Deciding Robustness for Lower SQL Isolation Levels

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

Frank Neven
frank.neven@uhasselt.be
Hasselt University and transnational University of Limburg

Christoph Koch
christoph.koch@epfl.ch
École polytechnique fédérale de Lausanne

Brecht Vandevoort*
brecht.vandevoort@uhasselt.be
Hasselt University and transnational University of Limburg

ABSTRACT
While serializability always guarantees application correctness, lower isolation levels can be chosen to improve transaction throughput at the risk of introducing certain anomalies. A set of transactions is robust against a given isolation level if every possible interleaving of the transactions under the specified isolation level is serializable. Robustness therefore always guarantees application correctness with the performance benefit of the lower isolation level. While the robustness problem has received considerable attention in the literature, only sufficient conditions have been obtained. The most notable exception is the seminal work by Fekete where he obtained a characterization for deciding robustness against snapshot isolation. In this paper, we address the robustness problem for the lower SQL isolation levels read uncommitted and read committed which are defined in terms of the forbidden dirty write and dirty read patterns. The first main contribution of this paper is that we characterize robustness against both isolation levels in terms of the absence of counter example schedules of a specific form (split and multi-split schedules) and by the absence of cycles in interference graphs that satisfy various properties. A critical difference with Fekete’s work is that the properties of cycles obtained in this paper have to take the relative ordering of operations within transactions into account as read uncommitted and read committed do not satisfy the atomic visibility requirement. A particular consequence is that the latter renders the robustness problem against read committed coNP-complete. The second main contribution of this paper is the coNP-hardness proof. For read uncommitted, we obtain LOGSPACE-completeness.

1 INTRODUCTION
To guarantee consistency during concurrent execution of transactions, most database management systems offer a serializable isolation level. Serializability ensures that the effect of concurrent execution of transactions is always equivalent to a serial execution where transactions are executed in sequence one after another. The database system thereby guarantees perfect isolation for every transaction. For application programmers perfect isolation is extremely important as it implies that they only need to reason about correctness properties of individual transactions. Ensuring serializability, however, comes at a non-trivial performance cost [21]. Database systems therefore provide the ability 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., [5]), 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 [20] 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 to guarantee perfect isolation at the performance cost of a lower isolation level.

Fekete et al. [16] 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 (what we and Fekete [15] call interference graph). This result was extended by Bernardi and Gotsman [10] 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. [11]. 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 to specify 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.
writes as well as dirty reads. The latter is a pattern of the form \( T \) reads \( z \), which has not yet committed. Both \( T \) and \( R \) write uncommitted and committed transactions, as their conflict graphs admit a cycle.

This means that for a given set of transactions, we need to check serializability. Alomari and Fekete [3] already studied robustness against isolation levels allowing rapid concurrent execution while guaranteeing perfect performance penalty, establishing robustness against these isolation levels.

To provide some insight into the technical challenges, we introduce some terminology by example (formal definitions are given in Section 2). As usual, a transaction is a sequence of read and write operations on objects followed by a commit. Consider for instance the set of transactions \( T = \{ T_1, T_2 \} \) with \( T_1 = W_1[x]R_1[z]W_1[y]C_1 \) and \( T_2 = W_2[z]R_2[y]W_2[x]C_2 \). Here, \( W_1[x] \) and \( R_1[x] \) denote a read and a write operation to object \( x \) by transaction \( T_1 \), whereas \( C_1 \) is the commit operation of \( T_1 \). A schedule for \( T \) then is an ordering of all operations occurring in transactions in \( T \). For instance, \( s_1 \) and \( s_2 \) as displayed in Figure 1 are schedules for \( T \). A schedule is not allowed under isolation level read uncommitted when it exhibits a dirty write: a pattern of the form \( W_1[x] \cdots W_2[x] \cdots C_1 \), that is, \( T_2 \) writes to an object that has been modified by a transaction \( T_1 \) that has not yet committed. Both \( s_1 \) and \( s_2 \) are allowed under read uncommitted.

The isolation level read committed prohibits dirty writes as well as dirty reads. The latter is a pattern of the form \( W_2[z] \cdots R_1[z] \cdots C_2 \). That is, \( T_1 \) reads an object that has been modified by a transaction \( T_2 \) that has not yet committed. The schedule \( s_1 \) is not allowed under read committed. Notice that \( s_1 \) and \( s_2 \) are not conflict serializable as their conflict graphs admit a cycle.

We start by studying robustness against read uncommitted. This means that for a given set of transactions, we need to check whether there is a counter example schedule that is allowed under read uncommitted and which is not serializable, that is, contains a cycle in its conflict graph. Notice that for \( T = \{ T_1, T_2 \} \) as defined above \( s_1 \) constitutes such a counter example. Furthermore, \( s_1 \) is of a very particular form. Indeed, \( s_1 \) can be seen as the schedule constructed by splitting \( T_2 \) into two parts \( (W_2[z] \cdots R_2[y]W_2[x]C_2) \) and placing the complete transaction \( T_1 \) in between. We call such schedules a split schedule. They can also be defined for sets of transactions consisting of more than two transactions by splitting one transaction in two parts and placing all other transaction in between (cf. Figure 2). We show that the existence of a counter example schedule that has the form of a split schedule provides a necessary and sufficient condition for deciding robustness against read uncommitted.

Fekete [15] introduced the notion of an interference graph for a set of transactions and obtained a characterization for deciding robustness against snapshot isolation in terms of the absence of a cycle with certain types of edges. We mimic his result by obtaining an additional characterization of deciding robustness against read uncommitted in terms of the absence of cycles in the interference graphs that are prefix-write-conflict-free.\(^2\) It is important to point out the main difference with the work of Fekete: snapshot isolation admits atomic visibility implying that cycles in the interference graph can refer to the global ordering of transactions and can ignore the ordering of operations within transactions. For read uncommitted, we can not rely on atomic visibility and need to take the specific conflicting operations into account that generate the edges in the interference graph. In addition, the notion of prefix-write-conflict-free cycle requires to isolate a single transaction (the one witnessing transferability, see Section 3 and the one that will be split in the counter example schedule) and determine non-existence of write-conflicts with respect to a prefix of this transaction (so the order of operations matters). That being said, the complexity of testing robustness against read uncommitted can be done very efficiently as we show it to be logspace-complete.

Next, we turn to robustness against read committed. Schedule \( s_2 \) shown in Figure 1 is allowed under read committed and is not serializable. It is hence a counter example showing that \( T \) is not robust against read committed. Notice that \( s_2 \) is not a split schedule. In fact, it can be argued that there is no split schedule for \( T \) that is allowed under read committed. This means that the existence of a counter example schedule in the form of a split schedule is not a necessary condition for deciding robustness against read committed. We show that counter examples do not need to take arbitrary forms either. We obtain a characterization for deciding robustness against read committed in terms of counter example schedules that take the form of multi-split schedules as illustrated in Figure 2. In contrast to a split schedule where one transaction is split open and all other transactions are inserted, a multi-split schedule can open several such transactions but needs to close them in sequence.

We obtain an equivalent characterization in terms of the absence of a multi-prefix-write-conflict-free cycle in the interference graph. The latter is a rather involved property of cycles that much more than the notion of prefix-write-conflict-free mentioned previously depends on the ordering of operations within transactions. Using this notion, we show that deciding robustness against read committed

---

1. See Section 2.2 for a definition of conflict graphs and how acyclicity implies serializability.

2. See Section 4 for a formal definition.
Split schedule for four transactions:

\[ T_1 \quad T_2 \quad T_3 \quad T_4 \]

Multi-split schedule for six transactions:

\[ T_1 \quad T_2 \quad T_3 \quad T_4 \quad T_5 \quad T_6 \]

opening phase sequential phase closing phase

Figure 2: Abstract presentation of split and multi-split schedule. The drawing omits a possible trailing sequence of non-interleaved transactions (cf. Definition 8 and Definitions 18).

Outline. We introduce the necessary definitions in Section 2. We introduce key notions in Section 3 in the context of robustness against no isolation level. We consider robustness against READ UNCOMMITTED and READ COMMITTED in Section 4 and Section 5, respectively. We discuss related work in Section 6 and conclude in Section 7.

2 DEFINITIONS

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 \( \text{Obj} \). For an object \( x \in \text{Obj} \), we denote by \( R[x] \) a read operation on \( x \) and by \( W[x] \) a write operation on \( x \). We also assume a special commit operation denoted by \( C \). A transaction \( T \) over \( \text{Obj} \) is a sequence of read and write operations on objects in \( \text{Obj} \) followed by a commit. In the sequel, we leave the set of objects \( \text{Obj} \) implicit when it is clear from the context and just say transaction rather than transaction over \( \text{Obj} \). We also sometimes just say reads and writes rather than read and write operations.

We assume that a transaction performs at most one write and at most one read per object. The latter is a common assumption (see, e.g., [15]) and is made to simplify exposition: all our results carry over to the more general setting in which multiple writes and reads per object are allowed.

Formally, we model a transaction as a linear order \( (T, \leq_T) \), where \( T \) is the set of \( (\text{read}, \text{write}, \text{commit}) \) operations occurring in the transaction and \( \leq_T \) encodes the ordering of the operations. As usual, we use \( \prec_T \) to denote the strict ordering.

For an operation \( b \in T \), we denote by \( \text{prefix}_b(T) \) the restriction of \( T \) to all operations that are smaller than or equal to \( b \) according to \( \leq_T \). Similarly, we denote by \( \text{postfix}_b(T) \) the restriction of \( T \) to all operations that are strictly larger than \( b \) according to \( \leq_T \). Throughout the paper, we interchangeably consider transactions both as linear orders as well as sequences. Therefore, \( T \) is then equal to the sequence \( \text{prefix}_b(T) \) followed by \( \text{postfix}_b(T) \) which we denote by \( \text{prefix}_b(T) \cdot \text{postfix}_b(T) \) for every \( b \in T \).

When considering a set \( \mathcal{T} \) of transactions, we assume that every transaction in the set has a unique id \( i \) and write \( T_i \) to make this id explicit. Similarly, to distinguish the operations from different transactions, we add this id as index to the operation. That is, we write \( R_i[x] \) and \( W_i[x] \) to denote a write and read on object \( x \) occurring in transaction \( T_i \); similarly \( C_i \) denotes the commit operation in transaction \( T_i \). Notice that this convention is consistent with the literature (see, e.g., [9, 15]).

A schedule \( s \) over a set \( \mathcal{T} \) of transactions is a sequence of all the operations occurring in transactions in \( \mathcal{T} \) in which the order of operations from different transactions is consistent with their order in the respective transactions. Formally, we model a schedule as a linear order \( (s, \leq_s) \) where \( s \) is the set containing all operations of transactions in \( \mathcal{T} \) and \( \leq_s \) encodes the ordering of these operations with the additional constraint that \( a \prec_T b \) implies \( a \prec_s b \) for every \( T \in \mathcal{T} \) and every \( a, b \in T \).

The absence of aborts in our definition of schedule is consistent with the common assumption [10, 15] that an underlying recovery mechanism will rollback transactions that interfere with aborted transactions.

A schedule \( s \) over a set of transactions \( \mathcal{T} \) is sequential if its transactions are not interleaved with operations from other transactions. That is, for every \( a, b, c \in s \) with \( a \prec_s b \prec_s c \) and \( a, c \in T \) implies \( b \in T \) for every \( T \in \mathcal{T} \). Adopting the view of schedules as sequences, the schedule \( s_1 = T_1 \cdot T_2 \cdot \ldots \cdot T_n \) is an example of a sequential schedule for the set of transactions \( \{T_1, T_2, \ldots, T_n\} \) as is any permutation of transactions in \( s_1 \).

2.2 Conflict Serializability

We say that two operations \( R_i \) and \( W_j \) from different transactions \( T_i \) and \( T_j \) are conflicting if both are operations on the same object, and at least one of them is a write. That is, \( R_i[x] \) and \( W_j[x] \), and \( W_i[x] \) and \( W_j[x] \) are conflicting operations while \( R_i[x] \) and \( R_i[x] \) or \( R_i[x] \) and \( W_j[x] \) are not. Furthermore, a commit operation never conflicts with any other
operation. Two schedules \( s \) and \( s' \) are conflict equivalent if they are over the same set \( T \) of transactions and if any pair of conflicting operations \( a \) and \( b \) is ordered the same in both, that is, \( a \leq_s b \) iff \( a \leq_{s'} b \).

Definition 1. A schedule \( s \) is conflict serializable if it is conflict equivalent to a sequential schedule.

A conflict graph \( CG(s) \) for schedule \( s \) over a set of transactions \( T \) is defined as usual [17]: it is the graph whose nodes are the transactions in \( T \) 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 \( T_j \) with \( b_i <_s a_j \). 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 \( \lambda \) 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 these transactions as a set of quadruples \((b_i, a_j, T_i, T_j)\) 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 only contain read and write operations, no 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
\[
(T_i, b_i, a_{i+1}, T_{i+1}) \cdots (T_n, b_n, a_1, T_1)(T_1, b_1, a_2, T_2) \cdots (T_{i-1}, b_{i-1}, a_i, T_i).
\]

We recall the following well-known result:

Theorem 2. [17] A schedule \( s \) is conflict serializable iff the conflict graph for \( s \) is acyclic.

2.3 Isolation Levels
We define isolation levels in terms of the concurrency phenomena that we want to exclude from schedules [9].

Let \( s \) be a schedule for a set \( T \) of transactions.

- Then, \( s \) exhibits a dirty write iff there are two different transactions \( T_i \) and \( T_j \) in \( T \) and an object \( x \) such that
  \[
  W_i(x) \leq_s W_j(x) \leq_C C_1.
  \]
  That is, transaction \( T_j \) writes to an object that has been modified earlier by \( T_i \), but \( T_j \) has not yet issued a commit.

- Furthermore, \( s \) exhibits a dirty read iff there are two different transactions \( T_i \) and \( T_j \) in \( T \) and an object \( x \) such that
  \[
  W_i(x) \leq_s R_j(x) \leq_C C_1.
  \]
  That is, transaction \( T_j \) reads an object that has been modified earlier by \( T_i \), but \( T_j \) has not yet issued a commit.

Definition 3. A schedule \( s \) 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. For convenience, we use the term no isolation to refer to the isolation level that allows all schedules.

Notice that every schedule is allowed under no isolation. Furthermore, every schedule allowed under read committed is also allowed under read uncommitted. It is customary to view an isolation level as a set of allowed schedules [17].

We say that an isolation level \( I \) is a restriction of an isolation level \( I' \), denoted \( I \subseteq I' \), if the fact that a schedule \( s \) is allowed under \( I \) implies that \( s \) is allowed under \( I' \) as well.

2.4 Robustness
Next, we define the robustness property [10] (also called acceptability in [15, 16]), which guarantees serializability for all schedules of a given set of transactions for a given isolation level.

Definition 4 Robustness. A set \( T \) of transactions is robust against an isolation level \( I \) if every schedule for \( T \) that is allowed under that isolation level is conflict serializable.

For an isolation level \( I \), ROBUSTNESS \( (I) \) is the problem to decide if a given set of transaction \( T \) is robust against \( I \). The following is an immediate observation:

Lemma 5. Let \( T \) be a set of transactions. Let \( I \) and \( I' \) be isolation levels with \( I \subseteq I' \). Then \( T \) is robust against \( I' \) implies that \( T \) is robust against \( I \).

3 NO ISOLATION LEVEL
We start by studying the toy isolation level no isolation that admits all schedules. The present section serves as a warm up for the remainder of the paper and allows us to discuss key notions like the interference graph, transferable cycle, and split schedule in a simplified setting.

We use a variant of the interference graph, as introduced by Fekete [15], 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 6. For a set of transactions \( T \), the interference graph \( IG(T) \) for \( T \) is the graph whose nodes are the transactions in \( T \) 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 \( \lambda \) 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 \( IG(T) \) as a set of quadruples of the form \((T_i, b_i, a_j, T_j)\). That is, \((T_i, b_i, a_j, T_j) \in IG(T)\) 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 \((T_i, b_i, a_j, T_j) \in IG(T)\) implies \((T_j, a_j, b_i, T_i) \in IG(T)\). Furthermore, the conflict graph \( CG(s) \) for a schedule \( s \) for \( T \) is always a subgraph of the interference graph \( IG(T) \) for \( T \). Therefore, every cycle in \( CG(s) \) is a cycle in \( IG(T) \). However, the converse is
not always true. Sometimes a cycle in IG(T) can be found that does not translate to a corresponding cycle in the conflict graph for any schedule for T. We therefore introduce the notion of a transferable cycle in an interference graph and show in Lemma 10 that whenever there is a transferable cycle in IG(T) there is a schedule s of a specific form called a split schedule (as defined in Definition 8) that admits a cycle in CG(s).

**Definition 7.** Let T be a set of transactions and C a cycle in IG(T). Then, C is non-trivial if for some pair of edges (T_i, b_j, a_j, T_j) and (T_j, b_j, a_i, T_i) in C the operations b_j and a_i 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_i, T_i) in C. We then say that C is transferable in T on operations (b_j, a_i).

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 8** Split schedule. Let T be a set of transactions and C a transferable cycle in IG(T). A split schedule for T based on C has the form

\[ \text{prefix}_s(T_1) \cdot T_2 \cdot \ldots \cdot T_m \cdot \text{postfix}_s(T_1) \cdot T_{m+1} \cdot \ldots \cdot T_n, \]

where

- \((T_m, b_m, a, T_1)\) and \((T_1, b, a, T_2)\) 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 CG(T) in the order as they occur; and,
- \(T_{m+1}, \ldots, T_n\) are the remaining transactions in T in an arbitrary order.

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

We say that a schedule s is a split schedule for T if there is a transferable cycle C in IG(T) such that s is a split schedule for T based on C. Figure 2 provides an abstract view of a split schedule omitting the trailing sequence \(T_{m+1} \cdot \ldots \cdot T_n\).

**Example 9.** Consider \(T = \{T_1, T_2, T_3\}\) with \(T_1 = R_1[x]W_1[y]C_1, T_2 = R_2[y]W_2[z]C_2\) and \(T_3 = R_3[z]R_3[x]W_3[x]W_3[z]C_3\). Then IG(T) is depicted in Figure 3. The cycle \(C_1\) consisting of the following edges \((T_1, W_1[y], R_2[y], T_2), (T_2, W_2[z], W_3[z], T_3), (T_3, W_3[x], R_1[x], T_1)\) is transferable in T on \(W_3[x]W_3[z]\) as \(W_3[x] < T_3 W_3[z]\). The cycle \(C_2\) consisting of the following edges \((T_1, W_1[y], R_2[y], T_2), (T_2, W_2[z], R_3[z], T_3), (T_3, W_3[x], R_1[x], T_1)\) is not transferable in T on \(W_3[x], R_3[z]\) as \(W_3[x] \neq T_3 R_3[z]\). The split schedule \(s_1\) for T based on \(C_1, T_3, W_3[x]\) is as follows:

\[
\text{prefix}_s(T_1) \cdot T_1 \cdot T_2 \cdot \text{postfix}_s(T_1)'
\]

with \(b = W_3[x]\). □

The following lemma collects some interesting properties of transactions.

**Lemma 10.** Let T be a set of transactions.

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

We are now ready to formulate a theorem that provides a characterization for deciding robustness against no isolation:

**Theorem 11.** A set T of transactions is not robust against isolation level no isolation if IG(T) contains a non-trivial cycle.

**Proof.** (1 → 2) Let s be a schedule for T that is not conflict serializable. Then there is a cycle C in its conflict graph CG(T) (by Theorem 2) which is a transferable cycle in IG(T) due to Lemma 10(1). Furthermore, a transferable cycle is non-trivial by definition.

(2 → 3) By Lemma 10(2) there is a transferable cycle C in IG(T). This cycle can be used to construct a split schedule for T.

(3 → 1) Immediate by Lemma 10(3). □

Next, we discuss the complexity of deciding robustness. Because the interference graph IG(T) of a set T of transactions is bidirectional, it has a natural undirected interpretation. In the next theorem, the upper bound is based on the result that undirected reachability is in logspace [18]. The lower-bound is obtained by a logspace-complete undirected acyclicity problem [14].

**Theorem 12.** Robustness (no isolation) is logspace-complete.

### 4 READ UNCOMMITTED

In this section, we discuss robustness against read uncommitted. This means that counter example schedules can no longer take arbitrary forms but must adhere to the read uncommitted isolation level. We therefore need additional requirements beyond non-triviality for cycles in interference graphs.

The work by Fekete et al. [15, 16] approaches the robustness problem by reasoning on cycles in interference graphs based on
the types of conflicts occurring in them without taking the specific operations responsible for these conflicts into account. Types of conflicts are, for instance, write-write, write-read, and read-write dependencies between transactions. In this view, it might be tempting to think that a characterization for robustness against read uncommitted can be found in terms of transferable cycles in IG(\(T\)) without write-write conflicts. However, consider \(T = (\{w_1[x], r_1[y], w_2[z], r_2[y], z\})\). Then, there is a transferable cycle \((T_1, r_1[y], w_2[y], T_2, r_2[z], w_1[x], T_1)\) without write-write conflicts but no counter example schedule can be found that is allowed under read uncommitted due to the presence of the leading write to \(x\) in both \(T_1\) and \(T_2\). Furthermore, a cycle of a schedule allowed under read uncommitted can still have write-write conflicts. Indeed, the schedule \(s_1 = r_1[x], w_2[x], w_2[y], c_2[y], w_1[y], c_1\) is allowed under read uncommitted since there is no dirty write but the (only) cycle in CG(s₁) has a write-write conflict on y.

The higher level explanation why it is necessary to reason about operations instead of transactions is that the isolation level read uncommitted (and read committed) does not guarantee atomic visibility requiring that either all or none of the updates of each transaction are visible to other transactions. More formally, a schedule \(s\) over a set of transactions \(T\) guarantees atomic visibility when \(w_i[x] <_s r_j[y] \iff w_i[y] <_s r_j[y]\) for all \(T_i, T_j \in T\). For instance, the schedule \(s_2 = r_1[x], r_2[y], w_2[x], w_2[y], c_2[r_1[y]], c_1\) is allowed under read uncommitted but does not guarantee atomic visibility as \(r_1[x] <_s w_2[x]\) but \(w_2[y] <_s r_1[y]\). When an isolation level guarantees atomic visibility it suffices to reason on the level of transactions rather than on the order of operations occurring in them [11]. For read uncommitted (and read committed), we do need to take the ordering of operations in individual transactions into account as will become apparent in the notion of prefix-write-conflict-free cycle as defined next.

**Definition 13.** Let \(T\) be a set of transactions and let \(C\) be a cycle in IG(\(T\)). Let \(T \in T\) 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 prefix-write-conflict-free cycle in \(T\) that conflicts with a write operation in a transaction in \(C \setminus \{T\}\).

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

**Example 14.** Cycle \(C_3\) of Example 9 is prefix-write-conflict-free in \(T_3\) on operations \((w_3[x], w_2[z])\). Indeed, there is no write operation in \(T_2\) or \(T_3\) to object \(x\). Notice that the split schedule \(s_1\) of Example 9 is allowed under read uncommitted. The next lemma shows that this is always the case.

**Lemma 15.** Let \(T\) be a set of transactions. Let \(C\) be a prefix-write-conflict-free cycle in IG(\(T\)). Then, there is a split schedule for \(T\) based on \(C\) that is allowed under isolation level read uncommitted.

**Proof.** Let \(T \in T\) and \(b, a \in T\) such that \(C\) is prefix-write-conflict-free in \(T\) on \((b, a)\). Let \(s\) be the split schedule based on \(C\), \(T\) and \(b\) as defined in Definition 8. As \(T\) is the only transaction whose operations are not consecutive in \(s\), the only possibility for a dirty write is when there is a write operation in prefix-write-conflict-free cycle that is allowed under isolation level read uncommitted.

We are now ready to formulate a theorem that provides a characterization for deciding robustness against read uncommitted in terms of the existence of prefix-write-conflict-free cycles. It readily follows from Lemma 15 and Lemma 10(3) that the existence of a prefix-write-conflict-free cycle is a sufficient condition for the existence of a counter example schedule. The next theorem establishes that it is also a necessary condition and in addition that always a counter example in the form of a split schedule can be found.

**Theorem 16.** Let \(T\) be a set of transactions. The following are equivalent:

1. \(T\) is not robust against isolation level read uncommitted;
2. IG(\(T\)) contains a prefix-write-conflict-free cycle; and,
3. there is a split schedule \(s\) for \(T\) that is allowed under read uncommitted.

**Proof Sketch.** (3→1) Immediate by Lemma 10(3).
(2→3) Follows from Lemma 15.
(1→2) Let \(T\) be a set of transactions that is not robust against isolation level read uncommitted. Towards a contradiction, suppose that IG(\(T\)) contains no prefix-write-conflict-free cycle. The following is then implied by Definition 13:

- For every cycle \(C\) in IG(\(T\)) that is transferable in some \(T_i \in C\) and on some pair of operations \((b, a)\), there is a write \(w_i[x] \in T_i\) with \(w_i[x] \leq_T b\) and a transaction \(T_k \in C\) different from \(T_i\) with a write \(w_k[x] \in T_k\).

By Theorem 2 and the definition of robustness (Definition 4) there is a schedule \(s\) for \(T\) under read uncommitted that admits a cycle \(C\) in CG(s). W.l.o.g., we can assume that \(C\) is a minimal cycle, that is, there is no cycle in CG(s) consisting of a strict subset of the transactions occurring in \(C\). By Lemma 10(1), \(C\) is a transferable cycle in IG(\(T\)). Furthermore, assumption (1) applies to \(C\).

When \(C\) is transferable in \(T\) on some operation \((b, a)\), we also say that \(T\) is a breakable transaction. The name comes from the fact that \(C\) can be split on \(T\) to create a split schedule. That is, \(T\) needs to be broken to create the split schedule.

The assumption (1) allows to derive the existence of conflicting write operations for neighboring transactions (of which at least one is breakable) in a transferable cycle. As the schedule \(s\) can not exhibit dirty writes, the ordering of these writes in \(s\) determines the ordering of the commits of the respective transactions in \(s\) as well. The general idea is now to order neighboring transactions (w.r.t. \(<_s\)) for all breakable transactions and extend this partial order to a complete order for all other transactions in \(C\). But as \(C\) is cyclic this means that there will be a transaction that is smaller than itself (w.r.t. \(<_s\)) which leads to the desired contradiction.

We distinguish two cases: \(C\) consists of only two edges and \(C\) contains strictly more than two edges. In the former case the simple structure allows for a more direct argument. In the latter case, we are sure that nodes have two different neighbors in the cycle but more care needs to be taken to compute the contradicting ordering in an iterative manner depending on the structure of breakable transactions.

\[\square\]
The following theorem establishes the complexity of deciding robustness against read uncommitted.

**Theorem 17.** robustness(read uncommitted) is logspace-complete.

5 READ COMMITTED

Next, we discuss robustness against read committed which means that counter example schedules must adhere to the read committed isolation level. This section contains two main results: (i) a characterization of robustness against read committed in terms of multi-split schedules and multi-prefix-conflict-free cycles (Theorem 23); and, (ii) comp-hardness of the associated decision problem (Theorem 27).

5.1 Multi-split schedules

We start by showing that when a counter example schedule exists, it can always take the form of a multi-split schedule based on a transferable cycle as defined below. In contrast to a split schedule where one transaction is split open and all other transactions are inserted in between in the order as they occur in the cycle, a multi-split schedule can open several transactions appearing consecutively in the cycle but needs to close them in sequence. Figure 2 provides an abstract view of a split schedule omitting the possible trailing sequence of non-interleaved transactions. To facilitate the definition of multi-split schedules, we assume that the first transaction in the cycle that the schedule is based on, is the first transaction that is opened.

**Definition 18.** Let \( T \) be a set of transactions and \( C \) a cycle in \( IG(T) \) that is transferable in its first transaction \( T_1 \) on operations \((b_1, a_1)\). A multi-split schedule for \( T \) based on \( C \) is any schedule of the form

\[
\text{prefix}_{\epsilon(T_i)}(T_1) \cdot \ldots \cdot \text{prefix}_{\epsilon(T_m)}(T_m) \cdot \text{postfix}_{\epsilon(T_1)}(T_1) \cdot \ldots \cdot \text{postfix}_{\epsilon(T_m)}(T_m) \cdot T_{m+1} \cdot \ldots \cdot T_n,
\]

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

1. \( \epsilon(T_i) = b_i \);
2. if \( \epsilon(T_{i-1}) \) then \( \epsilon(T_i) = C_i \); and,
3. if \( \epsilon(T_{i-1}) \neq C_i \) then \( \epsilon(T_i) = b_i \) or \( \epsilon(T_i) = C_i \) with the edge \((T_{i-1}, b_i, a_j, T_i)\) in \( C \) for some \( j \).

The transaction \( T_i \) is called open when \( \epsilon(T_i) \neq C_i \) and is closed otherwise. Notice that for a closed transaction \( T_i \), prefix_{\epsilon(T_i)}(T_i) = T_i\) and postfix_{\epsilon(T_i)}(T_i) is fully split when all transactions are open, that is, \( \epsilon(T_i) \neq C_i \) for all \( i \in [1, m] \).

We say that \( s \) is a multi-split schedule for \( T \) if it is a multi-split schedule for \( T \) based on some cycle \( C \). Notice that there is always a number \( k \geq 0 \) 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 19.** Let \( s \) be a multi-split schedule for a set of transactions \( T \) based on a cycle \( C \) in \( IG(T) \). Then \( C \) is also a cycle in \( CG(s) \).

The previous lemma does not imply that \( s \) is allowed under read committed. To this end, we introduce the definition of a multi-prefix-conflict-free cycle. First, we define the following notions.

Let \( T \) be a set of transactions, \( C \) a cycle in the interference graph \( IG(T) \), and \( T \) a transaction in \( T \). Then there is precisely one edge of the form \((T, b, a, T')\) in \( C \) for some \( b \in T \), \( T' \in T \), and \( a \in T' \). For ease of notation, we write \( bC(T) \) to denote \( b \) and \( aC(T) \) to denote \( a \). When \( C \) is clear from the context, we also write \( a(T) \) and \( b(T) \) for \( aC(T) \) and \( bC(T) \), respectively.

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 20.** Let \( T \) be a set of transactions and let \( C \) be a cycle in \( IG(T) \) containing transactions \( T \) and \( T' \). Then \( C \) is multi-prefix-conflict-free if \( C \) is transferable in \( T \) and for every transaction \( T_j \) that is equal to \( T' \) or occurs before \( T' \) in \( C[T] \) there is no write operation in prefix_{\epsilon(T_i)}(T_i) that

- conflicts with a read or write operation in prefix_{\epsilon(T_i)}(T_i) 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,
- conflicts with a read or write operation in postfix_{\epsilon(T_i)}(T_i) of some transaction \( T_j \) occurring strictly before \( T_i \) in \( C[T] \).

The next lemma says that when a multi-prefix-conflict-free cycle can be found, a corresponding counter example multi-split schedule witnessing non-robustness against read committed can be constructed. In Theorem 23, we show that the latter is also a necessary condition.

**Lemma 21.** Let \( T \) be a set of transactions. Let \( C \) be a cycle in \( IG(T) \) that is multi-prefix-conflict-free in \( T \) and \( T' \). Then, there is a multi-split schedule for \( T \) based on \( C \) that is allowed under isolation level read committed.

**Example 22.** Consider \( T = \{T_1, T_2, T_3\} \) with \( T_1 = W_1[x]W_1[y]C_1 \), \( T_2 = R_2[v]R_2[z]W_2[v]W_2[x]C_2 \) and \( T_3 = R_3[y]W_3[z]C_3 \). Then \( IG(T) \) is depicted in Figure 4. The cycle \( C \) consisting of the following edges \((T_1, W_1[x], W_1[x], T_2), (T_2, R_2[z], W_3[z], T_3), (T_3, R_3[y], W_1[y], T_1)\) is multi-prefix-conflict-free in \( T_1 \) and \( T_2 \). The multi-split schedule \( s \) for \( T \) based on \( C \) where \( T_1 \) and \( T_2 \) are open and \( T_3 \) is closed is as shown.
Then, $C$ and $R$ fashion. appending to $U$ s conflict graph of $C$ transactions not occurring in be the schedule obtained from $s$ read committed.

In the proof of Theorem 23, we show that any counter example schedule witnessing non-robustness against read committed can be transformed into one that is a multi-split schedule. Basically, in a multi-split schedule every transaction is represented by one or two blocks of consecutive operations. Indeed, an open transaction is represented by two blocks while closed transactions as well as trailing transactions are represented by one block. We refer to such blocks of consecutive operations within a transaction as a chunk. Formally, in a schedule $s$ for $T$, we call a maximal sequence of consecutive operations from the same transaction $T$ a chunk of $T$ in $s$. For instance, in Figure 1, $T_1$ is represented in $s_1$ by one chunk $(w_1[x], [y, z], w_1[y], c_1)$ while $T_2$ is represented by two chunks $(w_2[z])$ and $R_2[y]$, $w_2[x], r_2[c_2]$).

**Theorem 23.** Let $T'$ be a set of transactions. The following are equivalent:

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

**Proof Sketch.** $(3) \rightarrow (2)$ Let $s$ be the assumed multi-split schedule for $T'$ based on a cycle $C$ that is allowed under read committed. Then, $C$ is in $CG(s)$ by Lemma 19. Let $T \in C$ be the first transaction that appears in $s$. Let $T'$ denote the last transaction in $C$ that appears with two chunks in $s$. Then, $C$ is multi-prefix-conflict-free in $T$ and $T'$. Indeed, every transaction $T_i$ equal to $T'$ or occurring before $T'$ in $C$ has exactly two chunks in $s$. Assume there is a write operation $a$ in prefix$_b(T_i)$ (with $(T_i, b, a_{i+1}, T_{i+1})$ in $C$) and a conflicting read or write operation $b$ in prefix$_b(T_j)$ for transaction $T_j$ occurring after $T_i$ in $C$ (with $(T_j, b, a_{j+1}, T_{j+1})$ in $C$). Then, we have by definition of multi-split schedule that $a <_s b <_s c_1$, which contradicts with $s$ being allowed under read committed. The case $b$ in postfix$_b(T_j)$ with $T_j$ occurring before $T_i$ in $C$ implies $a <_s b <_s c_1$ as well.

$(2) \rightarrow (1)$ Follows immediately, as by Lemma 21 and Lemma 19 there is a schedule $s$ for $T'$ that is allowed under read committed and that has a cycle in $CG(s)$.

$(1) \rightarrow (3)$ By Theorem 2 there is a schedule $s_0$ for $T'$ allowed under read committed with a cycle $C$ in its conflict graph.

Let $\mathcal{U} \subseteq T'$ denote the transactions occurring in $C$ and let $s$ be the schedule obtained from $s_0$ by removing all operations from transactions not occurring in $C$. Notice that $C$ is a cycle in the conflict graph of $s$ and that $s$ is a schedule for $\mathcal{U}$ allowed under read committed. Moreover, if a multi-split schedule $s'$ exists for $\mathcal{U}$ that is allowed under read committed, we can easily obtain a multi-split schedule for $T'$ allowed under read committed by appending to $s'$ all missing transactions (those in $T' \setminus \mathcal{U}$) in a serial fashion.

The case where $\mathcal{U}$ contains precisely two transactions is treated in Lemma 32 in the appendix. Henceforth, we assume that $\mathcal{U}$ contains at least three transactions. Moreover, we assume that the following property applies to $s$ and $C$:

(i) $C$ is minimal in $CG(s)$ and contains at least three transactions; no schedule for $\mathcal{U}$ allowed under read committed exists with a cycle in its conflict graph mentioning a strict subset of the transactions in $C$. Furthermore, $s$ is allowed under read committed.

The construction requires four phases. In each phase, we transform schedule $s$ one step closer to the desired form. Eventually, we obtain a schedule $s'$ for $\mathcal{U}$ satisfying Properties (i-v):

(ii) Every transaction $T_i$ consists either of only one chunk or exactly two chunks in $s'$. In the latter case, the last operation of the first chunk of $T_i$ conflicts with an operation from transaction $T_{i+1}$ occurring after $T_i$ in $C$.

(iii) In the following, let $T_1$ be the transaction whose first operation occurs first in $s'$. Then $T_1$ consists of two chunks in $s'$. Furthermore, all pairs of chunks in $s'$ between the first and last chunk of $T_1$ and all pairs of chunks in $s'$ after the last chunk of $T_1$ appear in the same order as their corresponding transactions appear in $C$.

(iv) Every transaction (except $T_1$) has a chunk between the first and last chunk of $T_1$.

(v) If $T_1$ consists of only one chunk, then the transaction $T_{i+1}$ occurring after $T_1$ in $C$ (unless it is $T_1$) consists of only one chunk.

Notice that a schedule $s$ and cycle $C$ having Properties (i-v) indeed represent a multi-split schedule based on $C$ that is allowed under read committed, with as the mapping that maps $T_1$ on the last operation of its first chunk in $s$, which is either some read or write operation from $T_1$ (if $T_1$ has two chunks) or $C_1$ (if $T_1$ has only one chunk).

**5.2 Intermezzo: Properly colored cycles**

In this section, we study the complexity of a decision problem over colored graphs. Even though the problem is not directly related to deciding robustness, the reduction we present provides the no-frills intuition that will be central in the more complex reduction presented next in Section 5.3.

A **vertex-colored graph** is a tuple $G = (V, E, K, f)$ where $V$ is a finite set of nodes, $E \subseteq V \times V$ is the set of edges, $K$ is a finite set of colors, and $f$ maps each vertex in $V$ to a color in $K$. As before, a cycle $C$ is a non-empty sequence of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$ such that every vertex in $V$ does not occur in $C$ or occurs precisely twice. The latter in particular means that $C$ is simple. We say that $C$ is **properly colored** if for each two vertices $v_1$ and $v_2$ occurring in $C$ (not necessarily adjacent in $C$), $(v_1, v_2) \in E$ implies $f(v_1) \neq f(v_2)$. So, the induced subgraph of $G$ determined by the vertices occurring in $C$ should color adjacent vertices differently.

Let **ProperlyColoredCycle** be the problem to decide if a given colored graph contains a properly colored cycle. In this section, we show the following result:

**Proposition 24.** **ProperlyColoredCycle** is **NP-complete**.
As the upper-bound is straightforward, it remains to argue that \textsc{ProperlyColoredCycle} is also \textsc{NP}-hard. The proof is by a reduction from 3SAT. To this end, let \( \varphi \) be a propositional logic formula in 3CNF and let \( \text{Vars}(\varphi) \) be the set of variables occurring in \( \varphi \). We recall that \( \varphi \) is a conjunction of clauses \( C_i \) of the form \( L_{i,1} \lor L_{i,2} \lor L_{i,3} \) and each literal \( L_{i,t} \) equals \( x \) or \( \overline{x} \), with \( x \in \text{Vars}(\varphi) \). For ease of notation, we assume \( \text{Vars}(\varphi) = \{x_1, \ldots, x_m\} \) and we refer to the clauses in \( \varphi \) by \( C_{m+1}, \ldots, C_n \), thus with the variables and clauses having indices over disjoint intervals.

Next, we construct a vertex-colored graph \( G(\varphi) \) and show that \( G(\varphi) \) contains a properly colored cycle iff \( \varphi \) is satisfiable.

For the construction, we distinguish the following gadgets, which are disjoint subgraphs of \( G(\varphi) \):

- A variable gadget \( G(x_i) = (V_i, E_i) \) for every variable \( x_i \) in \( \varphi \) with vertices and edges as depicted in Figure 5a; the intuition is that \( v_{i,1} \) encodes the choice to make \( x_i \) \textit{true} and \( v_{i,2} \) encodes the choice to make \( x_i \) \textit{false}. A path from \( v_{i,\text{in}} \) to \( v_{i,\text{out}} \) then encodes the inverse truth assignment for \( x_i \);
- for every clause \( C_j = (v_{j,1}, E_j) \) for every clause \( C_j \) in \( \varphi \) with vertices and edges as depicted in Figure 5b; the intuition is that \( v_{j,1} \) encodes the literals \( L_{j,t} \) in \( C_j \). A path from \( v_{j,\text{in}} \) to \( v_{j,\text{out}} \) then encodes the choice of which literal in \( C_j \) is true.

Now, define \( G(\varphi) = (V_{\varphi}, E_{\varphi}, K_{\varphi}, f_{\varphi}) \) as the following vertex-colored graph:

- \( V_{\varphi} = \{v_0\} \cup V_1 \cup \cdots \cup V_n \) contains a special start vertex \( v_0 \) and the vertices necessary to describe gadgets \( G(x_i) \) and \( G(C_j) \) for every variable \( x_i \) and clause \( C_j \) in \( \varphi \);
- \( E_{\varphi} \) consists of the following edges:
  - edges \( E_1 \) and \( E_2 \) from gadgets \( G(x_i) \) and \( G(C_j) \) for every variable \( x_i \) and clause \( C_j \) in \( \varphi \);
  - edges from \( v_{i,\text{out}} \) to \( v_{i+1,\text{in}} \), for \( i \in [1, n-1] \), to chain all variable gadgets and clause gadgets after one other;
  - edges \( (v_0, v_{1,\text{in}}) \) and \( (v_{m,\text{out}}, v_0) \) to connect the chain with start node \( v_0 \) creating a cycle;
  - edges between variables in variable gadgets and their occurrence in clause gadgets:
  - an edge from each vertex \( v_{i,1} \) in a variable gadget to each vertex \( v_{j,1} \) in clause gadget \( C_j \) representing a literal \( L_{j,t} = x_i \) (recall that \( v_{i,1} \) encodes \textit{true} for \( x_i \));
  - an edge from each vertex \( v_{i,2} \) in variable gadgets to each vertex \( v_{j,1} \) in clause gadget \( C_j \) representing a literal \( L_{j,t} = \overline{x_i} \) (recall that \( v_{i,2} \) encodes \textit{false} for \( x_i \)).

We refer to these types of edges as consistency edges as appropriate coloring will ensure a consistent interpretation of the truth assignment.

- \( K_{\varphi} = K_{\text{variable}} \cup K_{\text{other}} \) with
  - \( K_{\text{variable}} = \{x, \overline{x} \mid x \in \text{Vars}(\varphi)\} \); and
  - \( K_{\text{other}} \) a set of \( |V_{\varphi}| - 3n + m \) colors distinct from \( K_{\text{variable}} \).

\( f_{\varphi} \) is defined as follows:

- \( f_{\varphi}(v_{i,1}) = x_i \) and \( f_{\varphi}(v_{i,2}) = \overline{x_i} \) for every \( x_i \in \text{Vars}(\varphi) \);
- \( f_{\varphi}(v_{j,1}) = L_{j,t} \) for \( j \in [m+1, n] \) and \( t \in \{1, 2, 3\} \);
- for all other vertices \( v \in V_{\varphi}, f(v) \) is assigned a different color in \( K_{\text{other}} \).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{gadgets}
\caption{Gadgets for the construction of \( G(\varphi) \).}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example}
\caption{G(\varphi_1) for \( \varphi_1 = (x_1 \lor x_2 \lor \overline{x}_3) \land (\overline{x}_1 \lor x_2 \lor x_3) \). For ease of exposition, vertices assigned with a unique color from \( K_{\text{other}} \) are left blank.}
\end{figure}

\begin{example}
Consider \( \varphi_1 = (x_1 \lor x_2 \lor \overline{x}_3) \land (\overline{x}_1 \lor x_2 \lor x_3) \). Then \( G(\varphi_1) \) is given in Figure 6. \hfill \Box
\end{example}

The following lemma then implies \textsc{NP}-hardness.

\begin{lemma}
Let \( \varphi \) be a propositional logic formula in 3CNF. Then, \( \varphi \) is satisfiable iff \( G(\varphi) \) has a properly colored cycle.
\end{lemma}

\begin{proof}
(\textit{if}) Assume \( C \) is a properly colored cycle. By construction of \( G(\varphi) \), a properly colored cycle always needs to go through each variable and clause gadget exactly once. Indeed, no cycle can use one of the shortcut consistency edges as the adjacent vertices carry the same color. Therefore, \( C \) picks for every variable \( x_i \) either the vertex \( v_{i,1} \) encoding \textit{true} or vertex \( v_{i,2} \) encoding \textit{false} in the variable gadget \( G(x_i) \). Furthermore, in every clause gadget \( G(C_j) \), \( C \) picks a single vertex \( v_{j,1} \) encoding literal \( L_{j,t} \) in \( C_j \). Let \( \xi \) be the truth assignment that maps every variable \( x_i \) to \textit{false} when \( v_{i,1} \) is picked by \( C \) and to \textit{true} when \( v_{i,2} \) is picked. So, the choices of \( C \) represent the complement of the truth assignment. Notice, that under \( \xi \) every clause \( C_j \) evaluates to true. Indeed, let \( L_{j,t} \) be the literal picked by \( C \). When \( L_{j,t} = x_i \) for some \( x_i \in \text{Vars}(\varphi) \), then the vertices \( v_{j,1} \) and \( v_{x_i,1} \) in \( G(\varphi) \) are connected with a consistency edge and are both labeled with the same color. As \( C \) is properly colored, this means that \( C \) must have picked the vertex \( v_{i,2} \) and \( \xi(x_i) = \xi(L_{j,t}) = \text{true} \). The same reasoning holds when \( L_{j,t} = \overline{x}_i \). It thus follows that \( \varphi \) evaluates to true under \( \xi \).

(\textit{only if}) Let \( \xi \) be a satisfying truth assignment for \( \varphi \). Then, let \( C \) be the path through \( G(\varphi) \) that starts and ends in \( v_0 \) and that picks in every variable gadget \( G(x_i) \), the vertex \( v_{i,1} \) when \( \xi(x_i) \) is \textit{false} and \( v_{i,2} \) otherwise. Furthermore, \( C \) picks in every clause gadget \( G(C_j) \) a literal \( L_{j,t} \) such that \( \xi(L_{j,t}) \) is \textit{true}. The only possibility to violate properly coloring is through the consistency edges as these are the only edges where endpoints carry the same color. Assume
two vertices $v_{i,1}$ (with $i \in \{1, m\}$) and $v_{j,\ell}$ (with $j \in \{m + 1, n\}$) are picked by $C$ that carry the same color. By construction, this color then is $x_j$ meaning that $\xi(x_j) = false$ by assumption on the choice of $C$ on variables. Furthermore, $\xi(C_{j,\ell}) = true$ by assumption on the choice of $C$ in clause gadgets. This leads to the desired contradiction. A similar argument can be made when $v_{i,2}$ and $v_{j,\ell}$ are picked by $C$. This concludes the proof.

5.3 comp-completeness

Next, we turn to the main result of this section showing that robustness(Read Committed) is comp-complete. The remainder of this section is devoted to the proof of the following theorem:

Theorem 27. robustness(Read Committed) is comp-complete.

Obviously, robustness(Read Committed) is in comp. Indeed, for a set of transactions $T$, just guess a counter example schedule $s$ over $T$; then check that $s$ is allowed under Read Committed and that $CO(s)$ has a cycle. As the size of the guessed schedule is linear in the size of $T$, and the checking step is in polynomial time, the latter procedure is in NP.

The remainder of this section is devoted to a reduction from the NP-complete 3SAT problem to the complement of robustness(Read Committed), from which Theorem 27 then follows. For this, let $\phi$ be a boolean formula in 3CNF given as input to 3SAT. Thus, $\phi$ is a conjunction of clauses $C_j$ of the form $L_{j,1} \lor L_{j,2} \lor L_{j,3}$ with literals $L_{j,\ell}$ that either equal a variable $x$ or a variable’s complement $\overline{x}$, with $x \in \text{Vars}(\phi)$. Analogous to Section 5.2, we assume $\text{Vars}(\phi) = \{x_1, \ldots, x_m\}$ and refer to the clauses in $\phi$ by $C_{m+1}, \ldots, C_n$.

Next, we define a set $T(\phi)$ of transactions that (we will later show) is not robust under isolation level Read Committed iff $\phi$ is satisfiable. The construction is similar to the construction of $G(\phi)$ in the previous section. In fact, we construct $T(\phi)$ so that there exists exactly one transaction for every vertex in $G(\phi)$. All transactions corresponding to vertices in (variable and clause) gadgets follow the following template $(\star)$:

- write to a distinguished object that identifies the vertex under consideration;
- read the objects that identify the successor vertices; and,
- read all objects that identify the predecessor vertices.

When the transaction corresponds to an inner vertex of a gadget (a vertex of the form $v_{j,\ell}$ with $\ell \in [1, 3]$), the above template is preceded by writes to objects $U^j_{\ell}$ to deal with consistency edges.

A formal construction of $T(\phi)$ is given below. We omit defining Obj explicitly, as the necessary objects can be derived straightforwardly from the below transaction definitions. For ease of exposition we also omit $C_i$ at the end of every transaction $T_i$

For every variable $x_j$ in $\phi$, $T(\phi)$ contains a variable gadget $T(\phi, j)$ consisting of the following four transactions:

$T_{i,\text{in}} : W_{j,1}[X_1], R_{j,1}[Y_{1}], R_{j,1}[Y_{2}], R_{j,1}[Z_{j-1}]$.

$T_{i,1} : \text{conflict-set}_{i,1}, W_{j,1}[Y_1], R_{j,1}[Z_1], R_{j,1}[X_1]$.

$T_{i,2} : \text{conflict-set}_{i,2}, W_{j,1}[Y_2], R_{j,2}[Z_1], R_{j,2}[X_1]$.

$T_{i,\text{out}} : W_{j,1}[Z_1], R_{j,1}[X_{j+1}], R_{j,1}[Y_{1}], R_{j,1}[Y_{2}]$.

with conflict-set$_{i,1}$ and conflict-set$_{i,2}$ a sequence of write operations that will be specified later.

In this construction, $T_{i,\text{in}}$ and $T_{i,\text{out}}$ represent the in- and out-vertices of the variable gadget $G(x_j)$, that is, vertices $v_{i,\text{in}}$ and $v_{i,\text{out}}$, respectively. In addition, the transactions $T_{i,1}$ and $T_{i,2}$ represent the remaining two inner vertices $v_{i,1}$ and $v_{i,2}$, respectively. Notice, that these transactions correspond to the template $(\star)$.

Indeed, consider for instance the transaction $T_{i,\text{in}}$ corresponding to vertex $v_{i,\text{in}}$ which is identified by object $X_1$ and who has successors $v_{i,1}$ and $v_{i,2}$ in $G(\phi)$ represented by objects $Y_1$ and $Y_2$, respectively. Furthermore, $v_{i,\text{in}}$ has exactly one predecessor $v_{i-1,\text{out}}$ identified by $Z_{j-1}$ when $i > 1$, and otherwise has $v_0$ as predecessor which in turn is identified by object $Z_0$.

For every clause $C_j$ in $\phi$, we have a gadget $U(\phi, j)$ consisting of the five following transactions:

$T_{j,\text{in}} : W_{j,1}[X_1], R_{j,1}[Y_{1}], R_{j,1}[Y_{2}], R_{j,1}[Z_{j-1}]$.

$T_{j,1} : W_{j,1}[Y_1], W_{j,1}[Z_1], R_{j,1}[X_j]$.

$T_{j,2} : W_{j,1}[Y_2], W_{j,2}[Z_1], R_{j,2}[X_j]$.

$T_{j,3} : W_{j,1}[Y_1], W_{j,1}[Z_1], R_{j,3}[X_j]$.

$T_{j,\text{out}} : W_{j,1}[Z_1], R_{j,1}[X_{j+1}], R_{j,1}[Y_{1}], R_{j,1}[Y_{2}]$.

In this construction, $T_{j,\text{in}}$ and $T_{j,\text{out}}$ represent the in- and out-vertices of the clause gadget $G(C_j)$. The transactions $T_{j,1}$, $T_{j,2}$, and $T_{j,3}$ represent the remaining three inner vertices of the clause gadget. Notice that the above transactions follow template $(\star)$ as well. Furthermore, every $j$-th inner vertex has the additional identifier $U_{j,2}$ that its corresponding transaction writes to.

Finally, $T(\phi)$ contains also the next transaction, corresponding to vertex $v_0$ in $G(\phi)$:

$T_0 : W_0[Z_0], R_0[X_1], W_0[X_{n+1}]$.

It remains to specify the conflict sets, whose purpose it is to represent the consistency edges in $G(\phi)$. For $i \in [1, m]$, conflict-set$_{i,1}$ consists of all $W_{i,1}[U^j_{\ell}]$ such that $L_{j,\ell} = x_i$ in clause $C_j$ for some $j \in [m+1, n]$ and $\ell \in \{1, 2, 3\}$. Similarly, conflict-set$_{i,2}$ consists of all $W_{i,2}[U^j_{\ell}]$ such that $L_{j,\ell} = \overline{x}_i$ in clause $C_j$ for some $j \in [m+1, n]$ and $\ell \in \{1, 2, 3\}$. That is, every occurrence of variable $x_i$ (respectively, $\overline{x}_i$) in the $\ell$-th position of a clause $C_j$ is witnessed by a write to $U^j_{\ell}$ in $T_{i,1}$ (respectively, $T_{i,2}$).

Let $\beta : \mathcal{U}_\phi \leftrightarrow T(\phi)$ be the bijection that associates the vertices in $G(\phi)$ with their corresponding transaction in $T(\phi)$. The following lemma details the correspondence between $T(\phi)$ and $G(\phi)$:

Lemma 28. For every $\upsilon, \upsilon' \in \mathcal{U}_\phi$:

(1) $(\upsilon, \upsilon') \in E_\phi$ implies there is an edge from $\beta(\upsilon)$ to $\beta(\upsilon')$ in the interference graph of $T(\phi)$; and,

(2) an edge from $\beta(\upsilon)$ to $\beta(\upsilon')$ in the interference graph of $T(\phi)$ implies either $(\upsilon, \upsilon') \in E_\phi$ or $(\upsilon', \upsilon) \in E_\phi$.

As $T(\phi)$ can be constructed in LOGSPACE, Theorem 27 then follows from Lemma 29 and Lemma 30.

Lemma 29. If there is a properly colored cycle in $G(\phi)$ then $T(\phi)$ is not robust against read committed.

Proof. Let $C_{\phi}$ be a properly colored cycle in $G(\phi)$. As argued in the proof of Lemma 26, $C_{\phi}$ passes through the special vertex $v_0$ as well as through every variable and clause gadget in $G(\phi)$. Let the
following sequence be the result of applying $\beta$ on the vertices in $C$ in the order as they appear in $C$ starting with $v_0$:

\[ T_0, T_{1,\text{in}}, T_{1,\text{out}}, \ldots, T_{n,\text{in}}, T_{n,\text{out}}. \]

Denote the set consisting of all transactions in this sequence by $T'$. By Lemma 28, there is a cycle $C_T$ in $IG(T')$ that corresponds to $C_\phi$. Then, $C_T$ is transferable in $T_0$ on operations $(R_0|X_1], w_0|X_{n+1})$.

Next, we construct a multi-split schedule for $T'$. To this end, we introduce the following notation. Let $b_0 = R_0|X_1]$ and let

\[
 b_{i,\alpha} = \begin{cases} 
 R_{1,\text{in}}[Y_i^\ell], & \text{if } \alpha = \text{in} \text{ and } T_{i,\ell} \text{ follows } T_{i,\text{in}} \text{ in } C_T \\
 R_{1,\alpha}[Z_i], & \text{if } \alpha \in \{1, 2, 3\} \\
 R_{1,\text{out}}[X_{i+1}], & \text{if } \alpha = \text{out} 
\end{cases}
\]

for every $i \in [1, n]$. Clearly, $b_0 \in T_0$ and notice further that every $b_{i,\alpha}$ occurs in $T_{i,\alpha}$. For $i \in [1, n]$, denote by $\text{prefix}_i$ the sequence

\[
 \text{prefix}_{b_{i,\text{in}}}(T_{i,\text{in}}), \text{prefix}_{b_{i,k_1}}(T_{i,k_1}), \text{prefix}_{b_{i,\text{out}}}(T_{i,\text{out}}),
\]

and by $\text{postfix}_i$ the sequence

\[
 \text{postfix}_{b_{i,\text{in}}}(T_{i,\text{in}}), \text{postfix}_{b_{i,k_1}}(T_{i,k_1}), \text{postfix}_{b_{i,\text{out}}}(T_{i,\text{out}})
\]

Now, let $s'$ be the schedule over $T'$ of the following form:

\[
 \text{prefix}_{b_0}(T_0) \cdot \text{prefix}_1 \cdots \text{prefix}_n \\
 \text{postfix}_{b_0}(T_0) \cdot \text{postfix}_1 \cdots \text{postfix}_n.
\]

Notice that $s'$ is indeed a multi-split schedule based on $C_T$ on operations $(R_0|X_1], w_0|X_{n+1})$ (c.f., Definition 18).

We argue in the full version of this paper that $s'$ is allowed under read committed.

To conclude the proof, it suffices to remark that the transactions occurring in $T'(\phi)$, $T'$ can be appended to $s'$ in a serial fashion and in arbitrary order to obtain the required schedule $s$ for $T'(\phi)$ that is allowed under read committed. Indeed, $s$ is clearly still allowed under read committed and has cycle $C_T$ in its conflict graph. By Theorem 2, $T'(\phi)$ is thus not robust against read committed. \(\square\)

Lemma 28(1) provides a direct way to obtain a set of transactions from a properly colored cycle thereby facilitating the proof of Lemma 29. The main difficulty in the proof of the next lemma stating the converse direction is that the interference graph for $T'(\phi)$ is bidirectional and can therefore contain cycles not corresponding to a cycle in $G(\phi)$ which is problematic for the reduction.

**Lemma 30.** If $T'(\phi)$ is not robust against read committed then there is a properly colored cycle in $G(\phi)$.

**Proof.** Assume $T'(\phi)$ is not robust for read committed. According to Theorem 23, there exists a multi-split schedule $s$ for $T'(\phi)$ based on some transferable cycle $C_T$ that is allowed under read committed. We argue that $C_T$ corresponds to a properly colored cycle in $G(\phi)$. To this end, we introduce some notation. For $i \in [1, n]$, let

\[
 \omega^{\text{in}}_i := (T_{i,\text{in}}, b_{i,\text{in}}, a_{k_1}, k_1, T_{i,k_1}); \\
 \omega^{\text{out}}_i := (T_{i,k_1}, b_{i,k_1}, a_{\text{out}}, T_{i,\text{out}}); \text{ and,} \\
 \omega^{\ell}_i := (T_{i,\text{out}}, b_{i,\text{out}}, a_{\ell+1,\text{in}}, T_{i,\ell+1,\text{in}});
\]

where

\[
 b_{i,\alpha} = \begin{cases} 
 R_{1,\text{in}}[Y_i^\ell], & \text{if } \alpha = \text{in} \text{ and } T_{i,\ell} \text{ follows } T_{i,\text{in}} \text{ in } C_T \\
 R_{1,\alpha}[Z_i], & \text{if } \alpha \in \{1, 2, 3\} \\
 R_{1,\text{out}}[X_{i+1}], & \text{if } \alpha = \text{out} 
\end{cases}
\]

and

\[
 a_{i,\alpha} = \begin{cases} 
 W_{1,\text{in}}[X_i], & \text{if } \alpha = \text{in} \\
 W_{1,\alpha}^\omega [Z_i], & \text{if } \alpha \in \{1, 2, 3\} \\
 W_{1,\text{out}}[Z_i], & \text{if } \alpha = \text{out} 
\end{cases}
\]

Furthermore, let $a_0 = W_0|X_{n+1}]$, $b_0 = R_0|X_1]$. We prove the following two claims to be true in the full version of this paper.

(C1) The cycle $C_T$ is transferable in $T_0$ on $(b_0, a_0)$ and has the following form:

\[
 (T_0, b_0, a_{1,\text{in}}, T_{1,\text{in}}), \omega_1^{\text{in}}, \omega_1^{\text{out}}, \omega_1^{1}, \omega_2^{\ell}, \omega_2^{2}, \omega_2^{3}, \ldots, \omega_{n-1}^{\text{in}}, \omega_n^{\text{in}}, \omega_n^{\text{out}}, (T_{n,\text{out}}, b_{n,\text{out}}, a_0, T_0).
\]

(C2) The schedule $s$ is fully split.

It follows immediately from Claim (C1) that $C_T$ directly corresponds to a valid cycle $C$ through each gadget in $G(\phi)$, that is, edges are followed in the correct direction. Towards a contradiction, assume that $C$ is not a properly colored cycle in $G(\phi)$. Then, by construction, as similarly colored nodes are only connected through consistency edges, there are two transactions $T_{i,k}$ and $T_{j,\ell}$ with $i \in [1, m]$, $j \in [m+1, n]$, $k \in \{1, 2\}$, and $\ell \in \{1, 2, 3\}$, corresponding to the two vertices with the same color in respectively a variable gadget $G(x_\ell)$ and a clause gadget $G(y)$.

In this case, both $T_{i,k}$ and $T_{j,\ell}$ contain a write operation on object $U_j^\ell$ in respectively prefix$_{b_{i,k}}(T_{i,k})$ and prefix$_{b_{j,\ell}}(T_{j,\ell})$. However, by Condition (C2) postfix$_{b_{i,k}}(T_{i,k})$ is not empty, implying that the conflicting write of $T_{j,\ell}$ happens after the write of $T_{i,k}$, but before the commit of $T_{i,k}$.

As a result, $s$ cannot be allowed under read committed, leading to the desired contradiction. \(\square\)

**6 RELATED WORK**

In this section, we discuss the papers that considered (variants of) the robustness problem.

**Sufficient conditions.** Fekete et al. [16] studied the robustness problem for snapshot isolation by extending traditional conflict graphs with extra information w.r.t. the type of each conflict. In contrast to our interference graphs, these static dependency graphs only capture the possible types of conflicts between transactions but not the specific operations responsible for these conflicts. Based on these graphs, a sufficient condition for robustness against snapshot isolation is presented, as well as possible approaches on how to modify transactions when robustness is not guaranteed. The performance of these approaches is studied by Alomari et al. [2].

Alomari and Fekete [3] provide a sufficient condition for robustness against read committed, both under a lock based and multi-transaction semantics. This work uses the same graph approach as in [16]. The provided condition, however, is not a necessary condition and can therefore not be used to characterize robustness against read committed.

Cerone et al. [11] provide a framework for uniformly specifying different isolation levels in a declarative way. A key assumption in their framework is atomic visibility, requiring that either all or none of the updates of each transaction are visible to other transactions. This assumption facilitates reasoning over isolation levels, since these isolation levels can be specified by consistency axioms on
the level of transactions instead of individual operations within each transaction. Bernardi and Gotsman [10] extended the work of Fekete et al. [16] by providing sufficient conditions for robustness against the different isolation levels that can be defined by this framework. Continuing on this line of work, Cerone, Gotsman and Yang [13] examined the relationship between consistency axioms restricting the allowed schedules over a set of transactions and the resulting properties of possible cycles in the static dependency graph for this set of transactions. In particular, they provide a more direct approach to derive robustness criteria based on static dependency graphs from arbitrary isolation levels specified by consistency axioms. Cerone and Gotsman [12] later refined the sufficient condition for robustness against snapshot isolation first obtained by Fekete et al. [16]. They furthermore obtained a sufficient condition for robustness against parallel snapshot isolation towards snapshot isolation (i.e., whether for a given workload every schedule allowed under parallel snapshot isolation is allowed under snapshot isolation). However, the declarative framework by Cerone et al. [11] providing the foundation on which the above work is built, cannot be used to study read committed (and hence read uncommitted) as it does not admit the atomic visibility condition.

Characterizations. As mentioned before Fekete [15] is the first work that provides a necessary and sufficient condition for deciding robustness against snapshot isolation. In particular, that work provides a characterization for acceptable allocations when every transaction runs under either snapshot isolation or strict two-phase locking (S2PL). The allocation then is acceptable when every possible execution respecting the allocated isolation levels is serializable. As a side result, this work indirectly provides a necessary and sufficient condition for robustness against snapshot isolation, since robustness against snapshot isolation holds if the allocation where each transaction is allocated to snapshot isolation is acceptable.

Beillahi et al. use an algorithmic approach to decide robustness against causal consistency [8] and snapshot isolation [7] by providing a polynomial time reduction from these problems to the reachability problem in transactional programs over a sequentially consistent shared memory. Their setting is slightly different from our setting, as they allow a nondeterministic execution of transactions. They furthermore group transactions under different processes. During execution, each process then runs its transactions sequentially but concurrently with other processes. Due to this different setting, they obtain complexity bounds that are considerably higher than our complexity results. In particular, they show that deciding robustness against causal consistency and snapshot isolation are EXPSPACE-complete in general, and PSPACE-complete if respectively the number of sites or the number of processes is fixed.

Transaction chopping. Instead of weakening the isolation level, transactions can also be split in smaller pieces to obtain performance benefits. However, this approach poses a new challenge, as not every serializable execution of these chopped transactions is necessarily equivalent to some serializable execution over the original transactions. A chopping of a set of transactions is correct if for every serializable execution of the chopping there exists an equivalent serializable execution of the original transactions. Shasha et al. [19] provide a graph based characterization for this correctness problem. It is interesting to note that robustness against no isolation corresponds to the correctness of fully chopped transactions. Indeed, if we would chop each operation of each transaction into its own chopped transaction, then every serializable schedule of this chopping would clearly correspond to a schedule over the original transactions allowed under no isolation and vice versa. However, this relation is no longer trivial when considering robustness against read uncommitted and read committed. In particular, a correspondence between transaction chopping correctness and robustness against read committed is not to be expected, as the former is decidable in polynomial time [19], whereas we showed that the latter to be \textsc{conp}-complete.

7 CONCLUSIONS

In this paper, we provided characterizations for robustness against the isolation levels read uncommitted and read committed, and used these to establish upper bounds on the complexity of the associated decision problem. We also obtained matching lower bounds. The obtained characterizations provide insight in to what robustness means in these settings and under which circumstances it can occur.

While the characterizations in this paper are not restricted to the traditional lock-based semantics of the SQL isolation levels as they are defined in terms of forbidden patterns [9], it would be interesting to see what kind of characterizations for robustness can be found in terms of a multi-version definition of the isolation levels [1]. A second immediate question pertains the \textsc{conp}-hardness result: are there natural restrictions that make the problem tractable. In an online context with millions of transactions, testing robustness against read committed is obviously not feasible and tractable restrictions or approximations would be desirable. On the other hand, in an offline context, where the set of transactions is generated through a finite (and small) set of transaction programs, as discussed next, intractability is not necessarily problematic.

The initial motivation for the study of robustness lies in the performance improvement gained by executing transactions at a weaker isolation level without the danger of introducing anomalies [16]. It is important to point out that robustness makes the most sense in settings where transactions can be grouped together or where the set of possible transactions is known beforehand. A natural occurrence of the latter is when transactions are generated by a finite set of parameterized transaction programs as for example in a banking application where customers can do a fixed number of financial transactions. Consider the parameterized transaction \( r = \text{R}[v][w][v]\text{R}[w]\text{R}[w] \) that represents a transfer from an account \( v \) to an account \( w \) and where \( v \) and \( w \) are variables. Any transactions \( T = \text{R}[x][w][x]\text{R}[y]\text{R}[y] \) with \( x, y \in \text{Obj} \) then is an instance of \( r \). For this example, it could even make sense to interpret \( v \) and \( w \) with the same object \( x \). However, in some scenarios it makes sense to disallow different variables to be interpreted by the same object. In future work, we will study the robustness problem w.r.t. a formalization of parameterized transactions. In such a setting the same characterizations continue to hold but the interference
graphs become infinitely large. Initial results show that depending on particular enforced disjoint variable domain constraints, the same complexities for robustness as in this paper can be obtained.

Robustness is a binary property: a set of transactions is robust against a given isolation level or it is not. When robustness does not hold, one can devise methods to make a set of transactions robust or one can split up transactions into maximally robust subsets. These questions have been considered for snapshot isolation [12, 16] and it would make sense to consider them w.r.t. the different isolation levels occurring in database systems [5]. An orthogonal, and undoubtedly more challenging, setting, is to depart from the requirement that every transaction has to be executed at the same isolation level. That is, for a given set of transaction programs, allocate every transaction to the optimal isolation level for suitable notions of optimality. An immediate interpretation of optimality could be the weakest possible isolation level for every transaction that guarantees overall robustness for the whole set. Fekete [15] studied, and solved, the allocation problem w.r.t. snapshot isolation and strict two-phase locking, but no results of this flavor have been obtained for other isolation levels.

REFERENCES
A PROOFS FOR SECTION 5 (READ COMMITTED)

Let \( T \) be a transaction. A subsequence \( B \) of \( T \) is a sequence of consecutive operations in \( T \). If \( a \) is the next operation in \( T \) following the last operation in \( B \) then \( B \cdot a \) is the subsequence \( B \) extended with \( a \). Let \( T' \) be a set of transactions and \( s \) be a schedule for \( T' \). Let \( T \in T' \) and let \( B \cdot a \) be a subsequence of \( T \). Then we denote by \( s(B;a) \) the schedule obtained from \( s \) by first removing all operations in \( B \) in \( s \) and then inserting them just before \( a \) in \( s \). More formally, let \( s = s_1 \cdot a \cdot s_2 \). Then, \( s(B;a) \) is the schedule \( s'_1 \cdot B \cdot a \cdot s_2 \) where \( s'_1 \) is obtained from \( s_1 \) by deleting every operation in \( B \). Such actions will be performed to merge chunks in a schedule in the proof of the following theorem.

**Lemma 31.** Let \( T' \) be a set of transactions and \( s \) a schedule for \( T' \) allowed under isolation level \( I \in \{ \text{no isolation, read uncommitted, read committed} \} \). Let \( B \cdot a \) be a subsequence of some transaction \( T_j \in T' \). The schedule \( s(B;a) \) for \( T' \) obtained from \( s \) by removing all operations in \( B \) and inserting them in front of \( a \) is allowed under \( I \) if at least one of the following conditions is true:

1. For every operation \( c \) that conflicts with an operation \( d \) in \( s \) we have \( c <_s d \) or \( C_k <_s a \), with \( C_k \) the commit of the transaction that \( c \) is in. 
2. Operation \( a \) equals \( C_j \) and \( T_j \) is the transaction whose commit occurs last in \( s \). 
3. For every operation \( c \) that conflicts with an operation \( d \) in \( s \) we have \( c <_s d \) or \( c <_s c \).

**Proof.** Observe that Condition (2) implies Condition (1), since \( C_k <_s C_j = a \) follows from the assumption that \( T_j \) is the transaction whose commit occurs last in \( s \). In the remainder of the proof we show Property (1) and Property (3). Let \( s' = s(B;a) \).

1. For this, let \( c_k \in T_k \) and \( d_j \in T_j \) be two arbitrary conflicting operations with \( c_k <_s j \). Towards a contradiction, suppose that \( c_k \) and \( d_j \) witness a forbidden phenomenon in \( s' \) for isolation level \( I \) (i.e., \( c_k <_s j \)). That is, a dirty-write if \( I = \text{read uncommitted} \); a dirty-write or dirty-read if \( I = \text{read committed} \). The proof is by case distinction:

- If \( c_k \neq B \) and \( d_j \neq B \), then the proof is straightforward. Indeed, the relative order between \( c_k \) and \( C_k <_s d_j \). Therefore, either \( c_k \) and \( d_j \) do not witness a forbidden phenomenon in \( s' \) or the phenomenon is already present in \( s \). Both contradict with our assumptions.

- If \( c_k \in B \), then \( T_k = T_l \) and \( c_k <_s a \). By Condition (1), \( d_j < c_k \) or \( C_j <_s a \). Note that, since \( s' \) is constructed from \( s \) by moving operations in \( B \) to the right, \( c_k <_s d_j \) implies \( c_k <_s d_j \). We conclude that \( d_j <_s C_j <_s a \), and hence \( d_j <_s C_j \). Again leading to a contradiction. We conclude that \( d_j <_s C_j < a \). But then \( c_k <_s C_k <_s d_j \), contradicting our assumption that \( c_k \) and \( d_j \) witness a forbidden phenomenon.

We conclude that \( s' \) is indeed allowed under \( I \).

2. The proof is analogous to the proof for Condition (1). \( c_k \) and \( d_j \) be again two arbitrary conflicting operations with \( c_k <_s d_j \) that we assume witness a forbidden phenomenon for isolation level \( I \) if \( c_k \neq B \) and \( d_j \neq B \), the proof argument is the same as in the proof for Property (1). The other two cases are as follows:

- If \( c_k \in B \), then \( T_k = T_l \) and \( c_k <_s a \). By Condition (3), \( d_j < c_k \) or \( a <_s d_j \). Analogous to the proof for Condition (1), the former cannot happen, and hence \( c_k <_s a <_s d_j \). Implies that the relative order between \( c_k \) and \( C_k \) is identical in \( s \) and \( s' \), again leading to a contradiction.

- If \( d_j \in B \), then \( T_j = T_l \) and \( d_j <_s a \). By Condition (3), \( c_k <_s d_j \) or \( a <_s c_k \). The former case is analogous to the proof for Condition (1), implying that the relative order between \( c_k \) and \( C_k \) is identical in \( s \) and \( s' \). The latter case cannot occur, as \( d_j <_s a <_s c_k \).

**Lemma 32.** Let \( T' \) be a set containing precisely two transactions. If \( T' \) is not robust against isolation level \( I = \text{read committed} \), then there is a multi-split schedule \( s \) for \( T' \) that is allowed under \( I = \text{read committed} \).

**Proof.** Let \( s \) be a schedule for \( T' \) that is allowed under \( I = \text{read committed} \) and contains a cycle. We call the transaction whose commit occurs first in \( s \) transaction \( T_1 \), and the other transaction \( T_2 \). Let \( c \) be the first operation from \( T_2 \) that conflicts with an operation \( d \) from \( T_1 \) such that \( c <_s d \). (Notice that \( c \) and \( d \) exist, due to existence of a cycle \( C \) in \( CG(s) \).) Next, we distinguish two cases:

1. **Case:** There is an operation \( a \) from \( T_1 \) that occurs before \( c \) in \( s \) and conflicts with an operation \( b \) from \( T_2 \). Let \( a \) be the last such operation in \( s \). Let \( s' \) be the schedule obtained from \( s \) by moving all operations from \( T_2 \) occurring after \( c \) to the chunk with \( C_2 \); all operations from \( T_2 \) occurring before \( c \) to the chunk with \( C_1 \). We conclude the case by observing that \( s' \) is indeed a multi-split schedule for \( T' \) based on cycle \( (T_1, a, b, T_2), (T_2, c, d, T_1) \) and function \( e \) with \( e(T_1) := a \) and \( e(T_2) := c \)
From a schedule \( s \) we either have that \( c \in C \) or \( c \notin C \). The second step of the construction satisfies Lemma 31(1) by the assumption of the case.

Recall that there is an edge \((T_1, a, b, T_2)\) in \( C \), for some operations \( a \) from \( T_1 \) and \( b \) from \( T_2 \) with \( a < s, b \). By assumption of the case, we have \( c < s, a \) thus \( a < s, b \) (by construction of \( s' \)).

Now it is straightforward to see that \( s' \) is a multi-split schedule for \( T \) based on the cycle \((T_2, c, d, T_1)\) and \((T_1, a, b, T_2)\) and function \( e \) with \( e(T_2) := c \) and \( e(T_1) := C_1 \).

**Theorem 23.** Let \( T \) be a set of transactions. The following are equivalent:

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

**Proof.** (Continued).

For convenience of notation, we refer in each phase by \( s' \) to the new version of \( s \).

**Phase 1:** From a schedule \( s \) for \( U \) under read committed with a cycle \( C \) in its conflict graph and with Property (i) we construct a schedule \( s' \) for \( U \) under read committed with cycle \( C' \in CG(s) \) and Properties (i-ii). For the construction, we iterate over the transactions in \( U \) in the opposite order as defined by \( C \), starting from the transaction whose commit occurs last in \( s \). For each visited transaction, we verify that it does not contradict Property (ii). If it does, then we rewrite \( s \) to a new schedule \( s' \) in which the property is made true for \( T_i \) and remains true for all earlier visited transactions. We continue the iterative process on the new schedule \( s' \) until Property (ii) is true.

The above procedure terminates as we never split chunks from other transactions than the selected one. Hence, the only possibly side effect on a transaction with Property (ii) in \( s \) is that its two chunks may become a single chunk in \( s' \).

Notice that our picking order has the following implications: The first transaction \( T_i \) that we pick has property \( C_{i+1} < s, C_i \), with \( T_i+1 \) the transaction following \( T_i \) in \( C \). Indeed, we start with the transaction that commits last in \( s \). For every next transaction \( T_i \), we can assume that Property (ii) is already true for \( T_{i+1} \).

For the rewriting step, we distinguish three cases:

1. \( C_{i+1} < s, C_i \) Let \( b \) be the first operation of \( T_i \) in \( s \) that conflicts with an operation from \( T_{i+1} \). Then let \( s' \) be the schedule obtained by (I) removing in \( s \) all operations in \( \text{prefix}_b(T_i) \) except \( b \) and inserting them in front of \( b \); (II) removing all operations in \( \text{postfix}_b(T_i) \) except \( C_1 \) and inserting them in front of \( C_1 \).

   The resulting schedule \( s' \) is allowed under read committed, because both steps (I) and (II) satisfy Lemma 31(1). Indeed, for (I) it follows from the choice of \( b \) that all operations \( c \) conflicting with an operation \( d \) in \( \text{prefix}_b(T_i) \) are from \( T_{i-1} \) and due to Property (i) thus occur before \( d \) in \( s \). For (II), if an operation \( c \) conflicts with an operation \( d \) in \( \text{prefix}_b(T_i) \) with \( c < s, d \), then the same argument applies. Otherwise, if \( d < s, c \) it follows from Condition (1) on \( s \) that \( c \) is from \( T_{i+1} \) and thus from the condition of the case that \( C_{i+1} < s, C_i \).

   Replacing the edge between \( T_i \) and \( T_{i+1} \) in \( C \) by \((T_i, b, c, T_{i+1})\), with \( c \) an operation from \( T_{i+1} \) that \( b \) conflicts with, results in a cycle that is in \( CG(s') \). Since \( C' \) mentions the same transactions as \( C \), Property (1) straightforwardly transfers from \( s \) and \( C \) to \( s' \) and \( C' \). Notice also that \( b \) (which is conflicting by assumption) is the last operation of the first chunk of \( T_i \) in \( s' \), thus \( s' \) has Property (ii) for transaction \( T_i \).

2. \( C_1 < s, C_{i+1} \) and there is an operation \( b \) in \( T_i \) that conflicts with an operation \( e \) from \( T_{i+1} \) with \( b < s, e < s, C_1 \) Let \( b \) denote the last operation in \( s \) with this property.

   Let \( s' \) be the schedule obtained by (I) removing in \( s \) all operations from \( \text{prefix}_b(T_i) \) except \( b \) and inserting them in front of \( b \); and (II) removing all operations in \( \text{postfix}_b(T_i) \) except \( C_1 \) and inserting them in front of \( C_1 \).

   To see that \( s' \) is allowed under read committed, we argue that both steps (I) and (II) satisfy Lemma 31(3). For step (I), this follows from the observation that \( T_{i+1} \) already has Property (ii) due to the order in which we select transactions. Existence of \( b \) thus implies that the first chunk of \( T_{i+1} \) is located between \( b \) and \( C_1 \) in \( s \). From this, we infer that for every operation \( c \) that conflicts with an operation \( d \) in \( \text{prefix}_b(T_i) \), we either have that \( c < s, d \) or, if \( d < s, e \), that \( c \) is from \( T_{i+1} \), due to Property (i) on \( s \), and thus that \( b < s, c \). For step (II), if an operation \( c \) conflicts with an operation \( d \) in \( \text{postfix}_b(T_i) \) it follows from our choice of \( b \) that either \( c < s, d \) or \( C_1 < s, c \), hence Lemma 31(3) applies.

   Due to the above observations and the fact that \( b \) is the last operation of the first chunk of \( T_i \) in \( s' \), Property (ii) is indeed true for transaction \( T_i \) in \( s' \).

   Notice that the above analysis implies that cycle \( C \) remains a cycle in \( CG(s') \). Hence, let \( C' \) equal \( C \). Now it follows straightforwardly from Property (i) on \( s \) and \( C \) that Property (i) is true for \( s' \) and \( C' \).

3. \( C_1 < s, C_{i+1} \) and there is no operation \( b \) in \( T_i \) that conflicts with an operation \( e \) from \( T_{i+1} \) with \( b < s, e < s, C_1 \) Let \( s' \) be the schedule obtained by removing all operations from \( T_i \) except \( C_1 \) from \( s \) and inserting them in front of \( C_1 \).

   To see that \( s' \) is allowed under read committed, we observe that for every operation \( c \) that conflicts with an operation \( d \) in \( T_i \), the assumption of the case implies that either \( c < s, d \) or \( C_1 < s, c \). Hence, Lemma 31(3) applies.

   We conclude that Property (ii) is indeed true for \( T_i \) in \( s' \) since \( T_i \) now has only one chunk.
Here, again, we let \( C' \) equal \( C \), as it is indeed a cycle in \( CG(s') \) (inferred from the earlier analysis on \( s' \)). That Property (i) is true for \( s' \) and \( C' \) follows immediately from Property (i) on \( s \), the fact that \( s' \) is allowed under \textsc{read committed} and because \( C' \) mentions the same transactions as \( C \).

**Phase 2:** From a schedule \( s \) for \( \mathcal{U} \) under \textsc{read committed} with a cycle \( C \) in its conflict graph and with Properties (i-ii) we construct a schedule \( s' \) for \( \mathcal{U} \) under \textsc{read committed} with cycle \( C' \in CG(s) \) and Properties (i-iii).

Let \( s' \) be the schedule obtained by sorting in \( s \) all chunks between the first chunk of \( T_1 \) and last chunk of \( T_1 \) based on the order of the transaction that they are part of in \( CG(T_1) \) and by sorting all chunks occurring after \( CG(T_1) \) according to the same order. Let \( C' \in CG(s) \) and Properties (i-iii).

That \( s' \) is under \textsc{read committed} follows straightforwardly from the following observation: an operation in a chunk from some transaction \( T_i \) can only conflict with an operation in chunks from transactions \( T_{i-1} \) and \( T_{i+1} \). Due to minimality of \( C \) in \( CG(s) \) and the fact that \( \mathcal{U} \) (thus also \( C \)) has three or more transactions, it follows that for chunks from transactions \( T_i \) and \( T_{i+1} \), either they are already in the correct order, or they contain no conflicting operations and thus can be swapped safely. Since we do not swap chunks containing conflicts, cycle \( C' \) is indeed a cycle in \( CG(s') \).

Property (i) on \( s' \) and \( C' \) follows from the fact that Property (i) is true on \( s \) and \( C \) and because \( C' \) equals \( C \). Property (ii) follows from the fact that Property (ii) is true on \( s \) and because we don’t split chunks to obtain \( s' \).

**Phase 3:** From a schedule \( s \) for \( \mathcal{U} \) under \textsc{read committed} with a cycle \( C \) in its conflict graph and with Properties (i-iii) we construct a schedule \( s' \) for \( \mathcal{U} \) under \textsc{read committed} with cycle \( C' \in CG(s) \) and Properties (i-iv).

Let \( T_i \) be the last transaction (w.r.t. the order defined in \( CG(T_1) \)) without chunk between the first and last chunk of \( T_1 \) in \( s \). Notice that \( i < n \), because \( i = n \) would imply there is no edge from \( T_n \) to \( T_1 \) in \( IG(s) \), which contradicts with Property (iii) on \( s \) and the assumption that \( C \) contains all transactions from \( \mathcal{U} \). For the same reason, transaction \( T_{i+1} \) must have two chunks in \( s \): one before \( C_1 \) and one after \( C_1 \). Indeed, if \( T_{i+1} \) is closed, there can be no edge from \( T_i \) to \( T_{i+1} \) in \( IG(s) \). We will denote the last operation occurring in the first chunk of \( T_{i+1} \) by a.

Let \( s' \) be the schedule obtained by moving all chunks occurring before \( T_{i+1} \) in \( s \) to their chunk after a or inserting on the right place after \( C_1 \) w.r.t. the ordered defined by \( CG(T_1) \) (if the transaction has only one chunk in \( s \)). Let \( C' \) equal \( C \).

That schedule \( s' \) is allowed under \textsc{read committed} follows from Lemma 31; particularly the fact that Lemma 31(3) applies to each individual swap. Property (i) follows from the assumption that Property (i) is true on \( s \) and \( C \) and by construction of \( C' \) (which equals \( C \)). Property (ii) follows from the assumption that Property (ii) is true on \( s \) and because we don’t split chunks to obtain \( s' \). Property (iii) and (iv) follow directly from the construction, taking \( T_{i+1} \) as \( T_1 \). Indeed, we do not split chunks and all repositionings are w.r.t. the order of transactions in \( C \). By choice of \( T_i \) all transactions occurring between \( T_{i-1} \) and \( T_1 \) in \( C \) already had a chunk between the first chunk of \( T_{i+1} \) and the last chunk of \( T_1 \) (and possibly a second chunk occurring after the second chunk of \( T_{i+1} \)). Transactions \( T_1 \) till \( T_i \) either already appear closed between the first and last chunk of \( T_{i+1} \) or are closed and put on the right position by the construction.

**Phase 4:** From a schedule \( s \) for \( \mathcal{U} \) under \textsc{read committed} with a cycle \( C \) in its conflict graph and with Properties (i-iv) we construct a schedule \( s' \) for \( \mathcal{U} \) under \textsc{read committed} with cycle \( C' \in CG(s) \) and Properties (i-v).

Let \( s' \) be the schedule obtained from \( s \) by iteratively picking a transaction \( T_i \) having two chunks in \( s \), with \( i \neq 1 \), and with \( T_{i-1} \) having only one chunk, then removing the second chunk of \( T_i \) and inserting it immediately after its first chunk.

This procedure clearly leads to a schedule with Property (v). The resulting schedule \( s' \) is also allowed under \textsc{read committed}. Indeed, suppose towards a contradiction that a pair of conflicting operations \( c \) and \( d \) exist with \( c <_s d \) witnessing a forbidden phenomenon for isolation level \( I \). Then either \( c \) or \( d \) must be from \( T_1 \) (as otherwise the phenomenon already occurred in \( s \)). If \( c <_s d \) with \( c \) from \( T_1 \), then \( d \) must be from \( T_{i+1} \) (due to Property (i) on \( s \) and \( C \)) and it follows from the construction that \( c <_{s'} C_1 <_{s'} d \), which contradicts with our assumption that \( c \) and \( d \) witness a forbidden phenomenon. Similarly, if \( c <_s d \) with \( d \) from \( T_1 \), then \( c \) must be from \( T_{i-1} \) (again due to Property (i)), which implies \( c <_{s'} C_{i-1} <_{s'} d \).

Properties (ii-iv) transfer from \( s \) to \( s' \), because we do not split chunks and because we do not remove chunks located between the first and second chunk of \( T_1 \).