relationship between the three notions for serializability is graphically depicted in Figure 3.

4 Isolation Levels

Most generally, an isolation level corresponds to a set of allowed schedules. In this section, we define the isolation levels RC, SI, and SSI.

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*

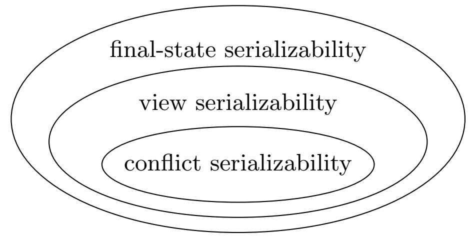


Figure 3: Different notions of serializability.

the commit order of s if the version of t written by T_j is installed after all versions of t installed by transactions committing before T_j commits, but before all versions of t installed by transactions committing after T_j commits. More formally, 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 $\mathtt{C}_j <_s \mathtt{C}_i$. For examples, consider schedules s_1 and s_2 from Example 2.1. In s_1 all transactions respect the commit order, while in schedule s_2 we have $\mathtt{W}_3[\mathtt{q}] \ll_{s_2} \mathtt{W}_2[\mathtt{q}]$ and $\mathtt{C}_2 <_{s_2} \mathtt{C}_3$.

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 $\mathit{first}(T)$. A read operation $\mathtt{R}_j[\mathtt{t}]$ in a transaction $T_j \in \mathcal{T}$ is *read-last-committed in s relative to an operation $a_j \in T_j$* (not necessarily different from $\mathtt{R}_j[\mathtt{t}]$) if the following holds:

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

So, $\mathtt{R}_j[\mathtt{t}]$ observes the most recently installed version of t (according to $\ll_s$) that is committed before a_j in s. The latter can be observed in schedule s_2 (w.r.t both the read itself as well as the start of the transaction), while in schedule s_1 there is a read $\mathtt{R}_1[\mathtt{t}]$ with $v_s(\mathtt{R}_1[\mathtt{t}]) = \mathtt{W}_2[\mathtt{t}]$ and $\mathtt{R}_1[\mathtt{t}] <_{s_1} \mathtt{C}_2$.

A transaction $T_j \in \mathcal{T}$ *exhibits a concurrent write in s* if there are two write operations b_i and a_j in s on the same object with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j$ and $\mathit{first}(T_j) <_s \mathtt{C}_i$. That is, transaction T_j writes to an object that has been modified earlier by a concurrent transaction T_i.

A transaction $T_j \in \mathcal{T}$ *exhibits a dirty write in s* if there are two write operations b_i and a_j in s with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j <_s \mathtt{C}_i$. That is, transaction T_j writes to an object that has been modified earlier by T_i, but T_i has not yet issued a commit. Notice that by definition a transaction exhibiting a dirty write always exhibits a concurrent write. In schedule s_1 (and s_2) the transaction T_3

witnesses a concurrent write since $\mathtt{W}_2[\mathtt{q}] \leq_{s_1} \mathtt{W}_3[\mathtt{q}]$ and $\mathit{first}(T_3) <_{s_1} \mathtt{C}_2$. But T_3 does not exhibit a dirty write since $\mathtt{C}_2 <_{s_1} \mathtt{W}_3[\mathtt{q}]$.

Definition 4.1. *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.*

Definition 4.2. *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, 18]).

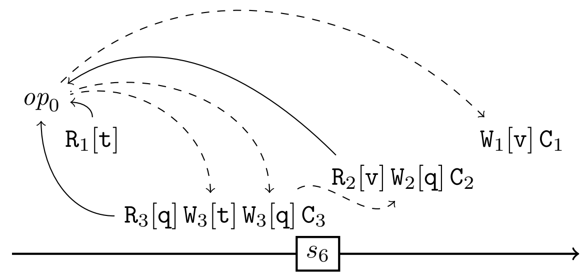
While RC and SI are defined on the granularity of a single transaction, SSI enforces a global condition on the schedule as a whole. For this, recall the concept of dangerous structures from [7]: three transactions $T_1, T_2, T_3 \in \mathcal{T}$ (where T_1 and T_3 are not necessarily different) form 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 ; and,
- $\mathtt{C}_3 <_s \mathtt{C}_1$ and $\mathtt{C}_3 <_s \mathtt{C}_2$.

Definition 4.3. *We say that the schedule s is allowed under SSI if every transaction is allowed under SI in s, and there is no dangerous structure in s.*

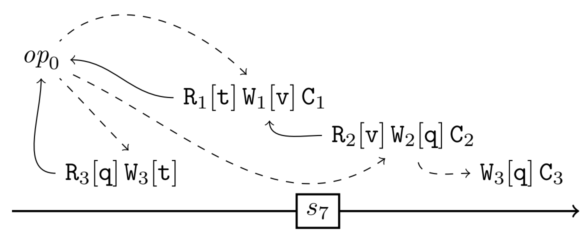
The latter definitions correspond to the ones in the literature (see, e.g., [12, 18]).

Example 4.1. *Consider the set of transactions $\mathcal{T} = \{T_1, T_2, T_3\}$ from Example 2.1. For a schedule in which the transactions of $\mathcal{T}$ are allowed under SI, consider s_6 over $\mathcal{T}$.*



It can be verified that all three transactions of $\mathcal{T}$ are indeed allowed under SI in s_6, but not under SSI, since $T_2 \to T_1 \to T_3$ is a dangerous structure.

We remark that the transactions in s_6 are also allowed under RC. For a schedule over $\mathcal{T}$ in which all transactions are allowed under RC but not under SI, consider consider schedule s_7.



It can be verified that all transactions of $\mathcal{T}$ are allowed under RC in s_7 but not under SI, because of the concurrent writes $\mathtt{W}_2[\mathtt{q}]$ and $\mathtt{W}_3[\mathtt{q}]$. □

5 Robustness

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

Definition 5.1 (Robustness). *Let I be an isolation level. A set of transactions $\mathcal{T}$ is* robust *against I if every schedule for $\mathcal{T}$ that is allowed under I is conflict serializable.*

In the next subsections, we relate robustness against different isolation levels to the non-existence of variants of a specific type of schedule, which we call a multi-version split schedule. These characterizations form the basis for algorithms to test robustness. In its most general form, a multi-version split schedule is defined as follows.

Definition 5.2 (Mv split schedule). *Let $\mathcal{T}$ be a set of transactions and $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$ a sequence of conflicting quadruples for $\mathcal{T}$ such that each transaction in $\mathcal{T}$ occurs in at most two different quadruples. A* multiversion split schedule *for $\mathcal{T}$ based on C is a multiversion schedule that has the following form:*

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

where

1. *there is no operation in T_1 conflicting with an operation in any of the transactions $T_3, \ldots, T_{m-1}$;*
2. *there is no write operation in $\mathsf{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in T_2 or T_m;*
3. *b_1 is rw-conflicting with a_2;*

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

5.1 Snapshot Isolation

We say that a multiversion split schedule s for some set $\mathcal{T}$ of transactions *satisfies the SI requirements* if there is no write operation in $\mathsf{postfix}_{b_1}(T_1)$ ww-conflicting with a write operation in T_2 or T_m; and b_m is rw-conflicting with a_1.

Proposition 5.1. *For a set of transactions $\mathcal{T}$, the following are equivalent:*

1. *$\mathcal{T}$ is not robust against SI;*
2. *there is a multiversion split schedule s for $\mathcal{T}$ based on some C that satisfies the SI requirements.*

Proof Sketch. $(2 \rightarrow 1)$ This direction is straightforward, as it can be verified that such a schedule is allowed under SI and is not conflict-serializable.

$(1 \rightarrow 2)$ Since $\mathcal{T}$ is not robust against SI, a schedule s for $\mathcal{T}$ exists that is allowed under SI but not conflict-serializable. Let $\Gamma = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_n, b_n, a_1, T_1)$ be a cycle in $SeG(s)$. W.l.o.g., we assume Γ is minimal and T_2 is the first transaction (among those occurring in Γ) to commit in s. That is, $\mathtt{C}_2 <_s \mathtt{C}_i$ for every other transaction T_i in Γ.

Next, let $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_n, b_n, a_1, T_1)$ be the sequence of conflicting quadruples derived from Γ. In the remainder of the proof, we argue that the multiversion split schedule s' for $\mathcal{T}$ based on C is indeed valid and satisfies the SI requirements. Condition 1 of Definition 5.2 is immediate by our assumption that Γ is a minimal cycle in $SeG(s)$. Since $\mathtt{C}_2 <_s \mathtt{C}_1$, the edge (T_1, b_1, a_2, T_2) in Γ must be based on a rw-antidependency in s, thereby proving Condition 3 of Definition 5.2. Indeed, by definition of SI, if $b_1 \rightarrow_s a_2$ would be a ww-dependency or a wr-dependency, then $\mathtt{C}_1 <_s \textit{first}(T_2)$. This rw-antidependency $b_1 \rightarrow_s a_2$ furthermore implies that T_1 and T_2 are concurrent in s, as otherwise these two operations would imply a wr-dependency in the opposite direction instead.

We next argue that there is no write operation in T_1 ww-conflicting with a write operation in T_2 or T_m. Since T_1 and T_2 are concurrent, and since SI does not allow concurrent writes, the result is immediate for T_2. If $T_2 = T_m$, the result is immediate for T_m as well. Otherwise, such a pair of conflicting write operations between T_1 and T_m would imply a ww-dependency from T_m to T_1 (as the opposite direction would contradict our assumption that Γ is minimal). But then the definition of SI implies that $\mathtt{C}_m <_s \textit{first}(T_1) <_s \mathtt{C}_2$, thereby contradicting our assumption that T_2 commits first.

To conclude the proof, we argue that b_m is rw-conflicting with a_1. Since $b_m \rightarrow_s a_1$ is a dependency in s, b_m wr- or ww-conflicting with a_1 would imply by definition of SI that $\mathtt{C}_m <_s \textit{first}(T_1) <_s \mathtt{C}_2$, again leading to the desired contradiction. $\square$

Algorithm 1 provides a direct decision procedure for robustness against SI based on the previous characterization. There, for a transaction T_1 and a set of transactions $\mathcal{T}$, we refer by si-graph$(T_1, \mathcal{T})$ to the graph containing as nodes all transactions in $\mathcal{T}$ that do not have an operation conflicting with an operation in T_1, and with an edge between transactions T_i and T_j if T_i has an operation conflicting with an operation in T_j.

The following theorem then readily follows:

Theorem 5.1. *[12] Deciding whether a set of transactions is robust against SI is in* PTIME.

An immediate corollary of the main result by Fekete [12] is that for a set of transactions $\mathcal{T}$ robustness against SI can be characterized by the absence of a specific structure, called pivots, in the interference graph $IG(\mathcal{T})$. In this graph, each transaction in $\mathcal{T}$ is represented by a node and edges summarize the possible dependencies between transactions. That is, if there exists a schedule s with a dependency between two transactions T_i and T_j, then $T_i \rightarrow T_j$ is an edge in $IG(\mathcal{T})$. An edge is furthermore referred to as an exposed edge if the dependency is a rw-antidependency and the two transactions are concurrent in s. A pivot is a transaction T_i part of a chord-free cycle in $IG(\mathcal{T})$ with two adjacent

Algorithm 1: Deciding robustness against SI.

Input : Set of transactions $\mathcal{T}$
Output : *True* iff $\mathcal{T}$ is robust against SI

def *reachable*(T_2, T_m, T_1)**:**
  if $T_2 = T_m$ **then**
    return *True*;
  for $b_2 \in T_2, a_m \in T_m$ **do**
    if b_2 *conflicts with* a_m **then**
      return *True*;
  $G :=$ si-graph$(T_1, \mathcal{T} \setminus \{T_1, T_2, T_m\})$;
  $TC :=$ reflexive-transitive-closure of G;
  for (T_3, T_{m-1}) *in* TC **do**
    for $b_2 \in T_2$, $a_3 \in T_3$, $b_{m-1} \in T_{m-1}$, $a_m \in T_m$ **do**
      if *(*b_2 *conflicts with* a_3 **and** b_{m-1} *conflicts with* a_m*)* **then**
        return *True*;
  return *False*;

for $T_1 \in \mathcal{T}$, $T_2 \in \mathcal{T} \setminus \{T_1\}$, $T_m \in \mathcal{T} \setminus \{T_1\}$ **do**
  if *reachable*(T_2, T_m, T_1) **then**
    for $b_1 \in T_1$ **do**
      if T_1, T_2, *and* T_m *have no ww-conflicting operations* **then**
        for $a_1 \in T_1$, $a_2 \in T_2$, $b_m \in T_m$ **do**
          if b_m *conflicts with* a_1 **and** b_1 *is rw-conflicting with* a_2 **and** b_m *is rw-conflicting with* a_1 **then**
            return *False*;
return *True*

exposed edges $T_{i-1} \to T_i$ and $T_i \to T_{i+1}$. It can be shown that every pivot implies a multiversion split schedule satisfying the SI requirements and vice versa. Intuitively, the rw-antidependencies $b_m \to_s a_1$ and $b_1 \to_s a_2$ in such a multiversion split schedule s correspond to exposed edges $T_m \to T_1$ and $T_1 \to T_2$ in $IG(\mathcal{T})$, thereby witnessing that T_1 is a pivot.

5.2 Read Committed

We say that a multiversion split schedule s for some set $\mathcal{T}$ of transactions *satisfies the RC requirements* if either b_m is rw-conflicting with a_1 or $b_1 <_{T_1} a_1$.

PROPOSITION 5.2. *[18] For a set of transactions $\mathcal{T}$, the following are equivalent:*

1. *$\mathcal{T}$ is not robust against RC;*
2. *there is a multiversion split schedule s for $\mathcal{T}$ based on some C that satisfies the RC requirements.*

PROOF SKETCH. $(2 \to 1)$ This direction is straightforward, as it can be verified that such a schedule is allowed under RC and is not conflict-serializable.

$(1 \to 2)$ The proof strategy is analogous to the proof of Proposition 5.1. In particular, let Γ and C be as in the proof of Proposition 5.1. We now argue that the multiversion split schedule based on C is valid and satisfies the RC requirements. Condition 1 of Definition 5.2 is again immediate by our assumption that Γ is a minimal cycle in $SeG(s)$. Towards Condition 3 of Definition 5.2, note that if $b_1 \to_s a_2$ is a ww-dependency or a wr-dependency, then by definition of RC, we have $\mathtt{C}_1 <_s a_2$, thereby contradicting our assumption that T_2 commits first. Furthermore, this rw-antidependency $b_1 \to_s a_2$ implies that $b_1 <_s \mathtt{C}_2$, as otherwise these operations would imply a wr-dependency in the opposite direction instead. Because of this, there can not be a write operation in $\mathsf{prefix}_{b_1}(T_1)$ conflicting with a write operation in T_2, as this would create a dirty write in s. If $T_m = T_2$, the result is immediate for T_m as well. Otherwise, if $T_m \neq T_2$, such a pair of ww-conflicting operations in T_1 and T_m would imply a ww-dependency from T_m to T_1 (as the opposite direction contradicts our assumption that Γ is a minimal cycle), and hence $\mathtt{C}_m <_s b_1 <_s \mathtt{C}_2$, again leading to the desired contradiction and thereby proving Condition 2 of Definition 5.2.

It remains to argue that the multiversion split schedule satisfies the RC requirements. Towards a contradiction, assume b_m is wr- or ww-conflicting with a_1 and $a_1 \leq_{T_1} b_1$. Then, by the definition of RC, we have $\mathtt{C}_m <_s a_1$, and hence $\mathtt{C}_m <_s b_1 <_s \mathtt{C}_2$, which contradicts our assumption that T_2 commits first. $\square$

Algorithm 1 can easily be adapted for RC, leading to the following result:

THEOREM 5.2. *[18] Deciding whether a set of transactions is robust against RC is in* PTIME.

Ketsman et al. [14] provide full characterizations for robustness against READ COMMITTED and READ UNCOMMITTED under lock-based semantics as opposed to the multiversion semantics that is used here. In addition, it is shown that the corresponding decision problems are complete for coNP and LOGSPACE, respectively. The coNP-hardness stems from the fact that counterexample schedules no longer take the simple form of a split schedule.

5.3 Robust allocations

In practice, an isolation level is not set uniformly on the level of the database or even on the level of the application but can be specified on the level of an individual transaction. Let $\mathcal{I} \subseteq \{\text{RC}, \text{SI}, \text{SSI}\}$. An *$\mathcal{I}$-allocation* $\mathcal{A}$ for a set of transactions $\mathcal{T}$ is a function mapping each transaction $T \in \mathcal{T}$ onto an isolation level $\mathcal{A}(T) \in \mathcal{I}$.

A schedule s over a set of transactions $\mathcal{T}$ *is allowed under an $\mathcal{I}$-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 say that a set of transactions $\mathcal{T}$ is *robust* against an allocation $\mathcal{A}$ when every schedule that is allowed under $\mathcal{A}$ is conflict-serializable. The *allocation problem* then consists of deciding whether a robust allocation exists and if so to find an optimal one.

In [21] it is shown that it can be decided in polynomial time whether a set of transactions is robust against a given $\{\text{RC}, \text{SI}, \text{SSI}\}$-allocation. That result is based on a notion of split schedules for allocations. Furthermore, it is shown that a unique optimal[3] allocation always exists and can be found in polynomial time as well. Fekete [12] provided a characterization for robust allocations when every transaction runs under either SNAPSHOT ISOLATION or strict two-phase locking (S2PL). He also obtained a polynomial time algorithm to compute the optimal allocation.

6 Transaction programs and templates

Transaction programs Previous work on static robustness testing [13, 2] for transaction programs is based on the following key insight: when a *schedule* is not serializable, then the dependency graph constructed from that schedule contains a cycle satisfying a condition specific to the isolation level at hand (*dangerous structure* for SNAPSHOT ISOLATION and the presence of a *counterflow edge* for RC). That insight is extended to a workload of *transaction programs* through the construction of a so-called static dependency graph where each program is represented by a node, and there is a conflict edge from one program to another if there can be a schedule that gives rise to that conflict. The absence of a cycle satisfying the condition specific to that isolation level then guarantees robustness while the presence of a cycle does not necessarily imply non-robustness.

[3]Informally, optimal means favoring RC over SI, and favoring SI over SSI.

DepositChecking:

$$\texttt{R}[\texttt{X} : \text{Account}\{\text{N, C}\}]$$
$$\texttt{U}[\texttt{Z} : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$$

Figure 4: Transaction template for DepositChecking.

Other work studies robustness within a framework for uniformly specifying different isolation levels in a declarative way [8, 4, 9]. 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*.

Transaction Templates The static robustness approach based on transaction templates [18] differs in two ways. First, it makes more underlying assumptions explicit within the formalism of transaction templates (whereas previous work departs from the static dependency graph that should be constructed in some way by the dba). Second, it allows for a decision procedure that is sound and complete for robustness testing against RC, allowing to detect larger subsets of transactions to be robust [18].

EXAMPLE 6.1. *Figure 4 shows the transaction template for DepositChecking, which is a part of the SmallBank benchmark. The template consists of two operations. The first operation is a read operation over variable* `X` *of type Account. In particular, the attributes Name (N) and CustomerID (C) are read. The second operation is an update operation over variable* `Z` *of type Checking. Such an update operation should be interpreted as a read immediately followed by a write that cannot be interleaved with other operations. In particular, the attributes CustomerID (C) and Balance (B) are read, immediately followed by a write to attribute Balance. We can now instantiate transactions from these templates by assigning tuples of the corresponding type to variables. For example, transaction* `R[t]U[v]C` *is a valid instantiation, but* `R[t]U[t]C` *is not, since object* `t` *cannot be of type Account and Checking at the same time.*

The formalization of transactions and conflict serializability in [18] and this paper is based on [12], generalized to operations over attributes of tuples and extended with `U`-operations that combine `R`- and `W`-operations into one atomic operation. These definitions are closely related to the formalization presented by Adya et al. [1], but we assume a total rather than a partial order over the operations in a schedule. There are also a few restrictions to the model: there needs to be a fixed set of read-only attributes that cannot be updated and which are used to select tuples for update. The most typical example of this are primary key values passed to transaction templates as parameters. The inability to update primary keys is not an important restriction in many workloads, where keys, once assigned, never get changed, for regulatory or data integrity reasons.

In [18], a PTIME decision procedure is obtained for robustness against RC for templates without functional constraints and [20] improves that result to NLOGSPACE. In addition, an experimental study was performed showing how an approach based on robustness and making transactions robust through promotion can improve transaction throughput. In particular, we show on the SmallBank and TPC-Ckv (based on TPC-C) benchmarks that in case of increasing contention our approach leads to practical performance improvements compared to when executed under SI or SSI. It should be noted that both benchmarks in their original form are not identified as robust. By promoting a small number of read operations such that they write the observed value back to the database, robustness is obtained without altering the semantics of these benchmark programs. In [20], the modeling power of transaction templates is extended with functional constraints, which are useful for capturing data dependencies like foreign keys. By more accurately modeling transaction programs, it becomes possible to recognize larger sets of workloads as robust.

7 Conclusion

Despite its practical relevance and challenging problems, concurrency control has only attracted limited attention from the database theory community. We hope that the present paper eases the barrier of entrance to this exciting topic. To spark further interest, we mention some open problems that we consider interesting.

Research on robustness for example has mostly focused on guaranteeing conflict-serializability. But as we explained in Section 3, there exist alternative definitions of serializability that can be used for a robustness analysis, like view-serializability and final-state serializability. It would be interesting to see if the robustness problem cast with one of the alternative definitions has similar characterizations as the ones obtained for conflict-serializability. Furthermore, while in theory one could expect that a more liberal definition of serializability leads to larger classes of transactions and templates that are robust, it is not clear if such a difference can be observed in practice.

Another direction for further research lies in the consideration of systems with a higher-degree of parallelism. The considered isolation levels, RC, SI, are mostly designed for conventional database systems utilizing a limited degree of parallelization. High isolation levels in many-core systems are known to be particularly challenging and therefore robustness analysis against lower-isolation levels that are meaningful in a highly parallel context would be particularly relevant.

Acknowledgments

This work is partly funded by FWO-grant G019921N.

8 References

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[2] Mohammad Alomari and Alan Fekete. Serializable use of read committed isolation level. In *AICCSA*, pages 1–8, 2015.

[3] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

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

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann, 1996.

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

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

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

[10] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Algebraic Laws for Weak

Consistency. In *CONCUR*, pages 26:1–26:18, 2017.
[11] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2):169–182, 2018.
[12] Alan Fekete. Allocating isolation levels to transactions. In *PODS*, pages 206–215, 2005.
[13] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.
[14] Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. Deciding robustness for lower SQL isolation levels. In *PODS*, pages 315–330, 2020.
[15] Christos H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.
[16] TPC-C. On-line transaction processing benchmark. `http://www.tpc.org/tpcc/`.
[17] Brecht Vandevoort. *Optimizing Concurrency Control: Robustness Against Read Committed Revisited*. PhD thesis, Hasselt University, 2021.
[18] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. Robustness against read committed for transaction templates. *PVLDB*, 14(11):2141–2153, 2021.
[19] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. Robustness against read committed: A free transactional lunch. In *PODS*, pages 1–14. ACM, 2022.
[20] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. Robustness against read committed for transaction templates with functional constraints. In *ICDT*, volume 220 of *LIPIcs*, pages 16:1–16:17, 2022.
[21] Brecht Vandevoort, Bas Ketsman, and Frank Neven. Allocating isolation levels to transactions in a multiversion setting. Manuscript, 2022.
[22] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.