Using Read Promotion and Mixed Isolation Levels for Performant Yet Serializable Execution of Transaction Programs

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

Alan Fekete
University of Sydney
alan.fekete@sydney.edu.au

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

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

Stijn Vansummeren
UHasselt, Data Science Institute
stijn.vansummeren@uhasselt.be

ABSTRACT

We propose a theory that can determine the lowest isolation level that can be allocated to each transaction program in an application in a mixed-isolation-level setting, to guarantee that all executions will be serializable and thus preserve all integrity constraints, even those that are not explicitly declared. This extends prior work applied to completely known transactions, to deal with the realistic situation where transactions are generated by running programs with parameters that are not known in advance. Using our theory, we propose an optimization method that allows for high throughput while ensuring that all executions are serializable. Our method is based on searching for application code modifications that are semantics-preserving while improving the isolation level allocation. We illustrate our approach to the SmallBank benchmark.

PVLDB Reference Format:
Brecht Vandevoort, Alan Fekete, Bas Ketsman, Frank Neven, and Stijn Vansummeren. Using Read Promotion and Mixed Isolation Levels for Performant Yet Serializable Execution of Transaction Programs. PVLDB, 18(9): 2846 - 2858, 2025.
doi:10.14778/3746405.3746412

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://tinyurl.com/repmila.

1 INTRODUCTION

Transaction management is a core capability for database management systems. While research continues to find ways to improve performance, especially utilising novel hardware [9, 12, 18, 19, 24–26, 30–33, 36], the bulk of application software runs on popular platforms whose concurrency control mechanisms are decades old and are known to suffer from bottlenecks that make serializable transactions perform poorly under contention [39, 45]. Under the narrative that many applications have domain-specific reasons why they do not need to be perfectly serializable, these platforms offer the application programmer a choice of isolation levels. As such, the programmer can select a weaker isolation level, such as the platform's default READ COMMITTED level [6], to improve performance when apt. However, there is not yet a well-grounded way for the programmer to decide when a decision to accept non-serializable isolation is justified.

Robustness of transactions to guarantee serializability. Recent theoretical studies have provided algorithms that can analyze the collection of transactions occurring in an application, and determine which isolation level to use for each of them in a mixed-isolation-level setting, while still guaranteeing the robustness of the application [43]. That is, every possible execution of the application's transactions will in fact be serializable, even though several of the transactions do not run with serializable isolation level. This task, dubbed the *allocation problem* [20], requires some characterisation of the concurrency control mechanism used by the platform. In particular, the conclusions on a platform using traditional shared and exclusive locks operating on single-version data will differ from the conclusions for multiversion systems that allow reading versions that have been overwritten (and therefore do not block reads) [20, 29]. The current theory for solving the allocation problem, however, assumes that all the transactions are completely known at allocation time, including all the items that will be read and written. This is not realistic: in practice applications execute programs with parameters that are provided at run-time, after allocation. For example, a student enrollment system will have a program that enrolls a student to a particular course. The concrete student id and the course code are only provided by the end-user when the program is executed.

Robustness of transaction templates. The key technical advance of this paper, is to provide an algorithm that can determine which isolation level can be allocated to each of a set of templates, where a template is an abstraction that aims to capture a transaction whose read and write set is determined based on some variables. The algorithm returns an allocation that is guaranteed to be robust on a platform such as PostgreSQL which offers multiversion concurrency control mechanisms. That is, every execution of any set of transactions that instantiate the templates with arbitrary values, will be serializable. Our algorithm, therefore, supports any number of instantiations per template while maintaining polynomial time in the template size. Unlike earlier work [43], which assumes a fixed set of concrete transactions and scales with their size, our execution time is independent of the number of concrete instantiations.

The template abstraction we use also has some restrictions. It assumes a fixed set of read-only attributes that cannot be modified and are used to select tuples for updates. A common example of

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746412

such attributes are primary keys. This assumption prevents predicate reads that could cause non-serializable executions not covered by our analysis. While it excludes workloads like TPC-C which involve predicate reads, in many cases this is not a major limitation as the inability to update primary keys is not a significant limitation, because keys are typically assigned once and remain unchanged due to regulatory or data integrity requirements. Furthermore, as the template abstraction loses some of the constraints in the transaction code, our analysis is conservative but safe: we may miss a desirable allocation which is robust, but the allocation found by our algorithm does indeed ensure that all executions are serializable. However, within the restrictions of the template abstraction, we can prove optimality of the allocation we identify, in the sense that we find the allocation that gives each template the lowest isolation level possible (i.e., prioritizing Read Committed over Snapshot Isolation, and Snapshot Isolation over the serializable level), such that any code that fits the template will be robust. While our template abstraction omits branches and loops for simplicity, we could handle them by treating each execution path as a separate template unfolding. Branches and bounded loops unfold easily, and for unbounded loops, it is sufficient to show that any non-robust behavior has a counterexample with loops unfolded only a fixed number of times (cf. [42]). The current paper does not consider depedencies like foreign keys. It is known that, in general, these dependencies can render the robustness property undecidable [41].

Optimization via read promotion and mixed-isolation level allocation. Our allocation algorithm can be used to find ways to deliver an application with the correctness guarantees of serializable execution (so all state invariants are preserved, including those which are not explicitly declared in the database), and yet better performance than when each transaction is executed at the serializable level. We propose to consider a space of different ways to modify the application code (while not changing its semantics), by "promoting" some read operations [21] so they are treated as identity updates, and thus set exclusive locks. For each of these different promotion choices, we use our allocation algorithm to determine the lowest isolation level. For each promotion choice and determined robust allocation we can then empirically measure the performance obtained; the promotion choice with best performance of its robust allocation, is how the application should be coded. We refer to this optimization approach as *read promotion and mixed isolation level allocation (RePMILA)*.

We illustrate *RePMILA* for the well-known benchmark SmallBank [3], and explore the performance obtained on PostgreSQL under a range of workload parameters. We find that some promotion choices have a robust allocation whose throughput is competitive with the throughput of the unmodified application running with all transactions using Read Committed. Unlike the latter, however, the robust allocation still guarantees serializable execution. Furthermore, the throughput under robust allocation can be twice the throughput achieved by running all transactions under the platform's serializable isolation level.

Contributions. The contributions of this paper are varied, with both theory and empirical results, and we propose guidance for practitioners. As theory, we offer a proof technique that allows demonstrating robustness for an allocation of multi-version isolation levels for transaction templates, and a polynomial-time algorithm that generates a unique lowest robust allocation. We give an experimental demonstration for the SmallBank application mix, that some promotion choices allow performance of a robust allocation, close to that of the default non-robust allocation (and much better than the naive use of Serializable isolation for all transactions). We consider that in this context *RePMILA* can be useful for practitioners as they seek performance while guaranteeing serializable execution.

Organization. The remainder of the paper is structured as follows. In Section 2 we illustrate *RePMILA* applied to SmallBank, demonstrating how each program is abstracted as a template, how the allocation algorithm determines the allocation of isolation levels for the templates, how the various promotion choices are generated, and what allocation is generated for each promotion choice. In Section 3 we show the measured performance for the different promotion choices, and compare with baselines where all programs are run with Serializable isolation level, and also where all are run at default Read Committed isolation (and thus undeclared data invariants can be violated). Section 4 presents the theory and includes the details of the allocation algorithm. We discuss related work in Section 5. Finally, Section 6 looks at implications and limitations of this work, and identifies some further research directions.

A full version of this paper is available as [38], containing all proofs, additional examples, further intuition regarding formalization, as well as the SQL code for the SmallBank benchmark.

2 READ PROMOTION AND MIXED ISOLATION LEVEL ALLOCATION (*REPMILA*)

To explain our approach, we work through the way it is applied to the well-known SmallBank benchmark application [3]. While it is not a real-world example, it has enough features to illustrate how we can identify a high-performing, robust allocation, and yet it is simple enough to fit in a conference paper.

SmallBank benchmark. The SmallBank [3] schema consists of the tables Account(Name, CustomerID), Savings(CustomerID, Balance), and Checking(CustomerID, Balance) (key attributes are underlined). The Account table associates customer names with IDs. The other tables contain the balance (numeric value) of the savings and checking accounts of customers identified by their ID. The application code interacts with the database via transaction programs:

- Balance(N) returns the total balance (savings and checking) for a customer with name N.
- DepositChecking(N,V) makes a deposit of amount V in the checking account of the customer with name N (see Figure 1).
- TransactSavings(N,V) makes a deposit or withdrawal V on the savings account of the customer with name N.
- Amalgamate(N_1,N_2) transfers all the funds from customer N_1 to customer N_2.
- Finally, WriteCheck(N,V) writes a check V against the account of the customer with name N, penalizing if overdrawing.

Transaction templates. We abstract transaction programs via transactions templates as illustrated in Figure 2. A transaction template consists of a sequence of read (R), write (W), and update (U) operations. Each operation accesses exactly one tuple. For instance, R[X : Account{N, C}] indicates that a read operation is performed

```
DepositChecking(N,V):
  SELECT CustomerId INTO :X FROM Account WHERE Name=:N;
  UPDATE Checking SET Balance = Balance+:V
    WHERE CustomerId=:X;
  COMMIT;
```

Figure 1: SQL code for DepositChecking.

Balance:

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

DepositChecking:

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

TransactSavings:

$R[X : \text{Account}\{N, C\}]$
$U[Y : \text{Savings}\{C, B\}\{B\}]$

Amalgamate:

$R[X_1 : \text{Account}\{N, C\}]$
$R[X_2 : \text{Account}\{N, C\}]$
$U[Y_1 : \text{Savings}\{C, B\}\{B\}]$
$U[Z_1 : \text{Checking}\{C, B\}\{B\}]$
$U[Z_2 : \text{Checking}\{C, B\}\{B\}]$

WriteCheck:

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

Figure 2: Transaction templates for SmallBank.

to a tuple X in relation Account on the attributes Name and CustomerID. We abbreviate the names of attributes by their first letter to save space. The set $\{N, C\}$ is the read set of the read operation. Similarly, W and U refer to write and update operations to tuples of a specific relation. Write operations have an associated write set while update operations contain a read set followed by a write set: e.g., $U[Z : \text{DepositChecking}\{C, B\}\{B\}]$ first reads the CustomerID and Balance of tuple Z and then writes to the attribute Balance. A U-operation is an atomic update that first reads the tuple and then writes to it. Templates serve as abstractions of transaction programs and represent an infinite number of possible workloads. For instance, let x, y, z (and their primed versions) be concrete database objects serving as interpretations of the variables X, Y, and Z. Then, disregarding attribute sets, $\{R[x]R[y]R[z]U[z], R[x']R[y']R[z']U[z']$,$R[x]U[z]\}$ is a workload consistent with the SmallBank templates as it contains two instantiations of WriteCheck and one instantiation of DepositChecking. We remark that $\{R[x]R[y]R[z]U[z']\}$ with $z \neq z'$ is not a valid workload as the two final operations in WriteCheck should be on the same object as required by the formalization. Typed variables effectively enforce domain constraints as we assume that variables that range over tuples of different relations can never be instantiated by the same value. For instance, in the transaction template for DepositChecking in Figure 2, X and Z can not be interpreted to be the same object.

Templates do not capture all constraints in the original programs, and may therefore overapproximate the transactions that can occur when the actual programs are executed. For instance, the workload $\{R[t]U[q], R[t]U[q']\}$ is consistent with the SmallBank templates (two instantiations of DepositChecking), but cannot occur in practice under the assumption that a customer can only have one checking account.

Lowest robust allocation. We are interested in determining the lowest isolation level for each separate template such that every execution that arises under the assigned levels will in fact be serializable. We refer to such as an allocation as *robust*. We consider the isolation levels of PostgreSQL: Read Committed (RC), Snapshot Isolation (SI), and Serializable Snapshot Isolation (SSI) where we rank them from *lower* to *higher* as RC < SI < SSI, under the assumption that throughput increases when isolation levels are lowered. The allocation algorithm that we describe in Section 4 finds that the allocation that maps DepositChecking to RC and all other templates to SSI is robust, and is in fact optimal in the sense that no isolation level can be lowered without breaking robustness.

	Promotion choices	Lowest robust allocation Bal	DC	TS	Am	WC	
(1)	no promotion	SSI	RC	SSI	SSI	SSI	(A)
(2)	WC: C						
(3)	Bal: S	SSI	SSI	SSI	SSI	SSI	(B)
(4)	Bal: S, WC: C						
(5)	Bal: C	SI	RC	RC	RC	SI	(C)
(6)	WC: S						
(7)	Bal: C, WC: S						
(8)	Bal: C, WC: C						
(9)	WC: S,C	SI	RC	RC	RC	RC	(D)
(10)	Bal: C, WC: S,C						
(11)	Bal: S,C	RC	RC	RC	RC	SI	(E)
(12)	Bal: S, WC: S						
(13)	Bal: S,C, WC: S						
(14)	Bal: S,C, WC: C						
(15)	Bal: S, WC: S,C	RC	RC	RC	RC	RC	(F)
(16)	Bal: S,C, WC: S,C						

Table 1: Lowest robust allocations for each promotion choices over the SmallBank benchmark, grouped by allocation. Promotion choices and allocations are labeled for easy reference.

Read promotion choices. None of the considered isolation levels allow dirty writes. This forces a transaction that wants to overwrite a change made by an earlier transaction, to wait until the earlier transaction either commits or aborts. Therefore, if we promote a read operation to an update (that is, a read operation that writes back the observed value), the semantics of the transaction remains unaffected but the lowest robust allocation might differ. Ignoring the read operations over the read-only Account table, the SmallBank benchmark contains 4 read operations over the Savings and Checking relations that are candidates for promotion, resulting in 16 possible promotion choices. For each promotion choice, we run our algorithm to detect the lowest robust allocation. The resulting allocations are summarized in Table 1. We denote each promotion choice by the read operations that are promoted. For example, *'Bal: S, WC: C'* promotes the read operation over the Savings relation in the Balance program, and the read operation over the Checking relation in the WriteCheck program. For convenience, the promotion choices are grouped by their lowest robust allocation. Notice that without promotion, as mentioned previously, only one out of the five programs (nl., DepositChecking) can be executed under an isolation level lower that SSI without giving up serializability. Furthermore, introducing a few promoted reads quickly leads to robust allocations where almost all programs are being executed under RC. However, the best promotion choice is not necessarily the one that allows the most programs to run under RC, for the simple reason that the newly introduced writes could have a negative impact on the overall performance. Throughput experiments are therefore needed to determine the best promotion choices, as we discuss in the next section.

3 EVALUATING *REPMILA* OVER SMALLBANK

Here we show experimentally the performance achieved by our approach when applied to the SmallBank benchmark.

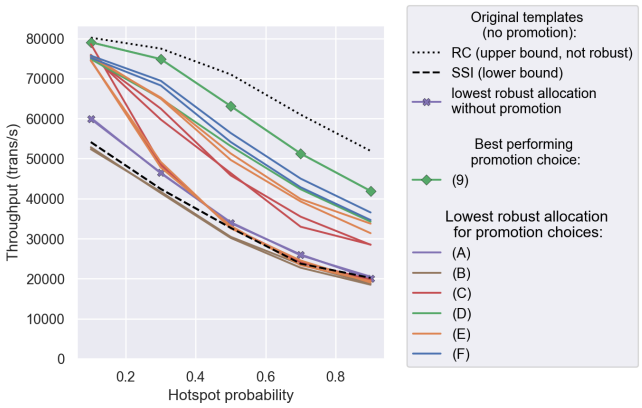

Figure 3: Throughput for the promotion choices mentioned in Table 1 for various hotspot probabilities.

Experimental setup. All experiments use PostgreSQL 16.2 as the database engine, running on a single machine with two 18-core Xeon Gold 6240 CPUs (2.6 GHz), 192 GB RAM and 200GB local SSD storage. A separate machine is used to query the database, with 100 concurrent clients executing randomly sampled programs from the SmallBank benchmark. If the database aborts a transaction, the client immediately retries the same program with the same parameters until it successfully commits. Each experiment runs for 60 seconds and is repeated 5 times to measure the average throughput. The database is populated with 18000 accounts. A small subset of 20 accounts act as a hotspot that will be accessed more frequently. The level of contention is varied by changing the probability of sampling a hotspot account during execution. Within the hotspot, uniform sampling is used to select an account. Unless otherwise specified, each of the five SmallBank programs has an equal probability of being sampled by the clients. Throughput is indicated in number of transactions per second. We implemented the allocation algorithm described in Section 4 in Python. Verifying whether an allocation is robust takes only a few seconds, whereas computing the lowest allocation for a specific promotion choice requires less than a minute. This runtime is acceptable since the computation is performed only once and can be executed offline.

3.1 *RePMILA* improves performance

Figure 3 displays the throughput for the different promotion choices and associated lowest robust allocations mentioned in Table 1 for increasing levels of contention. Promotion choices that result in the same lowest robust allocation are depicted in the same color and are not individually labeled to avoid adding complexity to the figure. While not all lines are easily discernible, we do see that the throughput for almost all promotion choices is higher than executing the unmodified templates under SSI. The exception is the bottom line indicating that these choices perform worse than executing the original unmodified programs under SSI. This is due to the introduction of additional writes, which do not provide allocation benefits as all templates still require the use of SSI.

None of the promotions can match the throughput levels that can be reached by the nonrobust allocation that executes all unmodified templates at RC. Nevertheless, the most performant promotion choice *'WC: S,C'* is a near match. Here, no reads are promoted in

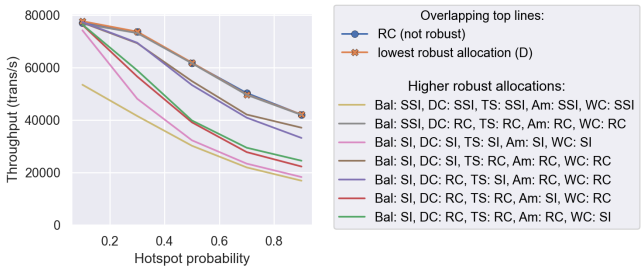

Figure 4: Throughput for promotion choice (9) 'WC: S,C' under its lowest and higher allocations.

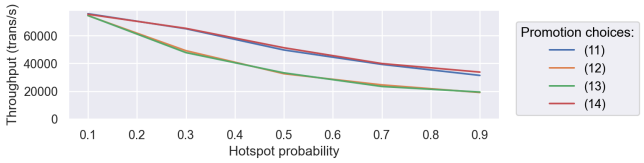

Figure 5: Throughput for different promotion choices sharing the same lowest robust allocation (E), i.e., the one that maps WriteCheck to SI and all the others to RC.

Balance and the lowest robust allocation assigns SI to Balance and RC to the others. In our experiments we observe that in this case there are no aborts due to concurrent writes.

We point out that the lowest robust allocation for the unmodified templates allows little to no improvement compared to running everything under SSI, especially when contention increases. We thus conclude that read promotions can increase throughput and that considering various promotion choices is helpful. In general, read promotions that allow to allocate RC tend to outperform those requiring SI or SSI. However, there are some notable exceptions: for example, *'WC: S,C'* still requires SI for Balance, but it outperforms all promotion choices that only require RC. Similarly, there are promotion choices (which are not discernible in the figure as individual lines are not labeled) where the lowest allocation assigns both Balance and WriteCheck to SI, yet these choices still outperform specific promotion choices that allocate only WriteCheck to SI, and RC to all others.

3.2 Lowest allocation outperforms higher ones

We defined a robust allocation as *lowest* when no isolation level can be reduced without compromising robustness, following the ordering RC < SI < SSI. As shown in Section 4, there always exists a *unique* lowest allocation with respect to this order. We verify that these unique lowest allocation consistently outperforms higher allocations, allowing us to focus solely on them. To this end, we examine the most performant promotion choice, *'WC: S,C'*, and compare its lowest robust allocation—where Balance is assigned SI and all other templates receive RC—against all alternative allocations that raise the isolation level for at least one of the five programs. Figure 4 demonstrates that the lowest robust allocation indeed outperforms all higher allocations. Additionally, we observe that its throughput matches that of the nonrobust allocation for *'WC: S,C'*, where all templates run under RC.

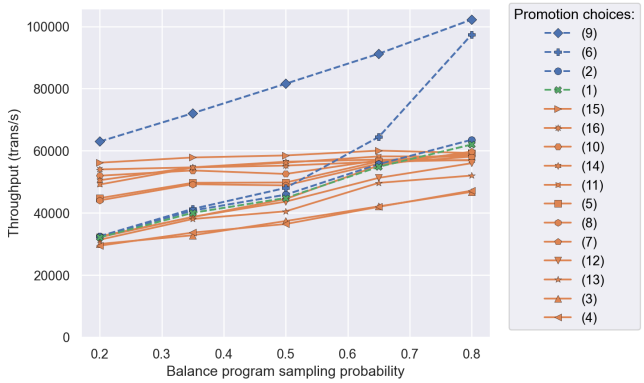

Figure 6: Throughput for different promotion choices varying the probability of executing Balance for a fixed hotspot probability of 0.5. Promotion choices that do not promote a read in Balance are in blue. Those that do are in orange.

3.3 Promotion choices within the same lowest robust allocation impact performance

As illustrated by Table 1, distinct promotion choices may lead to the same lowest robust allocation. For example, the four promotion choices *'Bal: S,C'*, *'Bal: S, WC: S'*, *'Bal: S,C, WC: S'* and *'Bal: S,C, WC: C'* all result in the same lowest robust allocation that maps WriteCheck to SI and all the others to RC. Since these promotion choices promote different reads, an identical lowest allocation does not necessarily lead to identical performance even under lower levels of contention, as is shown in Figure 5. There, we see that *'Bal: S,C'* and *'Bal: S,C, WC: C'* greatly outperform the others. This implies, in particular, that an experimental exploration of promotion choices cannot be limited to just a single promotion choice per unique robust allocation. A closer inspection reveals that the read promotions of the top performer, *'Bal: S,C, WC: C'*, form a strict *superset* of those of the next-best contender, *'Bal: S,C'*. In this case, the slight performance gain is obtained by taking an earlier lock due to the newly promoted read. Indeed, the promoted read in WriteCheck is over a tuple written to by a later write in the same transaction. Since the lock is taken earlier and since WriteCheck uses SI, potentially concurrent writes are detected earlier, thereby avoiding the amount of work wasted before an abort.

3.4 Impact of template frequency

We next illustrate that finding the best performing promotion choices is influenced as well by the template frequency. The previous experiments assumed uniform sampling over the five possible SmallBank programs. Since only Balance is read-only (assuming no promoted reads), this corresponds to a workload where only 20% of the transactions is read-only. To further explore the impact of promoting reads in read-only transactions, Figure 6 shows the throughput for different promotion choices when the probability of executing Balance is varied between 0.2 and 0.8, assuming a fixed hotspot probability of 0.5. The remaining four templates are sampled with equal probability. Overall, we see that promotion choices that do not promote a read in Balance start to outperform those that do when the probability of executing Balance increases. This observation is most pronounced for the promotion choice *'WC: S'*, which is among the worst performers when the probability of executing Balance is low, but becomes the second best performer when the probability is high, vastly outperforming all other promotion choices. Closer inspection reveals that in the latter case even the original programs (i.e., 'no promotion') outperform all promotion choices that promote a read in Balance.

3.5 Discussion

The SmallBank exploration in this section validates *RePMILA* as an effective optimization method, demonstrating that combining read promotion with the lowest robust allocation can double throughput compared to executing all transactions under SSI. Furthermore, it achieves throughput comparable to the unsafe yet default RC isolation level used by some platforms, all while preserving safety.

Since our performance gains stem from a tradeoff between reduced concurrency and fewer aborts inherent to the studied isolation levels, we expect our results to generalize to other systems supporting these levels. While the analysis in Table 1 should hold, the optimal promotion choice may vary (cf. Alomari et al.[3] for cross-system comparisons).

In theory, the number of promotion choices grows exponentially with the total number of read operations in templates. However, the number of required throughput experiments can be significantly reduced by the following guidelines: *(i)* ignoring reads from read-only tables (e.g., Accounts in SmallBank); *(ii)* avoiding promotion in read-only templates, particularly when they frequently appear in workloads (e.g., Balance); and, *(iii)* when multiple promotion choices yield the same lowest allocation, prioritizing those that promote reads occurring earlier in a template.

4 ALLOCATION ALGORITHM

We start by introducing all necessary terminology and borrow notation from [39, 43] along with some examples that we adapt and modify to suit our context. In particular, we discuss transactions, schedules, conflict-serializability, and isolation levels in Section 4.1–4.3. In Section 4.4–4.5 we introduce templates and their robustness. Finally, we present an algorithm for template robustness in Section 4.6 and an algorithm for finding the lowest robust allocation in Section 4.7.

4.1 Transactions and Schedules

A *relational schema* is a set Rels of relation names. For each $R \in \text{Rels}$, $\text{Attr}(R)$ is the finite set of associated attribute names and we fix an infinite set $\mathbf{Obj}_R$ of abstract objects called tuples. We assume that $\mathbf{Obj}_R \cap \mathbf{Obj}_S = \emptyset$ for all $R, S \in \text{Rels}$ with $R \neq S$. We denote by $\mathbf{Obj}$ the set $\bigcup_{R \in \text{Rels}} \mathbf{Obj}_R$ of all possible tuples. We require that for every $\mathtt{t} \in \mathbf{Obj}$ there is a unique relation $R \in \text{Rels}$ such that $\mathtt{t} \in \mathbf{Obj}_R$. We then say that $\mathtt{t}$ is of *type* R and denote the latter by $\text{type}(\mathtt{t}) = R$. A *database* $\mathbf{D}$ over schema Rels assigns to every relation name $R \in \text{Rels}$ a finite set $R^{\mathbf{D}} \subset \mathbf{Obj}_R$.

For a tuple $\mathtt{t} \in \mathbf{Obj}$, we distinguish three operations $\mathtt{R[t]}$, $\mathtt{W[t]}$, and $\mathtt{U[t]}$ on $\mathtt{t}$, denoting that $\mathtt{t}$ is read, written, or updated, respectively. We say that the operation is *on* the tuple $\mathtt{t}$. Here, $\mathtt{U[t]}$ is an atomic update and should be viewed as an atomic sequence of a read of $\mathtt{t}$ followed by a write to $\mathtt{t}$. To differentiate between the cases where we want to refer to an actual operation ($\mathtt{R}$, $\mathtt{W}$, or $\mathtt{U}$)

or to operations with a specific property (read or write), we employ the following terminology. A *read operation* is an R[t] or a U[t], and a *write operation* is a W[t] or a U[t]. Furthermore, an R-operation is an R[t], a W-operation is a W[t], and a U-operation is a U[t]. We also assume a special *commit* operation denoted C. To every operation o on a tuple of type R, we associate the set of attributes $\text{ReadSet}(o) \subseteq \text{Attr}(R)$ and $\text{WriteSet}(o) \subseteq \text{Attr}(R)$ containing, respectively, the set of attributes that o reads from and writes to. When o is an R-operation then $\text{WriteSet}(o) = \emptyset$. Similarly, when o is a W-operation then $\text{ReadSet}(o) = \emptyset$.

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

When considering a set $\mathcal{T}$ of transactions, we assume that every transaction in the set has a unique id i and write T_i to make this id explicit. Similarly, to distinguish the operations of different transactions, we add this id as a subscript to the operation. That is, we write $\text{W}_i[\text{t}]$, $\text{R}_i[\text{t}]$, and $\text{U}_i[\text{t}]$ to denote a W[t], R[t], and U[t] occurring in transaction T_i; similarly C_i denotes the commit operation in transaction T_i. This convention is consistent with the literature (see, *e.g.* [7, 20]). To avoid ambiguity of notation, we assume that a transaction performs at most one write, one read, and one update per tuple. The latter is a common assumption (see, *e.g.* [20]). All our results carry over to the more general setting in which multiple writes and reads per tuple are allowed.

A *(multiversion) schedule* s over a set $\mathcal{T}$ of transactions is a tuple $(O_s, \leq_s, \ll_s, v_s)$ where

- O_s is the set containing all operations of transactions in $\mathcal{T}$ as well as a special operation op_0 conceptually writing the initial versions of all existing tuples,
- $\leq_s$ encodes the ordering of these operations,
- $\ll_s$ is a *version order* providing for each tuple t a total order over all write operations on t occurring in s, and,
- v_s is a *version function* mapping each read operation a in s to either op_0 or to a write operation different from a in s (recall that a write operation is either a W[x] or a U[x]).

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

Figure 7 depicts an example of a schedule. There, the read operations on t in T_1 and T_4 both read the initial version of t instead of the version written but not yet committed by T_2. Furthermore, the read operation $\text{R}_2[\text{v}]$ in T_2 reads the initial version of v instead of the version written by T_3, even though T_3 commits before $\text{R}_2[\text{v}]$.

We say that a schedule s is a *single version schedule* if $\ll_s$ is compatible with $\leq_s$ and every read operation always reads the last written version of the tuple. Formally, for each pair of write operations a and b on the same tuple, $a \ll_s b$ iff $a <_s b$, and for every read operation a there is no write operation c on the same tuple as a with $v_s(a) <_s c <_s a$. A single version schedule over a set of transactions $\mathcal{T}$ is *single version serial* if its transactions are not interleaved with operations from other transactions. That is, for every $a, b, c \in O_s$ with $a <_s b <_s c$ and $a, c \in T$ implies $b \in T$ for every $T \in \mathcal{T}$.

4.2 Conflict-Serializability

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

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

We also say that a_j and b_i are conflicting operations. Commit operations and the special operation op_0 never conflict with any other operation. When a_j and b_i are conflicting operations in $\mathcal{T}$, we say that a_j *depends on* b_i in a schedule s over $\mathcal{T}$, denoted $b_i \rightarrow_s a_j$ if:

- *(ww-dependency)* b_i is ww-conflicting with a_j and $b_i \ll_s a_j$; or,
- *(wr-dependency)* b_i is wr-conflicting with a_j and $b_i = v_s(a_j)$ or $b_i \ll_s v_s(a_j)$; or,
- *(rw-antidependency)* b_i is rw-conflicting with a_j and $v_s(b_i) \ll_s a_j$.

Intuitively, a ww-dependency from b_i to a_j implies that a_j writes a version of a tuple that is installed after the version written by b_i. A wr-dependency from b_i to a_j implies that b_i either writes the version observed by a_j, or it writes a version that is installed before the version observed by a_j. A rw-antidependency from b_i to a_j implies that b_i observes a version installed before the version written by a_j.

For example, the dependencies $\text{U}_2[\text{t}] \rightarrow \text{W}_4[\text{t}]$, $\text{U}_3[\text{v}] \rightarrow \text{R}_4[\text{v}]$ and $\text{R}_4[\text{t}] \rightarrow \text{U}_2[\text{t}]$ are respectively a ww-dependency, a wr-dependency and a rw-antidependency in schedule s presented in Figure 7.

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

Definition 4.1. A schedule s is *conflict-serializable* if it is conflict-equivalent to a single version serial schedule.

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

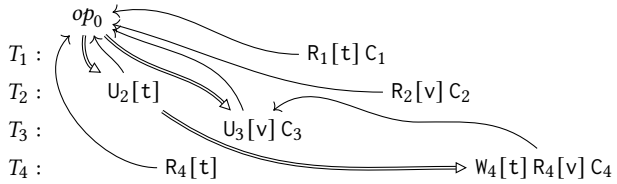

Figure 7: A schedule s with v_s (single lines) and $\ll_s$ (double lines) represented through arrows.

T_j with all possible choices of operations with $b_i \rightarrow_s a_j$. Henceforth, we refer to these quadruples simply as edges. Notice that edges cannot contain commit operations. A *cycle* Γ in $SeG(s)$ is a non-empty sequence of edges $(T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_n, b_n, a_1, T_1)$ in $SeG(s)$, in which every transaction is mentioned exactly twice.

Theorem 4.2 (implied by [2]). *A schedule s is conflict-serializable iff $SeG(s)$ is acyclic.*

Our formalisation of transactions and conflict serializability is based on [20], generalized to operations over attributes of tuples and extended with U-operations that combine R- and W-operations into one atomic operation. These definitions are closely related to the formalization presented by Adya et al. [2], but we assume a total rather than a partial order over the operations in a schedule.

4.3 Isolation Levels

Let $\mathcal{I}$ be a class of isolation levels. An *$\mathcal{I}$-allocation* $\mathcal{A}$ for a set of transactions $\mathcal{T}$ is a function mapping each transaction $T \in \mathcal{T}$ onto an isolation level $\mathcal{A}(T) \in \mathcal{I}$. When $\mathcal{I}$ is not important or clear from the context, we also say allocation rather than $\mathcal{I}$-allocation. In this paper, we consider the following isolation levels: read committed (RC), snapshot isolation (SI), and serializable snapshot isolation (SSI). So, in general, $\mathcal{I} = \{\text{RC}, \text{SI}, \text{SSI}\}$. Before we define what it means for a schedule to consist of transactions adhering to different isolation levels, we introduce some necessary terminology. Some of these notions are illustrated in Example 4.5 below.

Let s be a schedule for a set $\mathcal{T}$ of transactions. Two transactions $T_i, T_j \in \mathcal{T}$ are said to be *concurrent* in s when their execution overlaps. That is, if $\mathit{first}(T_i) <_s \mathtt{C}_j$ and $\mathit{first}(T_j) <_s \mathtt{C}_i$. We say that a write operation o_j on $\mathtt{t}$ in a transaction $T_j \in \mathcal{T}$ *respects the commit order of s* if the version of $\mathtt{t}$ written by T_j is installed after all versions of $\mathtt{t}$ installed by transactions committing before T_j commits, but before all versions of $\mathtt{t}$ installed by transactions committing after T_j commits. More formally, if for every write operation o_i on $\mathtt{t}$ in a transaction $T_i \in \mathcal{T}$ different from T_j we have $o_j \ll_s o_i$ iff $\mathtt{C}_j <_s \mathtt{C}_i$. We next define when a read operation $a \in T$ reads the last committed version relative to a specific operation. For RC this operation is a itself while for SI this operation is $\mathit{first}(T)$. Intuitively, these definitions enforce that read operations in transactions allowed under RC act as if they observe a snapshot taken right before the read operation itself, while under SI they observe a snapshot taken right before the first operation of the transaction. A read operation o_j on $\mathtt{t}$ in a transaction $T_j \in \mathcal{T}$ is *read-last-committed in s relative to an operation $a_j \in T_j$* (not necessarily different from o_j) if the following holds:

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

The first condition says that o_j either reads the initial version or a committed version, while the second condition states that o_j observes the most recently committed version of $\mathtt{t}$ (according to $\ll_s$). A transaction $T_j \in \mathcal{T}$ *exhibits a concurrent write in s* if there is another transaction $T_i \in \mathcal{T}$ and there are two write operations b_i and a_j in s on the same object with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j$ and $\mathit{first}(T_j) <_s \mathtt{C}_i$. That is, transaction T_j writes to an object that has been modified earlier by a concurrent transaction T_i.

A transaction $T_j \in \mathcal{T}$ *exhibits a dirty write in s* if there are two write operations b_i and a_j in s on the same object with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j <_s \mathtt{C}_i$. That is, transaction T_j writes to an object that has been modified earlier by T_i, but T_i has not yet issued a commit. Notice that by definition a transaction exhibiting a dirty write always exhibits a concurrent write. Transaction T_4 in Figure 7 exhibits a concurrent write, since it writes to $\mathtt{t}$, which has been modified earlier by a concurrent transaction T_2. However, T_4 does not exhibit a dirty write, since T_2 has already committed before T_4 writes to $\mathtt{t}$.

Definition 4.3. Let s be a schedule over a set of transactions $\mathcal{T}$. A transaction $T_i \in \mathcal{T}$ is *allowed under isolation level read committed (RC) in s* if:

- each write operation in T_i respects the commit order of s;
- each read operation $b_i \in T_i$ is read-last-committed in s relative to b_i; and
- T_i does not exhibit dirty writes in s.

A transaction $T_i \in \mathcal{T}$ is *allowed under isolation level snapshot isolation (SI) in s* if:

- each write operation in T_i respects the commit order of s;
- each read operation in T_i is read-last-committed in s relative to $\mathit{first}(T_i)$; and
- T_i does not exhibit concurrent writes in s.

We then say that the schedule s is allowed under RC (respectively, SI) if every transaction is allowed under RC (respectively, SI) in s. The latter definitions correspond to the ones in the literature (see, e.g., [20, 39]). We emphasize that our definition of RC is based on concrete implementations over multiversion databases, found in e.g. PostgreSQL, and should therefore not be confused with different interpretations of the term Read Committed, such as lock-based implementations [7] or more abstract specifications covering a wider range of concrete implementations (see, e.g., [2]). In particular, abstract specifications such as [2] do not require the read-last-committed property, thereby facilitating implementations in distributed settings, where read operations are allowed to observe outdated versions. When studying robustness, such a broad specification of RC is not desirable, since it allows for a wide range of schedules that are not conflict-serializable.

While RC and SI are defined on the granularity of a single transaction, SSI enforces a global condition on the schedule as a whole. For this, recall the concept of dangerous structures from [13]: three transactions $T_1, T_2, T_3 \in \mathcal{T}$ (where T_1 and T_3 are not necessarily different) form a *dangerous structure* $T_1 \rightarrow T_2 \rightarrow T_3$ in s if:

- there is a rw-antidependency from T_1 to T_2 and from T_2 to T_3 in s;

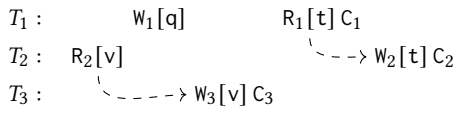

Figure 8: Example of a dangerous structure $T_1 \to T_2 \to T_3$ with the required rw-antidependencies represented through dashed arrows.

- T_1 and T_2 are concurrent in s;
- T_2 and T_3 are concurrent in s;
- $C_3 \leq_s C_1$ and $C_3 <_s C_2$; and
- if T_1 is read-only, then $C_3 <_s \mathit{first}(T_1)$.

Note that this definition of dangerous structures slightly extends upon the one in [13], where it is not required for T_3 to commit before T_1 and T_2. In the full version [14] of that paper, it is shown that, if all transactions are allowed under SI, such a structure can only lead to non-serializable schedules if T_3 commits first. Furthermore, Ports and Grittner [35] show that if T_1 is a read-only transaction, this structure can only lead to non-serializable behavior if T_3 commits before T_1 starts. Actual implementations of SSI (e.g., PostgreSQL [35]) therefore include this optimization when monitoring for dangerous structures to reduce the number of aborts due to false positives. It is interesting to note that presence of a dangerous structure on itself does not necessarily mean that the schedule s is non-conflict-serializable, as our definition does not require a cycle in the serialization graph $SeG(s)$. However, if all transactions are allowed under SI, then every cycle in $SeG(s)$ implies a dangerous structure as part of the cycle [21, 35]. Stated differently, the absence of dangerous structures is a sufficient condition for conflict-serializability when all transactions are allowed under SI.

We are now ready to define when a schedule is allowed under a (mixed) allocation of isolation levels.

Definition 4.4. A schedule s over a set of transactions $\mathcal{T}$ *is allowed under an allocation* $\mathcal{A}$ over $\mathcal{T}$ if:

- for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) = \text{RC}$, T_i is allowed under RC;
- for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) \in \{\text{SI}, \text{SSI}\}$, T_i is allowed under SI; and
- there is no dangerous structure $T_i \to T_j \to T_k$ in s formed by three (not necessarily different) transactions $T_i, T_j, T_k \in \{T \in \mathcal{T} \mid \mathcal{A}(T) = \text{SSI}\}$.

We denote the allocation mapping all transactions to RC (respectively, SI) by $\mathcal{A}_{\text{RC}}$ (respectively, $\mathcal{A}_{\text{SI}}$). We illustrate some of the just introduced notions through an example.

Example 4.5. Consider the schedule s in Figure 7. Transaction T_1 is concurrent with T_2 and T_4, but not with T_3; all other transactions are pairwise concurrent with each other. The second read operation of T_4 is a read-last-committed relative to itself but not relative to the start of T_4. The read operation $R_2[v]$ of T_2 is read-last-committed relative to the start of T_2, but not relative to itself, so an allocation mapping T_2 to RC is not allowed. All other read operations are read-last-committed relative to both themselves and the start of the corresponding transaction. None of the transactions exhibits a dirty write. Only transaction T_4 exhibits a concurrent write (witnessed by the write operation $U_2[t]$ in T_2). Due to this, an allocation mapping T_4 on SI or SSI is not allowed. The transactions $T1 \to T2 \to T3$ form a dangerous structure, therefore an allocation mapping all three transactions T_1, T_2, T_3 on SSI is not allowed. All other allocations, that is, mapping T_4 on RC, T_2 on SI or SSI and at least one of T_1, T_2, T_3 on RC or SI, is allowed. □

4.4 Transaction Templates

Transaction templates are transactions where operations are defined over typed variables. Types of variables are relation names in Rels and indicate that variables can only be instantiated by tuples from the respective type. We fix an infinite set of variables **Var** that is disjoint from **Obj**. Every variable $X \in \mathbf{Var}$ has an associated relation name in Rels as type that we denote by $\text{type}(X)$.

Definition 4.6. A *transaction template* τ is a transaction over **Var**. In addition, for every operation o in τ over a variable X, $\text{ReadSet}(o) \subseteq \text{Attr}(\text{type}(X))$ and $\text{WriteSet}(o) \subseteq \text{Attr}(\text{type}(X))$.

For an operation o in a transaction template τ, we denote by $\mathit{var}(o)$ the variable over which o is defined. Notice that operations in transaction templates are defined over typed variables whereas they are over **Obj** in transactions. Indeed, the transaction template for Balance in Figure 2 contains a read operation $o = R[X : \text{Account}\{N, C\}]$. As explained in Section 2, the notation $X : \text{Account}\{N, C\}$ is a shorthand for $\text{type}(X) = \text{Account}$ and $\text{ReadSet}(o) = \{N, C\}$.

Recall that we denote variables by capital letters X, Y, Z and tuples by small letters t, v. A variable assignment μ is a mapping from **Var** to **Obj** such that $\mu(X) \in \mathbf{Obj}_{\text{type}(X)}$. By $\mu(\tau)$, we denote the transaction T obtained by replacing each variable X in τ with $\mu(X)$. A variable assignment for a database **D** maps every variable to a tuple occurring in a relation in **D**. We say that a transaction T is *instantiated* from a template τ over a database **D** if there is a variable assignment μ for **D** such that $T = \mu(\tau)$. As a slight abuse of notation, we will frequently write $\mu(o)$ for an operation o in τ to denote the corresponding operation in T.

A set of transactions $\mathcal{T}$ is *consistent* with a set of transaction templates $\mathcal{P}$ and database **D**, if every transaction in $\mathcal{T}$ is instantiated from a template in $\mathcal{P}$ over **D**. That is, for every transaction T in $\mathcal{T}$ there is a transaction template $\tau \in \mathcal{P}$ and a variable assignment μ_T for **D** such that $\mu_T(\tau) = T$.

We extend the notion of allocations towards transaction templates. For a class of isolation levels $\mathcal{I}$, a *template $\mathcal{I}$-allocation* $\mathcal{A}^{\mathcal{P}}$ for a set of transaction templates $\mathcal{P}$ is a function mapping each template $\tau \in \mathcal{P}$ onto an isolation level $\mathcal{A}^{\mathcal{P}}(\tau) \in \mathcal{I}$. When $\mathcal{I}$ is not important or clear from the context, we will frequently refer to $\mathcal{A}^{\mathcal{P}}$ as a template allocation rather than template $\mathcal{I}$-allocation.

Let $\mathcal{T}$ be a set of transactions consistent with a set of transaction templates $\mathcal{P}$ and a database **D**, and let $\mathcal{A}^{\mathcal{P}}$ be a template $\mathcal{I}$-allocation for $\mathcal{P}$. An allocation $\mathcal{A}$ for $\mathcal{T}$ is *consistent* with $\mathcal{A}^{\mathcal{P}}$ and **D** if for every transaction $T \in \mathcal{T}$ there is a template $\tau \in \mathcal{P}$ such that T is instantiated from τ over **D** and $\mathcal{A}(T) = \mathcal{A}^{\mathcal{P}}(\tau)$.

4.5 Transaction and Template Robustness

We first define the robustness property [8] (also called *acceptability* in [20, 21]) over a given set of transactions $\mathcal{T}$, which guarantees serializability for all schedules over $\mathcal{T}$ for a given allocation.

Definition 4.7 (Transaction robustness). A set of transactions $\mathcal{T}$ is *robust* against an allocation $\mathcal{A}$ for $\mathcal{T}$ if every schedule for $\mathcal{T}$ that is allowed under $\mathcal{A}$ is conflict-serializable.

We refer to $\mathcal{A}$ as a *robust allocation*. The *(transaction) robustness problem* is then to decide whether a given allocation for a set of transactions $\mathcal{T}$ is a robust allocation. A polynomial time algorithm for transaction robustness is given in [43].

We next lift robustness to the level of templates by requiring transaction robustness for all possible template instantiations and all possible databases. Let $\mathcal{P}$ be a set of transaction templates and $\mathbf{D}$ be a database. Then, $\mathcal{P}$ is *robust against a template allocation* $\mathcal{A}^{\mathcal{P}}$ *over* $\boldsymbol{D}$ if for every set of transactions $\mathcal{T}$ that is consistent with $\mathcal{P}$ and $\mathbf{D}$ and for every allocation $\mathcal{A}$ for $\mathcal{T}$ consistent with $\mathcal{A}^{\mathcal{P}}$ and $\mathbf{D}$, it holds that $\mathcal{T}$ is robust against $\mathcal{A}$.

Definition 4.8 (Template robustness). A set of transaction templates $\mathcal{P}$ is *robust* against a template allocation $\mathcal{A}^{\mathcal{P}}$ for $\mathcal{P}$ if $\mathcal{P}$ is robust against $\mathcal{A}^{\mathcal{P}}$ for every database $\mathbf{D}$.

4.6 Deciding Template Robustness

We are now ready to present our first algorithmic result: a polynomial-time algorithm for template robustness. In Section 4.7, we demonstrate how this algorithm can be applied to identify the lowest robust allocation.

Outline of approach. We recall that template robustness is defined over all possible database instances. Consequently, any approach that considers all possible instantiations of transaction templates and then applies the transaction robustness algorithm from [43] is infeasible due to the infinite number of possible instantiations. The algorithm proposed in this paper addresses this challenge using the following approach. We first introduce the concept of a sequence of potentially conflicting quadruples (which can be seen as an abstraction of a path in a serialization graph induced by the templates). We then obtain some conditions that characterize when this sequence can be converted to a counterexample schedule, that is, a non-conflict-serializable schedule that is still allowed under the given allocation. Furthermore, the schedule is of a very specific form to which we refer as a split schedule and only requires the existence of four different tuples per relation in the database. For the decision problem it then suffices to check for the existence of a sequence satisfying the above mentioned conditions, for which we present a polynomial time algorithm.

Sequence of potentially conflicting quadruples. Let τ_i and τ_j be two (not necessarily different) templates in $\mathcal{P}$, and let o_i and p_j be two operations in τ_i and τ_j, respectively. We then say that o_i is *potentially conflicting* with p_j if o_i and p_j are over variables of the same type (i.e., $\text{type}(var(o_i)) = \text{type}(var(p_j))$) and at least one of the following conditions holds:

- (potential ww-conflict) $\text{WriteSet}(o_i) \cap \text{WriteSet}(p_j) \neq \emptyset$;
- (potential wr-conflict) $\text{WriteSet}(o_i) \cap \text{ReadSet}(p_j) \neq \emptyset$; or
- (potential rw-conflict) $\text{ReadSet}(o_i) \cap \text{WriteSet}(p_j) \neq \emptyset$.

In this case, we also say that the tuple $(\tau_i, o_i, p_j, \tau_j)$ is a *potentially conflicting quadruple* over $\mathcal{P}$. Intuitively, a potentially conflicting quadruple represents a pair of operations that leads to conflicting operations whenever the corresponding variables are instantiated with the same tuple by a variable assignment. We will sometimes refer to the operation o as an *outgoing operation* and the operation p as an *incoming operation* in the quadruple (τ_i, o, p, τ_j).

Towards our algorithm, we consider *sequences of potentially conflicting quadruples* $C = (\tau_1, o_1, p_2, \tau_2), (\tau_2, o_2, p_3, \tau_3), \ldots, (\tau_{n-1}, o_{n-1}, p_n, \tau_n), (\tau_n, o_n, p_1, \tau_1)$ over a set of templates $\mathcal{P}$, where each tuple is a potentially conflicting quadruple over $\mathcal{P}$, and where multiple occurrences of the same template in C are allowed (i.e., $\tau_i = \tau_j$ is allowed, even when $i \neq j$). Notice in particular that C starts and ends with the same template τ_1.

We will use a sequence C of potentially conflicting quadruples to construct a non-conflict-serializable schedule, where a cycle in the serialization graph is formed by the operations in C. For this to work, care must be taken to ensure that the variables of the operations occurring in a potentially conflicting quadruple are assigned the same tuple in the database instance. Indeed, otherwise there would be no dependency between these operations in the schedule. Let o and p be two operations from templates τ_i and τ_j respectively, where τ_i and τ_j both occur in a sequence of potentially conflicting quadruples $C = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_n, o_n, p_1, \tau_1)$. We then say that the variables $var(o)$ and $var(p)$ are *connected* in C if

- $i = j$ and $var(o) = var(p)$ *(connected within the same template)*;
- there exists a quadruple (τ_i, o, p, τ_j) or (τ_j, p, o, τ_i) in C *(connected between templates)*; or
- there exists a variable $\mathtt{X}$ occurring in a template in C such that both $var(o)$ and $var(p)$ are connected to $\mathtt{X}$ *(transitivity)*.

Intuitively, connected variables must be assigned the same tuple for the variable assignments to be valid while ensuring that the desired dependencies are in place.

Mapping to a database with four elements per relation. Before we define variable assignments μ_i for each template τ_i in C, we first introduce a special database instance $\mathbf{D}_4$ over the considered schema Rels containing four tuples per relation in Rels. We refer to these tuples as $\mathtt{t}_1^R$, $\mathtt{t}_2^R$, $\mathtt{t}_3^R$, and $\mathtt{t}_4^R$ for each $R \in$ Rels and use $\mathbf{D}_4$ to construct the variable assignments μ_i. To this end, we first define four *type mappings* c_1, c_2, c_3 and c_4 that map each relation $R \in$ Rels to a tuple of the corresponding type in $\mathbf{D}_4$. Formally, we set $c_i(R) = \mathtt{t}_i^R$ for each $i \in \{1, 2, 3, 4\}$ and $R \in$ Rels. For each template τ_i in C, we define the *canonical variable assignments* μ_i over $\mathbf{D}_4$ as follows:

$$\mu_1(\mathtt{X}) = \begin{cases} c_1(\text{type}(\mathtt{X})) & \text{if } \mathtt{X} \text{ is connected to } var(o_1), \\ c_2(\text{type}(\mathtt{X})) & \text{if } \mathtt{X} \text{ is connected to } var(p_1) \text{ and} \\ & \text{not to } var(o_1), \\ c_4(\text{type}(\mathtt{X})) & \text{otherwise.} \end{cases}$$

For every μ_i with $1 < i \leq m$,

$$\mu_i(\mathtt{X}) = \begin{cases} c_1(\text{type}(\mathtt{X})) & \text{if } \mathtt{X} \text{ is connected to } var(o_1), \\ c_2(\text{type}(\mathtt{X})) & \text{if } \mathtt{X} \text{ is connected to } var(p_1) \\ & \text{not to } var(o_1), \\ c_3(\text{type}(\mathtt{X})) & \text{otherwise.} \end{cases}$$

By construction, type mapping c_4 is used exclusively for τ_1, whereas c_3 is only used for $\tau_2, \ldots, \tau_n$; c_1 is used for all variables connected to $var(o_1)$; and, c_2 is used for all variables connected to $var(p_1)$, unless $var(o_1)$ and $var(p_1)$ are connected in C in which case c_1 is used as well as by transitivity variables connected to $var(p_1)$ are also connected to $var(o_1)$.

From a sequence to a split schedule. For a given sequence C of potentially conflicting quadruples over a set of templates $\mathcal{P}$, we define the *canonical set of transactions* $\mathcal{T}_C$ as the set obtained by applying the canonical variable assignment μ_i to each template τ_i occurring in C. A template allocation $\mathcal{A}^{\mathcal{P}}$ over $\mathcal{P}$ then induces a *canonical allocation* $\mathcal{A}_C$ over $\mathcal{T}_C$ in a natural way: for each template τ_i in C, we allocate the corresponding transaction $T_i = \mu_i(\tau_i)$ to the isolation level $\mathcal{A}_C(T_i) = \mathcal{A}^{\mathcal{P}}(\tau_i)$. By construction, $\mathcal{T}_C$ is consistent with $\mathcal{P}$ and $\mathbf{D}_4$, and $\mathcal{A}_C$ is consistent with $\mathcal{A}^{\mathcal{P}}$ and $\mathbf{D}_4$. For a transaction T and operation $o \in T$, we denote by $\text{prefix}_o(T)$ the subsequence of T containing all operations up to and including o, and by $\text{postfix}_o(T)$ the subsequence of T containing all operations strictly after o. This notation extends to templates in a natural way.

Definition 4.9. Let $\mathcal{P}$ be a set of transaction templates, $\mathcal{A}^{\mathcal{P}}$ a template allocation over $\mathcal{P}$, and $C = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_n, o_n, p_1, \tau_1)$ a sequence of potentially conflicting quadruples over $\mathcal{P}$. A *template split schedule s for $\mathcal{P}$ and $\mathcal{A}^{\mathcal{P}}$ induced by C* is a multiversion schedule over the canonical set of transactions $\mathcal{T}_C$ of the form:

$$\text{prefix}_{\mu_1(o_1)}(\mu_1(\tau_1)) \cdot \mu_2(\tau_2) \cdot \ldots \cdot \mu_n(\tau_n) \cdot \text{postfix}_{\mu_1(o_1)}(\mu_1(\tau_1)),$$

where

(1) s is allowed under the canonical allocation $\mathcal{A}_C$ induced by $\mathcal{A}^{\mathcal{P}}$;
(2) $\mu_i(o_i) \rightarrow_s \mu_j(p_j)$ for each quadruple $(\tau_i, o_i, p_j, \tau_j) \in C$; and
(3) there is no operation in $\mu_1(\tau_1)$ conflicting with an operation in any of the transactions $\mu_3(\tau_3), \ldots, \mu_{n-1}(\tau_{n-1})$.

Notice that the cycle of dependencies between the operations in C implies that such a schedule is not conflict-serializable. The next proposition then readily follows:

Proposition 4.10. *Let s be a template split schedule for a set of templates $\mathcal{P}$ and template allocation $\mathcal{A}^{\mathcal{P}}$ over $\mathcal{P}$ induced by a sequence of potentially conflicting quadruples C. Then, s is non-conflict-serializable and allowed under $\mathcal{A}^{\mathcal{P}}$.*

Conditions characterizing the existence of a counterexample schedule. We introduce a set of conditions that must be satisfied by a sequence of potentially conflicting quadruples for a template split schedule to exist.

Proposition 4.11. *Let $\mathcal{P}$ be a set of transaction templates, $\mathcal{A}^{\mathcal{P}}$ a template allocation over $\mathcal{P}$, and $C = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_n, o_n, p_1, \tau_1)$ a sequence of potentially conflicting quadruples over $\mathcal{P}$. A template split schedule for $\mathcal{P}$ and $\mathcal{A}^{\mathcal{P}}$ induced by C exists if and only if the following conditions hold:*

(1) there is no operation o in τ_1 potentially conflicting with an operation p in any of the templates $\tau_3, \ldots, \tau_{n-1}$ with $var(o)$ and $var(p)$ connected in C;
(2) there is no write operation o in $\text{prefix}_{o_1}(\tau_1)$ potentially ww-conflicting with a write operation p in τ_2 or τ_n where $var(o)$ and $var(p)$ are connected in C;
(3) if $\mathcal{A}^{\mathcal{P}}(\tau_1) \in \{SI, SSI\}$, then there is no write operation o in $\text{postfix}_{o_1}(\tau_1)$ potentially ww-conflicting with a write operation p in τ_2 or τ_n with $var(o)$ and $var(p)$ connected in C;
(4) o_1 is potentially rw-conflicting with p_2;
(5) o_n is potentially rw-conflicting with p_1 or ($\mathcal{A}^{\mathcal{P}}(\tau_1) = RC$ and $o_1 <_{\tau_1} p_1$);
(6) $\mathcal{A}^{\mathcal{P}}(\tau_1) \neq SSI$ or $\mathcal{A}^{\mathcal{P}}(\tau_2) \neq SSI$ or $\mathcal{A}^{\mathcal{P}}(\tau_n) \neq SSI$;
(7) if $\mathcal{A}^{\mathcal{P}}(\tau_1) = SSI$ and $\mathcal{A}^{\mathcal{P}}(\tau_2) = SSI$, then there is no operation o in τ_1 potentially wr-conflicting with an operation p in τ_2 with $var(o)$ and $var(p)$ connected in C; and
(8) if $\mathcal{A}^{\mathcal{P}}(\tau_1) = SSI$ and $\mathcal{A}^{\mathcal{P}}(\tau_n) = SSI$, then there is no operation o in τ_1 potentially rw-conflicting with an operation p in τ_n with $var(o)$ and $var(p)$ connected in C.

Intuitively, these conditions enforce the desired cycle of dependencies, while ensuring that the schedule is allowed under the allocation (cf. Definition 4.9). For example, conditions (1) and (2) ensure that no dirty writes are present, and condition (3) additionally avoids concurrent writes for transactions allocated to SI or SSI. In the proof we argue that if a condition is not satisfied, then at least one of the requirements of Definition 4.9 is not met.

Decision algorithm. The next proposition shows that it suffices to find a split schedule to decide template robustness.

Proposition 4.12. *Let $\mathcal{P}$ be a set of transaction templates and let $\mathcal{A}^{\mathcal{P}}$ be an allocation for $\mathcal{P}$. The following are equivalent:*

- *$\mathcal{P}$ is not robust against $\mathcal{A}^{\mathcal{P}}$;*
- *there exists a template split schedule s for $\mathcal{P}$ and $\mathcal{A}^{\mathcal{P}}$ induced by a sequence of potentially conflicting quadruples C over $\mathcal{P}$.*

It readily follows from Proposition 4.10 that a split schedule witnesses non-robustness. The reverse direction is more involved, and relies on the argument that we can extract a sequence C satisfying the conditions in Proposition 4.11 from an arbitrary non-conflict-serializable schedule allowed under the allocation.

Proposition 4.11 then offers a concrete way to find a split schedule via sequences of potentially conflicting quadruples. However, a naive enumeration is not feasible, as these sequences can have an arbitrary length. Instead, we propose an algorithm that iterates over all possible choices for operations o_1 and p_1 in a template $\tau_1 \in \mathcal{P}$, constructing a graph referred to as pt-conflict-graph$(o_1, p_1, \tau_1, h, \mathcal{P})$ with $h \in \{1, 2\}$, which we will define next. Intuitively, the existence of a sequence C satisfying the conditions in Proposition 4.11 corresponds to reachability between specific nodes in this graph, and some additional conditions that the algorithm will verify separately.

Let $\mathcal{P}$ be a set of transaction templates, o_1 and p_1 two (not necessarily different) operations occurring in a template $\tau_1 \in \mathcal{P}$, and let $h \in \{1, 2\}$. The directed graph pt-conflict-graph$(o_1, p_1, \tau_1, h, \mathcal{P})$ has nodes of the form (τ, o, c, k) for all $\tau \in \mathcal{P}$, $o \in \tau$, $c \in \{O, P, N\}$ and $k \in \{\text{in}, \text{out}\}$ satisfying the following conditions:

(1) if $c = O$, there is no operation $o_1' \in \tau_1$ over the same variable as o_1 such that o_1' is potentially conflicting with an operation $o' \in \tau$ and $var(o') = var(o)$;
(2) if $c = P$, there is no operation $o_1' \in \tau_1$ over the same variable as p_1 such that o_1' is potentially conflicting with an operation $o' \in \tau$ and $var(o') = var(o)$.

Intuitively, the value h indicates whether $var(o_1)$ and $var(p_1)$ are connected in the sequence C that will be constructed by the algorithm, where $h = 1$ indicates that they are connected and $h = 2$ indicates that they are not. For each node (τ, o, c, k), the value of c indicates that o is connected to o_1 ($c = O$), to p_1 ($c = P$), or to neither ($c = N$) in C. Lastly, the value of k indicates whether o is an incoming or outgoing operation in a quadruple in C. The previous two conditions then guarantee that the sequence C constructed by the algorithm satisfies Condition 1 in Proposition 4.11.

Algorithm 1: Deciding template robustness

Input : Set of transaction templates $\mathcal{P}$ and template allocation $\mathcal{A}^{\mathcal{P}}$ for $\mathcal{P}$
Output: *True* iff $\mathcal{P}$ is robust against $\mathcal{A}^{\mathcal{P}}$

foreach $o_1, p_1 \in \tau_1$ *with* $\tau_1 \in \mathcal{P}$ **do**
  if $var(o_1) = var(p_1)$ **then** $H := \{1\}$ **else** $H := \{1, 2\}$;
  foreach $h \in H$ **do**
    $G :=$ pt-conflict-graph$(o_1, p_1, \tau_1, h, \mathcal{P})$;
    $TC :=$ transitive closure of G;
    foreach $o_2, p_2 \in \tau_2$; $o_n, p_n \in \tau_n$ *with* $\tau_2, \tau_n \in \mathcal{P}$ **do**
      if o_1 *not potentially conflicting with* p_2 ***or***
        o_n *not potentially conflicting with* p_1 **then**
        continue;
      if $var(o_2) = var(p_2)$ **then** $C_{o2} := \{O\}$ **else** $C_{o2} := \{N, P\}$;
      if $var(o_n) = var(p_n)$ **then** $C_{pn} := \{P\}$ **else** $C_{pn} := \{N, O\}$;
      foreach $c_{o2} \in C_{o2}$, $c_{pn} \in C_{pn}$ **do**
        if *Reachable*$(\tau_2, o_2, p_2, c_{o2}, \tau_n, o_n, p_n, c_{pn}, h, TC)$ ***and***
          ValidSchedule$(\tau_1, o_1, p_1, \tau_2, o_2, p_2, c_{o2}, \tau_n, o_n, p_n, c_{pn}, h, \mathcal{A}^{\mathcal{P}})$ **then**
          return False;
return True;

The graph pt-conflict-graph$(o_1, p_1, \tau_1, h, \mathcal{P})$ contains an edge from a node (τ, o, c, k) to a node (τ', o', c', k') if and only if one of the following conditions hold:

(3) $k =$ out, $k' =$ in, $c = c'$, and o is potentially conflicting with o';
(4) $k =$ in, $k' =$ out, $\tau = \tau'$, $var(o) \neq var(o')$ and $(c, c') \in \{(O, P), (O, N), (N, N), (N, P)\}$;
(5) $k =$ in, $k' =$ out, $\tau = \tau'$, $var(o) = var(o')$ and $c = c'$; or
(6) $k =$ in, $k' =$ out, $\tau = \tau'$, $var(o) = var(o')$, $c = O$, $c' = P$ and $h = 1$.

By construction, these edges propagate connectedness with respect to $var(o_1)$ and $var(p_1)$ in the sequence C that will be constructed by the algorithm. Condition (3) requires potentially conflicting operations for each quadruple in C, while maintaining connectedness (recall that the variables over which the operations in a quadruple are defined are always connected). Within a template, connectedness with o_1 or p_1 only propagates if the variables are the same (Condition (4) and Condition (5)). Condition (6) covers the special case when we assume that $var(o_1)$ and $var(p_1)$ are connected (i.e., $h = 1$). In that case, every variable connected to $var(o_1)$ is connected to $var(p_1)$ as well.

The template robustness algorithm is displayed as Algorithm 1. The algorithm iterates over all possible choices for operations in τ_1, τ_2 and τ_n, tracking connectedness of $var(o_2)$ and $var(p_n)$ with respect to $var(o_1)$ and $var(p_1)$. For each such choice, the algorithm then verifies whether p_n is reachable from o_2 in the pt-conflict-graph, thereby witnessing the existence of a sequence of potentially conflicting quadruples satisfying Condition (1) of Proposition 4.11, followed by the verification of the remaining conditions. The implementation of *Reachable* and *ValidSchedule* are straightforward and provided in detail in [38].

Algorithm 2: Computing the lowest robust allocation.

Input : Set of templates $\mathcal{P}$
Output: Lowest robust template allocation $\mathcal{A}^{\mathcal{P}}$ for $\mathcal{P}$

for $\tau \in \mathcal{P}$ **do** $\mathcal{A}^{\mathcal{P}}[\tau] :=$ SSI ;
for $\tau \in \mathcal{P}$ **do**
  if $\mathcal{P}$ *is robust against* $\mathcal{A}^{\mathcal{P}}[\tau \mapsto RC]$ **then**
    $\mathcal{A}^{\mathcal{P}} := \mathcal{A}^{\mathcal{P}}[\tau \mapsto \text{RC}]$;
  else if $\mathcal{P}$ *is robust against* $\mathcal{A}^{\mathcal{P}}[\tau \mapsto SI]$ **then**
    $\mathcal{A}^{\mathcal{P}} := \mathcal{A}^{\mathcal{P}}[\tau \mapsto \text{SI}]$;
return $\mathcal{A}^{\mathcal{P}}$;

Theorem 4.13. *Let $\mathcal{P}$ be a set of templates and $\mathcal{A}^{\mathcal{P}}$ a template allocation for $\mathcal{P}$. Algorithm 1 decides whether $\mathcal{P}$ is robust against $\mathcal{A}^{\mathcal{P}}$ in time polynomial in the size of $\mathcal{P}$.*

4.7 Finding Lowest Robust Allocations

We first show that there always exists a unique lowest robust allocation and then present a polynomial time algorithm to find it.

As discussed in Section 2, we assume a total order RC < SI < SSI over the three considered isolation levels, expressing our preference for lower isolation levels over higher ones. In the following, let $\mathcal{A}_1^{\mathcal{P}}$ and $\mathcal{A}_2^{\mathcal{P}}$ be two template allocations for a set of templates $\mathcal{P}$. We write $\mathcal{A}_1^{\mathcal{P}} \leq \mathcal{A}_2^{\mathcal{P}}$ if $\mathcal{A}_1^{\mathcal{P}}(\tau) \leq \mathcal{A}_2^{\mathcal{P}}(\tau)$ for all $\tau \in \mathcal{P}$. Furthermore, we write $\mathcal{A}_1^{\mathcal{P}} < \mathcal{A}_2^{\mathcal{P}}$ if $\mathcal{A}_1^{\mathcal{P}} \leq \mathcal{A}_2^{\mathcal{P}}$ and there exists a template $\tau \in \mathcal{P}$ such that $\mathcal{A}_1^{\mathcal{P}}(\tau) < \mathcal{A}_2^{\mathcal{P}}(\tau)$. For an isolation level $\mathcal{I}$, we denote by $\mathcal{A}_1^{\mathcal{P}}[\tau \mapsto \mathcal{I}]$ the template allocation for $\mathcal{P}$ where $\mathcal{A}_1^{\mathcal{P}}[\tau \mapsto \mathcal{I}](\tau) = \mathcal{I}$ and $\mathcal{A}_1^{\mathcal{P}}[\tau \mapsto \mathcal{I}](\tau') = \mathcal{A}_1^{\mathcal{P}}(\tau')$ for all other templates $\tau' \in \mathcal{P}$. That is, the template allocation derived from $\mathcal{A}_1^{\mathcal{P}}$ by setting the isolation level of τ to $\mathcal{I}$, while leaving all other templates unchanged.

Definition 4.14. Let $\mathcal{P}$ be a set of templates robust against a template allocation $\mathcal{A}_1^{\mathcal{P}}$ for $\mathcal{P}$. Then, $\mathcal{A}_1^{\mathcal{P}}$ is *lowest* if there is no other allocation $\mathcal{A}_2^{\mathcal{P}}$ for $\mathcal{P}$ such that $\mathcal{A}_2^{\mathcal{P}} < \mathcal{A}_1^{\mathcal{P}}$ and $\mathcal{P}$ is robust against $\mathcal{A}_2^{\mathcal{P}}$.

The next propositions extend some results for allocations for transactions presented in [43] towards template allocations:

Proposition 4.15. *Let $\mathcal{P}$ be a set of templates, and let $\mathcal{A}_1^{\mathcal{P}}$ and $\mathcal{A}_2^{\mathcal{P}}$ be two template allocations for $\mathcal{P}$.*

(1) If $\mathcal{A}_1^{\mathcal{P}} \leq \mathcal{A}_2^{\mathcal{P}}$ and $\mathcal{P}$ is robust against $\mathcal{A}_1^{\mathcal{P}}$, then $\mathcal{P}$ is robust against $\mathcal{A}_2^{\mathcal{P}}$.
(2) If $\mathcal{P}$ is robust against $\mathcal{A}_1^{\mathcal{P}}$ and $\mathcal{A}_2^{\mathcal{P}}$, then $\mathcal{P}$ is robust against $\mathcal{A}_2^{\mathcal{P}}[\tau \mapsto \mathcal{A}_1^{\mathcal{P}}(\tau)]$ for every $\tau \in \mathcal{P}$.

Proposition 4.16. *There exists a unique lowest template allocation for every set of templates $\mathcal{P}$.*

Algorithm 2 outlines the procedure for determining the unique lowest robust allocation. Initially, all templates are assigned the allocation SSI. The algorithm then iterates through each template, evaluating whether the associated isolation level can be safely reduced using Algorithm 1 for the robustness test.

Theorem 4.17. *For a set of templates $\mathcal{P}$, Algorithm 2 computes the unique lowest robust template allocation in time polynomial in the size of $\mathcal{P}$.*

5 RELATED WORK

Shortly after the initial papers that defined serializability and studied its theoretical complexity [34], the IBM System R team published an account of weaker isolation levels and locking-based algorithms that achieved those by releasing shared locks early, or even not taking them at all [23]. Much later, the multiversion Snapshot Isolation mechanism was described, and shown to allow some non-serializable executions, despite avoiding all the anomalies mentioned in the SQL specification [7]. The serializable multiversion SSI mechanism was proposed by Cahill [14] and was implemented (with optimizations) in PostgreSQL [35].

There has been a variety of approaches used to define isolation properties abstractly. For definitions and proofs that mechanisms achieve serializability, theory was developed especially by Bernstein and Goodman, including for multiversion and even distributed protocols [10, 11]. For defining lower isolation, Gray et al. [23] and the later SQL specification used the notion of anomalies, that is patterns of read and write operations that occur in the system's histories and can lead to situations that do not happen in serial executions. Berenson et al. [7] showed that this approach was inadequate to deal with Snapshot Isolation, and proposed some alternatives that were later seen as also inadequate. Adya developed a theory framework for defining these isolation levels abstractly, based on graphs showing dependency edges between operations[1, 2]. Much ongoing work has built on Adya's style. Cerone et al. [15] and Crooks et al. [17] offer different approaches based on abstract state or a "client-centric" definition style.

Shasha et al. introduced the approach of showing conditions which can prove robustness, showing when serializable execution is ensured despite using mechanisms that in general allow non-serializable behavior (in this case, dividing a transaction into segments separated by COMMIT operations) [37]. Fekete et al. gave a theory that could prove robustness for Snapshot Isolation [21]. This paper also introduced the technique of making an application robust through modifying application code by "promoting" a read to also do an identity update of the item. Alomari et al. examined performance comparisons of promotion choices and other ways to modify application code for robustness [5]. Further work in this line showed a sufficient condition to prove robustness for Read Committed Isolation [4]. Recent research has introduced the concept of split schedules, establishing necessary and sufficient conditions for ensuring transaction robustness across various isolation levels [28, 29, 39, 40, 43]. However, a proof technique utilizing split-schedules does not always lead to an efficient algorithm: under a lock-based semantics robustness testing can become coNP-complete [27] or even undecidable [41]. Other work examines robustness within a framework that declaratively specifies different isolation levels in a uniform manner [8, 15, 16], relying on the key assumption of atomic visibility, which ensures that either all or none of a transaction's updates are visible to other transactions.

Fekete [20] introduced the allocation question: choosing the concurrency control for each transaction separately from a set of available isolation mechanisms; this paper dealt with choosing either 2-phase locking, or snapshot isolation. Recently, other combinations of choices have been considered, such as combinations of read committed, snapshot isolation and/or serializable snapshot isolation [43]. In [44], the allocation problem is studied in the context of view- rather than conflict-serializability, and is observed that for the isolation levels of PostgreSQL, both problems coincide.

Beyond handling specific transactions that access explicitly identified items, practical scenarios often require working with application code, where the accessed items may vary at runtime. This variability can arise, for example, from values retrieved in earlier queries or user-provided parameters. Various approaches have been explored to represent such applications and reason about the different explicit transactions they may generate during execution. The concept of a transaction program was introduced in [21], while the abstraction of a transaction template, which we adopt here, was presented in [39]. Adding constraints such as foreign keys quickly makes robustness undecidable [41], though restricted classes of constraints still allow decidability. Vandevoort et al. [42] explored a more expressive variant of transaction programs, offering a sufficient condition for ensuring robustness under read committed isolation. This approach, which relies on a formalism for transaction programs to define potential workloads, stands in contrast to methods like IsoDiff [22], where transactions are derived from concrete execution traces.

6 CONCLUSIONS

We introduced a novel optimization method, which can enhance performance without requiring modifications to the database system's internals. An evaluation on SmallBank demonstrates that *RePMILA* can achieve throughput comparable to the unsafe yet default RC isolation level used by some platforms, while maintaining safety. Additionally, it can double throughput compared to executing all under the serializable isolation level.

Our approach builds on the theoretical framework for solving the mixed allocation problem [43], which we extend in this paper from fully specified transactions to transaction templates. However, this extension brings several limitations that are not yet addressed in the current formalization. First, it does not account for data dependencies such as foreign keys which more accurately model transactions that can effectively occur. Second, it lacks support for transaction programs that include control structures (e.g., loops and conditionals) as well as operations like inserts, deletes, and predicate reads—scenarios that can trigger the phantom problem. To address these challenges, we can build on ideas from [41, 42], though substantial theoretical work is still needed—making this a key direction for future research.

ACKNOWLEDGMENTS

This work was partly funded by FWO-grant G019921. Stijn Vansummeren was partially supported by the Bijzonder Onderzoeksfonds (BOF) of Hasselt University (Belgium) under Grant No. BOF20ZAP02. The resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government.

REFERENCES

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. MIT, Cambridge, MA, USA.

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

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

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

[5] Mohammad Alomari, Alan D. Fekete, and Uwe Röhm. 2014. Performance of program modification techniques that ensure serializable executions with snapshot isolation DBMS. *Inf. Syst.* 40 (2014), 84–101. https://doi.org/10.1016/J.IS.2013.10.002

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

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

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

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

[10] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (1983), 465–483.

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

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

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

[14] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.* 34, 4 (2009), 20:1–20:42.

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

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

[17] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *PODC*. 73–82.

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

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

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

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

[22] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *PVLDB* 13, 11 (2020), 2773–2786.

[23] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, G. M. Nijssen (Ed.). North-Holland, 365–394.

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

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

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

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

[28] Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. 2022. Concurrency control for database theorists. *SIGMOD Rec.* 51, 4 (2022), 6–17.

[29] Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. 2022. Deciding Robustness for Lower SQL Isolation Levels. *ACM Trans. Database Syst.* 47, 4 (2022), 13:1–13:41. https://doi.org/10.1145/3561049

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

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

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

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

[34] Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J.ACM* 26, 4 (1979), 631–653.

[35] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *PVLDB* 5, 12 (2012), 1850–1861.

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

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

[38] Brecht Vandevoort, Alan Fekete, Bas Ketsman, Frank Neven, and Stijn Vansummeren. 2025. Using Read Promotion and Mixed Isolation Levels for Performant Yet Serializable Execution of Transaction Programs (full version). (2025). http://arxiv.org/abs/2501.18377.

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

[40] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2022. Robustness Against Read Committed: A Free Transactional Lunch. In *PODS*. 1–14.

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

[42] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2023. Detecting Robustness against MVRC for Transaction Programs with Predicate Reads. In *EDBT*. 565–577.

[43] Brecht Vandevoort, Bas Ketsman, and Frank Neven. 2023. Allocating Isolation Levels to Transactions in a Multiversion Setting. In *PODS*. 69–78.

[44] Brecht Vandevoort, Bas Ketsman, and Frank Neven. 2024. When View- and Conflict-Robustness Coincide for Multiversion Concurrency Control. *Proc. ACM Manag. Data* 2, 2 (2024), 91.

[45] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB* 8, 3 (2014), 209–220.