Robustness Against Read Committed: A Free Transactional Lunch

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

Bas Ketsman
Vrije Universiteit Brussel
Brussels, Belgium
bas.ketsman@vub.be

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

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

ABSTRACT

Transaction processing is a central part of most database applications. While serializability remains the gold standard for desirable transactional semantics, many database systems offer improved transaction throughput at the expense of introducing potential anomalies through the choice of a lower isolation level. Transactions are often not arbitrary but are constrained by a set of transaction programs defined at the application level (as is the case for TPC-C for instance), implying that not every potential anomaly can effectively be realized. The question central to this paper is the following: when - within the context of specific transaction programs - do isolation levels weaker than serializability, provide the same guarantees as serializability? We refer to the latter as the robustness problem. This paper surveys recent results on robustness testing against (multiversion) read committed focusing on complete rather than sufficient conditions. We show how to lift robustness testing to transaction templates as well as to programs to increase practical applicability. We discuss open questions and highlight promising directions for future research.

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

databases, transactions, isolation levels, robustness

ACM Reference Format:
Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2022. Robustness Against Read Committed: A Free Transactional Lunch. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3517804.3524162

1 INTRODUCTION

Clients interact with a DBMS by issuing transactions which, on a conceptual level, are complex operations involving multiple database objects. A transaction might, for instance, transfer a specific amount of money from a savings to a checking account of the same customer. The DBMS guarantees that transactions are executed completely or not at all (*atomicity*), even in the case of failures. Concurrent access to data is facilitated by interleaving operations in transactions while guaranteeing consistency through a serializable isolation level. Serializability ensures that the effect of concurrent execution of transactions is always equivalent to a serial execution. The database system thereby guarantees perfect isolation for every transaction which is of paramount importance to application programmers as it allows them to restrict attention on the correctness properties of individual transactions.

Ensuring serializability, however, comes at a non-trivial performance cost [48]. Many approaches to increase transaction throughput have been proposed: improved or novel pessimistic (cf., e.g., [24, 34, 36, 42, 49]) or optimistic (cf., e.g., [10, 11, 16, 17, 22, 23, 25, 27–29, 31, 37, 38, 50, 51]) algorithms, as well as approaches based on coordination avoidance (cf., e.g., [18, 19, 30, 33, 35, 40, 41]). An orthogonal approach offered by many database systems is to trade off isolation guarantees for improved performance by offering a variety of isolation levels. Even though isolation levels lower than serializability are often configured by default (see, e.g., [4]), executing transactions concurrently under such isolation levels is not without risk as it can introduce certain anomalies. Sometimes, however, a set of transactions can be executed at an isolation level lower than serializability without introducing any anomalies. This is for instance the case for the TPC-C benchmark application [43] running under SNAPSHOT ISOLATION. In such a case, the set of transactions is said to be *robust* against a particular isolation level. More formally, *a set of transactions is robust against a given isolation level if every possible interleaving of the transactions allowed under the specified isolation level is serializable.* Detecting robustness is highly desirable as it allows guaranteeing perfect isolation at the performance cost of a lower isolation level.

Fekete et al [21] initiated the study of robustness in the context of SNAPSHOT ISOLATION, referring to it as the *acceptability* problem, and providing a sufficient condition in terms of the absence of cycles with specific types of edges in the static dependency graph. These algorithms are sound but not complete as they can result in false negatives but never in false positives. This result was extended by Bernardi and Gotsman [9] by providing sufficient conditions for deciding robustness against the different isolation levels that can be defined in a declarative framework as introduced by Cerone et al [12]. This framework provides a uniform specification of various isolation levels (including SNAPSHOT ISOLATION) that admit atomic visibility, a condition requiring that either all or none of the updates of each transaction are visible to other transactions. The atomic visibility assumption is key as it allows specifying isolation levels by consistency axioms on the level of transactions rather than on the granularity of individual operations within each transaction. The sufficient conditions are again based on the absence of cycles with certain types of edges. A related but different notion is that of transaction chopping which splits transactions into smaller pieces to obtain performance benefits and is correct if, for every serializable execution of the chopping, there is an equivalent serializable execution of the original transactions [39]. Cerone et al. [13, 14] studied chopping under various isolation levels.

While robustness received quite a bit of attention in the literature, most existing work focuses on SNAPSHOT ISOLATION [2, 6, 20, 21] or higher isolation levels [7, 9, 12, 15]. So far, robustness against lower isolation levels such as different variations of the Read Committed isolation level has received only little attention (one notable exception being the work by Alomari and Fekete [3]). This might seem surprising, as Read Committed is very common, often even the default isolation level in quite a number of database systems [5], and also one of the few isolation levels providing highly available transactions [4]. As Read Committed provides a low performance penalty, establishing robustness against this isolation level allows rapid concurrent execution while guaranteeing perfect isolation.

In this paper, we survey recent work on the robustness problem against (multiversion) read committed [26, 45–47]. A distinguishing approach is that we initially focus on sound *and* complete algorithms and set steps towards practical application of these results through increasingly more general formalisations of transaction programs.

In Section 3, we study robustness for the simplified setting where transactions are mere sequences of reads and writes whose structure as well as the objects in the database that they will access, is known beforehand. This setting is most similar to the one considered by Fekete [20] who obtained sound and complete characterizations for deciding robustness against SNAPSHOT ISOLATION. We provide complete characterizations for robustness against single-version as well as multiversion read committed.

In practice, transactions are often generated by a (finite) number of transaction programs (for instance, made available through an API). The TPC-C benchmark [43] for example consists of five different transaction programs, from which an unlimited number of concrete transactions can be instantiated. In Section 4, we extend (part of) the previous results to a formalization of transaction programs that we refer to as transaction templates, which, conceptually, are functions with parameters. Robustness then becomes a static property that can be tested offline at API design time. When the set of templates passes the test, the database isolation level can be set to read committed for that API without fear for introducing anomalies. We obtain a sound and complete algorithm to decide robustness for transaction templates against multiversion read committed. Larger sets of transaction templates can be determined to be robust when taking additional constraints into account (like foreign key constraints). While such constraints render the robustness problem undecidable in general, there are restrictions that make the problem decidable and even tractable.

A major shortcoming of the machinery presented in Section 4 is that it can not be extended to predicate reads, inserts or deletes, operations that are common in real world transactions. We therefore depart from our quest in obtaining sound and complete characterizations in Section 5 and discuss how robustness can be tested for more general transaction programs formalized as BTPs incorporating predicate reads, inserts, deletes as well as control structures. A main advantage is that once SQL transaction programs are translated into the formalism of BTP, robustness testing is fully automatic and no longer requires the intervention of a database administrator to predict possible conflicts as is the case in earlier work on robustness.

In Section 6 we discuss direction for future work. The treatment in this paper is rather high level. We therefore refer to [44] for a more in depth discussion and missing formal definitions.

2 PRELIMINARIES

We start by introducing the necessary definitions. Our formalization of transactions and conflict serializability is based on [20]. 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.

2.1 Transactions and Schedules

For natural numbers i and j with $i \leq j$, denote by $[i, j]$ the set $\{i, \ldots, j\}$. We fix an infinite set of objects **Obj**. For an object $\mathtt{t} \in$ **Obj**, we denote by $\mathtt{R[t]}$ a *read* operation on $\mathtt{t}$ and by $\mathtt{W[t]}$ a *write* operation on $\mathtt{t}$. We also assume a special *commit* operation denoted by $\mathtt{C}$. A *transaction* T is a sequence of read and write operations on objects in **Obj** followed by a commit. 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 an operation $b \in T$, we denote by $\mathsf{prefix}_b(T)$ the restriction of T to all operations that are smaller than or equal to b according to $\leq_T$. Similarly, we denote by $\mathsf{postfix}_b(T)$ the restriction of T to all operations that are strictly larger than b according to $\leq_T$. We interchangeably consider transactions both as linear orders as well as sequences.

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 from different transactions, we add this id as index to the operation. That is, we write $\mathtt{W}_i\mathtt{[t]}$ and $\mathtt{R}_i\mathtt{[t]}$ to denote a write and read on object $\mathtt{t}$ 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.* [8, 20]). To avoid ambiguity of notation, we assume that a transaction performs at most one read operation and at most one write operation per object. The latter is a common assumption (see, *e.g.* [20]). All results carry over to the more general setting in which multiple writes and reads per tuple are allowed.

Example 1. *As a running example, consider the set of transactions* $\mathcal{T} = \{T_1, T_2, T_3\}$ *with* $T_1 = \mathtt{R}_1[\mathtt{t}]\,\mathtt{W}_1[\mathtt{v}]\,\mathtt{C}_1$, $T_2 = \mathtt{R}_2[\mathtt{v}]\,\mathtt{W}_2[\mathtt{q}]\,\mathtt{C}_2$ *and* $T_3 = \mathtt{R}_3[\mathtt{q}]\,\mathtt{R}_3[\mathtt{t}]\,\mathtt{W}_3[\mathtt{t}]\,\mathtt{W}_3[\mathtt{q}]\,\mathtt{C}_3$. *If we take* $b = \mathtt{R}_3[\mathtt{t}]$ *in transaction* T_3, *then* $\mathrm{prefix}_b(T_3) = \mathtt{R}_3[\mathtt{q}]\,\mathtt{R}_3[\mathtt{t}]$ *and* $\mathrm{postfix}_b(T_3) = \mathtt{W}_3[\mathtt{t}]\,\mathtt{W}_3[\mathtt{q}]\,\mathtt{C}_3$. □

A *(multiversion) schedule* s over a set $\mathcal{T}$ of transactions is a tuple $(O_s, \leq_s, \ll_s, v_s)$ where O_s is the set containing all operations of transactions in $\mathcal{T}$ as well as a special operation op_0 conceptually writing the initial versions of all existing objects, $\leq_s$ encodes the ordering of these operations, $\ll_s$ is a *version order* providing for each object t a total order over all write operations on t occurring in s, and v_s is a *version function* mapping each read operation a in s to either op_0 or to a write operation different from a 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. This separate version order is needed, as for a pair of write operations on the same object, the order in which these versions are installed does not necessarily coincide with $\leq_s$.

A schedule s is a *single version schedule* if $\ll_s$ coincides 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 a schedule is consistent with the common assumption [9, 20] that an underlying recovery mechanism will roll back transactions that interfere with aborted transactions.

Example 2. *Consider the set of transactions* $\mathcal{T}$ *given in Example 1. Let* s_1 *be the schedule where the ordering* $\leq_{s_1}$ *of operations is*

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

the version order $\ll_{s_1}$ *equals* $op_0 \ll_{s_1} \mathtt{W}_3[\mathtt{t}]$ *for object* t, $op_0 \ll_{s_1} \mathtt{W}_1[\mathtt{v}]$ *for object* v *and* $op_0 \ll_{s_1} \mathtt{W}_2[\mathtt{q}] \ll_{s_1} \mathtt{W}_3[\mathtt{q}]$ *for object* q, *and the version function* v_{s_1} *is*

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

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* q *that should precede the version installed by transaction* T_3. *Furthermore,* $v_{s_1}(\mathtt{R}_3[\mathtt{t}]) = op_0$ *and* $v_{s_1}(\mathtt{R}_1[\mathtt{t}]) = \mathtt{W}_3[\mathtt{t}]$ *imply that* T_3 *observes the initial version of* 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* t, $op_0 \ll_{s_2} \mathtt{W}_1[\mathtt{v}]$ *for object* v *and* $op_0 \ll_{s_2} \mathtt{W}_3[\mathtt{q}] \ll_{s_2} \mathtt{W}_2[\mathtt{q}]$ *for object* q, *and the version function* v_{s_2} *is*

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

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* q *installed by* T_3 *should precede the version of* T_2, *even though this version of* T_3 *is installed later according to* $\leq_{s_2}$. *Furthermore, the read operation* $\mathtt{R}_1[\mathtt{t}]$ *does not read the most recent version, as it observes the initial version of* t *rather than the more recent version written by* $\mathtt{W}_3[\mathtt{t}]$. □

2.2 Conflict Serializability

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

- *(ww-conflict)* $a_j = \mathtt{W}_j[\mathtt{t}]$ and $b_i = \mathtt{W}_i[\mathtt{t}]$; or,
- *(wr-conflict)* $a_j = \mathtt{W}_j[\mathtt{t}]$ and $b_i = \mathtt{R}_i[\mathtt{t}]$; or,
- *(rw-conflict)* $a_j = \mathtt{R}_j[\mathtt{t}]$ and $b_i = \mathtt{W}_i[\mathtt{t}]$.

In this case, we also say that a_j and b_i are conflicting operations. Furthermore, commit operations and the special operation op_0 never conflict with any other operation. When a_j and b_i 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:[1]

- *(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 $v_s(b_i) \ll_s a_j$.

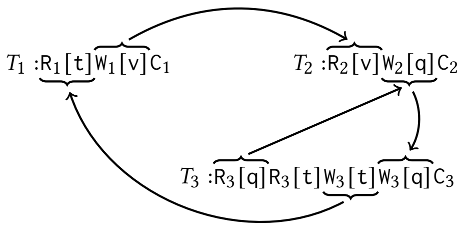
Intuitively, a ww-dependency from b_i to a_j implies that a_j writes a version of an object that is installed after the version written by b_i. A wr-dependency from b_i to a_j implies that b_i either writes the version observed by a_j, or it writes a version that is installed before the version observed 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. *Consider schedule* s_2 *as defined in Example 2. In this schedule, the dependency* $\mathtt{W}_3[\mathtt{q}] \to_{s_2} \mathtt{W}_2[\mathtt{q}]$ *is a ww-dependency since* $\mathtt{W}_3[\mathtt{q}] \ll_{s_2} \mathtt{W}_2[\mathtt{q}]$. *This 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* $\mathtt{R}_1[\mathtt{t}] \to_{s_2} \mathtt{W}_3[\mathtt{t}]$ *is a rw-antidependency, witnessed by* $v_{s_2}(\mathtt{R}_1[\mathtt{t}]) = op_0 \ll_{s_2} \mathtt{W}_3[\mathtt{t}]$. □

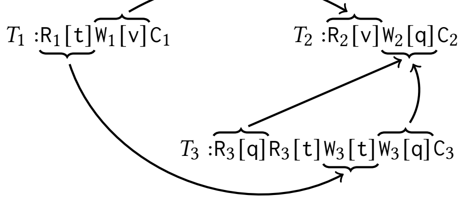
Two schedules s and s' are *conflict equivalent* if they are over the same set $\mathcal{T}$ of transactions and for every pair of conflicting operations a_j and b_i, $b_i \to_s a_j$ iff $b_i \to_{s'} a_j$.

Definition 4. A schedule s is *conflict serializable* if it is conflict equivalent to a single version serial schedule.

[1]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]W$_1$[v]C$_1$
T_2 : R$_2$[v]W$_2$[q]C$_2$
T_3 : R$_3$[q]R$_3$[t]W$_3$[t]W$_3$[q]C$_3$

(a) $CG(s_1)$


T_1 : R$_1$[t]W$_1$[v]C$_1$
T_2 : R$_2$[v]W$_2$[q]C$_2$
T_3 : R$_3$[q]R$_3$[t]W$_3$[t]W$_3$[q]C$_3$

(b) $CG(s_2)$

Figure 1: Conflict graphs for schedules s_1 and s_2 as defined in Example 2.

A *conflict graph*[2] $CG(s)$ 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 $b_i \rightarrow_s a_j$. Since we are usually not only interested in the existence of conflicting 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 $b_i \in T_i$ that conflicts with an operation $a_j \in T_j$ and $b_i <_s a_j$. For ease of notation, we choose to represent $CG(s)$ 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 conflicting operations b_i and a_j. Henceforth, we refer to these quadruples simply as edges. Notice that edges cannot contain commit operations.

A *cycle* C in $CG(s)$ 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 $CG(s)$, 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 C, we denote by $C[T_i]$ the cycle obtained from C by letting T_i start and conclude the cycle while otherwise respecting the order of transactions in C. That is, $C[T_i]$ 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 5 ([32]). *A schedule s is conflict serializable iff the conflict graph for s is acyclic.*

[2]The term serialization graph is used to denote the multiversion equivalent of the (single version) conflict graph. We use the term conflict graph in Section 3 (also for multiversion conflict graphs) but switch to serialization graph in Section 5 to be consistent with [46].

Example 6. *The conflict graphs for schedules s_1 and s_2 in Example 2 are given in Figure 1. Since $CG(s_1)$ contains cycles, we conclude that s_1 is not conflict serializable. The conflict graph $CG(s_2)$ 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 $T_1 \cdot T_3 \cdot T_2$.* □

2.3 Isolation Levels

An *isolation level* is a set of constraints over all possible schedules. A schedule s *is allowed under an isolation level* $\mathcal{I}$ if s adheres to all constraints set forth by $\mathcal{I}$. We define isolation levels in terms of the concurrency phenomena that we want to exclude from schedules [8]. It is accustomed to view an isolation level as a set of allowed schedules [32]. We say that an isolation level $\mathcal{I}$ is a *restriction* of an isolation level $\mathcal{I}'$, denoted $\mathcal{I} \subseteq \mathcal{I}'$, if the fact that a schedule s is allowed under $\mathcal{I}$ implies that s is allowed under $\mathcal{I}'$ as well.

3 ROBUSTNESS FOR TRANSACTIONS

We start out by considering robustness in a restricted setting where the set of transactions under consideration is assumed to be known in advance. Even though this restricted setting has little direct practical relevance for the simple reason that transaction sets $\mathcal{T}$ are usually not known in advance, we show in the next section that the obtained results can be generalized to offline robustness testing on the application level where interaction with the database happens through an API consisting of a fixed set of transaction programs.

The robustness property [9] (also called *acceptability* in [20, 21]) guarantees serializability for all schedules of a given set of transactions for a given isolation level:

Definition 7 (Robustness). A set $\mathcal{T}$ of transactions is *robust* against an isolation level if every schedule for $\mathcal{T}$ that is allowed under that isolation level is conflict serializable.

For an isolation level $\mathcal{I}$, robustness($\mathcal{I}$) is the problem to decide if a given set of transactions $\mathcal{T}$ is robust against $\mathcal{I}$.

3.1 Single Version Read (Un)Committed

For a single version schedule $s = (O_s, \leq_s, \ll_s, v_s)$, the version order $\ll_s$ and version function v_s are by definition implied by $\leq_s$. Because of this, dependencies between conflicting operations in s can be identified more directly based on the ordering of operations $\leq_s$ itself rather than $\ll_s$ and v_s:

Proposition 8. *Let a_j and b_i be two conflicting operations occurring in a single version schedule s over a set of transactions $\mathcal{T}$. Then $b_i \rightarrow_s a_j$ iff $b_i <_s a_j$. Furthermore, (T_i, b_i, a_j, T_j) is an edge in $CG(s)$ iff $b_i <_s a_j$.*

Since $\ll_s$ and v_s can be derived from $\leq_s$ for a single version schedule s, we facilitate presentation in this subsection by omitting $\ll_s$ and v_s in our notation, denoting s by the tuple $(O_s, \leq_s)$ instead.

Let s be a single version schedule for a set $\mathcal{T}$ of transactions.

- Then, *s exhibits a dirty write* iff there are two different transactions T_i and T_j in $\mathcal{T}$ and an object t such that

$$\mathtt{W}_i[\mathtt{t}] <_s \mathtt{W}_j[\mathtt{t}] <_s \mathtt{C}_i.$$

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.

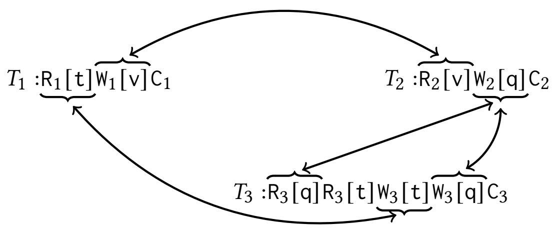


Figure 2: $IG(\mathcal{T})$ for $\mathcal{T} = \{T_1, T_2, T_3\}$ as defined in Example 13.

- Furthermore, s *exhibits a dirty read* iff there are two different transactions T_i and T_j in $\mathcal{T}$ and an object t such that
$$\mathtt{W}_i[\mathtt{t}] <_s \mathtt{R}_j[\mathtt{t}] <_s \mathtt{C}_i.$$
That is, transaction T_j reads an object that has been modified earlier by T_i, but T_i has not yet issued a commit.

Definition 9. A single version schedule is *allowed under isolation level* READ UNCOMMITTED if it exhibits no dirty writes, and it is *allowed under isolation level* READ COMMITTED if, in addition, it also exhibits no dirty reads.

Notice that READ COMMITTED is a restriction of READ UNCOMMITTED, as every schedule allowed under READ COMMITTED is also allowed under READ UNCOMMITTED.

We use a variant of the interference graph, as introduced by Fekete [20], which essentially lifts the notion of a conflict graph from schedules to sets of transactions. Consistent with our definition of conflict graph, we expose conflicting operations via an explicit labeling of edges.

Definition 10. For a set of transactions $\mathcal{T}$, the *interference graph* $IG(\mathcal{T})$ for $\mathcal{T}$ is the graph whose nodes are the transactions in $\mathcal{T}$ and where there is an edge from T_i to T_j if there is an operation in T_i that conflicts with some operation in T_j. Again, we assume a labeling function λ mapping each edge to a set of pairs of conflicting operations. Formally, $(b_i, a_j) \in \lambda(T_i, T_j)$ iff there is an operation $b_i \in T_i$ that conflicts with an operation $a_j \in T_j$.

For convenience, just like for conflict graphs, we choose to represent $IG(\mathcal{T})$ as a set of quadruples of the form (T_i, b_i, a_j, T_j). That is, $(T_i, b_i, a_j, T_j) \in IG(\mathcal{T})$ iff there is an edge (T_i, T_j) and $(b_i, a_j) \in \lambda(T_i, T_j)$. Again, we then refer to these quadruples simply as edges.

Notice that $(T_i, b_i, a_j, T_j) \in IG(\mathcal{T})$ implies $(T_j, a_j, b_i, T_i) \in IG(\mathcal{T})$. Furthermore, the conflict graph $CG(s)$ for a schedule s for $\mathcal{T}$ is always a subgraph of the interference graph $IG(\mathcal{T})$ for $\mathcal{T}$. Therefore, every cycle in $CG(s)$ is a cycle in $IG(\mathcal{T})$. However, the converse is not always true. Sometimes a cycle in $IG(\mathcal{T})$ can be found that does not translate to a corresponding cycle in the conflict graph for any schedule for $\mathcal{T}$. We therefore introduce the notion of a *transferable cycle* in an interference graph and show in Lemma 14 that whenever there is a transferable cycle in $IG(\mathcal{T})$ there is a single version schedule s of a specific form called a split schedule (as defined in Definition 12) that admits a cycle in $CG(s)$.

Definition 11. Let $\mathcal{T}$ be a set of transactions and C a cycle in $IG(\mathcal{T})$. Then, C is *non-trivial* if for some pair of edges (T_i, b_i, a_j, T_j) and (T_j, b_j, a_k, T_k) in C the operations b_j and a_j are different. Furthermore, C is *transferable* if $b_j <_{T_j} a_j$ for some pair of edges (T_i, b_i, a_j, T_j) and (T_j, b_j, a_k, T_k) in C. We then say that C is transferable in T_j on operations (b_j, a_j).

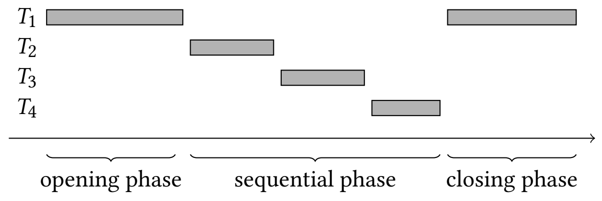


Figure 3: Abstract presentation of a split schedule for four transactions. The drawing omits a possible trailing sequence of non-interleaved transactions (cf. Definition 12).

When a cycle is transferable in T on (b, a), we create a split schedule by splitting T between b and a, inserting all other transactions from the cycle in the created opening while maintaining their ordering and appending at the end all other transactions not occurring in the cycle in an arbitrary order. Notice that split schedules exhibit a cycle in their conflict graph. Split schedules are formally defined as follows:

Definition 12 (Split schedule)**.** Let $\mathcal{T}$ be a set of transactions and C a transferable cycle in $IG(\mathcal{T})$. A *split schedule* for $\mathcal{T}$ based on C is a single version schedule having the form
$$\mathsf{prefix}_b(T_1) \cdot T_2 \cdot \ldots \cdot T_m \cdot \mathsf{postfix}_b(T_1) \cdot T_{m+1} \cdot \ldots \cdot T_n,$$
where

- (T_m, b_m, a, T_1) and (T_1, b, a_2, T_2) is a pair of edges in C and C is transferable in T on (b, a);
- $T_1, \ldots, T_m$ are the transactions in $C[T_1]$ in the order as they occur[3]; and,
- $T_{m+1}, \ldots, T_n$ are the remaining transactions in $\mathcal{T}$ in an arbitrary order.

More specifically, we say that the above schedule is a split schedule for $\mathcal{T}$ based on C, T_1 and b.

We say that a single version schedule s is a split schedule for $\mathcal{T}$ if there is a transferable cycle C in $IG(\mathcal{T})$ such that s is a split schedule for $\mathcal{T}$ based on C. Figure 3 provides an abstract view of a split schedule omitting the trailing sequence $T_{m+1} \cdots T_n$.

EXAMPLE 13. *Consider* $\mathcal{T} = \{T_1, T_2, T_3\}$ *with* $T_1 = \mathtt{R}_1[\mathtt{t}]\,\mathtt{W}_1[\mathtt{v}]\,\mathtt{C}_1$, $T_2 = \mathtt{R}_2[\mathtt{v}]\,\mathtt{W}_2[\mathtt{q}]\,\mathtt{C}_2$ *and* $T_3 = \mathtt{R}_3[\mathtt{q}]\,\mathtt{R}_3[\mathtt{t}]\,\mathtt{W}_3[\mathtt{t}]\,\mathtt{W}_3[\mathtt{q}]\,\mathtt{C}_3$. *Then* $IG(\mathcal{T})$ *is depicted in Figure 2. The cycle* C_1 *consisting of the following edges*
$$(T_1, \mathtt{W}_1[\mathtt{v}], \mathtt{R}_2[\mathtt{v}], T_2), (T_2, \mathtt{W}_2[\mathtt{q}], \mathtt{W}_3[\mathtt{q}], T_3), (T_3, \mathtt{W}_3[\mathtt{t}], \mathtt{R}_1[\mathtt{t}], T_1)$$
is transferable in T_3 *on* $(\mathtt{W}_3[\mathtt{t}], \mathtt{W}_3[\mathtt{q}])$ *as* $\mathtt{W}_3[\mathtt{t}] <_{T_3} \mathtt{W}_3[\mathtt{q}]$. *The cycle* C_2 *consisting of the following edges*
$$(T_1, \mathtt{W}_1[\mathtt{v}], \mathtt{R}_2[\mathtt{v}], T_2), (T_2, \mathtt{W}_2[\mathtt{q}], \mathtt{R}_3[\mathtt{q}], T_3), (T_3, \mathtt{W}_3[\mathtt{t}], \mathtt{R}_1[\mathtt{t}], T_1)$$

[3]Recall from Section 2.2 that $C[T_1]$ denotes the cycle obtained from C by letting T_1 start and conclude the cycle while otherwise respecting the order of transactions in C.

is not transferable in T_3 on $(\mathtt{W}_3[\mathtt{t}], \mathtt{R}_3[\mathtt{q}])$ as $\mathtt{W}_3[\mathtt{t}] \not<_{T_3} \mathtt{R}_3[\mathtt{q}]$. The split schedule s_1 for $\mathcal{T}$ based on C_1, T_3, and $\mathtt{W}_3[\mathtt{t}]$ is as follows:

$$\underbrace{\mathtt{R}_3[\mathtt{q}]\mathtt{R}_3[\mathtt{t}]\mathtt{W}_3[\mathtt{t}]}_{\mathsf{prefix}_b(T_3)}\ \underbrace{\mathtt{R}_1[\mathtt{t}]\mathtt{W}_1[\mathtt{v}]\mathtt{C}_1}_{T_1}\ \underbrace{\mathtt{R}_2[\mathtt{v}]\mathtt{W}_2[\mathtt{q}]\mathtt{C}_2}_{T_2}\ \underbrace{\mathtt{W}_3[\mathtt{q}]\mathtt{C}_3}_{\mathsf{postfix}_b(T_3)},$$

with $b = \mathtt{W}_3[\mathtt{t}]$. □

The following lemma collects some interesting properties of transactions.

Lemma 14. *Let $\mathcal{T}$ be a set of transactions.*

(1) If a schedule s for $\mathcal{T}$ has a cycle C in its conflict graph, then C is a transferable cycle in $IG(\mathcal{T})$.
(2) If there is a non-trivial cycle C in $IG(\mathcal{T})$ then there is a transferable cycle C' in $IG(\mathcal{T})$.
(3) Let s be a split schedule for $\mathcal{T}$ based on a transferable cycle C in $IG(\mathcal{T})$. Then C is a cycle in $CG(s)$.

Read Uncommitted. For a cycle to guarantee the existence of a schedule allowed under READ UNCOMMITTED, we need an additional property:

Definition 15. Let $\mathcal{T}$ be a set of transactions and let C be a cycle in $IG(\mathcal{T})$. Let $T \in \mathcal{T}$ and $b, a \in T$. Then, C is *prefix-write-conflict-free in T on operations (b, a)* if C is transferable in T on operations (b, a) and there is no write operation in $\mathsf{prefix}_b(T)$ that conflicts with a write operation in a transaction in $C \setminus \{T\}$.[4]

Furthermore, C is *prefix-write-conflict-free* if it is prefix-write-conflict-free in T on (b, a) for some $T \in \mathcal{T}$ and some operations $b, a \in T$.

Example 16. *Cycle C_1 of Example 13 is prefix-write-conflict-free in T_3 on operations $(\mathtt{W}_3[\mathtt{t}], \mathtt{W}_3[\mathtt{q}])$. Indeed, there is no write operation in T_2 or T_1 to object $\mathtt{t}$. Notice that the split schedule s_1 of Example 13 is allowed under* READ UNCOMMITTED. □

The next theorem then characterizes robustness against READ UNCOMMITTED:

Theorem 17 ([26]). *Let $\mathcal{T}$ be a set of transactions. The following are equivalent:*

(1) $\mathcal{T}$ is not robust against isolation level READ UNCOMMITTED*;*
(2) $IG(\mathcal{T})$ contains a prefix-write-conflict-free cycle; and,
(3) there is a split schedule s for $\mathcal{T}$ that is allowed under READ UNCOMMITTED.

Using the above characterization, the following theorem establishes the complexity of deciding robustness against READ UNCOMMITTED:

Theorem 18 ([26]). ROBUSTNESS(READ UNCOMMITTED) *is* LOGSPACE*-complete.*

Read Committed. Robustness against READ COMMITTED can be characterized in terms of the absence of counterexample schedules of a particular form:

Definition 19. Let $\mathcal{T}$ be a set of transactions and C a cycle in $IG(\mathcal{T})$ that is transferable in its first transaction T_1 on operations (b_1, a_1). A *multi-split schedule for $\mathcal{T}$ based on C* is any schedule of the form

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

with $T_1, \ldots, T_m$ denoting the transactions occurring in C in the order as they occur, and with $T_{m+1}, \ldots, T_n$ denoting the remaining transactions in $\mathcal{T}$ in an arbitrary order. Here, ε is a function that maps each transaction occurring in C to one of its operations and that satisfies the following conditions: for every $i > 1$,

(1) $\varepsilon(T_1) = b_1$;
(2) if $\varepsilon(T_{i-1}) = \mathtt{C}_{i-1}$ then $\varepsilon(T_i) = \mathtt{C}_i$;[5] and,
(3) if $\varepsilon(T_{i-1}) \neq \mathtt{C}_{i-1}$ then $\varepsilon(T_i) = b_i$ or $\varepsilon(T_i) = \mathtt{C}_i$ with the edge (T_i, b_i, a_j, T_j) in C where $j = 1$ if $i = m$ and $j = i + 1$ otherwise.

The transaction T_i is called open when $\varepsilon(T_i) \neq \mathtt{C}_i$ and is closed otherwise. Notice that for a closed transaction T_i, $\mathsf{prefix}_{\varepsilon(T_i)}(T_i) = T_i$ and $\mathsf{postfix}_{\varepsilon(T_i)}(T_i)$ is empty. A multi-split schedule is *fully split* when all transactions are open, that is, $\varepsilon(T_i) \neq \mathtt{C}_i$ for all $i \in [1, m]$.

We say that s is a multi-split schedule for $\mathcal{T}$ if it is a multi-split schedule for $\mathcal{T}$ based on some cycle C. Notice that there is always a number $k > 0$ such that the first k transactions occurring in C are open and the others (if any) are closed. In a *fully split* schedule there are no closed transactions.

The next lemma establishes that a multi-split schedule gives rise to a cycle in the corresponding conflict graph.

Lemma 20 ([26]). *Let s be a multi-split schedule for a set of transactions $\mathcal{T}$ based on a cycle C in $IG(\mathcal{T})$. Then C is also a cycle in $CG(s)$.*

Notice that the previous lemma only implies that s is not conflict serializable, but does not state whether s is allowed under READ COMMITTED. However, a multi-split schedule can only serve as a valid counterexample for robustness against READ COMMITTED if it is allowed under READ COMMITTED. To this end, we introduce the definition of a multi-prefix-conflict-free cycle.

In the following definition, T and T' intuitively refer to the first open and last open transaction in the multi-split schedule that can be constructed from a multi-prefix-conflict-free cycle.

Definition 21. Let $\mathcal{T}$ be a set of transactions and let C be a cycle in $IG(\mathcal{T})$ containing transactions T and T'. Then C is *multi-prefix-conflict-free* in T and T' if C is transferable in T and for every transaction T_i that is equal to T' or occurs before T' in $C[T]$ there is no write operation in $\mathsf{prefix}_{b(T_i)}(T_i)$ that[6]

- conflicts with a read or write operation in $\mathsf{prefix}_{b(T_j)}(T_j)$ of some transaction T_j occurring after T_i but before or equal to T' in $C[T]$; or,
- conflicts with a read or write operation in some transaction T_j occurring after T' in $C[T]$; or,

[4]We abuse notation here and denote the set of transactions occurring in C also by C.

[5]Recall that $\mathtt{C}_{i-1}$ and $\mathtt{C}_i$ are the commit operations of transactions T_{i-1} and T_i, respectively.

[6]Recall from Section 2.2 that $C[T]$ denotes the cycle obtained from C by letting T start and conclude the cycle while otherwise respecting the order of transactions in C.

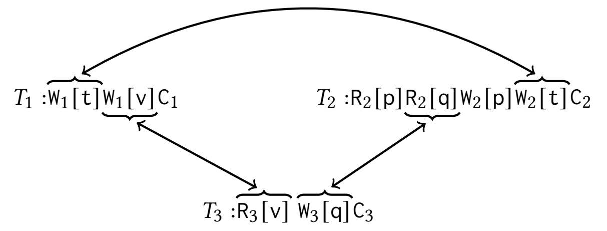

T_1 :W$_1$[t]W$_1$[v]C$_1$
T_2 :R$_2$[p]R$_2$[q]W$_2$[p]W$_2$[t]C$_2$
T_3 :R$_3$[v] W$_3$[q]C$_3$

Figure 4: $IG(\mathcal{T})$ for $\mathcal{T} = \{T_1, T_2, T_3\}$ as defined in Example 22.

- conflicts with a read or write operation in $\text{postfix}_{b(T_j)}(T_j)$ of some transaction T_j occurring strictly before T_i in $C[T]$.

Example 22. *Consider $\mathcal{T} = \{T_1, T_2, T_3\}$ with $T_1 = \mathtt{W}_1[\mathtt{t}]\,\mathtt{W}_1[\mathtt{v}]\,\mathtt{C}_1$, $T_2 = \mathtt{R}_2[\mathtt{p}]\,\mathtt{R}_2[\mathtt{q}]\,\mathtt{W}_2[\mathtt{p}]\,\mathtt{W}_2[\mathtt{t}]\,\mathtt{C}_2$ and $T_3 = \mathtt{R}_3[\mathtt{v}]\,\mathtt{W}_3[\mathtt{q}]\,\mathtt{C}_3$. Then $IG(\mathcal{T})$ is depicted in Figure 4. The cycle C consisting of the following edges*

$$(T_1, \mathtt{W}_1[\mathtt{t}], \mathtt{W}_2[\mathtt{t}], T_2), (T_2, \mathtt{R}_2[\mathtt{q}], \mathtt{W}_3[\mathtt{q}], T_3), (T_3, \mathtt{R}_3[\mathtt{v}], \mathtt{W}_1[\mathtt{v}], T_1)$$

is multi-prefix-conflict-free in T_1 and T_2. The multi-split schedule s for $\mathcal{T}$ based on C where T_1 and T_2 are open and T_3 is closed is as follows:

$$\underbrace{\mathtt{W}_1[\mathtt{t}]}_{\text{prefix}_{b_1}(T_1)}\ \underbrace{\mathtt{R}_2[\mathtt{p}]\mathtt{R}_2[\mathtt{q}]}_{\text{prefix}_{b_2}(T_2)}\ \underbrace{\mathtt{R}_3[\mathtt{v}]\mathtt{W}_3[\mathtt{q}]\mathtt{C}_3}_{T_3}\ \underbrace{\mathtt{W}_1[\mathtt{v}]\mathtt{C}_1}_{\text{postfix}_{b_1}(T_1)}\ \underbrace{\mathtt{W}_2[\mathtt{p}]\mathtt{W}_2[\mathtt{t}]\mathtt{C}_2}_{\text{postfix}_{b_2}(T_2)},$$

with $b_1 = \mathtt{W}_1[\mathtt{t}]$ and $b_2 = \mathtt{R}_2[\mathtt{q}]$. Notice that s is allowed under read committed. □

The next theorem then characterizes robustness against read committed:

Theorem 23 ([26]). *Let $\mathcal{T}$ be a set of transactions. The following are equivalent:*

(1) $\mathcal{T}$ is not robust against isolation level read committed*;*
(2) $IG(\mathcal{T})$ contains a multi-prefix-conflict-free cycle; and
(3) there is a multi-split schedule s for $\mathcal{T}$ that is allowed under read committed.

Using the above characterization, the following theorem establishes the complexity of deciding robustness against read committed:

Theorem 24 ([26]). robustness(read committed) *is* conp*-complete.*

3.2 Multiversion Read Committed

We shift attention to multiversion read committed (mvrc), a variation of the Read Committed isolation level over multiversion schedules. This isolation level allows a transaction to read the last committed version of an object instead of waiting for the other transaction to commit.

For a schedule s, the version order $\ll_s$ corresponds to the commit order in s if for every pair of write operations $a_j \in T_j$ and $b_i \in T_i$, we have $b_i \ll_s a_j$ iff $\mathtt{C}_i <_s a_j$. We say that a schedule s is *read-last-committed (RLC)* if $\ll_s$ corresponds to the commit order and for every read operation a_j in s on some tuple $\mathtt{t}$ the following holds:

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

That is, a_j observes the most recent version of $\mathtt{t}$ (according to the order of commits) that is committed before a_j.

Definition 25. A schedule is *allowed under isolation level* multiversion read committed (mvrc) if it is read-last-committed and does not exhibit dirty writes.

Only schedules with a very particular structure have to be considered: *multiversion split schedules.*

Definition 26 (Multiversion 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 conflict quadruples for $\mathcal{T}$ s.t. each transaction in $\mathcal{T}$ occurs in at most two quadruples. A *multiversion split schedule* for $\mathcal{T}$ based on C is a multiversion schedule that has the form

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

where

(1) there is no write operation in $\text{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in any of the transactions $T_2, \ldots, T_m$;
(2) $b_1 <_{T_1} a_1$ or b_m is rw-conflicting with a_1; and,
(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.

Schematically, a multiversion split schedule is as in Figure 3. Intuitively, Condition (1) guarantees that s is allowed under mvrc, while Condition (2) and (3) ensure that C corresponds to a cycle in $CG(s)$.

The following theorem characterizes non-robustness in terms of the existence of a multiversion split schedule. The proof shows that for any counterexample schedule allowed under mvrc, a counterexample schedule can be constructed that is a multiversion split schedule, and that, conversely, any multiversion split schedule s gives rise to a cycle in the conflict-graph $CG(s)$.

Theorem 27 (follows from [45]). *For a set of transactions $\mathcal{T}$, this is equivalent:*

(1) $\mathcal{T}$ is not robust against mvrc;
(2) there is a multiversion split schedule s for $\mathcal{T}$ based on some C.

The above characterization for robustness against mvrc leads to a polynomial time algorithm that cycles through all possible split schedules. For this, we need to introduce the following notion. For a transaction T_1, an operation $b_1 \in T_1$ and a set of transactions $\mathcal{T}$ with $T_1 \notin \mathcal{T}$, define prefix-conflict-free-graph$(b_1, T_1, \mathcal{T})$ as the graph containing as nodes all transactions in $\mathcal{T}$ that do not contain a ww-conflict with an operation in $\text{prefix}_{b_1}(T_1)$. 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.

Theorem 28 (follows from [45]). *Algorithm 1 decides whether a set of transactions $\mathcal{T}$ is robust against mvrc in time $O(max\{k.|\mathcal{T}|^3, k^3.\ell\})$, with k the total number of operations in $\mathcal{T}$ and ℓ the maximum number of operations in a transaction in $\mathcal{T}$.*

4 STATIC ROBUSTNESS TESTING FOR TEMPLATES

We next consider robustness testing as a static and offline problem on the application level. We assume that transactions can only be

Algorithm 1: Deciding transaction robustness against MVRC.

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

generated through an API consisting of a fixed set of transaction programs. For instance, the TPC-C benchmark [43] consists of five different transaction programs, from which an infinite number of concrete transactions can be instantiated. A finite set of transaction programs, like TPC-C, is robust against MVRC iff every set of transactions that can be instantiated from these programs, is robust against MVRC. If the answer is yes, then the isolation level can be safely set to MVRC without giving up on serializability.

Our approach is based on a formalization of transaction programs, called *transaction templates*, facilitating fine-grained reasoning for robustness against MVRC. Key aspects of the formalization are the following:

- Conceptually, *transaction templates* are functions with parameters, and can, for instance, be derived from stored procedures inside a database system. The abstraction generalizes transactions as usually studied in concurrency control research – sequences of read and write operations – by making the objects worked on variable, determined by input parameters. Such parameters are *typed* to add additional power to the analysis.
- We support *atomic updates* (that is, a read followed by a write of the same database object, to make a relative change to its value) allowing us to identify some workloads as robust that otherwise would not be.
- Furthermore, we model database objects read and written at the granularity of fields, rather than just entire tuples, decoupling conflicts further and allowing to recognize additional cases that would not be recognizable as robust on the tuple level.
- Dependencies between tuples are modeled by *functional constraints*.

There are also a few restrictions to the model. We assume there is 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. We

Account(<u>Name</u>, CustomerID, IsPremium)
Savings(<u>CustomerID</u>, Balance, InterestRate)
Checking(<u>CustomerID</u>, Balance)

Figure 5: Tables of the SmallBank$^+$ benchmark. Underlined attributes are primary keys.

show in Section 5 how to surpass these restrictions at the expense of completeness.

4.1 Formalization of Transaction Templates

We introduce transaction templates by means of an example and refer to [45] for a formal definition. We present SmallBank$^+$, a small extension of the SmallBank benchmark [2], to exemplify the modeling power of transaction templates.

The SmallBank schema consists of three tables as given in Figure 5. 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 a set of predefined transaction programs.

- Balance(N): returns the total balance (savings and checking) for the 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 with name N.
- Amalgamate(N_1,N_2): transfers all the funds from the customer with name N_1 to the customer with name 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. the original SmallBank benchmark [2].

The corresponding transaction templates are given in Figure 6. In short, a transaction template is a sequence of read (R), write (W) and update (U) statements over typed variables (X, Y, ...) with additional equality and disequality constraints. E.g., R[Y : Savings{C, I}] in template GoPremium indicates that a read operation is performed to a tuple in relation Savings on the attributes CustomerID and InterestRate. We abbreviate the names of attributes by their first letter to save space. The set $\{C, I\}$ is the read set. 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., U[X : Account{N, C}{I}] first reads the Name and CustomerID of tuple X and then writes to the attribute InterestRate. Furthermore, the description of these transaction programs implies certain dependencies between the accessed tuples. For example, the transaction program Balance first

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 6: Transaction templates for SmallBank⁺.

T_1 : $\mathtt{U}_1[\mathtt{a}_1\{\text{N,C}\}\{\text{I}\}]\,\mathtt{R}_1[\mathtt{s}_1\{\text{C,I}\}]$ $\qquad\qquad\qquad\qquad\qquad\qquad$ $\mathtt{U}_1[\mathtt{s}_1\{\text{C}\}\{\text{I}\}]\,\mathtt{C}_1$
T_2 : $\qquad\qquad\qquad\qquad\qquad\qquad$ $\mathtt{R}_2[\mathtt{a}_1\{\text{N,C}\}]\,\mathtt{U}_2[\mathtt{s}_1\{\text{C, B}\}\{\text{B}\}]\,\mathtt{C}_2$

Figure 7: Example schedule.

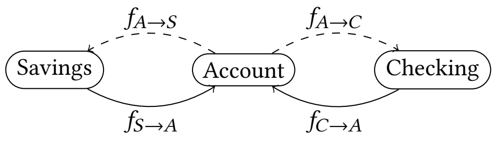


Figure 8: Schema graph for the SmallBank⁺ benchmark. The dashed edges correspond to the multi-tree schema graph for the schema restricted to $f_{A\to S}$ and $f_{A\to C}$.

reads a tuple of type Account and then uses the value of the CustomerID attribute to read the corresponding Savings and Checking tuples. To capture these 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 account is associated to a unique Account tuple. This is modelled through the functions $f_{C\to A}$ and $f_{S\to A}$ with an analogous interpretation. Notice that the equality constraints in Figure 6 imply that these functions are bijections and each other's inverses.

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 in $\mathbf{D}$ that satisfies all the constraints in τ such that $\mu(\tau) = T$. For instance, consider a database $\mathbf{D}$ with tuples $\mathtt{a}_1, \mathtt{a}_2, \ldots$ of type Account, $\mathtt{s}_1, \mathtt{s}_2, \ldots$ of type Savings, and $\mathtt{c}_1, \mathtt{c}_2, \ldots$ of type Checking with $f^{\mathbf{D}}_{A\to S}(\mathtt{a}_i) = \mathtt{s}_i$, $f^{\mathbf{D}}_{A\to C}(\mathtt{a}_i) = \mathtt{c}_i$, $f^{\mathbf{D}}_{S\to A}(\mathtt{s}_i) = \mathtt{a}_i$, $f^{\mathbf{D}}_{C\to A}(\mathtt{c}_i) = \mathtt{a}_i$ for each i. Then, for $\mu_1 = \{\mathtt{X} \to \mathtt{a}_1, \mathtt{Y} \to \mathtt{s}_1\}$, $\mu_1(\text{GoPremium}) = \mathtt{U}[\mathtt{a}_1]\,\mathtt{R}[\mathtt{s}_1]\,\mathtt{U}[\mathtt{s}_1]$ is an instantiation of GoPremium whereas $\mu_2(\text{GoPremium})$ with $\mu_2 = \{\mathtt{X} \to \mathtt{a}_1, \mathtt{Y} \to \mathtt{s}_2\}$ is not as the functional constraint $\mathtt{Y} = f_{A\to S}(\mathtt{X})$ is not satisfied. Indeed, $\mu_2(\mathtt{Y}) = \mathtt{s}_2 \neq \mathtt{s}_1 = f^{\mathbf{D}}_{A\to S}(\mathtt{a}_1) = f^{\mathbf{D}}_{A\to S}(\mu_2(\mathtt{X}))$. We then say that a set of transactions is *consistent* with a set of templates if every transaction is an instantiation of a transaction template.

The specification of read and write sets for each operation enables a more fine grained analysis of conflicts on the granularity of attributes. Consider for example the schedule s in Figure 7 over transactions T_1 and T_2, where T_1 is an instantiation of GoPremium, and T_2 is an instantiation of TransactSavings (we show the read and write sets to facilitate the discussion). Disregarding attributes, we would conclude that this schedule s is not serializable, as there is a rw-antidependency from $\mathtt{R}_1[\mathtt{s}_1]$ to $\mathtt{U}_2[\mathtt{s}_1]$, and a ww-dependency from $\mathtt{U}_2[\mathtt{s}_1]$ to $\mathtt{U}_1[\mathtt{s}_1]$. Notice however that $\mathtt{U}_2[\mathtt{s}_1]$ in T_2 only writes to attribute Balance, whereas the operations in T_1 only read and/or write to the attributes CustomerID and InterestRate. Analogously, there is no conflict between operation $\mathtt{U}_1[\mathtt{a}_1]$ in T_1 and $\mathtt{R}_2[\mathtt{a}_1]$ in T_2, as the former only writes to the attribute IsPremium, which is not read by the latter. We can therefore safely conclude that there are no conflicting operations between T_1 and T_2, and, as a result, that the schedule is serializable! This analysis on the granularity of attributes should be contrasted with concurrency control implementations that operate on the level of tuples, e.g. by placing locks on tuples rather than specific attributes. The consequences of these differences are discussed in more detail in Section 8 of [45]. We emphasize in particular that the attribute-based characterizations for robustness are still applicable as sufficient conditions for tuple-based systems, without requiring changes to the database system. Intuitively, a tuple-based concurrency control implementation will allow only a subset of the schedules that are possible under an attribute-based implementation, since a tuple-based implementation will detect more conflicts compared to an attribute-based one.

4.2 Detecting robust subsets

Deciding robustness against MVRC for transaction templates with only functional constraints that are variable equalities, is in polynomial time [45]. Algorithm 1 cannot be applied directly to test robustness for transaction templates as there are infinitely many sets of transactions consistent with a given set of transaction templates. However, Algorithm 1 can be generalized to interference graphs on the level of transaction templates representing potential conflicts: potentially conflicting operations lead to conflicting operations when the variables of these operations are mapped to the same tuple by a variable assignment. The crux underlying the decision algorithm is that when a counterexample multiversion split schedule exists, the needed variable instantiations can be compactly represented.

Deciding robustness against MVRC for transaction templates with general functional constraints is undecidable [47]. When functional constraints satisfy certain restrictions, decidability can be retained. For instance, when functional constraints admit multi-tree bijectivity robustness is in NLOGSPACE [47]. The unary functions associated to the database schema induce a schema graph mapping tuples of one relation to another (see Figure 8 for the SmallBank$^+$ benchmark). Informally, multi-tree bijectivity means that there should be a partitioning of the functions $(f_1, g_1), \ldots, (f_n, g_n)$ such that each f_i and g_i are each other's inverses and the schema graph restricted to one choice for each (f_i, g_i) is a multi-tree.[7] Furthermore, when the schema graph is acyclic robustness is in EXPSPACE [47].

4.3 Robustness of SmallBank$^+$ and TPC-Ckv

Figure 9 gives an overview of the maximal subsets robust against MVRC that are detected for the SmallBank$^+$ and TPC-Ckv benchmarks (TPC-Ckv is a version of TPC-C where selections are restricted to be key-based). Transaction templates are presented in abbreviated form (e.g., Bal refers to Balance). To assess the effect of the different features of the abstraction, we consider different settings: 'Only R & W' is the setting where updates are modeled through a read followed by a write and where attributes and functional constraints are ignored (that is, conflicts are considered on the level of entire tuples and violations against functional constraints are allowed).

The setting 'Atomic Updates' is the extension that models updates explicitly as atomic updates. This setting already allows detecting relatively large robust sets compared to the 'Only R & W' setting. Indeed, for SmallBank$^+$ the set {Am, DC, TS} is a robust subset indicating that any schedule using any number of instantiations of just these three templates that is allowed under MVRC is serializable! Also, for TPC-Ckv larger robust subsets are detected.

Next, 'Attr conflicts' includes attributes in the analysis (that is, conflicts are specified on the level of attributes). This difference in granularity has a profound effect for TPC-Ckv as can be seen in the third row of Figure 9: a robust subset of four templates (out of five!) is found: {Del, Pay, NO, SL}. For SmallBank$^+$ there is no improvement as almost all conflicts for this benchmark are based on the same Balance attributes in Savings and Checking. We emphasize that the analysis of conflicts on the granularity of attributes is still relevant if the database system's concurrency control subsystem works at the granularity of tuples rather than individual attributes: robustness on attribute-level conflicts implies robustness on these systems.

Finally, 'Func constraints' includes functional constraints in the setting. For SmallBank$^+$, the addition of functional constraints allows adding GoPremium to each maximal robust set, which should be contrasted with the fact that without functional constraints, the set {GoPremium} on itself is not even robust against MVRC. For TPC-Ckv, the addition of functional constraints does not lead to larger subsets robust against MVRC.

4.4 When robustness fails: promotion

When a set of transaction templates is not robust against MVRC code modification techniques can be applied to make it robust. Alomari and Fekete [3] presented a technique that relies on the 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.

We propose a template modification technique based on insights from Definition 26: an equivalent set of transaction templates robust against MVRC 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 template, but the newly introduced write operation will trigger concurrency mechanisms in the database. We emphasize that this is a general technique that can *always* be used to construct an equivalent robust set of templates: Definition 26 requires that operation b_1 is rw-conflicting with a_1 (Condition (3)), but not ww-conflicting with a_1 (Condition (1)), so promoting *all* R-operations to U-operations is sufficient to guarantee robustness against MVRC. For example, the SmallBank$^+$ benchmark can be made robust against MVRC by changing all read operations on tuples of type Savings and Checking in templates Balance and WriteCheck to atomic updates. We refer to [45] for a more detailed discussion.

5 STATIC ROBUSTNESS TESTING FOR PROGRAMS

A drawback of the approach in Section 4 is that it can not be extended to take updates to key attributes or predicate reads into account. In this section, we discuss an orthogonal approach where we require soundness but no longer completeness. We present a formal model BTP for transaction programs taking predicate reads, inserts, deletes as well as control structures into account. A robustness test is then obtained by testing for the absence of certain cycles in a so-called summary graph. The main advantage of the presented approach is that the construction of the summary graph only depends on the BTP formalization of the corresponding SQL programs and can be constructed automatically: no intervention from a domain specialist or database administrator to predict possible conflicts is necessary.

5.1 Auction Example

We illustrate the approach through a running example based on an auction service, where the database schema consists of three relations: Buyer($\underline{\text{id}}$, calls), Bids($\underline{\text{buyerId}}$, bid), and Log($\underline{\text{id}}$, buyerId, bid),

[7] A graph is a multi-tree is there is at most one directed path between any pair of vertices. Every multi-tree is acyclic but not vice-versa.

	SmallBank⁺	TPC-Ckv
Only R & W	{Bal}	{OS, SL}
Atomic Updates	{Am, DC, TS}, {Bal, DC}, {Bal, TS}	{Del, Pay, SL}, {NO, SL}, {Pay, OS, SL}
Attr conflicts	{Am, DC, TS}, {Bal, DC}, {Bal, TS}	{Del, Pay, NO, SL}, {Pay, OS, SL}
Func constraints	{Am, DC, TS, GP}, {Bal, DC, GP}, {Bal, TS, GP}	{Del, Pay, NO, SL}, {Pay, OS, SL}

Figure 9: Subsets of the SmallBank⁺ and TPC-Ckv benchmarks robust against MVRC by analysis setting.

```
FindBids(:B, :T):
  UPDATE Buyer --q1
  SET calls = calls + 1
  WHERE id = :B;

  SELECT bid --q2
  FROM Bids
  WHERE bid >= :T;

  COMMIT;
```

Auction schema
Buyer(<u>id</u>,calls)
Bids(<u>buyerId</u>, bid)
Log(<u>id</u>,buyerId,bid)
Foreign keys
f_1: Bids(BuyerId) → Buyer(id)
f_2: Log(BuyerId) → Buyer(id)

	BTP
FindBids	$q_1; q_2$
PlaceBid	$q_3; q_4; (q_5 \mid \varepsilon); q_6$

```
PlaceBid(:B, :V):
  UPDATE Buyer --q3
  SET calls = calls + 1
  WHERE id = :B;

  SELECT bid into :C --q4
  FROM Bids
  WHERE buyerId = :B;

  IF :C < :V: --q5
    UPDATE Bids
    SET bid = :V
    WHERE id = :B;
  ENDIF;

  :logId = uniqueLogId();

  INSERT INTO Log --q6
  VALUES(:logId, :B, :V);

  COMMIT;
```

Figure 10: Auction schema, SQL code and BTP formalization for FindBids(B, T) and PlaceBid(B, V)

q	type(q)	rel(q)	PReadSet(q)	ReadSet(q)	WriteSet(q)
FindBids					
q_1	key upd	Buyer	⊥	{calls}	{calls}
q_2	pred sel	Bids	{bid}	{bid}	⊥
PlaceBid					
q_3	key upd	Buyer	⊥	{calls}	{calls}
q_4	key sel	Bids	⊥	{bid}	⊥
q_5	key upd	Bids	⊥	{}	{bid}
q_6	ins	Log	⊥	⊥	{id, buyerId, bid}

Figure 11: Query details for BTPs FindBids and PlaceBid.

where the primary key for each relation is underlined and buyerId in Bids and Log is a foreign key referencing Buyer(id). The relation Buyer lists all potential buyers, Bids keeps track of the current bid for each potential buyer, and Log keeps a register of all bids. Each buyer can interact with the auction service through API calls. For logging purposes, the attribute Buyer(calls) counts the total number of calls made by the buyer. The API interacts with the database via two transaction programs: FindBids(B, T) and PlaceBid(B, V) whose SQL code is given in Figure 10. FindBids returns all current bids above threshold T, whereas PlaceBids increases the bid of buyer B to value V (if V is higher than the current bid, otherwise the current bid remains unchanged) and inserts this newly placed bid as a new tuple in Log. Both programs increment the number of calls for B.

5.2 Basic Transaction Programs

We introduce the formalism of *basic transaction programs* (BTP) to overestimate the set of schedules that can arise when executing transaction programs as given in Figure 10. A BTP is a sequence of statements that only retains the information necessary to detect robustness against MVRC: the type of statement (insert, key-based selection/update/delete, or predicate-based selection/update/delete), the relation that is referred to, and the attributes that are read from, written to, and that are used in predicates. In particular, BTPs ignore the concrete predicate selection condition.

Formally, a BTP is a sequence of statements $q_1; \ldots; q_k$. For example, FindBids is modeled by $q_1; q_2$, where q_1 and q_2 are two statements reflecting the corresponding SQL statements in Figure 10. Each statement q_i is supplemented with additional information as detailed in Figure 11. There, type(q_i) refers to the type of statement: an insert, a key-based or predicate-based selection, update or delete; rel(q_i) is the relation under consideration; ReadSet(q_i) are the attributes read by q_i; WriteSet(q_i) those written by q_i; and, PReadSet(q_i) the attributes used for predicates in the WHERE part of the query. We use ⊥ to indicate that a specific function is not applicable to a statement. For example, q_1 in FindBids is a key-based update over relation Buyer, since the corresponding SQL query selects exactly one tuple based on the primary key attribute Buyer(id). This statement reads and then overwrites the value for attribute Buyer(calls), and therefore ReadSet(q_1) = WriteSet(q_1) = {calls}. Since this statement is not predicate-based, we have PReadSet(q_1) = ⊥. Statement q_2 is a predicate-based selection over relation Bids. The predicate `id = :B` in the corresponding SQL statement only uses the attribute Bids(bid), and therefore PReadSet(q_2) = {bid}. Therefore, ReadSet(q_2) = {bid}.

BTPs incorporate conditional branching and loops as well. Indeed, PlaceBid is modeled by $q_3; q_4; (q_5 \mid \varepsilon); q_6$ supplemented with additional information as depicted in Figure 11. Here, $(q_5 \mid \varepsilon)$ denotes the branching corresponding to the IF-statement in the SQL program: either q_5 is executed (if the condition in the SQL program evaluates to true), or nothing is executed (if the condition evaluates to false). We note that an ELSE-clauses can be modeled by replacing ε by a corresponding statement. Analogously, BTPs allow loop(P)

to express iteration, where P is an arbitrary sequence of statements. Intuitively, loop(P) specifies that P can be repeated for an arbitrary yet finite number of iterations.

A set of transaction programs $\mathcal{P}$ induces an infinite set of possible schedules where each transaction in the schedule is an instantiation of a program in $\mathcal{P}$ as informally explained next by means of an example. Consider the schedule s over transactions T_1, T_2 and T_3 presented in Figure 12. Here, T_1 and T_2 are instantiations of PlaceBid and T_3 is an instantiation of FindBids (when considered as a BTP). Furthermore, $\mathtt{t}_1$ and $\mathtt{t}_2$ are tuples of relation Buyer, $\mathtt{v}_1$, $\mathtt{v}_2$ and $\mathtt{v}_3$ are tuples of Bids, and $\mathtt{l}_1$ and $\mathtt{l}_2$ are tuples of Log. The operation $\mathtt{R}_1[\mathtt{t}_1]$ (respectively $\mathtt{W}_1[\mathtt{t}_1]$) indicates that transaction T_1 reads (respectively writes to) tuple $\mathtt{t}_1$, and operation $\mathtt{I}_1[\mathtt{l}_1]$ indicates that T_1 inserts a new tuple $\mathtt{l}_1$ into the database. The operation $\mathsf{PR}_3[\text{Bids}]$ in T_3 is a predicate read that evaluates a predicate over all tuples in relation Bids.

Figure 12 further illustrates how each statement in a BTP leads to one or more operations over tuples. For example, the key-based update q_3 in PlaceBid results in two operations $\mathtt{R}_1[\mathtt{t}_1]$ and $\mathtt{W}_1[\mathtt{t}_1]$. Notice in particular that these two operations are over the same tuple $\mathtt{t}_1$ of relation Buyer = rel(q_3), where the first operation reads the value for attribute Buyer(calls) and the second operation overwrites the value for this attribute, as indicated by ReadSet(q_3) and WriteSet(q_3). The predicate-based selection statement q_2 of FindBids results in a larger number of operations in T_3. First, the predicate read $\mathsf{PR}_3[\text{Bids}]$ evaluates a predicate over all tuples in Bids = rel(q_2), where only attribute Bids(bid) is used in the predicate, indicated by PReadSet(q_2). This predicate intuitively corresponds to the WHERE clause of the corresponding SQL statement, but in our formalism, we will only specify the attributes needed in the predicate rather than the predicate itself. Then, T_3 reads three tuples of relation Bids. For each such tuple, only the value of attribute Bids(Bid) is read, as specified by ReadSet(q_2). Also notice how T_1 is an instantiation of PlaceBid where the if-condition evaluates to false, whereas for T_2 it evaluates to true, witnessed by the presence of q_5 in T_2 and its absence in T_1.

5.3 Foreign Keys

Schedules should respect foreign keys. Two instantiations of PlaceBid that access the same tuple $\mathtt{t}_1$ of relation Bids also need to access the same Buyer $\mathtt{v}_1$ as Bids(buyerId) is a foreign key referencing Buyer(Id). Such information can be used to rule out inadmissible schedules (that could otherwise inadvertently cause a set of transaction programs to not be robust). For example, the schedule s' obtained from s by substituting $\mathtt{t}_1$ with $\mathtt{t}_2$ in T_1 violates the foreign key constraint and is therefore not admissible.

5.4 MVRC, Dependencies and Conflict Serializability

We extend some definitions presented in Section 2 and the definition of MVRC presented in Section 3.2 to include predicate reads. When a database is operating under isolation level Multiversion Read Committed (MVRC), each read operation reads the most recently committed version of a tuple, and write operations cannot overwrite uncommitted changes. Furthermore, for each tuple $\mathtt{t}$ in some relation R, a predicate read $\mathsf{PR}_i[R]$ over a relation R evaluates its predicate over the most recently committed version of $\mathtt{t}$. For example, under the assumption that s in Figure 12 is allowed under MVRC, $\mathtt{R}_2[\mathtt{t}_1]$ will observe the version of $\mathtt{t}_1$ written by $\mathtt{W}_1[\mathtt{t}_1]$, as T_1 committed before $\mathtt{R}_2[\mathtt{t}_1]$. Read operation $\mathtt{R}_3[\mathtt{v}_1]$ on the other hand will not see the changes made by $\mathtt{W}_2[\mathtt{v}_1]$, as the commit of T_2 occurs after $\mathtt{R}_3[\mathtt{v}_1]$. The predicate read $\mathsf{PR}_3[\text{Bids}]$ evaluates the predicate over the version of $\mathtt{v}_1$ before the version written by $\mathtt{W}_2[\mathtt{v}_1]$, as the commit of T_2 occurs after $\mathsf{PR}_3[\text{bids}]$.

We say that two operations occurring in two different transactions are conflicting if they are over the same tuple, access a common attribute of this tuple, and at least one of these two operations overwrites the value for this common attribute. These conflicts introduce dependencies between operations. For example, $\mathtt{W}_1[\mathtt{t}_1]$ in T_1 and $\mathtt{R}_2[\mathtt{t}_1]$ in T_2 are conflicting, as the former modifies the value for attribute Buyer(calls) and the latter reads this value. We therefore say that there is a wr-dependency from $\mathtt{W}_1[\mathtt{t}_1]$ to $\mathtt{R}_2[\mathtt{t}_1]$, denoted by $\mathtt{W}_1[\mathtt{t}_1] \to_s \mathtt{R}_2[\mathtt{t}_1]$. Similarly, since we assume that s is allowed under MVRC, $\mathtt{R}_3[\mathtt{v}_1]$ observes a version of $\mathtt{v}_1$ before the changes made by $\mathtt{W}_2[\mathtt{v}_1]$. We therefore say that there is an rw-antidependency from $\mathtt{R}_3[\mathtt{v}_1]$ to $\mathtt{W}_2[\mathtt{v}_1]$, denoted by $\mathtt{R}_3[\mathtt{v}_1] \to_s \mathtt{W}_2[\mathtt{v}_1]$. Since the predicate read $\mathsf{PR}_3[\text{Bids}]$ observes a version of $\mathtt{v}_1$ before the version written by $\mathtt{W}_2[\mathtt{v}_1]$, we say that there is a predicate rw-antidependency from $\mathsf{PR}_3[\text{Bids}]$ to $\mathtt{W}_2[\mathtt{v}_1]$, denoted by $\mathsf{PR}_3[\text{Bids}] \to_s \mathtt{W}_2[\mathtt{v}_1]$. The serialization graph $SeG(s)$ contains transactions as nodes and edges correspond to dependencies. It is well-known that a schedule is conflict serializable if there is no cycle in $SeG(s)$.

A dependency from a transaction T_i to a transaction T_j is counterflow if T_j commits before T_i (that is, the direction of the dependency is opposite to the commit order). In our running example, the dependencies $\mathtt{R}_3[\mathtt{v}_1] \to_s \mathtt{W}_2[\mathtt{v}_1]$ and $\mathsf{PR}_3[\text{Bids}] \to_s \mathtt{W}_2[\mathtt{v}_1]$ are counterflow dependencies, as T_3 commits after T_2. Alomari and Fekete [3] showed that if a schedule is allowed under MVRC, then every cycle in the serialization graph contains at least one counterflow dependency. We refer to cycles containing at least one counterflow dependency as a type-I cycle. In [46], we refine this condition and show that every such cycle must either contain an adjacent-counterflow pair or an ordered-counterflow pair, as well as a non-counterflow dependency, and refer to the latter as a type-II cycle . As every type-II cycle is a type-I cycle but not vice-versa, this refinement will allow us to identify larger sets of programs to be robust against MVRC.

5.5 Linear Transaction Programs

We refer to BTPs without branching and loops as linear transaction programs (LTP). For each BTP an equivalent set of LTPs can be derived by unfolding all branching statements and loops. FindBids is also an LTP and PlaceBid can be unfolded into two LTPs PlaceBid$_1$:= $q_3; q_4; q_5; q_6$ and PlaceBid$_2$:= $q_3; q_4; q_6$. Loop unfolding gives rise to an infinite number of LTPs. However, we will show that for detecting robustness against MVRC it suffices to limit loop unfoldings to at most two iterations.

$$\begin{array}{lllllllllll}
T_1: & \underbrace{\mathtt{R}_1[\mathtt{t}_1]\mathtt{W}_1[\mathtt{t}_1]}_{q_3} & \underbrace{\mathtt{R}_1[\mathtt{v}_1]}_{q_4} & \underbrace{\mathtt{I}_1[\mathtt{l}_1]}_{q_6}\mathtt{C}_1 & & & & & & & \\
T_2: & & & & \underbrace{\mathtt{R}_2[\mathtt{t}_1]\mathtt{W}_2[\mathtt{t}_1]}_{q_3} & \underbrace{\mathtt{R}_2[\mathtt{v}_1]}_{q_4} & & \underbrace{\mathtt{W}_2[\mathtt{v}_1]}_{q_5} & & \underbrace{\mathtt{I}_2[\mathtt{l}_2]}_{q_6}\mathtt{C}_2 & \\
T_3: & & & & & & \underbrace{\mathtt{R}_3[\mathtt{t}_2]\mathtt{W}_3[\mathtt{t}_2]}_{q_1} & & \underbrace{\mathtt{PR}_3[\mathtt{Bids}]\mathtt{R}_3[\mathtt{v}_1]\mathtt{R}_3[\mathtt{v}_2]\mathtt{R}_3[\mathtt{v}_3]}_{q_2} & & \mathtt{C}_3
\end{array}$$

Figure 12: Example schedule s where T_1 and T_2 are instantiations of PlaceBid and T_3 is an instantiation of FindBids.

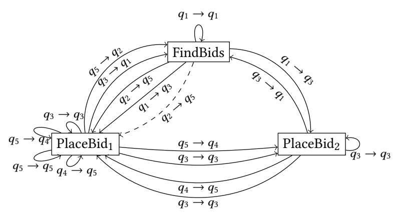


Figure 13: Summary graph containing a type-I but no type-II cycles.

5.6 Detecting Robustness against MVRC

A set $\mathcal{P}$ of LTPs is robust against MVRC if every allowed schedule is serializable. We therefore lift the just mentioned condition from serialization graphs to summary graphs. The summary graph $SuG(\mathcal{P})$ summarizes all serialization graphs for all possible schedules allowed under MVRC over transactions instantiated from programs in $\mathcal{P}$. Here, nodes in $SuG(\mathcal{P})$ are programs in $\mathcal{P}$ and if a schedule allowed under MVRC exists with a dependency $b_i \to a_j$, then an edge is added from P_i to P_j where b_i is an operation in transaction T_i instantiated from a program $P_i \in \mathcal{P}$ and a_j is an operation in transaction T_j instantiated from $P_j \in \mathcal{P}$. That edge is annotated with statements P_i and P_j and is dashed when the dependency is counterflow. The summary graph for the three LTPs FindBids, PlaceBid$_1$ and PlaceBid$_2$ is visualized in Figure 13. If we consider for example the dependency $\mathtt{W}_1[\mathtt{t}_1] \to_s \mathtt{R}_2[\mathtt{t}_1]$, we see that $SuG(\mathcal{P})$ has a corresponding edge from PlaceBid$_2$ to PlaceBid$_1$, labeled with q_3 and q_3. Analogously, the counterflow dependency $\mathtt{R}_3[\mathtt{v}_1] \to_s \mathtt{W}_2[\mathtt{v}_1]$ is witnessed by the counterflow edge from FindBids to PlaceBid$_1$ in $SuG(\mathcal{P})$. We present a formal algorithm constructing the graph $SuG(\mathcal{P})$ for a given set of LTPs in [46].

Let s be an arbitrary schedule allowed under MVRC where transactions are instantiations of $\mathcal{P}$. As each dependency in the serialization graph $SeG(s)$ is witnessed by an edge in the summary graph $SuG(\mathcal{P})$, it immediately follows that each cycle in $SeG(s)$ is witnessed by a cycle in $SuG(\mathcal{P})$. So, when $SuG(\mathcal{P})$ does not contain a type-II cycle, we can safely conclude that $\mathcal{P}$ is robust against MVRC. Indeed, the absence of such cycles indicates that no schedule allowed under MVRC exists with a cycle in its serialization graph, implying that every such schedule is serializable. The presence of a type-II cycle does not necessarily imply non-robustness as there might not be a single schedule in which the corresponding cycle is realized. However, in that case, the conservative approach is to attest non-robustness to avoid false positives. Algorithm 2 in [46] follows this conservative approach and determines $\mathcal{P}$ to be robust iff $SuG(\mathcal{P})$ does not contain a type-II cycle.

It can be shown that the summary graph in Figure 13 does not contain a type-II cycle. The set {FindBids, PlaceBid} is therefore identified as robust against MVRC. The SQL programs presented in Figure 10 can thus be safely executed under isolation level MVRC, without risking non-serializable behavior. This improves over earlier work, as the summary graph does contain a type-I cycle (e.g., between FindBids and PlaceBid$_1$), and, hence, the method of [3] can not identify {FindBids, PlaceBid} as robust.

A major advantage of the presented approach is that once a set of SQL transaction programs is translated in BTP, the construction of the summary graph is fully automated. This should be contrasted with earlier work [3, 21] where the construction of the static dependency graph is a manual step that should be performed by a database specialist. This is of course a difficult problem as the decision to place an edge requires reasoning over all possible schedules. In [46], we show the effectiveness of the approach in detecting larger subsets of transaction programs to be robust against MVRC as before.

6 RESEARCH DIRECTIONS

Even though database concurrency control has not received much attention from the database theory research community in the past decade, we do think that many interesting challenges remain unanswered. We discuss some possible directions for future work:

- All work on robustness is cast in the setting of conflict-serializability even though more general notions exist like view-serializability or final-state serializability [32]. It would be interesting to see whether similar characterizations for robustness testing as obtained for SNAPSHOT ISOLATION [20], READ COMMITTED [26] and MULTIVERSION READ COMMITTED [45] can be obtained for those settings as well. How these can be generalized to transaction templates and whether they make a difference in practice.
- The general aim of this work is to develop machinery for choosing an adequate isolation level. Robustness provides ease of mind: increased throughput guarantees of the lower isolation level, while not giving up on serializability. But what can be done when a workload is not robust? One option is to make workloads robust through code modification techniques (e.g., [3, 45]) but these come with a performance penalty as well. From a theory perspective two orthogonal approaches come to mind. Robustness as considered in this paper is a binary property: a workload is robust, or it is not robust. Is it possible to quantify failure of robustness, as in, the probability that a schedule is not serializable? Another direction is to consider mixed isolation levels where you allow different transactions to be executed under different isolation levels. Rather than deciding robustness, we want to find an optimal allocation of isolation levels. The allocation problem has been studied by Fekete [20] for SNAPSHOT ISOLATION and SERIALIZABLE, and mixed isolation levels have been considered by [1], but a general theory for the allocation problem is lacking.

- Robustness for transaction templates is undecidable when taking functional constraints into account. While some decidable restrictions have been obtained, we do not have a good understanding of precisely what is needed for undecidability. We think that larger decidable settings can still be obtained.
- The results mentioned in Section 5 are a first formal step towards robustness testing in practice. A shortcoming is that the predicate conditions used in the SQL statements are not taken into account. It would be interesting to see how the present approach can be generalized to that end. This might require reasoning on intermediate representations of the transaction program code and could benefit from technology from the programming languages and compilers field.

ACKNOWLEDGMENTS

This work is funded by FWO-grant G019921N.

REFERENCES

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *ICDE*. 67–78.

[2] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE*. 576–585.

[3] Mohammad Alomari and Alan Fekete. 2015. Serializable use of Read Committed isolation level. In *AICCSA*. 1–8.

[4] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *PVLDB* 7, 3 (2013), 181–192.

[5] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. HAT, Not CAP: Towards Highly Available Transactions. In *USENIX HotOS*. 24–24.

[6] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019. Checking Robustness Against Snapshot Isolation. In *CAV*. 286–304.

[7] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019. Robustness Against Transactional Causal Consistency. In *CONCUR*. 1–18.

[8] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*. 1–10.

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

[10] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. 2015. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *SIGMOD*. 1295–1309.

[11] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR*. 9–20.

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

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

[14] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2015. Transaction Chopping for Parallel Snapshot Isolation. In *DISC*, Vol. 9363. 388–404.

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

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*. 1243–1254.

[17] Bailu Ding, Lucja Kot, Alan J. Demers, and Johannes Gehrke. 2015. Centiman: elastic, high performance optimistic concurrency control by watermarking. In *SoCC*. 262–275.

[18] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *PVLDB* 10, 5 (2017), 613–624.

[19] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *PVLDB* 8, 11 (2015), 1190–1201.

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

[21] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528.

[22] Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, and Aoying Zhou. 2019. Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. *PVLDB* 12, 5 (2019), 584–596.

[23] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *PVLDB* 13, 5 (2020), 629–642.

[24] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. 2009. Improving OLTP Scalability using Speculative Lock Inheritance. *PVLDB* 2, 1 (2009), 479–489.

[25] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*. 603–614.

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

[27] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*. 1675–1687.

[28] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.

[29] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD*. 21–35.

[30] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *PVLDB* 13, 11 (2020), 2047–2060.

[31] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*. 677–689.

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

[33] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *SIGMOD*. 527–542.

[34] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *SIGMOD*. 1583–1598.

[35] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-latency, Geo-replicated Transactions. *PVLDB* 12, 11 (2019), 1747–1761.

[36] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2012. Lightweight Locking for Main Memory Database Systems. *PVLDB* 6, 2 (2012), 145–156.

[37] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. 2014. Reducing Database Locking Contention Through Multi-version Concurrency. *PVLDB* 7, 13 (2014), 1331–1342.

[38] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting. In *SIGMOD*. 245–258.

[39] Dennis E. Shasha, François Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.* 20, 3 (1995), 325–363.

[40] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP transactions via learned abort prediction. In *aiDM*. 1:1–1:8.

[41] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*. 1–12.

[42] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-Aware Lock Scheduling for Transactional Databases. *PVLDB* 11, 5 (2018), 648–662.

[43] TPC-C. 1992. On-Line Transaction Processing Benchmark. (1992). http://www.tpc.org/tpcc/.

[44] Brecht Vandevoort. 2021. *Optimizing Concurrency Control: Robustness Against Read Committed Revisited*. Ph. D. Dissertation. Universiteit Hasselt.

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

[46] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2022. Detecting Robustness against MVRC for Transaction Programs with Predicate Reads. (2022). Manuscript.

[47] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2022. Robustness against Read Committed for Transaction Templates with Functional Constraints. In *ICDT*, Vol. 220. 16:1–16:17.

[48] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann.

[49] Cong Yan and Alvin Cheung. 2016. Leveraging Lock Contention to Improve OLTP Application Performance. *PVLDB* 9, 5 (2016), 444–455.

[50] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*. 1629–1642.

[51] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-Memory Databases. *PVLDB* 9, 6 (2016), 504–515.