When is it safe to run a transactional workload under Read Committed?

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

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

Christoph Koch
École Polytechnique Fédérale de Lausanne
christoph.koch@epfl.ch

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

ABSTRACT

The popular isolation level multiversion Read Committed (RC) exchanges some of the strong guarantees of serializability for increased transaction throughput. Nevertheless, transaction workloads can sometimes be executed under RC while still guaranteeing serializability at a reduced cost. Such workloads are said to be robust against RC. This paper provides a high level overview of deciding robustness against RC. In particular, we discuss how a sound and complete test can be obtained through the formalization of transaction templates. We then increase the modeling power of transaction templates by extending them with functional constraints which are useful for capturing data dependencies like foreign keys. We show that the incorporation of functional constraints can identify more workloads as robust than would otherwise be the case. Even though the robustness problem becomes undecidable in its most general form, we establish that various restrictions on functional constraints lead to decidable and even tractable results that can be used to model and test for robustness against RC for practical scenarios.

1. INTRODUCTION

The gold standard for desirable transactional semantics is serializability, and much research and technological development has gone into creating systems that provide the greatest possible transaction throughput. Nevertheless, in practice, a hierarchy of alternative isolation levels of different strengths is available, allowing users to trade off semantic guarantees for better performance. One prominent example is the isolation level (multiversion) Read Committed (RC), which does not guarantee serializability but which can be implemented more efficiently than isolation level Serializable. The central question that we address in this paper is: *When is it safe to run a transactional workload under RC?*

Various researchers have studied the so-called transactional robustness problem [1–4, 6–10, 16, 18], which revolves around deciding whether for a given workload a lower isolation level than Serializable is sufficient to guarantee serializability. Specifically, a set of transaction programs is called robust against a given isolation level if every possible interleaving of the executed programs allowed under the specified isolation level, is serializable. That there is a real chance that nontrivially robust workloads do exist is probably best demonstrated by the fact that the well-known TPC-C benchmark is robust against Snapshot Isolation [10]. More specifically, any schedule allowed under Snapshot Isolation that exclusively contains transactions that are instantiations of transaction programs from TPC-C is serializable.

Executing a transaction program over a database results in a sequence of read and write operations on concrete database objects that we simply refer to as a transaction. When multiple programs are executed concurrently, the resulting sequence of interleaved transactions is referred to as a schedule. Within such a schedule, transactions can introduce dependencies when they access the same database objects. For example, if a transaction T_1 writes to an object and another transaction T_2 afterwards reads this object, then T_2 depends on T_1. The dependency graph $CG(s)$ of a schedule s is the graph whose vertices are the transactions of s and whose edges represent dependencies between transactions. It is well-known that a schedule is (conflict) serializable if and only if its dependency graph is acyclic [14].

Robustness for a given workload specified as a set of concrete transactions is obviously decidable as one can simply enumerate all schedules allowed under the chosen isolation level and verify whether their dependency graphs are acyclic. There are, however, more efficient approaches that are based on characterisations in terms of a particular counterexample schedule that must exist when a set of transactions is not robust against the isolation level. These are the following: split schedules and multisplit schedules for Read Uncommitted and Read Committed in single-version databases where concurrency control is implemented based on read and write locks [12]; multiversion split schedules for (multiversion) Read Committed [16]; and a multiversion split schedule for Snapshot Isolation [9, 13]. Deciding robustness then reduces to searching for a counterexample schedule adhering to a specific form.

These results do not immediately extend to the setting where a workload is specified as a set of transaction programs (as, for instance, the TPC-C benchmark). Indeed, a transaction program represents a potentially infinite number of concrete transaction instantiations giving rise to an infinite number of possible schedules. A frequently used approach for detecting robustness in this setting is based on summarizing all potential schedules in a static dependency graph [6,10]. More specifically, the vertices in this graph are the transaction programs, and there is an edge from a program P_1 to a program P_2 if there exists a pair of transactions T_1 and T_2 instantiated from P_1 and P_2, respectively, where T_2 depends on T_1. By construction this graph contains all possible dependency graphs of schedules that can be derived from the given set of transaction programs. In particular, every cycle in the dependency graph of a non-serializable schedule is witnessed by a cycle in the static dependency graph. A sufficient condition for robustness therefore relies on the absence of cycles in the static dependency graph. Notice that this is only a sufficient condition for robustness, as a cycle in the static dependency graph does not necessarily imply the existence of a non-serializable schedule.

Additional isolation level specific conditions have been identified that must hold for a cycle in a dependency graph witnessing non-robustness: *dangerous structure* for Snapshot Isolation [10] and the presence of a *counterflow edge* for Read Committed [2]. Robustness against these isolation levels is therefore guaranteed as long as all cycles in the static dependency graph do not satisfy the corresponding condition. We stress that such approaches depend on the construction of a static dependency graph which in itself is a non-trivial task. Indeed, a manual analysis of each pair of programs is required to identify possible dependencies. IsoDiff [11] automates this construction by analyzing an execution trace (i.e., a set of concrete transactions obtained by executing the transaction programs). But the disadvantage of this approach is that it might miss programs that are not executed frequently and are therefore not in the execution trace. Therefore, IsoDiff might falsely identify applications as robust against a lower isolation level.

Our work on robustness differs from the just mentioned approach as follows: (1) We introduce a formalism to express transaction programs, referred to as *transaction templates*, that facilitates reasoning in terms of counterexample schedules and in essence allows to construct the static dependency graph automatically without manual intervention; and, (2) we provide a decision procedure for robustness against multiversion Read Comitted that is both sound and complete. In comparison to earlier work, which only offered methods that are sound but not complete, our approach enables the detection of more and larger groups of workloads to be robust against Read Committed. A restriction of our approach is that we must assume that there is a fixed set of read-only attributes that cannot be updated and which are used to select tuples. The most typical example of this are primary key values passed to transaction templates as parameters. We refer to [16] for a more in depth discussion. In [20], we present a sufficient (but no longer complete) condition for testing robustness for an extension of templates where all attributes of tuples can be modified and that incorporates inserts, deletions, and predicate reads.

This paper is intended as a more accessible overview of the main ideas presented in [16] and [18]. Many concepts are only introduced informally and we refer to the original papers for more details. For a more complete overview of our work, we refer to [17]. For an introduction to concurrency control aimed at database theorists we refer to [14] and [13]. For a more complete discussion of the related work we refer to [16, 18].

2. DEFINITIONS

We present the necessary definitions based on a small extension of the SmallBank benchmark [1], which we will use as a running example throughout the paper. The SmallBank schema consists of three tables: Account(Name, CustomerID, IsPremium), Savings(CustomerID, Balance, InterestRate), and Checking(CustomerID, Balance). Underlined attributes are primary keys. The Account table associates customer names with IDs and keeps track of the premium status (Boolean); CustomerID is a `UNIQUE` attribute. The other tables contain the balance (numeric value) of the savings and checking accounts of customers identified by their ID. Account (CustomerID) is a foreign key referencing both the columns Savings (CustomerID) and Checking (CustomerID). The interest rate on a savings account is based on a number of parameters, including the account status (premium or not). The application code can interact with the database only through the following transaction programs:

- Balance(N): returns the total balance (savings & checking) for a customer with name N.
- DepositChecking(N,V): makes a deposit of amount V on the checking account of the customer with name N.
- TransactSavings(N,V): makes a deposit or withdrawal V on the savings account of the customer named N.
- Amalgamate(N_1,N_2): transfers all the funds from N_1 to N_2.
- WriteCheck(N,V): writes a check V against the account of the customer with name N, penalizing if overdrawing.
- GoPremium(N): converts the account of the customer with name N to a premium account and updates the interest rate of the corresponding savings account. This transaction program is an extension w.r.t. [1].

Databases. A *relational schema* consists of a set of relations, where a finite set of attribute names Attr(R) is associated to every relation R. A *database (instance)* **D** over a relational schema assigns a finite set of tuples to each relation R in this schema, and we furthermore say that these tuples are of type R. Each such tuple of type R maps the attribute names in Attr(R) to a value.

EXAMPLE 2.1. We will refer to the following SmallBank database instance by $\mathbf{D}_1$ and use it as a running example:

Account	Name	CustomerID	IsPremium
t :	Alice	123	true
t′ :	Bob	456	false

Savings	CustomerID	Balance	InterestRate
v :	123	\$2500	0.50
v′ :	456	\$450	0.50

Checking	CustomerID	Balance
q :	123	\$50
q′ :	456	\$30

Transactions and schedules. We distinguish three operations R[t], W[t], and U[t] on a tuple t, denoting that tuple t is read, written, or updated, respectively. The operation U[t] is an atomic update and should be viewed as an atomic sequence of a read of t followed by a write to t. We will use the following terminology: a *read operation* is an R[t] or a U[t], and a *write operation* is a W[t] or a U[t]. Furthermore, an R-operation is an R[t], a W-operation is a W[t], and a U-operation is a U[t]. We also assume a special *commit* operation denoted C. To every operation o on a tuple of type R, we associate the set of attributes $\text{ReadSet}(o) \subseteq \text{Attr}(R)$ and $\text{WriteSet}(o) \subseteq \text{Attr}(R)$ containing, respectively, the set of attributes that o reads from and writes to. When o is an R-operation then $\text{WriteSet}(o) = \emptyset$. Similarly, when o is a W-operation then $\text{ReadSet}(o) = \emptyset$.

EXAMPLE 2.2. *Say, we want to* read the balance of the savings account of customer 123 *and* set the balance of the savings account of customer 123 to \$1000. *The former translates to operation* R[v] *of type Savings with ReadSet*(R[v]) $= \{\mathit{CustomerID}, \mathit{Balance}\}$*, and the latter to operation* W[v] *of type Savings with WriteSet*(W[v]) $= \{\mathit{Balance}\}$*. Notice that* v *refers to a specific tuple in* $\boldsymbol{D}_1$ *as defined by Example 2.1.*
Suppose we want to increase the balance of the savings account of customer 123 by \$150. *The associated operation becomes* U[v] *of type Savings with ReadSet*(U[v]) $= \{\mathit{CustomerID}, \mathit{Balance}\}$ *and WriteSet*(U[v]) $= \{\mathit{Balance}\}$*.*

A *transaction* T is a sequence of read and write operations followed by a commit. For two operations $a, b \in T$, we use $a <_T b$ to denote the fact that a precedes b in transaction T.

EXAMPLE 2.3. *Consider* an increase of the savings account balance of the customer named Alice by \$150 *over database* $\boldsymbol{D}_1$*. The associated transaction consists of two operations:* R[t]U[q] *with* U[q] *as in Example 2.2 and* R[t] *of type Account with ReadSet*(R[t]) $= \{Name, CustomerID\}$*.*

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 $\mathtt{W}_i$[t], $\mathtt{R}_i$[t], and $\mathtt{U}_i$[t] to denote a W[t], R[t], and U[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.* [5, 9]). To avoid ambiguity of notation, we assume that a transaction performs at most one write, one read, and one update per tuple. The latter is a common assumption (see, *e.g.* [9]) and all our results carry over to the more general setting in which multiple writes and reads per tuple are allowed.

A *(multiversion) schedule* s over a set $\mathcal{T}$ of transactions is a total order over the operations in $\mathcal{T}$, consistent with the order of operations in each transaction $T_i \in \mathcal{T}$. For a pair of operations a, b occurring in s, we use $a <_s b$ to denote the fact that a precedes b in s. Each write operation in s creates a new *version* of the tuple it writes to, and each read operation observes a specific version of the tuple it reads from. Note that read operations do not necessarily read the most recently created version. Indeed, multiversion databases such as Postgres and Oracle maintain multiple versions of each tuple, and the version observed by a read operation depends on the chosen isolation level. We say that a schedule is a *single version schedule* if every read operation observes the most recently created version. 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 triple of operations a, b, c occurring in s with $a <_s b <_s c$ and $a, c \in T$ implies $b \in T$ for every $T \in \mathcal{T}$. Figure 1a and 1c show three example schedules.

The absence of aborts in our definition of schedule is consistent with the common assumption [6, 9] 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.

Conflict Serializability. Let b_i and a_j be two operations on the same tuple from different transactions T_i and T_j in a set of transactions $\mathcal{T}$. We then say that b_i is *conflicting* with a_j if:

- *(ww-conflict)* $\text{WriteSet}(b_i) \cap \text{WriteSet}(a_j) \neq \emptyset$; or,
- *(wr-conflict)* $\text{WriteSet}(b_i) \cap \text{ReadSet}(a_j) \neq \emptyset$; or,
- *(rw-conflict)* $\text{ReadSet}(b_i) \cap \text{WriteSet}(a_j) \neq \emptyset$.

In this case, we also say that b_i and a_j are conflicting operations. When b_i and a_j are conflicting operations in $\mathcal{T}$, we say that a_j *depends on* b_i in a schedule s over $\mathcal{T}$, denoted $b_i \rightarrow_s a_j$ if:[1]

- *(ww-dependency)* b_i is ww-conflicting with a_j and the version written by a_j is created after the version written by b_i; or,
- *(wr-dependency)* b_i is wr-conflicting with a_j and b_i either creates the version observed by a_j, or it creates a version that is created before the version observed by a_j; or,
- *(rw-antidependency)* b_i is rw-conflicting with a_j and b_i observes a version created before the version written by a_j.

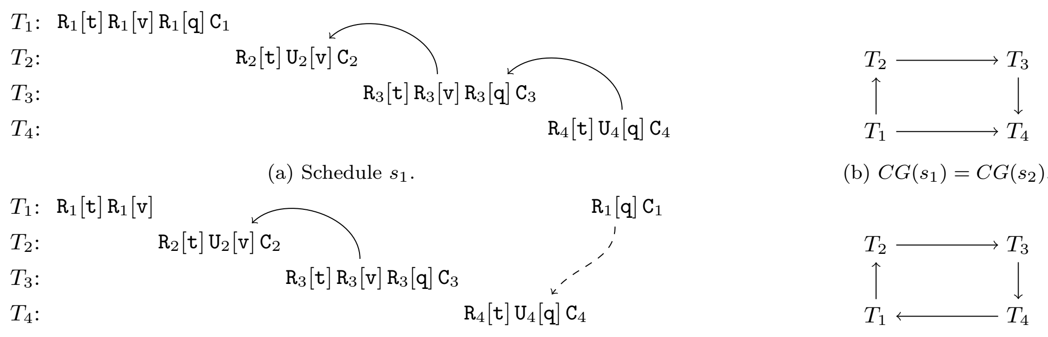
EXAMPLE 2.4. *For examples of rw-antidependencies, consider operations* $\mathtt{R}_1$[v]*,* $\mathtt{U}_2$[v] *and* $\mathtt{R}_1$[q]*,* $\mathtt{U}_4$[q] *in schedule* s_2 *(Figure 1c without dashed arrow). Particularly notice that the write operation for the latter anti dependency occurs before the accompanying read operation. Examples of wr-dependencies are operations* $\mathtt{U}_2$[v] *and* $\mathtt{R}_3$[v] *in schedules* s_1 *and* s_2*, as well as operations* $\mathtt{U}_4$[q] *and* $\mathtt{R}_1$[q] *in schedule* s_3*. None of the schedules in Figure 1 observe a ww-dependency.*

Two schedules s and s' are *conflict equivalent* if they are over the same set $\mathcal{T}$ of transactions and for every pair of conflicting operations a_j and b_i, $b_i \rightarrow_s a_j$ iff $b_i \rightarrow_{s'} a_j$.

DEFINITION 2.5. *A schedule s is* conflict serializable *if it is conflict equivalent to a single version serial schedule.*

EXAMPLE 2.6. *Schedule* s_1 *and* s_2 *in Figure 1 are conflict equivalent. The pairs of conflicting operations are* $\mathtt{R}_1$[v] *and* $\mathtt{U}_2$[v]*,* $\mathtt{U}_2$[v] *and* $\mathtt{R}_3$[v]*,* $\mathtt{R}_1$[q] *and* $\mathtt{U}_4$[q]*, and* $\mathtt{R}_3$[q] *and* $\mathtt{U}_4$[q]*. Equivalence of the dependencies is straightforward, except perhaps for* $\mathtt{R}_1[\mathtt{q}] \rightarrow_{s_1} \mathtt{U}_4[\mathtt{q}]$ *and* $\mathtt{R}_1[\mathtt{q}] \rightarrow_{s_2} \mathtt{U}_4[\mathtt{q}]$*, which are due to a wr-dependency in* s_1 *and rw-antidependency in* s_2*.*

[1]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'.


T_1: $R_1[t]\, R_1[v]\, R_1[q]\, C_1$
T_2: $R_2[t]\, U_2[v]\, C_2$
T_3: $R_3[t]\, R_3[v]\, R_3[q]\, C_3$
T_4: $R_4[t]\, U_4[q]\, C_4$
T_1: $R_1[t]\, R_1[v]$ $R_1[q]\, C_1$
T_2: $R_2[t]\, U_2[v]\, C_2$
T_3: $R_3[t]\, R_3[v]\, R_3[q]\, C_3$
T_4: $R_4[t]\, U_4[q]\, C_4$

(a) Schedule s_1.

$$\begin{tikzcd} T_2 \arrow[r] & T_3 \arrow[d] \\ T_1 \arrow[u] \arrow[r] & T_4 \end{tikzcd}$$

(b) $CG(s_1) = CG(s_2)$.

(c) Schedule s_2 (with solid and dashed arrows) and s_3 (with only solid arrows).

$$\begin{tikzcd} T_2 \arrow[r] & T_3 \arrow[d] \\ T_1 \arrow[u] & T_4 \arrow[l] \end{tikzcd}$$

(d) $CG(s_3)$.

Figure 1: All schedules are multi-version. Read operations observe a particular version of the accessed object. This version is either the result of a write operation (in that case indicated through an arrow pointing to the respective write) or the initial version present at the start of the schedule (in that case without arrow).

A *conflict graph* $CG(s)$ 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_i has an operation b_i that conflicts with an operation a_j in T_j and $b_i \rightarrow_s a_j$. Conflict graphs for schedules s_1, s_2, and s_3 are given in Figure 1b and 1d.

THEOREM 2.7 ([14]). *A schedule s is conflict serializable iff the conflict graph for s is acyclic.*

Multiversion Read Committed. Let s be a schedule for a set $\mathcal{T}$ of transactions. Then, *s exhibits a dirty write* iff there are two ww-conflicting operations a_j and b_i in s on the same tuple $\mathtt{t}$ with $a_j \in T_j$, $b_i \in T_i$ and $T_j \neq T_i$ such that $b_i <_s a_j <_s \mathtt{C}_i$. That is, transaction T_j writes to an attribute of a tuple that has been modified earlier by T_i, but T_i has not yet issued a commit. We say that a schedule s is *read-last-committed (RLC)* if every read operation a_j in s on some tuple $\mathtt{t}$ observes the most recent version of $\mathtt{t}$ that is committed before a_j. Note that this version is different from any version of $\mathtt{t}$ created before a_j that has not been committed yet. Note in particular that a schedule cannot exhibit dirty reads, defined in the traditional way [5], if it is read-last-committed.

EXAMPLE 2.8. *Schedules s_1 and s_2 are both RLC. Schedule s_3 is not RLC, since in s_3 operation $\mathtt{R}_1[\mathtt{q}]$ observes the original version of object $\mathtt{q}$ instead of the version written by $\mathtt{U}_4[\mathtt{q}]$. Clearly, all three schedules are free of dirty writes, since none of the involved transactions write to a shared object (i.e., there are no ww-dependencies at all).*

DEFINITION 2.9. *A schedule is* allowed under isolation level *read committed (RC) if it is read-last-committed and does not exhibit dirty writes.*

It follows from the properties described in Example 2.8 that schedules s_1 and s_3 are allowed under isolation level RC, while schedule s_2 is not.

3. TRANSACTION TEMPLATES

Figure 2 displays the transaction templates for SmallBank. In short, a transaction template is a sequence of read (R), write (W) and update (U) statements over typed variables (X, Y, ...) (the additional (dis)equality constraints at the end of each template in Figure 2 will be discussed in Section 5 and can be ignored for now). For instance, $\mathtt{R}[\mathtt{X} : \text{Account}\{\mathtt{N},\mathtt{C}\}]$ indicates that a read operation is performed to a tuple in relation Account on the attributes Name and CustomerID. We abbreviate the names of attributes by their first letter to save space. The set $\{N, C\}$ is the read set of the read operation. Similarly, W and U refer to write and update operations to tuples of a specific relation. Write operations have an associated write set while update operations contain a read set followed by a write set: e.g., $\mathtt{U}[\mathtt{Z} : \text{Checking}\{\mathtt{C},\mathtt{B}\}\{\mathtt{B}\}]$ first reads the CustomerID and Balance of tuple Z and then writes to the attribute Balance. All R-, W- and U-operations always access exactly one tuple. A U-operation is an atomic update that first reads the tuple and then writes to it.

Templates serve as abstractions of transaction programs and represent an infinite number of possible transactions. More formally, a transaction T over a database $\mathbf{D}$ is an *instantiation* of a transaction template τ if there is a variable mapping μ from the variables in τ to tuples of the corresponding type in $\mathbf{D}$ such that $\mu(\tau) = T$, where we use $\mu(\tau)$ to denote the transaction obtained by replacing each variable X in τ by its corresponding tuple $\mu(\mathtt{X})$. We then say that a set of transactions $\mathcal{T}$ over tuples in $\mathbf{D}$ is *consistent* with a set of templates $\mathcal{P}$ and database $\mathbf{D}$ if every transaction in $\mathcal{T}$ is an instantiation of a transaction template in $\mathcal{P}$.

EXAMPLE 3.1. *Disregarding attribute sets,* $\{\mathtt{R}[\mathtt{t}]\,\mathtt{R}[\mathtt{v}]\,\mathtt{R}[\mathtt{q}], \mathtt{R}[\mathtt{t}]\,\mathtt{U}[\mathtt{v}], \mathtt{R}[\mathtt{t}]\,\mathtt{U}[\mathtt{q}]\}$ *is a set of transactions consistent with the SmallBank templates as it contains an instantiation of Balance, DepositChecking, and TransactSavings. Furthermore,* $\{\mathtt{R}[\mathtt{t}]\,\mathtt{R}[\mathtt{v}]\,\mathtt{R}[\mathtt{q}]\,\mathtt{U}[\mathtt{q}']\}$ *with* $\mathtt{q} \neq \mathtt{q}'$ *is not a valid set of transactions as the two final operations in WriteCheck should be on the same object as required by the formalization. Typed variables effectively enforce domain constraints as we assume that variables that range over tuples of different relations can never be instantiated by the same value.*

4. ROBUSTNESS

We define the robustness property [6] (also called *acceptability* in [9,10]), which guarantees serializability for all schedules of a given set of transactions for a given isolation level.

DEFINITION 4.1 (TRANSACTION ROBUSTNESS). *A set $\mathcal{T}$ of transactions is* robust *against RC if every schedule over*

Balance:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{N, C}\}]$
$\mathtt{R}[\mathtt{Y} : \text{Savings}\{\text{C, B}\}]$
$\mathtt{R}[\mathtt{Z} : \text{Checking}\{\text{C, B}\}]$
$\mathtt{Y} = f_{A\to S}(\mathtt{X}),\ \mathtt{X} = f_{S\to A}(\mathtt{Y})$
$\mathtt{Z} = f_{A\to C}(\mathtt{X}),\ \mathtt{X} = f_{C\to A}(\mathtt{Z})$

DepositChecking:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{N, C}\}]$
$\mathtt{U}[\mathtt{Z} : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$
$\mathtt{Z} = f_{A\to C}(\mathtt{X}),\ \mathtt{X} = f_{C\to A}(\mathtt{Z})$

TransactSavings:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{N, C}\}]$
$\mathtt{U}[\mathtt{Y} : \text{Savings}\{\text{C, B}\}\{\text{B}\}]$
$\mathtt{Y} = f_{A\to S}(\mathtt{X}),\ \mathtt{X} = f_{S\to A}(\mathtt{Y})$

Amalgamate:

$\mathtt{R}[\mathtt{X}_1 : \text{Account}\{\text{N, C}\}]$
$\mathtt{R}[\mathtt{X}_2 : \text{Account}\{\text{N, C}\}]$
$\mathtt{U}[\mathtt{Y}_1 : \text{Savings}\{\text{C, B}\}\{\text{B}\}]$
$\mathtt{U}[\mathtt{Z}_1 : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$
$\mathtt{U}[\mathtt{Z}_2 : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$
$\mathtt{X}_1 \neq \mathtt{X}_2,$
$\mathtt{Y}_1 = f_{A\to S}(\mathtt{X}_1),\ \mathtt{X}_1 = f_{S\to A}(\mathtt{Y}_1)$
$\mathtt{Y}_2 = f_{A\to S}(\mathtt{X}_2),\ \mathtt{X}_2 = f_{S\to A}(\mathtt{Y}_2)$
$\mathtt{Z}_1 = f_{A\to C}(\mathtt{X}_1),\ \mathtt{X}_1 = f_{C\to A}(\mathtt{Z}_1)$
$\mathtt{Z}_2 = f_{A\to C}(\mathtt{X}_2),\ \mathtt{X}_2 = f_{C\to A}(\mathtt{Z}_2)$

WriteCheck:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{N, C}\}]$
$\mathtt{R}[\mathtt{Y} : \text{Savings}\{\text{C, B}\}]$
$\mathtt{R}[\mathtt{Z} : \text{Checking}\{\text{C, B}\}]$
$\mathtt{U}[\mathtt{Z} : \text{Checking}\{\text{C, B}\}\{\text{B}\}]$
$\mathtt{Y} = f_{A\to S}(\mathtt{X}),\ \mathtt{X} = f_{S\to A}(\mathtt{Y})$
$\mathtt{Z} = f_{A\to C}(\mathtt{X}),\ \mathtt{X} = f_{C\to A}(\mathtt{Z})$

GoPremium:

$\mathtt{U}[\mathtt{X} : \text{Account}\{\text{N, C}\}\{\text{I}\}]$
$\mathtt{R}[\mathtt{Y} : \text{Savings}\{\text{C, I}\}]$
$\mathtt{U}[\mathtt{Y} : \text{Savings}\{\text{C}\}\{\text{I}\}]$
$\mathtt{Y} = f_{A\to S}(\mathtt{X}),\ \mathtt{X} = f_{S\to A}(\mathtt{Y})$

Figure 2: Transaction templates for SmallBank.

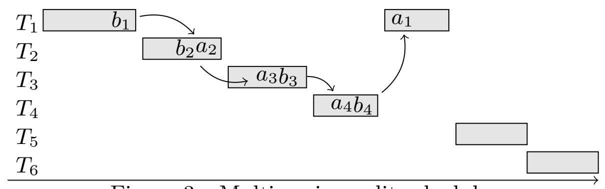


Figure 3: Multiversion split schedule.

$\mathcal{T}$ *that is allowed under RC is conflict serializable.*

Our characterization of transaction robustness for a set of transactions $\mathcal{T}$ is based on counterexample schedules of a very specific form, called *multiversion split schedules*. These schedules are obtained by splitting a transaction $T_1 \in \mathcal{T}$ into two parts and interleaving other transactions $T_2, T_3, \ldots, T_m$ with $m \geq 2$ in between in a sequential fashion. If there are any remaining transactions $T_{m+1}, T_{m+2}, \ldots, T_n$ in $\mathcal{T}$, they are placed at the end of the schedule in an arbitrary order. For such a schedule s to be a valid multiversion split schedule, we furthermore require that s is allowed under RC, and that there is a dependency in s from transaction T_i to transaction T_{i+1} for all $i \in \{1, \ldots, m-1\}$, as well as a dependency from T_m to T_1. Figure 3 depicts a schematic multiversion split schedule for $m = 4$, with arrows indicating the required dependencies. In this figure, T_5 and T_6 depict trailing transactions. Furthermore, schedules s_2 and s_3 in Figure 1 are examples of multiversion split schedules.

By construction, a multiversion split schedule has a cycle in its conflict graph $CG(s)$, and is therefore not conflict serializable. Since every multiversion split schedule is allowed under RC, the existence of a multiversion split schedule for a set of transactions $\mathcal{T}$ witnesses the fact $\mathcal{T}$ is not robust against RC. The next theorem shows that the converse is also true, i.e., if a set of transactions $\mathcal{T}$ is not robust against RC, then we can always construct a multiversion split schedule for $\mathcal{T}$.

Theorem 4.2 ([16]). *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 $\mathcal{T}$.*

Theorem 4.3 ([16]). *Deciding robustness against RC for a set of transaction is decidable in* PTIME.

Algorithm 1: Deciding transaction robustness against RC.

```
Input  :  Set of transactions $\mathcal{T}$
Output:  $True$ iff $\mathcal{T}$ is robust against RC
for $T_1 \in \mathcal{T}$ do
    for $b_1$ a read operation in $T_1$ do
        $G$ := prefix-conflict-free-graph($b_1, T_1, \mathcal{T}\setminus\{T_1\}$);
        $TC$ := reflexive-transitive-closure of $G$;
        for $(T_2, T_m)$ in $TC$ do
            for $a_1 \in T_1$, $a_2 \in T_2$, $b_m \in T_m$ do
                if $a_1$ conflicts with $b_m$ and $b_1$ is rw-conflicting with $a_2$ and ($b_1 <_{T_1} a_1$ or $b_m$ is rw-conflicting with $a_1$ ) then
                    return $False$
return $True$
```

A polynomial time algorithm for deciding robustness that cycles through all possible split schedules is presented as Algorithm 1. Define prefix-conflict-free-graph($b_1, T_1, \mathcal{T}$), for a transaction T_1, an operation $b_1 \in T_1$ and a set of transactions $\mathcal{T}$ with $T_1 \notin \mathcal{T}$, as the graph containing as nodes all transactions in $\mathcal{T}$ that do not contain a ww-conflict with an operation in $\mathsf{prefix}_{b_1}(T_1)$. Furthermore, there is an edge between two transactions T_i and T_j if T_i has an operation that conflicts with an operation in T_j.

Let $\mathcal{P}$ be a set of transaction templates and $\mathbf{D}$ be a database. Then, $\mathcal{P}$ is *robust against RC over $\mathbf{D}$* if for every set of transactions $\mathcal{T}$ that is consistent with $\mathcal{P}$ and $\mathbf{D}$, it holds that $\mathcal{T}$ is robust against RC.

Definition 4.4 (Template Robustness). *A set of transaction templates $\mathcal{P}$ is* robust *against RC if $\mathcal{P}$ is robust against RC for every database $\mathbf{D}$.*

Theorem 4.5 ([16]). *Deciding robustness against RC for a set of transaction templates is decidable in* PTIME.

An extension of Algorithm 1 is provided as Algorithm 2 in [16]. The crux underlying the extension is that, although a single transaction template can have infinitely many instantiations, the concrete choice of tuples as well as the number of different instantiations for the same template that need to be considered to find a multiversion split schedule (if it exists) is bounded.

$$
\begin{array}{lllllll}
T_1\text{:} & \mathtt{U}_1[\mathtt{t}]\,\mathtt{R}_1[\mathtt{v}] & & & & \swarrow \mathtt{U}_1[\mathtt{v}]\,\mathtt{C}_1 \\
T_2\text{:} & & \mathtt{U}_2[\mathtt{t}']\,\mathtt{R}_2[\mathtt{v}]\,\mathtt{U}_2[\mathtt{v}]\,\mathtt{C}_2 & & &
\end{array}
$$

Figure 4: Non-serializable schedule under RC.

5. FUNCTIONAL CONSTRAINTS

Tuples in a database instance are often related to each other, for example through foreign keys. To capture such data dependencies, we use unary functions mapping tuples from one relation to tuples from another relation. Existence of these functions can often be derived from the database schema, but the precise mapping from tuples to other tuples is always instance-specific.

EXAMPLE 5.1. *Reconsider the schema of the SmallBank benchmark discussed in Section 2. To capture the dependencies between tuples induced by the foreign keys, we use two unary functions: $f_{A\to S}$ maps a tuple of type Account to a tuple of type Savings, while $f_{A\to C}$ maps a tuple of type Account to a tuple of type Checking. As Account(CustomerID) is* UNIQUE, *every savings and checking accounts is associated to a unique Account tuple. This is modelled through the functions $f_{S\to A}$ and $f_{C\to A}$ with an analogous interpretation. For database instance $\boldsymbol{D}_1$ introduced in Example 2.1, these functions are defined as follows:*

$$
\begin{array}{ll}
f_{A\to S}(\mathtt{t}) = \mathtt{v} & f_{A\to C}(\mathtt{t}) = \mathtt{q} \\
f_{A\to S}(\mathtt{t}') = \mathtt{v}' & f_{A\to C}(\mathtt{t}') = \mathtt{q}' \\
f_{S\to A}(\mathtt{v}) = \mathtt{t} & f_{C\to A}(\mathtt{q}) = \mathtt{t} \\
f_{S\to A}(\mathtt{v}') = \mathtt{t}' & f_{C\to A}(\mathtt{q}') = \mathtt{t}'
\end{array}
$$

Transaction program instances often access various tuples where some of them are related via such functions. For example, every instance of the program GoPremium in our running example first reads an Account tuple to look up the CustomerID C for the given Name N and then updates the Savings tuple with the corresponding CustomerID C (cf. function $f_{A\to S}$). However, the formalism for transaction templates introduced in Section 3 does not capture this information, thereby allowing program instances that cannot occur in practice, potentially leading to workloads that are falsely identified as not robust against RC.

EXAMPLE 5.2. *Ignoring the functional constraints of template GoPremium in Figure 2, Figure 4 shows a schedule over two instances of GoPremium over database instance $\boldsymbol{D}_1$. Since this schedule is allowed under RC but not serializable, we can conclude that {GoPremium} is not robust against RC. Note however that T_2 cannot occur in practice, as the Account tuple $\mathtt{t}'$ has a different CustomerID than the Savings tuple $\mathtt{v}$ in $\boldsymbol{D}_1$.*

To include these dependencies between different tuples in a program instance, we extend transaction templates with *functional constraints*. For a template τ with variables $\mathtt{X}$ and $\mathtt{Y}$ and a function f, an *equality constraint* is an expression of the form $\mathtt{X} = f(\mathtt{Y})$, and a *disequality* constraint is an expression of the form $\mathtt{X} \neq \mathtt{Y}$. Intuitively, these functional constraints reduce the number of possible instantiations of a template. More formally, a transaction T over a database $\mathbf{D}$ is an instantiation of a template τ, witnessed by a variable mapping μ, if μ satisfies all functional constraints of τ. That is, for every equality constraint $\mathtt{X} = f(\mathtt{Y})$ in τ, $\mu(\mathtt{X}) = f(\mu(\mathtt{Y}))$ holds in $\mathbf{D}$ and for every disequality constraint $\mathtt{X} \neq \mathtt{Y}$ in τ, we have $\mu(\mathtt{X}) \neq \mu(\mathtt{Y})$.

EXAMPLE 5.3. *Reconsider transactions T_1 and T_2 over database instance $\boldsymbol{D}_1$ in Example 5.2. The GoPremium template in Figure 2 has two functional constraints: $\mathtt{Y} = f_{A\to S}(\mathtt{X})$ and $\mathtt{X} = f_{S\to A}(\mathtt{Y})$. Transaction T_1 is an instantiation of GoPremium, as both $\mathtt{v} = f_{A\to S}(\mathtt{t})$ and $\mathtt{t} = f_{S\to A}(\mathtt{v})$ hold in $\boldsymbol{D}_1$ (cf. Example 5.1). However, T_2 is not an instantiation of GoPremium, as $\mathtt{v} = f_{A\to S}(\mathtt{t}')$ and $\mathtt{t}' = f_{A\to S}(\mathtt{v})$ do not hold in $\boldsymbol{D}_1$.*

Functional constraints do not replace the more usual data consistency constraints like key constraints, functional dependencies or denial constraints, ... The latter are intended to verify data consistency, whereas the former are intended to verify whether a set of transactions instantiated from templates are indeed consistent with these templates. The abstraction of functional constraints provides a straightforward mechanism to capture dependencies between tuples implied by foreign key constraints but is not limited to those. For the SmallBank benchmark, for example, we can infer from the fact that Account(CustomerID) is UNIQUE that each checking and savings account is associated to exactly one Account tuple, even though no foreign key from respectively Checking and Savings to Account is defined in the schema.

We say that a transaction template is a *variable transaction template* when it does not contain any functional constraints and an *equality transaction template* when all constraints are equality constraints. We denote these sets by **VarTemp** and **EqTemp**, respectively. For an isolation level $\mathcal{I}$ and a class of transaction templates $\mathcal{C}$, T-ROBUSTNESS($\mathcal{C}$,$\mathcal{I}$) is the problem to decide if a given set of transaction templates $\mathcal{P} \in \mathcal{C}$ is robust against $\mathcal{I}$. When $\mathcal{C}$ is the class of all transaction templates, we simply write T-ROBUSTNESS($\mathcal{I}$).

Unfortunately, adding functional constraints to transaction templates renders the robustness problem undecidable, even when disequality constraints are not allowed:

THEOREM 5.4 ([18]). T-ROBUSTNESS*(**EqTemp**,RC) is undecidable.*

The proof is a reduction from *Post's Correspondence Problem (PCP)* [15] and relies on cyclic dependencies between functional constraints. We emphasize that robustness is defined w.r.t. all possible database instances, instead of one specific database instance. In particular, the undecidability proof shows that there is no bound on the required minimal size of a database instance required to construct a counterexample, if such a counterexample exists.

To obtain decidable fragments, we introduce restrictions on the structure of functional constraints. The *schema graph* $SG(\mathsf{Rels}, \mathsf{Funcs})$ of a schema over the set of relations Rels and with Funcs as the set of implied functions is a directed multigraph having the relations in Rels as nodes, and in which there are as many edges from a node $R \in \mathsf{Rels}$ to node $S \in \mathsf{Rels}$ as there are functions $f \in \mathsf{Funcs}$ with $dom(f) = R$ and $range(f) = S$. We say that a schema is *acyclic* if the multigraph $SG(\mathsf{Rels}, \mathsf{Funcs})$ is acyclic and that it is a *multi-tree* if there is at most one directed path between any two nodes in $SG(\mathsf{Rels}, \mathsf{Funcs})$.

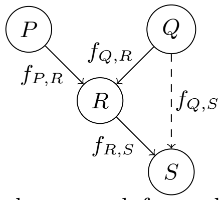


Figure 5: Acyclic schema graph for a schema over four relations and four functions. If we remove function $f_{Q,S}$ (dashed edge), the resulting schema graph is a multi-tree.

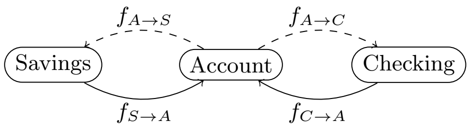


Figure 6: Schema graph for the SmallBank benchmark. The dashed edges correspond to the multi-tree schema graph for the schema restricted to $f_{A\to S}$ and $f_{A\to C}$.

Example 5.5. *Consider the schema over four relations P, Q, R and S, and three functions $f_{P,R}$, $f_{Q,R}$, $f_{R,S}$ with dom$(f_{i,j}) = i$ and range$(f_{i,j}) = j$ for each function $f_{i,j}$. The corresponding schema graph with solid lines is given in Figure 5. This schema is a multi-tree, as there is at most one path between any pair of nodes. Notice that the definition of a multi-tree is more general than a forest, as a node can still have multiple parents (e.g., node R in our example). Adding the function $f_{Q,S}$ with dom$(f_{Q,S}) = Q$ and range$(f_{Q,S}) = S$ results in the schema graph given in Figure 5 that is still acyclic, but no longer a multi-tree as there are now two paths from Q to S.*

Figure 6 displays the schema graph for the SmallBank benchmark that can be seen to be cyclic.

5.1 Templates admitting multi-tree bijectivity

Notice that the equality constraints for all templates in the SmallBank benchmark in Figure 2 imply that the functions $f_{A\to S}$ and $f_{S\to A}$ are bijections and act as each others inverses. Indeed, for every occurrence of a constraint $\mathtt{Y} = f_{A\to S}(\mathtt{X})$ in a template τ, there is a corresponding constraint $\mathtt{X} = f_{S\to A}(\mathtt{Y})$ in τ as well, and vice versa. For $f_{A\to C}$ and $f_{C\to A}$, the observation is analogous.

We say that a set of transaction templates admits *multi-tree bijectivity* if we can partition the functions in pairs (f_i, g_i) such that f_i and g_i are each others inverses (i.e., $\mathtt{Y} = f_i(\mathtt{X})$ occurs in a template τ iff $\mathtt{X} = g_i(\mathtt{Y})$ occurs in τ as well), and every restriction of the schema graph obtained by choosing a single function f from each pair (f_i, g_i) is a multi-tree. We refer to [18] for a more formal definition of multi-tree bijectivity. We denote the class of all sets of templates admitting multi-tree bijectivity by **MTBTemp**.

Example 5.6. *The SmallBank benchmark admits multi-tree bijectivity, witnessed by the partitioning $\{(f_{A\to S}, f_{S\to A}), (f_{A\to C}, f_{C\to A})\}$ and the observation that every schema graph obtained by removing either $f_{A\to S}$ or $f_{S\to A}$, as well as either $f_{A\to C}$ or $f_{C\to A}$, is a multi-tree. For example, the schema graph restricted to $f_{A\to S}$ and $f_{A\to C}$ is a tree and therefore also a multi-tree, as shown in Figure 6.*

The next theorem allows disequalities whereas Theorem 5.4 does not require them.

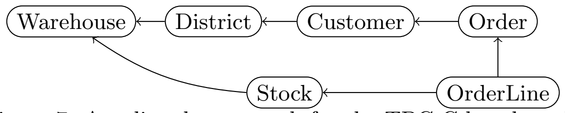


Figure 7: Acyclic schema graph for the TPC-C benchmark.

Theorem 5.7 ([18]). t-robustness*(**MTBTemp**,RC) is decidable in* nlogspace.

The intuition behind the previous result is based on a non-deterministic algorithm guessing the counterexample multi-version split schedule by iteratively adding a new transaction to the schedule, while maintaining a schedule allowed under RC and without contradicting earlier functional constraints. The crux of this approach relies on the fact that for this fragment, the algorithm must keep track of only a constant number of tuples per relation.

5.2 Templates over acyclic schemas

We denote by **AcycTemp** the class of all sets of transaction templates over acyclic schemas. As a concrete example, the schema graph for an abstraction of the TPC-C benchmark[2] is given in Figure 7. Since this schema graph does not contain any cycles, the TPC-C benchmark is situated within **AcycTemp**. Notice in particular how this acyclic schema graph corresponds to the hierarchical structure of many-to-one relationships inherent to the schema for this benchmark. For example, every OrderLine belongs to exactly one Order, and every Order is related to exactly one Customer, but the opposite is never true (i.e., a Customer can be related to multiple Orders, each of which can be related to multiple OrderLines). In general, the results presented in this section can be applied to all workloads over schemas with such a hierarchical structure.

Theorem 5.8 ([18]). t-robustness*(**AcycTemp**,RC) is decidable in* expspace.

The underlying algorithm is similar to the one for Theorem 5.7, but for this fragment the number of tuples that must be maintained to avoid contradicting earlier functional constraints can be exponential.

Lower complexities can be obtained by assuming further restrictions on either the set of transaction templates or the paths in the schema graph. Towards the former, notice that from the instantiation of a variable $\mathtt{X}$ in a template τ with a tuple from a database instance $\mathbf{D}$, we can sometimes infer the tuple from $\mathbf{D}$ that must be assigned to a variable $\mathtt{Y}$ in τ if $\mathtt{X}$ is related to $\mathtt{Y}$ via one or more functional constraints in τ. In that case, we say that $\mathtt{X}$ implies $\mathtt{Y}$ in τ. Informally, a template is said to be *restricted* if for every combination of variables $\mathtt{X}$, $\mathtt{Y}$, $\mathtt{Z}$ and $\mathtt{W}$ in τ with $\mathtt{X}$ implying $\mathtt{Z}$ and $\mathtt{Y}$ implying $\mathtt{W}$ in τ, either $\mathtt{Z}$ implies $\mathtt{W}$ or $\mathtt{W}$ implies $\mathtt{Z}$ in τ. We denote by **AcycResTemp** the class of all sets of restricted transaction templates over acyclic schemas. A more formal definition of restricted templates and the resulting class **AcycResTemp** can be found in [18].

Theorem 5.9 ([18]).

[2]Since our formalism does not support predicate reads (cf. Section 1), we modified the benchmark to only include key-based lookups. More details, as well as the corresponding transaction templates can be found in [19].

1. T-ROBUSTNESS*(**AcycResTemp**,RC) is decidable in* EXPTIME.

2. T-ROBUSTNESS*(**AcycTemp**,RC) is decidable in* PSPACE *when the number of paths between any two nodes in the schema graph is bounded by a constant k.*

Regarding (1), all templates in TPC-C with the exception of NewOrder are restricted. Regarding (2), when the schema graph is a multi-tree then $k = 1$ and for TPC-C $k = 2$ (recall that in general there can be an exponential number of paths), leading to a more practical algorithm for robustness in those cases.

6. CONCLUSION

We presented a high level overview on our recent work on deciding robustness against RC for transaction templates. It should be clear that these techniques can only be used when the set of allowed transaction templates is known beforehand (for instance, when exposed through an API) and does not apply when completely arbitrary transactions can be inserted. As already mentioned in the introduction, our techniques only work when there is a fixed set of read-only attributes that cannot be updated and which are used to select tuples for update. We do not believe these restrictions can be easily lifted. Nevertheless, we do think that the insights obtained through the study of transaction templates can aid in establishing sufficient (but no longer complete) conditions for testing robustness against RC for general transaction programs. We present some initial results in this direction in [20].

A pertinent question is what can be done when a set of transaction templates is found *not* to be robust against RC. In [16], we present a template modification technique based on the insight that an equivalent set of transaction templates robust against RC can be created by promoting `R`-operations to `U`-operations that write back the read value. Such a change does not alter the effect of the transaction templates, but the newly introduced write operation will trigger concurrency mechanisms in the database. Alomari and Fekete [2] presented a modification technique that relies on adding new tuples to the database that act as locks for problematic combinations of transactions, thereby enforcing that these transactions cannot be interleaved with each other and therefore ensuring robustness against RC. A completely different approach is to no longer require that all transactions have to be executed under the same isolation level but only assign stricter isolation levels to problematic transactions. The corresponding *allocation problem* has been investigated on the level of transaction for Snapshot Isolation and Two-phase Locking [9] and RC, Snapshot Isolation, and Serializable Snapshot Isolation [21]. It would be interesting to investigate the allocation problem in the context of transaction templates.

Acknowledgments

This work is partly funded by FWO-grant G019921N.

7. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

[13] B. Ketsman, C. Koch, F. Neven, and B. Vandevoort. Concurrency control for database theorists. *SIGMOD Rec.*, 51(4):6–17, jan 2023.

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

[15] E. L. Post. A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc.*, pages 264–268, 1946.

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

[17] B. Vandevoort, B. Ketsman, C. Koch, and F. Neven. Robustness against read committed: A free transactional lunch. In *PODS*, pages 1–14. ACM, 2022.

[18] B. Vandevoort, B. Ketsman, C. Koch, and F. Neven. Robustness against read committed for transaction templates with functional constraints. In *ICDT*, volume 220 of *LIPIcs*, pages 16:1–16:17, 2022.

[19] B. Vandevoort, B. Ketsman, C. Koch, and F. Neven. Robustness against read committed for transaction templates with functional constraints (full version). `https://arxiv.org/abs/2201.05021`, 2022.

[20] B. Vandevoort, B. Ketsman, C. Koch, and F. Neven. Detecting robustness against MVRC for transaction programs with predicate reads. To appear in EDBT, 2023.

[21] B. Vandevoort, B. Ketsman, and F. Neven. Allocating isolation levels to transactions in a multiversion setting. Manuscript, 2022.