Constraint-Preserving Snapshot Isolation

Stephen J. Hegner
Umeå University
Department of Computing Science
SE-901 87 Umeå, Sweden
`hegner@cs.umu.se`
`http://www.cs.umu.se/~hegner`

Corrected 15 July 2017

Abstract

A method for detecting potential violations of integrity constraints of concurrent transactions running under snapshot isolation (SI) is presented. Although SI provides a high level of isolation, it does not, by itself, ensure that all integrity constraints are satisfied. In particular, while current implementations of SI enforce all internal integrity constraints, in particular key constraints, they fail to enforce constraints implemented via triggers. One remedy is to turn to serializable SI (SSI), in which full serializability is guaranteed. However, SSI comes at the price of either a substantial number of false positives, or else a high cost of constructing the full direct serialization graph. In this work, a compromise approach, called *constraint-preserving snapshot isolation (CPSI)*, is developed, which while not guaranteeing full serializability, does guarantee that all constraints, including those enforced via triggers, are satisfied. In contrast to full SSI, CPSI requires testing concurrent transactions for conflict only pairwise, and thus involves substantially less overhead while providing a foundation for resolving conflicts via negotiation rather than via abort and restart. As is the case with SSI, CPSI can result in false positives. To address this, a hybrid approach is also developed which combines CPSI with a special version of SSI called CSSI, resulting in substantially fewer false positives than would occur using either approach alone.

1 Introduction

A hallmark feature of any modern database-management system (DMBS) is support for concurrent transactions. The guiding principle, embodied as *isolation* in the classical ACID (atomicity, correctness, isolation, durability) properties for transactions [10], mandates that concurrent transactions not interfere with one another. The gold standard for such isolation is *view serializability* [18, Sec. 2.4], in which the result of executing a set of transactions concurrently must be the same as executing them in some serial order, one after the other.[1]

[1] In practice, a stronger form, known as *conflict serializability* [18, Sec. 4.3], is often used. Although it excludes some schedules which are in fact view serializable, they are typically anomalous in form and arise

The classical, single-version, lock-based implementation of view serializability is via *two-phase locking*, or *2PL* for short. The ideas were first presented in the seminal paper [8], and given a firm theoretical foundation in [21].[2] In order to avoid undesirable and unmanageable interaction, in particular nonrecoverable schedules in which the result of a committed transaction must be reversed, a stronger version of 2PL, known as *rigorous two-phase locking (rigorous 2PL)* [4], or more commonly nowadays *strong strict two-phase locking (SS2PL)*, is generally regarded as an essential extension in implementation. 2PL and SS2PL typically receive thorough coverage in modern textbooks on DBMSs, and the student is often left with the impression that they are widely used in practice. It may therefore come as something of a surprise to learn that they do not enjoy widespread use in real systems. Indeed, some of the most widely used systems do not even implement 2PL, and even the few which do typically provide something much weaker as the default isolation level. The drawback to SS2PL is that all exclusive (write) locks must be held until the end of the transaction, and so a writer of a data object will block all readers of that data object until it finishes (commits). So, while it guarantees both view serializability and recoverable schedules, SS2PL does so at a price of greatly limited concurrency. In most cases, the consequent performance hit is too great, and so weaker levels of isolation, which admit higher levels of concurrency, are the choice.

Of course, the goal is to achieve both a strong level of isolation and a high level of concurrency. Progress towards this goal has been made possible by advances in memory size and processor speed, which has in turn led to the the emergence, over the past few decades, of DBMSs employing multiversion concurrency control (MVCC), in which several version of a data object may coexist. This notion, which has its roots in early work on optimistic concurrency control [16], has in turn opened the door to newer models of isolation, of which *snapshot isolation (SI)* has become one of the most prevalent. In SI, each transaction operates on its own private copy of the database (its *snapshot*). Upon completion (commit), the results of the updates which a transaction has made on its snapshot must be integrated into the stable (permanent) database. If there is a write conflict; that is, if several concurrent transactions write the same data object, only one transaction is allowed to commit and have its updates become part of the stable database. The others must abort if they are not naturally terminated in some other way.

In contrast to the situation surrounding SS2PL, readers are not blocked by writers in SI.[3] It is only conflicts between concurrent writers which limit concurrency. Furthermore SI avoids many of the update anomalies associated with policies such as read uncommitted (RU) and read (latest) committed (RC), including in particular dirty and nonrepeatable reads, respectively [6, p. 61]. Nevertheless, it does permit certain types of undesirable behavior which do not occur under view serialization, such as write skew [2, A5B] [9, Ex. 2.2], as well as a certain type of read-only anomaly [9, Ex. 2.3].[4]

rarely in practice. Conflict serializability is particularly attractive in situations in which schedules are to be tested for serializability, since it admits a test of low deterministic polynomial order (detecting cycles in a graph), while deciding view serializability is NP-complete. It is conflict serializability which is used as the formalization throughout this paper.

[2] Schedulers which employ 2PL actually guarantee schedules which are conflict serializable.

[3] Although it is largely true that readers are not blocked by writers under SI, it is not completely true, due to the way internal integrity constraints are enforced in real systems. See Summary 2.4 for a more complete discussion.

[4] The `SERIALIZABLE` level of isolation, as defined in the SQL standard, does not ensure view serializability, or

Because true view serializability [18, Sec. 2.4] is the gold standard for isolation of transactions, there has been substantial recent interest in augmenting SI to achieve such true serializability, the idea being to achieve the desirable properties of true serializability while exploiting the efficiency of SI. In particular, a strategy known as *serializable SI* (SSI), has been developed [9], [5]. In stark contrast to SS2PL, SSI is an optimistic approach. Rather than forcing transactions to wait until a needed resource is available, it relies much more on allowing them to proceed, and then resolving concurrency conflicts by requiring one or more transactions to terminate without committing.

Unfortunately, any algorithm which is capable of deciding whether or not a set of concurrent transactions running under SI exhibits view-serializable isolation must, in the worst case, examine all transactions. It is instructive to show how this worst case arises. Let $n_0 \geq 2$ be a natural number and let $\mathbf{E}_0$ be a database schema which includes n_0 integer-valued data objects d_0, d_1, ..., d_{n_0-1}. Let τ_{0i} be the transaction which replaces the value of d_i with the current value of $d_{(i+1) \bmod n_0}$; i.e., which executes $d_i \leftarrow d_{(i+1) \bmod n_0}$. Running the set $\mathbf{T}_0 = \{\tau_{0i} \mid 0 \leq i < n_0\}$ of transactions concurrently under snapshot isolation results in a permutation of the values of the d_i's, with the new value of d_i being the old value of $d_{(i+1) \bmod n_0}$, since each transaction sees the old values of the d_i's in its snapshot. However, no serial schedule of $\mathbf{T}_0$ can produce this permutation result. Indeed, if τ_{0i} is run first and commits before any other transaction begins, then the old value of d_i will be overwritten before $\tau_{0((i+1) \bmod n_0)}$ is able to read it. Thus, $\mathbf{T}_0$ is not view serializable. Formally, there is a read-write dependency [9, Def. 2.2] (or *antidependency* [1, 4.4.2]) from $\tau_{0(i \bmod n_0)}$ to $\tau_{0((i+1) \bmod n_0)}$ for data object $d_{(i+1) \bmod n_0}$, meaning that $\tau_{0(i \bmod n_0)}$ reads $d_{(i+1) \bmod n_0}$ and $\tau_{0((i+1) \bmod n_0)}$ is the first transaction to write $d_{(i+1) \bmod n_0}$ after the start of $\tau_{0(i \bmod n_0)}$. These dependencies are represented using the *multiversion direct serialization graph (DSG)* or *multiversion conflict graph* [1], as illustrated in Fig. 1.1. As argued above, (and in general since this graph contains a cycle [1, Sec. 5.3]), no view serialization is possible. However, if any transaction is removed from $\mathbf{T}_0$, the remaining set is serializable. Indeed, if τ_{0i} is removed, then execution in the serial order $\tau_{0(i+1)}\tau_{0(i+2)} \ldots \tau_{0(n_0-1)}\tau_{00}\tau_{01} \ldots \tau_{0(i-1)}$ is equivalent to concurrent execution under SI. Thus, for any natural number n, there is a set $\mathbf{T}$ of n concurrent transactions whose execution under SI is not equivalent to any serial schedule, but execution of any proper subset of $\mathbf{T}$ under SI is equivalent to a serial execution. In other words, to determine whether a set of n transactions run under SI is view serializable, a test involving all n transactions must be performed.

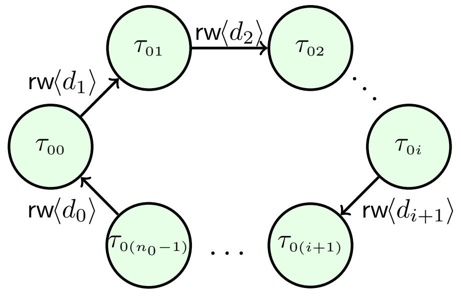


Figure 1.1: An SI rw-conflict cycle of length n_0

To address this complexity issue, conditions which identify necessary but not sufficient conditions have been developed. More specifically, a *dangerous structure* is a sequence of two consecutive read-write edges between concurrent transactions which occurs in a cycle of the DSG. For example, in the graph of Fig. 1.1, any two consecutive edges form a dangerous

any other reasonable notion of serializability. In particular, SI qualifies as a serializable level of isolation under that standard, and is offered as the implementation of `SERIALIZABLE` by some systems. See [2] for a further discussion.

structure. It has been shown that such a dangerous structure must be present, in a cycle of the DSG, for nonserializable behavior to occur [5]. However, while identifying dangerous structures requires examining at most three transactions at a time, to determine whether a dangerous structure lies within a cycle requires examining all transactions in the worst case, since the entire DSG must be constructed. As a compromise, any sequence of two consecutive read-write edges between concurrent transactions may raise a red flag which requires one or more transactions to abort, without a check of whether they lie within a cycle. Indeed, this is exactly the approach which is taken in SSI [5]. This will result in false positives, although benchmarks reported in [5] are impressive. Recently, PostgreSQL, as of version 9.1, became the first widely used DBMS to put SSI into practice, employing a variant for the implementation of its serializable isolation level [19].

It should be noted that in *Precisely SSI (PSSI)* [20], the entire DSG is constructed, thus avoiding virtually all false positives. Although a prototype has been implemented, this strategy has not yet been tested on a large scale, at least to the knowledge of this author. It thus remains open how well it would perform for transaction mixes involving long cycles.

Despite the impressive performance statistics in the benchmark results, and the recent use in a widely used DBMS, it must be acknowledged that SSI and PSSI are not appropriate for all application domains. In particular, in any setting which involves long-running and interactive transactions, a policy for enforcing isolation which depends upon aborts and/or waits, as do SSI and PSSI, is highly undesirable. Interactive business processes are one such domain, and it is in particular the context of cooperative transactions within that setting [14, 12] which motivated the work of this paper. In such settings, there are two complementary goals. The first is to minimize the number of false positives, and the second is to identify ways to perform more amicable solutions to conflicts which cannot be avoided. The focus of this paper is to provide a framework which addresses both goals. On the one hand, it provides a method for guaranteeing acceptable behavior while avoiding many of the false positives which occur with SSI. On the other hand, when conflicts are indicated, it provides information which greatly facilitates the identification of which transactions need to negotiate and cooperate in order to resolve the problem.

To this end, it is important to begin by noting that nonserializable behavior for transactions has been, and will continue to be, an acceptable compromise in many circumstances. Indeed, for reasons of efficiency, lower levels of isolation, such as RU and RC, are routinely used in real-world transaction mixes. On the other hand, results which violate integrity constraints, even those expressed via triggers or within application programs, are almost never acceptable. Separating the two, and providing checks for full serializability only when necessary, provides an avenue for much more efficient support for long-running and interactive transactions.

The central topic of this paper is an augmentation of SI, named *constraint-preserving SI (CPSI)*, which ensures that all integrity constraints will be satisfied. The isolation level which it enforces is strictly weaker than that guaranteed by SSI, in that nonserializable behavior which does not result in constraint violation is not ruled out. To understand the scope of this approach, it is important to recognize that integrity constraints may be partitioned into two groups. First, there are the *internal constraints*, which are declared using the DDL (data definition language) of the DBMS itself, such as key constraints; these are always enforced internally by the system. Second, there are the *extended constraints*, which fall outside of the DDL and are expressed typically via update triggers or directly within the application code of the transactions. While the theory of CPSI is applicable to both classes of constraints, current relational DBMSs already enforce constraints which lie in the first group completely in a very efficient manner, so that no constraint violation can occur under SI. For a further discussion, see Summary 2.4. Consequently, the techniques of this paper are addressed primarily towards the second group, which are particularly common in interactive business processes, but which may fail to be enforced completely under SI unless additional measures are taken.

An example will help illustrate the idea. Let m_1 and n_1 each be positive integers with $m_1 \geq 1$ and $n_1 \geq 2$, and let $\mathbf{E}_1$ be the database schema with two $m_1 \times n_1$ arrays of integer-valued data objects $\{d_{ij} \mid (0 \leq i \leq m_1 - 1) \wedge (0 \leq j \leq n_1 - 1)\}$ and $\{e_{ij} \mid (0 \leq i \leq m_1 - 1) \wedge (0 \leq j \leq n_1 - 1)\}$. For each i with $0 \leq i \leq m_1 - 1$, let φ^d_{1i} be the constraint defined by $\sum_{j=0}^{n_1-1} d_{ij} \geq 1000$, let φ^e_{1i} be the constraint defined by $\sum_{j=0}^{n_1-1} e_{ij} \geq 1000$, and assume that $\mathbf{E}_1$ is constrained by the set $\{\varphi^d_{1i} \mid 0 \leq i \leq m_1\} \cup \{\varphi^e_{1i} \mid 0 \leq i \leq m_1\}$. In concrete terms, think of each d_{ij} and each e_{ij} as the balance in a bank account, with each row identifying a set of related accounts. The constraints φ^d_{1i} and φ^e_{1i} mandate that that the total balance in each row of related accounts be at least 1000. Now, for $0 \leq i \leq m_1 - 1$, $0 \leq j \leq n_1 - 1$, let τ^d_{1ij} be the transaction which executes the assignment $d_{ij} \leftarrow d_{ij} - e_{ij}$ if $d_{ij} + d_{i(j+1) \bmod n_1} - e_{ij} \geq 1000$ holds, and the assignment will not result in a constraint violation. If either of these conditions fails, the transaction executes the identity update. Thus, the assignment, which may be thought of as a withdrawal of the value of e_{ij} from account d_{ij}, is executed iff both $d_{ij} + d_{i(j+1) \bmod n_1} - e_{ij} \geq 1000$ and $\sum_{k=0}^{n_1-1} d_{ik} > 1000 + e_{ij}$ hold before the update is executed. (Both conditions are necessary; the first does not imply the second since the balances may be negative.) Similarly, let τ^e_{1ij} be the transaction which executes the assignment $e_{ij} \leftarrow e_{ij} - d_{ij}$ if $e_{ij} + e_{i(j+1) \bmod n_1} - d_{ij} \geq 1000$ holds, and the assignment will not result in a constraint violation. Otherwise, the transaction executes the identity update. These transactions form the basis for a generalization of the write-skew example of [2].

Each transaction performs two distinct flavors of read. Consider τ^d_{1ij}. First, it must read both $d_{i(j+1) \bmod n_1}$ and e_{ij} (as well as the initial-snapshot value of d_{ij}) in order to determine which ground update to perform. A *ground update* is a specific change on a specific (set of) data objects. For example, if the current database state has $d_{ij} = d_{i(j+1) \bmod n_1} = 1000$ and $e_{ij} = 600$, then the ground update for τ^d_{1ij} (to be performed if there are no constraint violations) is $1000 \overset{d_{ij}}{\rightsquigarrow} 400$, indicating that the value of d_{ij} is to be changed from 1000 to 400. Since this ground update decreases the value of d_{ij}, the transaction must read every element in $h^d_{1ij} = \{d_{ik} \mid 0 \leq k \leq m_1\} \setminus \{d_{ij}\}$ in order to determine whether the constraint will remain satisfied after the update. (The values of the d_{ij} may be negative, so even if $d_{ij} + d_{i(j+1) \bmod n_1}$ is a large positive sum, there is no way to guarantee that φ^d_{1i} will hold without reading all values involved in the sum, except for d_{ij}, which will be modified.) The set $\{d_{i(j+1) \bmod n_1}, e_{ij}\}$ is called the *grounding context* of τ^d_{1ij} while h^d_{1ij} is called its *integrity context*, for the given ground update. An analogous construction applies to each τ^e_{ij}. For technical reasons, the data objects which are potentially written are never included in these contexts.

This distinction makes it possible to classify read-write dependencies in the DSG, and retain only those which can lead to a constraint violation. For example, since the transaction τ^d_{1ij} must always read e_{ij} and τ^e_{1ij} must always read d_{ij}, for any state from which both transactions will perform a write, they form a loop in the DSG when run concurrently, as illustrated in Fig. 1.2. While such a loop can (and in this instance does) signal a loss of serializability, it cannot, by itself, signal a possible constraint violation. Only cycles in the DSG which involve reads of the integrity context can result in constraint violations. A *guard function* for a transaction assigns to each state of the database a set of data objects, called the *guard object*, which are sufficient to read in order to determine whether the ground update associated with that state (as the initial snapshot of the transaction) will result in a constraint violation. For example, in the context of τ_{ij}, any ground update which reduces the value of d_{ij} has h^d_{1ij} as its unique minimal guard object, while any ground update which increases the value of d_{ij} or leaves it unchanged has the empty set as unique minimal guard object (since an increase in the value of d_{ij} can never result in a constraint violation).

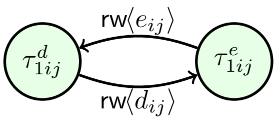

$\mathsf{rw}\langle e_{ij}\rangle$
τ^d_{1ij}
τ^e_{1ij}
$\mathsf{rw}\langle d_{ij}\rangle$

Figure 1.2: An SI rw-conflict cycle of length 2 which cannot result in a constraint violation

The central result of this paper is that if a schedule of transactions running under SI is not constraint preserving, then there is some pair of concurrent transactions for which each writes the guard object of the other. Thus, the only case for which an update of the transaction τ_{ij} can cause a constraint violation is if there is a concurrent transaction of the form τ_{ik} with $j \neq k$ and loop of the form shown in Fig. 1.3. In that figure, the edge labelled $\mathsf{gw}\langle d_{ij}\rangle$ from τ^d_{1ij} to τ^d_{1ik} indicates both that the data object d_{ij} is in the guard of τ^d_{1ij} and that τ^d_{1ik} is the first transaction to write d_{ij} after τ^d_{1ij} starts. It is thus similar to an rw-edge, save for that the read of the data object d_{ij} is replaced by its presences in the guard of τ^d_{1ij}. For the possibility of a constraint violation to exist, there must be two concurrent transactions, each connected to the other via gw-edges. Such a cycle defined by two gw-edges is called a *gw-conflict.*

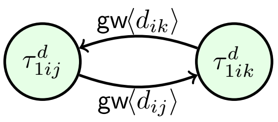

$\mathsf{gw}\langle d_{ik}\rangle$
τ^d_{1ij}
τ^d_{1ik}
$\mathsf{gw}\langle d_{ij}\rangle$

Figure 1.3: An SI gw-conflict cycle of length 2

CPSI uses gw-conflicts as its basis. While a gw-conflict involves only two transactions, as opposed to three for a dangerous structure of SSI, its major advantages lie elsewhere. First of all, using gw-conflicts places a bound on which transactions may conflict under SI. For example, in the context of $\mathbf{E}_1$, the only pairs of transactions from the set $\{\tau^x_{ij} \mid (x \in \{d, e\}) \wedge (0 \leq i \leq m_0 - 1) \wedge (0 \leq j \leq n_0 - 1)\}$ which have any possibility to conflict are of the form $\{\tau^x_{ij_1}, \tau^x_{ij_2}\}$, with $x \in \{d, e\}$ and $j_1 \neq j_2$. No similar characterization is possible when full serializability is required, since there are no underlying constraints on the schema which limit concurrency. Second, and central to the effective management of long-running and interactive transactions, a gw-conflict provides specific information which may be used in support of a cooperative mode between transactions, to allow them to decide whether the conflict can actually result in a constraint violations for the proposed updates, and to negotiate changes if need be. In that context, such a strategy is often far preferable to abort and rerun.

If the transactions in a schedule read the entire guard of the ground updates which they execute, then this test will be free of false positives. However, if there are circumstances under which a transaction can get away with reading only part of the guard, there may be gw-loops without the possibility of constraint violation. For such false positives to be possible, two things must hold. First of all, the integrity constraints must have a certain disjunctive character. Second, the transactions must be conditional in a way which allows them to verify integrity by looking at only part of the guard.

The schema $\mathbf{E}_1$ does not exhibit the necessary disjunctive character of its constraints, so, regardless of the update, the entire associated guard object must be read. To illustrate the kind of constraint which is required, consider the schema $\mathbf{E}'_1$, which is identical to $\mathbf{E}_1$, except that all data values are now required to be nonnegative. In this context, the potential update of the transaction τ^d_{1ij} always satisfies the integrity constraint. Indeed, if $d_{ij} + d_{i(j+1) \bmod n_1} - e_{ij} \geq 1000$ holds before the update $d_{ij} \leftarrow d_{ij} - e_{ij}$, then $d_{ij} + d_{i(j+1) \bmod n_1} \geq 1000$ will hold afterwards. Since the values of the other d_{ik}s must be nonnegative, that is enough to guarantee that φ^d_{1i} is satisfied. Thus, $\{d_{i(j+1) \bmod n_1}\}$ is a sufficient integrity context for this transaction when run on $\mathbf{E}'_1$. It is the basis for what is called a *conditional guard function*. A conditional guard is not sufficient, in general, to determine whether a ground update will result in a constraint violation, but it is sufficient to make that determination within the specific context that the ground update is applied.

Detecting constraint violation in $\mathbf{E}'_1$ via the presence of a gw-loop need no longer be free of false positives. To illustrate, fix i with $0 \leq i \leq m_0$ and consider the set $\mathbf{T}_{10} = \{\tau^d_{1ij} \mid 1 \leq j \leq n_1 - 1\}$, run in the context of $\mathbf{E}'_1$. If any proper subset of $\mathbf{T}_{10}$ is run concurrently, there will be no constraint violation. On the other hand, if the entire set is run concurrently, constraint violation is possible. To see this, suppose that the transactions commit in the order $\tau^d_{i0}, \tau^d_{i1}, \tau^d_{i2}, \ldots, \tau^d_{i(n_1-1)}$. The entire situation is depicted in Fig. 1.4; note in particular that there is a gw-dependency between every pair of distinct transactions. Remember that a guard function needs to be extensive enough to guarantee the correctness of the associated ground update, regardless of whether or not it is applied conditionally.

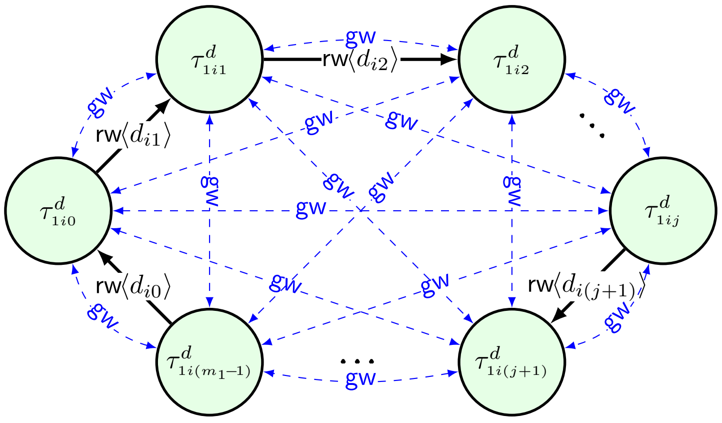

τ^d_{1i1}
τ^d_{1i2}
τ^d_{1i0}
τ^d_{1ij}
$\tau^d_{1i(m_1-1)}$
$\tau^d_{1i(j+1)}$
rw$\langle d_{i2}\rangle$
rw$\langle d_{i1}\rangle$
rw$\langle d_{i0}\rangle$
rw$\langle d_{i(j+1)}\rangle$
gw

Figure 1.4: An SI constraint-conflict cycle of length n_0 with a gw-dependency between every pair of vertices

To see how a constraint violation may arise, let M_{10} be the database with $d_{ij} = e_{ij} = 1000$ for all i, j with $1 \leq i \leq m_1 - 1$ and $i \leq j \leq n_1 - 1$. If any proper subset of T_{10} is run concurrently on M_{10}, there will be no constraint violation, since at least one of the d_{ij}s will be left at 1000. If all n_0 transactions are run concurrently, the resulting state will violate φ^d_{1i}, since in the resulting database, it will be the case that $d_{ij} = 0$ for all $0 \leq j \leq n_0 - 1$. Thus, for any positive integer n, it is possible to find a set of n transactions which is not constraint preserving, while any proper subset is. Hence, CPSI can produce false positives if transactions do not read the associated full guard objects.

Although CPSI may involve false positives, as illustrated by the example above, that is not as large a limitation as might first appear. In major database systems, while transactions are responsible for carrying out their assigned tasks correctly, the system itself ensures that the internal integrity constraints are maintained. A similar design strategy seems appropriate for extended constraints and CPSI. The most direct way to support extended constraints in a schema-wide fashion is via triggers. However, triggers check the correctness of ground updates; they do not, and in general can not, take into account the way in which the proposed update was obtained. Thus, it is difficult, if not impossible, to implement conditional guards via triggers. This means that the trigger must read the entire (ordinary) guard and, in that case, CPSI will not produce false positives, This issue is discussed further in Discussion 6.4, together with some "tricks" which a trigger can use to limit its reads. Nevertheless, in most practical implementations, a trigger will read the entire guard associated with the ground update to be verified for integrity.

Beyond that, it must be remembered that CPSI is designed for mixes of long-running transactions which may involve human interaction. In that setting, as already noted, CPSI offers more specific information than does SSI about conflict, information which will be critical to any process of resolution which involves further examination and negotiation, as opposed to abort and rerun. Also, the typical case is rarely the worst case. Although Fig. 1.4 may appear condemning, it is far more likely that only a few of the n_0 transactions will ever run concurrently. In that case, an approach in which sets of conflicting transactions can negotiate an abort-free solution and proceed with their tasks will likely prove to be feasible. In other words, false positives are far less damaging if their premises may be examined, and then the behavior of the transactions altered, than if a false positive always involves discarding the work of at least one transaction completely.

Finally, CPSI and SSI involve independent tests; there is no reason why both cannot be used, with a result of fewer false positives than either produces by itself. This observation leads to the second contribution of this paper, the development of an an enhanced CPSI in which basic CPSI is combined with a special version of SSI which is fine tuned to the task of detecting constraint violation only.

It is worth noting that there is another proposed solution to supporting SI with constraint preservation, namely *SI+IC* of [17, Sec. 4]. It is similar in isolation level to CPSI in that it augments SI in order to ensure constraint preservation, without guaranteeing fully serializability. However, in contrast to CPSI, it does not distinguish internal constraints from those which are external; rather, it essentially requires that all constraints be managed internally by the DBMS. Whenever an update is to be performed by a transaction, it checks which other updates have already been performed, on local data copies, by concurrent transactions, in order to be certain that no constraint violation can occur, just as is done by current DBMSs for internal constraints. (See Summary 2.4 for more information.) This approach offers the obvious advantage of providing a uniform solution which avoids essentially all false positives. However, the price is that the underlying DBMS must be modified to support all types of constraints which are used. In the case of those of constraints of the form φ^d_{1i} on the schema $\mathbf{E}_1$, which are not supported by current relational DBMSs (except via triggers), a major augmentation of the DBMS would be required. On the other hand, as presented more fully in Discussion 5.25, extending a current relational DBMS to support CPSI is a much more modest endeavor.

This paper is based upon [13], but has been revised and extended greatly. In terms of the overall approach, there are three fundamental changes. First of all, the approach of [13] is to use views of the main schema as database objects, in order to present the results within a very general context. While this allows the use of very general data objects, for example, projections on relations, it also results in a formulation which is difficult to understand without specialized knowledge in mathematics and database theory. In this paper, the main results are presented within a more typical context for transactions, in which a database schema is defined by a set of mutable objects together with some integrity constraints. Although there is some sacrifice in terms of generality, the results remain more than general enough to apply to real systems, while becoming much more understandable.

Secondly, the results of this paper generalize those of [13] in a very significant way. The work of [13] is based upon a *static* model of transaction reads and writes; the data items which are read and written are fixed for all executions of the transaction, regardless of the initial snapshot. While this approach is suitable for simple examples, it is not a realistic reflection of the behavior of real transactions, which determine their read and write sets dynamically, based upon the input (qua initial snapshot). In this paper, a *dynamic* model is employed for modelling the reads and writes of all transactions, while retaining all of the results of the earlier paper.

Lastly, the presentation in [13] does not address the issue of false positives in CPSI. In this paper, that topic is discussed in some detail, and a new enhanced CPSI which, as noted above, combines ordinary CPSI with a special version of SSI, is developed.

The organization of the paper is as follows. Sec. 2 provides an informal overview of snapshot isolation and its implementation, while Sec. 3 provides the basic database and transaction framework upon which the work is based. In Sec. 4, a formal model of snapshot isolation is developed, upon which the main theory of the paper is based. In Sec. 5, the main theory of CPSI is developed, while in Sec. 6, some extensions of CPSI which involve combining it with SSI, in order to reduce the occurrence of false positives, are presented. Finally, Sec. 7 provides some conclusions and further directions.

2 An Overview of Snapshot Isolation

The purpose of this section is to provide an informal description of snapshot isolation with a particular focus upon significant differences between its implementation in actual systems and the formal model which is used here. It is assumed that the reader has a basic understanding of transactions, and in particular serializability, as presented in, for example, [23], [3], and Chapters 14-15 of the textbook [22].

Summary 2.1 (SI and Concurrency of transactions) Before presenting the theory, it is appropriate to sketch the model of snapshot isolation (hereafter *SI*) which is used. A *transaction* performs read and write operations on data objects. Each transaction has a start time, as well as an end time at which its writes are *committed* to the database. Two transactions are *concurrent* if the start time of one lies between the start and end times of the other. Concurrency is not transitive; if T_1 is concurrent with T_2, and T_2 is concurrent with T_3, then T_1 and T_3 need not be concurrent. Therefore, while it makes sense to speak of a pair of transactions being concurrent, stating that a larger set of transactions is concurrent requires further clarification.

In SI, the transaction T always operates on a private copy of the database, called the *snapshot*, taken at the start time of the transaction. While it is running, it does not see updates performed by other (necessarily concurrent) transactions, and those other transactions do not see its updates. Thus, with SI, readers are never in conflict with concurrent writers.[5] The only kind of conflict which must be resolved is between two (or more) writers of the same data object. Before a transaction commits, its updates must be written to the stable database. In general, a transaction is never allowed to overwrite the value of a data object which was written by another transaction with which it ran concurrently. There are several ways in which such conflicts may be resolved; the two most important are considered next.

Summary 2.2 (The FCW model of snapshot isolation) With the *first-committer wins (FCW)* rule, transactions under SI run, up to their commit point, with no awareness of the behavior of other, concurrent transactions. The check for conflict is made at commit time. When a transaction T has finished and its writes must be committed to the stable database, it is allowed to commit only if no other concurrent transaction T' has written a data object which which T has written. If any other transaction T' which is concurrent with T, and which has already committed has written a data object which T has also written, then T is not allowed to commit.

FCW is particularly appealing from a theoretical point of view, because it is a perfect match of the black-box model of a transaction, to be presented in Definition 3.12 below. In that model, the order and timing of the internal operations are not modelled, and FCW does not require any knowledge of those internal operations.

Summary 2.3 (The FUW model of snapshot isolation) The rule for conflict resolution which is commonly used in practice is called *first updater wins (FUW)*. With FUW, if some other concurrent transaction T' writes a data object which T later is to write, then T is blocked from continuing to operate, even on its private copy, until T' commits (in which case T is not allowed to continue) or T' aborts (in which case T is allowed to continue).

FUW offers the advantage (over FCW) that conflicts are detected and addressed earlier in the execution of a transaction, and so appropriate action may be taken at an earlier point in time, before a transaction performs a long sequence of operations for which it is already known (or at least likely) that they cannot be committed. Unfortunately, the black-box model of transaction is not suitable for modelling SI under FUW, since the timing of internal operations, which is not retained in the black-box model, is of importance.

Fortunately, for the purposes of this work, while FCW and FUW differ in implementation, they are identical in the definition of a conflict; namely, that concurrent transactions may not both write the same data object. They furthermore produce identical results when there is no write conflict. Thus, for a study of conflict and constraint violation, FCW may be used in lieu of FUW with no loss of generality. FCW will be used in this paper because it admits a simpler conceptual model of a transaction which does not involve the order or points in time at which the transaction performs internal operations on its private copy.

Summary 2.4 (Enforcement of integrity constraints under SI in practice) In most implementations of SQL under FUW, constraints which may be specified using the DDL (data definition language), particularly primary-key and uniqueness constraints, are enforced immediately, and unless checking is declared to be `deferrable` and then `deferred`, foreign-key constraints are enforced immediately as well. These checks incorporate even pending updates

[5] This is not quite true in practice. See Summary 2.4 for an elaboration.

made by other concurrent transactions to their local snapshots. The reason for these immediate checks is to avoid the need to verify constraints at commit time, possibly against updates made by other, concurrent transactions. A rather comprehensive formal model of how such constraints are handled in real systems may be found in [17, Sec. 4].

This internal management of integrity constraints might appear to limit the applicability of the results of this paper, since such constraint violations will already be detected by the constraint-maintenance process, and need not be handled separately by the transaction manager. However, this applies only to built-in constraints. In many applications, particularly business processes, complex constraints are often implemented via triggers and even in application programs. Such "implemented" constraints are not covered by this process. In SI, a trigger will execute immediately upon execution of the update on the local snapshot; updates made by other concurrent transactions will not be taken into account. On the other hand, the ideas developed in this paper are applicable to integrity constraints implemented by triggers and even by application programs.

3 Data Objects, Schemata, Updates, and Transactions

A characterizing feature of the work of this paper, as compared to most other work on isolation of transactions, is the central rôle which the preservation of integrity constraints plays. For an effective treatment, this mandates the use of more elaborate and formal models of both database schemata and of transactions themselves than is necessary in studies of transactions which do not address integrity constrains explicitly. The purpose of this section is to establish the necessary formal framework. It is assumed that the reader has basic understanding of database systems in general, and of the relational model in particular, as presented in standard textbooks such as [7], [22], and [15]. It is also assumed that the reader has a basic understanding of database transactions, as presented in those textbooks.

Notation 3.1 (Some mathematical shorthand) Before presenting the notions which are DBMS specific, it is useful to establish some mathematical notation of a more general nature.

It will often be necessary to assert that a partial function f is defined on an argument x. The shorthand $f(x)\!\downarrow$ will be used in this regard. In order to avoid the need to state independently that a partial function is defined on an argument, a statement such as $f(x)\!\downarrow\in Y$ will be used to indicate that both $f(x)\!\downarrow$ and $f(x) \in Y$. Similarly, $f(x)\!\uparrow$ denotes that f is undefined on x. If f_1 and f_2 are both partial functions, a statement of the form $f_1(x) = f_2(x)$ means that either both $f_1(x)\!\downarrow$ and $f_2\!\downarrow\!(x)$ hold, and these two values are the same, or else both $f_1(x)\!\uparrow$ and $f_2(x)\!\uparrow$ hold.

$\mathbb{N}$ denotes the set $\{0, 1, 2, \ldots\}$ of natural numbers, while $\mathbb{Z}$ denotes the set of all integers, positive, negative, and zero.. For $i, j \in \mathbb{Z}$, $[i, j]$ denotes the set $\{i, i + 1, \ldots, j\}$ of integers between i and j inclusive, while $[i, \text{-}]$ denotes the set of all integers which are greater than or equal to i.

$\mathsf{Card}(X)$ denotes the cardinality of the set X.

For composition of functions, the convention $(f \circ g)(x) = g(f(x))$ is followed. That is, $f \circ g$ means apply f first, and then g.

Definition 3.2 (Data objects and unconstrained database schemata) In the study of concurrent transactions and isolation, database schemata are typically modelled as sets of *mutable objects*; that is, objects which have a value which may be altered. These mutable objects are often called *data objects* or *data items* [1], [5], [7, Sec. 21.1.2], [22, Sec. 14.2]. Formally, a *simple data object* x is characterized by a set $\mathsf{States}\langle x\rangle$, the *states* of x.

A *compound data object* is a set $\mathbf{x}$ of simple data objects. A *database* over $\mathbf{x}$ is a function $M : \mathbf{x} \to \bigcup_{x\in\mathbf{x}} \mathsf{States}\langle x\rangle$ with the property that $M(x) \in \mathsf{States}\langle x\rangle$ for each $x \in \mathbf{x}$. The set of all databases over $\mathbf{x}$ is denoted $\mathsf{DB}(\mathbf{x})$.

For technical reasons, it is convenient to allow a compound data object to be the empty set $\emptyset$. In that case $\mathsf{DB}(\emptyset)$ is an empty function; that is, a function on domain $\emptyset$. There is only one such function, so the empty database object has just one possible database, which will be denoted by ϕ_{DB}.

Note that each simple data object x may be regarded as a compound data object $\{x\}$. The term *data object*, without qualification, will mean either a simple data object (via this identification) or a complex data object.

An *unconstrained database schema* $\mathbf{D}$ is given by a complex data object $\mathsf{DObj}\langle\mathbf{D}\rangle$. A *database* of $\mathbf{D}$ is a database over $\mathsf{DObj}\langle\mathbf{D}\rangle$. The set of all databases of $\mathbf{D}$ is denoted $\mathsf{DB}(\mathbf{D})$. Thus, $\mathsf{DB}(\mathbf{D})$ is shorthand for $\mathsf{DB}(\mathsf{DObj}\langle\mathbf{D}\rangle)$.

Definition 3.3 (Constrained database schemata) Database schemata are almost always constrained, in the sense that only certain databases are *legal*. Typically, whether or not a database $M \in \mathsf{DB}(\mathbf{D})$ of the schema $\mathbf{D}$ is legal is determined by a set $\mathsf{Constr}(\mathbf{D})$ of *(integrity) constraints*, with M legal iff $M \models \varphi$ for every $\varphi \in \mathsf{Constr}(\mathbf{D})$; i.e., iff M is a model of every member of $\mathsf{Constr}(\mathbf{D})$. For the purposes of this work, it does not matter how the constraints are specified. However, how they are enforced is significant when implementation is considered. As elaborated in Summary 2.4, some constraints, typically those which are specified via the data-definition language (DDL), are enforced internally by the DBMS. In the relational setting with SQL, these include functional and foreign-key dependencies, as well as others which may be specified via SQL directives such as `CHECK` [6, Sec. 14.4]. More general constraints, which are defined using programs known as *triggers* [22, Sec. 5.3], [7, Sec. 26.1], [15, Chap. 7], are not enforced internally. This distinction is significant because while constraints which are specified in the DDL are enforced even under classical snapshot isolation, those specified via triggers or other means which lie outside of the DDL are not. In considering any implementation, it is useful to separate these two flavors of constraints in the formalism, since only those constraints which are not implemented internally require special attention in order to obtain a correct and satisfactory model of constraint preservation under snapshot isolation.

Formally, a *constrained database schema* is a triple $\mathbf{D} = \langle \mathsf{DObj}\langle\mathbf{D}\rangle, \mathsf{LDB}(\mathbf{D}), \mathsf{ELDB}(\mathbf{D})\rangle$ in which $\mathsf{DObj}\langle\mathbf{D}\rangle$ is a set of data objects, $\mathsf{LDB}(\mathbf{D})$ is a subset of $\mathsf{DB}(\mathsf{DObj}\langle\mathbf{D}\rangle)$, called the set of *legal databases* of $\mathbf{D}$, and $\mathsf{ELDB}(\mathbf{D})$ is a subset of $\mathsf{LDB}(\mathbf{D})$, called the set of *extended legal databases*, or *x-legal databases*, of $\mathbf{D}$. The notation $\mathsf{DB}(\mathbf{D})$, as shorthand for $\mathsf{DB}(\mathsf{DObj}\langle\mathbf{D}\rangle)$, will be used for constrained database schemata as well. Thus, $\mathsf{ELDB}(\mathbf{D}) \subseteq \mathsf{LDB}(\mathbf{D}) \subseteq \mathsf{DB}(\mathbf{D})$.

Although constraints themselves are not part of the formal model, it will nevertheless be useful to be able to classify them, since they will be used in examples. To this end, an *internal constraint* is one which is used to determine which databases are in $\mathsf{LDB}(\mathbf{D})$, while an *extended constraint* is one which is used to determine which databases are in $\mathsf{ELDB}(\mathbf{D})$. Thus, in a typical setting, the internal constraints are those implemented via the DDL and enforced even under SI, while the extended constraints are those implemented via triggers, and so not enforced under unaugmented SI.

Unless stated otherwise, all database schemata in this work are assumed to be constrained. Thus, *database schema* is a synonym for *constrained database schema*.

Discussion 3.4 (Granularities in the relational model) Since the relational model is ubiquitous in database systems, it is instructive to indicate the most common ways in which data objects may be modelled.

To begin, assume that $\mathbf{R}$ is a relational schema with relation names $\{R_1, R_2, \ldots, R_m\}$. The simplest model in terms of data objects uses *relation-level granularity*. In this model, for each $i \in [1, m]$ there is one simple data object $\mathsf{RelObj}\langle R_i \rangle$ with the set $\mathsf{States}\langle \mathsf{RelObj}\langle R_i \rangle \rangle$ consisting of possible relations for R_i. Integrity constraints are not enforced at the level of the data objects; in particular, there is no requirement that the elements of $\mathsf{States}\langle \mathsf{RelObj}\langle R_i \rangle \rangle$ satisfy any key constraints.

Although simple and natural, relation-level granularity has a serious shortcoming for transaction modelling. In virtually all models of transaction concurrency, overlap of transactions is measured in terms of mutual data objects for which at least one transaction is a writer. Thus, if transactions T_1 and T_2 both write some data object X, or if one writes X and the other reads X, it may not be possible to allow them to run concurrently under a given level of isolation. Relation-level granularity is very *coarse*, in the sense that there are only a few, very large data objects. From the point of view of increasing potential concurrency, is it advantageous to divide the database schema into a larger number of smaller data objects, resulting in a *finer* level of granularity. In the context of the relational model, there are several choices.

For *key-level granularity*, it must be assumed that each relation has a (unique by definition) primary key, which is always the case when a schema is defined using SQL. Then, rather than having a single simple data object for each relation, there is a simple data object for each key value of each relation. In effect, the single data object $\mathsf{RelObj}\langle R_i \rangle$ of relation-level granularity is divided up into distinct data objects, one for each possible value of the primary key. For simplicity, assume that this key is identified by a single attribute K_i. For an element $a \in \mathsf{Dom}\langle K_i \rangle$; i.e., a value a which lies in the domain of the attribute K_i, define the data object $\sigma_{K_i=a}\langle R_i \rangle$ as the selection on the instance of R_i of those tuples for which the value on attribute K_i is a. A bit of care is necessary in defining $\mathsf{States}\langle \sigma_{K_i=a}\langle R_i \rangle \rangle$, because this selection may be empty. In addition to each tuple t over R_i whose value for attribute K_i is a, $\mathsf{States}\langle \sigma_{K_i=a}\langle R_i \rangle \rangle$ must also include an additional state None which indicates that R_i does not contain a tuple with key value a. The set $\{\sigma_{K_i=a}\langle R_i \rangle \mid i \in [1, m] \wedge a \in \mathsf{Dom}\langle K_i \rangle\}$ of simple data objects then recaptures all of $\mathbf{R}$.

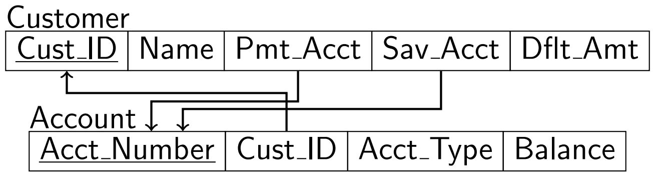
With *tuple-level granularity*, no keys are necessary. In this case, there is a simple data object for each possible tuple of each relation. More precisely, for each $i \in [1, M]$ and each possible tuple t of R_i, define the data object $\sigma_t\langle R_i \rangle$ via $\mathsf{States}\langle \sigma_t\langle R_i \rangle \rangle = \{\mathbf{true}, \mathbf{false}\}$. The state of $\sigma_t\langle R_i \rangle$ tells whether tuple t is present (**true**) in relation R_i or not present (**false**). In general, this granularity is even finer than key level, but if each relation has a primary key, it offers no advantage in terms of concurrency.

In some cases, for modelling conflict between transactions, real systems use *page-level granularity*, in which each simple data object consists of those tuples which are stored in the same physical page in the underlying DBMS. This model has obvious implementation advantages, but is more complex from a user and analysis point of view, since the storage pattern is not part of the logical relational model, and may in fact change over time. It will not be discussed further here.

The theory presented in this paper is independent of the level of granularity. It only requires that the database schema be modelled as a set of data objects. Of course, the choice of granularity for a given modelling instance may well affect the amount of concurrent behavior which is possible.

Examples 3.5 (An example relational schema and a useful simplification) It is useful to have a running example which may be used to illustrate the ideas developed in this paper. The example is first presented in the relational context, and then simplified to one which retains the essential features for illustrating CPSI while avoiding many of the details which characterize full transaction modelling within the relational context.

The relational schema, named $\mathbf{E}_2$, is depicted in Fig. 3.1. Primary keys are underlined and



For all Customer tuples c: c.Pmt_Acct->Balance + c.Sav_Acct->Balance ≥ 500

Figure 3.1: The example relational schema $\mathbf{E}_2$

foreign keys are shown via arrows, as is the convention in depicting relational schemata. There are two relations, one for customers and one for accounts. Each customer has an ID, a name, a payments[6] account Pmt_Acct, a savings account Sav_Acct, and a default transaction amount Dflt_Amt. An account has an identifying account number, an associated customer, a type, and a balance. For simplicity, no field in either relation may be null. Note that a customer may have more than two accounts, but it is the designated payments and savings accounts which are involved in the main extended constraint in this example. This constraint is displayed in the line under the diagram of Fig. 3.1, using in a notation borrowed from programming languages such as C. Specifically, the sum of the balances of the payment account and the savings account of each customer must be at least 500.

Under the current SQL standard, there is no way to express such a constraint within the DDL (data-definition language); it must be expressed using triggers or within application programs. In particular, as elaborated in Summary 2.4, while the primary- and foreign-key dependencies will be enforced by the relational system, constraints which are expressed via triggers, such as this one must be, are not, even under unaugmented SI. As a concrete example, consider the transaction τ_{2p}, which withdraws 100 from the payments account of a given customer c, and the transaction τ_{2s}, which withdraws 100 from the savings account of the same customer. Suppose that the sum of the balances is initially 600. Then each transaction, run in isolation, is correct, since the end balance will be 500. However, when the two transactions are run concurrently under SI, the result after both commit will be a sum of only 400, in violation of the constraint. This is the classical example of write skew, introduced in [2, Par. A5B].

[6] *Transactions* account is a more descriptive term, but to avoid any possible confusion with database transactions, the term *payments* has been chosen instead.

Although presentation of the full model of $\mathbf{E}_2$ for key-level granularity as described in Discussion 3.4 would be instructive, its use as a running example would involve a substantial amount of notational overhead. In particular, it would involve developing a language for extracting fields from data objects which are tuples, as well as for following pointers (represented as foreign keys). To keep the data model as simple as possible, with the goal of maintaining focus on the issues surrounding constraint preservation for concurrent transactions, the simplified schema $\mathbf{E}_3$, described in Table 1, is employed instead. This schema recaptures essential features

Name	Description
w	Finite set of active customers (read only)
x_c	There is a simple data object x_c for each $c \in \mathsf{States}\langle w \rangle$. x_c is the balance c.Pmt_Acct->Balance associated with customer c.
y_c	There is a simple data object y_c for each $c \in \mathsf{States}\langle w \rangle$. y_c is the balance c.Sav_Acct->Balance associated with customer c.
z_c	There is a simple data object z_c for each $c \in \mathsf{States}\langle w \rangle$. z_c is c.Dflt_Amt associated with customer c.

Constraint for each c occurring in w: $x_c + y_c \geq 500$

Table 1: The simple data objects of the schema $\mathbf{E}_3$

of full key-level granularity, as described in Discussion 3.4, while remaining simple enough to avoid the need to develop a formalism for extracting fields from simple data objects which are tuples. Both the set of customers (via w) and the set of accounts (via x and y) are fixed, so there is no need to recapture all possible tuples via the None idea described in Discussion 3.4. Furthermore, the associations of transaction and savings accounts to a given customer are fixed. Finally, values for customer name and account type are omitted; although useful in a real application, they are peripheral at best to the theory developed here. Observe in particular that the only constraint of $\mathbf{E}_3$, given at the bottom of Table 1, is in fact an extended constraint. Except for the requirement that balances be numbers, there are no other constraints, since the aspects of $\mathbf{E}_2$ which involve such constraints have been fixed (made unchangeable) in $\mathbf{E}_3$.

The focus now returns to the general case, with the above schema used in clarifying examples.

Notation 3.6 (Notational convention) Throughout the rest of this paper, unless stated specifically to the contrary, take $\mathbf{D} = \langle \mathsf{DObj}\langle \mathbf{D} \rangle, \mathsf{LDB}(\mathbf{D}), \mathsf{ELDB}(\mathbf{D}) \rangle$ to be a (constrained) database schema.

Definition 3.7 (Subschemata and subobjects) A *(syntactic) subobject* of $\mathbf{D}$ is any subset of $\mathsf{DObj}\langle \mathbf{D} \rangle$. Formally, define $\mathsf{SubObj}\langle \mathbf{D} \rangle = \mathbf{2}^{\mathsf{DObj}\langle \mathbf{D} \rangle} = \{\mathbf{y} \mid \mathbf{y} \subseteq \mathsf{DObj}\langle \mathbf{D} \rangle\}$. A database on a set of objects restricts naturally to a smaller set. Let $\mathbf{y} \subseteq \mathbf{x} \in \mathsf{SubObj}\langle \mathbf{D} \rangle$, and let $M \in \mathsf{DB}(\mathbf{x})$. The *restriction* of M to $\mathbf{y}$ is the database on $\mathbf{y}$ defined by $M_{|\mathbf{y}}$, the function M restricted to $\mathbf{y}$.

Every $\mathbf{y} \in \mathsf{SubObj}\langle \mathbf{D} \rangle$ defines a constrained database schema in a natural way. Specifically, the *database schema* defined on $\mathbf{y}$ from $\mathbf{D}$ is the triple $[\![\mathbf{D}|\mathbf{y}]\!] = \langle \mathbf{y}, \mathsf{LDB}\langle \mathbf{D}|\mathbf{y} \rangle, \mathsf{ELDB}\langle \mathbf{D}|\mathbf{y} \rangle \rangle$ in which $\mathsf{LDB}\langle \mathbf{D}|\mathbf{y}\rangle = \{M_{|\mathbf{y}} \mid M \in \mathsf{LDB}(\mathbf{D})\}$. and $\mathsf{ELDB}\langle \mathbf{D}|\mathbf{y}\rangle = \{M_{|\mathbf{y}} \mid M \in \mathsf{ELDB}(\mathbf{D})\}$. Thus, the legal (resp. extended legal) databases of $[\![\mathbf{D}|\mathbf{y}]\!]$ are precisely those members of $\mathsf{DB}(\mathbf{y})$ which are the restriction of some element of $\mathsf{LDB}(\mathbf{D})$ (resp. $\mathsf{ELDB}(\mathbf{D})$).

Definition 3.8 (Updates) A *(syntactic) update* on $\mathbf{D}$ is a pair $\langle M_1, M_2\rangle \in \mathsf{LDB}(\mathbf{D}) \times \mathsf{DB}(\mathbf{D})$. M_1 is the current or old state before the update, and M_2 is the new state afterwards. The set of all syntactic updates on $\mathbf{D}$ is denoted $\mathsf{SynUpdates}(\mathbf{D})$.

In a syntactic update, while the state before the update must be legal, the resulting state need not be. If the resulting state is legal, the update itself is called legal. Formally, $\langle M_1, M_2\rangle \in \mathsf{SynUpdates}(\mathbf{D})$ is *legal* if $M_2 \in \mathsf{LDB}(\mathbf{D})$. The set of all legal updates on $\mathbf{D}$ is denoted $\mathsf{LUpdates}(\mathbf{D})$. Observe that $\mathsf{LUpdates}(\mathbf{D}) \subseteq \mathsf{SynUpdates}(\mathbf{D})$.

The pair $\langle M_1, M_2\rangle \in \mathsf{LUpdates}(\mathbf{D})$ is *extended legal* (or *x-legal*) if $M_1, M_2 \in \mathsf{ELDB}(\mathbf{D})$. The set of all extended legal updates on $\mathbf{D}$ is denoted $\mathsf{ELUpdates}(\mathbf{D})$. Note that $\mathsf{ELUpdates}(\mathbf{D}) \subseteq \mathsf{LUpdates}(\mathbf{D})$.

Updates are often identified by name; therefore, it is convenient to have a shorthand for their components. To this end, if $u \in \mathsf{SynUpdates}(\mathbf{D})$, then write $u^{(1)}$ and $u^{(2)}$ for the values of the state before and after the update, respectively; i.e., $u = \langle u^{(1)}, u^{(2)}\rangle$. For $\mathbf{u} \subseteq \mathsf{SynUpdates}(\mathbf{D})$, define $\mathbf{u}^{(1)} = \{u^{(1)} \mid u \in \mathbf{u}\}$ and $\mathbf{u}^{(2)} = \{u^{(2)} \mid u \in \mathbf{u}\}$.

The set $\mathbf{u} \subseteq \mathsf{SynUpdates}(\mathbf{D})$ is *legal* if $\mathbf{u} \subseteq \mathsf{LUpdates}(\mathbf{D})$, and *extended legal* or *x-legal* if it is legal and, in addition, for every $u \in \mathbf{u}$, $u^{(2)} \in \mathsf{ELDB}(\mathbf{D})$ whenever $u^{(1)} \in \mathsf{ELDB}(\mathbf{D})$. In other words, $\mathbf{u}$ is x-legal if it is legal and it preserves x-legality.

A set of updates is complete if there is an update for every legal database of $\mathbf{D}$. Formally, $\mathbf{u} \subseteq \mathsf{SynUpdates}(\mathbf{D})$ is *complete* if for every $M \in \mathsf{LDB}(\mathbf{D})$, there is a $u \in \mathbf{u}$ with $u^{(1)} = M$.

The *composition* $u_1 \circ u_2$ of two updates $u_1, u_2 \in \mathsf{SynUpdates}(\mathbf{D})$ is just their composition in the sense of mathematical relations. More precisely, $u_1 \circ u_2 = \{(M_1, M_3) \mid (\exists M_2 \in \mathsf{LDB}(\mathbf{D}))((M_1, M_2) \in u_1 \wedge (M_2, M_3) \in u_2)\}$.

The restriction operator of Definition 3.7 extends naturally to updates. Specifically, for $u \in \mathsf{SynUpdates}(\mathbf{D})$, $\mathbf{y} \subseteq \mathsf{DObj}\langle \mathbf{D}\rangle$, define $u_{|\mathbf{y}} = \langle {u^{(1)}}_{|\mathbf{y}}, {u^{(2)}}_{|\mathbf{y}}\rangle$. For a set $\mathbf{u} \subseteq \mathsf{SynUpdates}(\mathbf{D})$ of updates, $\mathbf{u}_{|\mathbf{y}} = \{u_{|\mathbf{y}} \mid u \in \mathbf{u}\}$.

There is a form of selection on sets of updates which is also useful. For $M \in \mathsf{LDB}(\mathbf{D})$. define the *trimming* of $\mathbf{u}$ to M to be $\mathsf{Trim}_M\langle \mathbf{u}\rangle = \{u \in \mathbf{u} \mid u^{(1)} = M\}$. It will prove useful to extend this a bit more, to subschemata. If $\mathbf{y} \subseteq \mathbf{x} \subseteq \mathsf{DObj}\langle \mathbf{D}\rangle$, $M \in \mathsf{LDB}\langle \mathbf{D}|\mathbf{x}\rangle$, and $\mathbf{u} \subseteq \mathsf{SynUpdates}([\![\mathbf{D}|\mathbf{y}]\!])$, then $\mathsf{Trim}_M\langle \mathbf{u}\rangle$ is shorthand for $\mathsf{Trim}_{M_{|\mathbf{y}}}\langle \mathbf{u}\rangle$. In other words, if M involves a superset of the attributes of $\mathbf{u}$, then restrict M to the attributes of $\mathbf{u}$ before performing the trimming.

In general, for any $M \in \mathsf{LDB}(\mathbf{D})$, $\mathbf{u}(M)$ denotes $\{u^{(2)} \mid (u \in \mathbf{u}) \wedge (u^{(1)} = M)\} = \{u^{(2)} \mid u \in \mathsf{Trim}_M\langle \mathbf{u}\rangle\}$. Call an update set $\mathbf{u} \subseteq \mathsf{SynUpdates}(\mathbf{D})$ *functional* if for every $M \in \mathsf{LDB}(\mathbf{D})$, $\mathbf{u}(M)$ contains at most one element. Finally, for $\mathbf{M} \subseteq \mathsf{LDB}(\mathbf{D})$, define $\mathbf{u}(\mathbf{M}) = \{\mathbf{u}(M) \mid M \in \mathbf{M}\}$.

Definition 3.9 (Representation via ground updates) Although a database schema often consists of a very large number of data objects, any single transaction updates only a very few, leaving the rest unchanged. It is therefore useful to have a notation which identifies only the changes.

A *simple ground update* on $\mathbf{D}$ is a statement of the form $a \overset{x}{\rightsquigarrow} b$ in which $x \in \mathsf{DObj}\langle \mathbf{D}\rangle$ and $a, b \in \mathsf{States}\langle x\rangle$ with $a \neq b$. In this case, the simple update is said to *act on* x, *from* a *to* b. The semantics is clear; the data object x is updated from state value a to state value b. A *compound* *ground update* on $\mathbf{D}$ is a set S of simple ground updates, subject to the condition that distinct elements act on distinct data objects. The *domain* of S is the set of all data objects involved. More precisely, $\mathsf{Domain}\langle S\rangle = \{x \mid (\exists a)(\exists b)(a \overset{x}{\rightsquigarrow} b \in S)\}$.

Given a compound ground update S on $\mathbf{D}$, $\mathsf{Upd}_{\mathbf{D}}\langle S\rangle$ denotes the subset of $\mathsf{SynUpdates}(\mathbf{D})$ consisting of those updates which change the simple data objects in $\mathsf{Domain}\langle S\rangle$ according to the elements of S and leave all other simple data objects unchanged. More precisely, $\mathsf{Upd}_{\mathbf{D}}\langle S\rangle$ is the set of all $u \in \mathsf{SynUpdates}(\mathbf{D})$ with the following two properties.

(s-i) For each $a \overset{x}{\rightsquigarrow} b \in S$, $(u^{(1)})_{|\{x\}} = a$ and $(u^{(2)})_{|\{x\}} = b$.

(s-ii) For each $x \in \mathsf{DObj}\langle\mathbf{D}\rangle \setminus \mathsf{Domain}\langle S\rangle$, $(u^{(1)})_{|\{x\}} = (u^{(2)})_{|\{x\}}$.

Examples 3.10 (Representation of updates) Working within the context of the schema $\mathbf{E}_3$ introduced in Examples 3.5, let $c_1, c_2 \in \mathsf{States}\langle w\rangle$, and let $S_{31} = \{300 \overset{x_{c_1}}{\rightsquigarrow} 210, 300 \overset{x_{c_2}}{\rightsquigarrow} 260\}$. Then $\mathsf{Upd}_{\mathbf{E}_3}\langle S_{31}\rangle$ consists of all pairs $(M_1, M_2) \in \mathsf{LDB}(\mathbf{E}_3) \times \mathsf{DB}(\mathbf{E}_3)$ with the property that $M_1(x_{c_1}) = M_1(x_{c_2}) = 300$, $M_2(x_{c_1}) = 210$, $M_2(x_{c_2}) = 210$, and $M_1(x) = M_2(x)$ for all other $x \in \mathsf{DObj}\langle\mathbf{E}_3\rangle$.

These operations need not be applied to the entire main schema. For example, let $\mathbf{y}_{31} = \{x_{c_1}, y_{c_1}, x_{c_2}, y_{c_2}\} \subseteq \mathsf{DObj}\langle\mathbf{E}_3\rangle$. Then $\mathsf{Upd}_{[\![\mathbf{E}_3|\mathbf{y}_{31}]\!]}\langle S_{31}\rangle$ consists of all pairs $(M_1, M_2) \in \mathsf{LDB}([\![\mathbf{E}_3|\mathbf{y}_{31}]\!]) \times \mathsf{DB}([\![\mathbf{E}_3|\mathbf{y}_{31}]\!])$ with the property that $M_1(x_{c_1}) = M_1(x_{c_2}) = 300$, $M_2(x_{c_1}) = 210$, $M_2(x_{c_2}) = 260$, $M_1(y_{c_1}) = M_2(y_{c_1})$, and $M_1(y_{c_2}) = M_2(y_{c_2})$.

Choosing $\mathbf{y}_{32} = \{x_{c_1}, x_{c_2}\}$, $\mathsf{Upd}_{[\![\mathbf{E}_3|\mathbf{y}_{32}]\!]}\langle S_{31}\rangle$ contains just one update. This construction is central and is examined further in Definition 3.14 and Examples 3.15.

Definition 3.11 (Updateable objects) It is useful to have a notation which combines the update executed by a transaction together with the database context in which that update operates. To that end, define an *updateable object* over $\mathbf{D}$ to be a pair $\langle\mathbf{c}, \mathbf{u}\rangle$ in which $\mathbf{c} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$, called the *context*, and $\mathbf{u} \subseteq \mathsf{SynUpdates}([\![\mathbf{D}|\mathbf{c}]\!])$. The updateable object $\langle\mathbf{c}, \mathbf{u}\rangle$ is *functional*, (resp. *complete*, resp. *legal*, resp. *x-legal* precisely in the case that $\mathbf{u}$ has that property, and it is called *singleton* just in case $\mathbf{u}$ consists of just one update; i.e., $\mathbf{u} = \{u\}$ for some $u \in \mathsf{SynUpdates}([\![\mathbf{D}|\mathbf{c}]\!])$.

Updateable objects on all of $\mathbf{D}$ occur frequently in this work, so it is advantageous to have a special notation for them. The updateable object $\langle\mathsf{DObj}\langle\mathbf{D}\rangle, \mathbf{u}\rangle$ will be abbreviated to $\langle\mathbf{D}, \mathbf{u}\rangle$.

The trimming operation extends naturally to updateable objects. Specifically, for $\langle\mathbf{c}, \mathbf{u}\rangle$, $\mathbf{x} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$ with $\mathbf{c} \subseteq \mathbf{x}$, and $M \in \mathsf{LDB}\langle\mathbf{D}|\mathbf{x}\rangle$, define $\mathsf{Trim}_M\langle\langle\mathbf{c}, \mathbf{u}\rangle\rangle = \langle\mathbf{c}, \mathsf{Trim}_M\langle\mathbf{u}\rangle\rangle$.

The restriction operation of Definition 3.8 extends naturally to updateable objects as well. For $\mathbf{y} \subseteq \mathbf{c}$, define $\langle\mathbf{c}, \mathbf{u}\rangle_{|\mathbf{y}} = \langle\mathbf{y}, \mathbf{u}_{|\mathbf{y}}\rangle$.

Finally, it is convenient to have a notation for extracting the components of an updateable object. If $\mathbf{o}$ is an updateable object, then $\mathsf{DObject}\langle\mathbf{o}\rangle$ denotes the data object of $\mathbf{o}$ while $\mathsf{UpdSet}\langle\mathbf{o}\rangle$ denotes the set of updates of $\mathbf{o}$. More concretely, if $\mathbf{o} = \langle\mathbf{c}, \mathbf{u}\rangle$, then $\mathsf{DObject}\langle\mathbf{o}\rangle = \mathbf{c}$ and $\mathsf{UpdSet}\langle\mathbf{o}\rangle = \mathbf{u}$.

Definition 3.12 (Black-box transactions) In this work, a *black-box* model of transactions is employed in which the internal operations are hidden; just the interaction with the environment is modelled. Formally, a *black-box transaction* T over $\mathbf{D}$ is represented by an updateable object $\langle\mathbf{D}, \mathcal{U}_T\rangle$ which is functional, complete, and x-legal. This update represents precisely the updates which the transaction performs, when run in isolation.

The requirement that $\mathcal{U}_T$ be functional ensures that the transaction is deterministic; there is at most one action for each input. Completeness ensures that T is defined on all inputs (although it may execute the identity update for some). Extended legality is mandated by *consistency* condition of ACID [10] is satisfied; that is, T must produce an x-legal result whenever the input snapshot is x-legal.

A transaction, as defined in this paper, *always* executes a legal update whose result is furthermore x-legal if the input is. If a transaction must terminate for any reason (an abort, for example), then it is modelled as a complete transaction which perform the identity update. However, a transaction will accept legal inputs which are not x-legal; the x-legality of the output is not guaranteed in that case, although the ordinary legality is.

The set of all black-box transactions over $\mathbf{D}$ is denoted $\mathsf{BBTrans}_{\mathbf{D}}$.

Discussion 3.13 (A framework for expressing example transactions) The abstract representation of an update as an order pair, and transaction semantics as a set of x-legal updates, is an appropriate one for the theory presented here. Nevertheless, it is essential to have a more compact representation for examples. Real transactions are expressed in some sort of programming language, and in order to cover the general case, a Turing-complete language would be required. For the purposes of this paper, it is neither practical nor necessary to develop such a complete language. Rather, a simple language, based upon assignment statements and conditionals will be used. While far from Turing complete, it will prove to be more than adequate to express the examples necessary to illustrate the ideas of this paper. Since it is assumed that the reader is familiar with traditional imperative programming languages, the ideas will be sketched only briefly, with emphasis upon the special properties necessary to describe the behavior of transactions.

Taking $\mathbf{D}$ to be the database schema, the context of the language is a subschema $[\![\mathbf{D}|\mathbf{c}]\!]$. The variables of the language are just the simple data objects in $\mathbf{c}$. Expressions are formed in the usual way; for example, if $x, y \in \mathbf{c}$, then $x + y$ is a proper expression provided that x and y take numerical values. Programs are constructed using a combination of assignment statements and conditional statements. Rather than provide a formal grammar and semantics, the ideas will be illustrated via example.

Working within the context of $\mathbf{E}_3$ of Examples 3.5, and for the moment using the entire subschema $\mathbf{E}_3 = [\![\mathbf{E}_3|\mathsf{DObj}\langle\mathbf{E}_3\rangle]\!]$, consider the transaction τ_{31} defined by the following statement, in which $c_1, c_2 \in \mathsf{States}\langle w\rangle$.

$$\begin{aligned}\mathsf{if}\ (z_{c_1} < 100)\wedge(z_{c_2} < 50)\ \mathsf{then}\ &\{x_{c_1} \leftarrow x_{c_1} - z_{c_1},\ x_{c_2} \leftarrow x_{c_2} - z_{c_2}\}\\ \mathsf{else}\ &\{x_{c_1} \leftarrow x_{c_1} - 100,\ x_{c_2} \leftarrow x_{c_2} - 50\}\ \mathsf{endif}\end{aligned}$$

Recall in particular the constraints and names of data objects identified in Table 1. The semantics is similar to that for an imperative programming language. The main difference is that that the assignments are only performed if the result is x-legal. If it is not, then the identity update is performed instead. To illustrate via example, first consider the state $M_{311} \in \mathsf{ELDB}(\mathbf{E}_3)$ in which $x_{c_1} = 300$, $y_{c_1} = 300$, $z_{c_1} = 90$, $x_{c_2} = 300$, $y_{c_2} = 300$, $z_{c_2} = 40$. The values of the other data objects are not relevant. The *grounding* of this assignment to M_{311} is obtained by evaluating each expression against M_{311}. The result is the compound ground update $S_{31} = \{300 \overset{x_{c1}}{\rightsquigarrow} 210, 300 \overset{x_{c2}}{\rightsquigarrow} 260\}$ of Examples 3.10. The result is $M'_{311} \in \mathsf{ELDB}(\mathbf{E}_3)$ with $x_{c_1} = 210$, $x_{c_2} = 260$, and with the state of all other data objects the same as for M_{311}.

Next, consider the state $M_{312} \in \mathsf{ELDB}(\mathbf{E}_3)$ in which $x_{c_1} = 250$, $y_{c_1} = 300$, $z_{c_1} = 90$, $x_{c_2} = 300$, $y_{c_2} = 300$, $z_{c_2} = 40$. In this case, the *potential* compound ground update is $\{250 \overset{x_{c_1}}{\rightsquigarrow} 160, 300 \overset{x_{c_2}}{\rightsquigarrow} 260\}$. Were the assignment $250 \overset{x_{c_1}}{\rightsquigarrow} 160$, to be executed on M_{312}, the constraint $x_{c_1} + y_{c_1} \geq 500$ would be violated. Therefore, the update is not allowed; neither assignment is executed and the transaction performs the identity update. The set of *actual* compound ground updates is $\emptyset$.

Now consider the state $M_{313} \in \mathsf{LDB}(\mathbf{E}_3) \setminus \mathsf{ELDB}(\mathbf{E}_3)$ in which $x_{c_1} = 150$, $y_{c_1} = 300$, $z_{c_1} = 90$, $x_{c_2} = 300$, $y_{c_2} = 300$, $z_{c_2} = 40$. Such an initial snapshot M_{313} must be considered, even though it does not lie in $\mathsf{ELDB}(\mathbf{D})$, since in pure, unaugmented SI, there is a possibility that a transaction will need to process a state which is not x-legal. For the purposes of this work, it suffices to assume that the update of the transaction results in some legal state, not necessarily x-legal. The actual nature of that state is not relevant to this work, since the goal is to preserve x-legality, not to repair snapshots which are not x-legal. Nevertheless, a real transaction will almost certainly perform some subset of the elements of the ground update.

As a last example, consider the transaction τ_{32} defined by the simple assignment set $\{x_{c_1} \leftarrow y_{c_1}, y_{c_1} \leftarrow x_{c_1}\}$. The result is a swap of the values of x_{c_1} and y_{c_1}; evaluation is always in parallel. The order of evaluation of statements does not matter.

It should also be noted that these updates could be applied to $[\![\mathbf{E}_3|\mathbf{c}_{31}]\!]$ with $\mathbf{c}_{31} = \{x_{c_1}, x_{c_2}, y_{c_1}, y_{c_2}, z_{c_1}, z_{c_2}\}$; the states of the other data objects do not matter.

In summary, there are three basic principles which are always followed.

Parallel evaluation on the initial snapshot: All expressions are evaluated against the same initial database; no statement ever uses the result of another statement or a value which has been committed by a concurrent transaction.

No overlap: A program never assigns more than one value to any data object. Programs which would perform more than one assignment to a data object are not allowed.

Preservation of x-legality: If the input database is x-legal, then the program must preserve the property. If the statements would not preserve that property, they are not executed.

Definition 3.14 (Write sets and write trimming) Returning to the general context of a constrained schema $\mathbf{D}$, let $\langle \mathbf{c}, \mathbf{u} \rangle$ be an updateable object over $\mathbf{D}$. The *write set* of $\langle \mathbf{c}, \mathbf{u} \rangle$, denoted $\mathsf{WSet}\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle$, is the largest $\mathbf{x} \subseteq \mathbf{c}$ with the property that for every $x \in \mathbf{x}$, there is a $u \in \mathbf{u} \cap \mathsf{LUpdates}([\![\mathbf{D}|\mathbf{c}]\!])$ with $u^{(1)}{}_{|\{x\}} \neq u^{(2)}{}_{|\{x\}}$. In other words, $x \in \mathbf{c}$ is in the write set of $\langle \mathbf{c}, \mathbf{u} \rangle$ if there is some legal $u \in \mathbf{u}$ which alters the state of x. The *write updates* of $\langle \mathbf{c}, \mathbf{u} \rangle$, denoted $\mathsf{WUpd}\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle$, is $\mathbf{u}_{|\mathsf{WSet}\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle}$. The *write object* of $\langle \mathbf{c}, \mathbf{u} \rangle$, denoted $\mathsf{WObj}\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle$, is defined to be $\langle \mathsf{WSet}\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle, \mathsf{WUpd}\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle\rangle$. Call $\langle \mathbf{c}, \mathbf{u} \rangle$ a *full write object* if $\langle \mathbf{c}, \mathbf{u} \rangle = \mathsf{WObj}\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle$.

For $M \in \mathsf{LDB}(\mathbf{D})$, define the *write trim* of $\langle \mathbf{c}, \mathbf{u} \rangle$ to M as $\mathsf{WTrim}_M\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle = \langle \mathbf{c}, \mathbf{u} \rangle_{|\mathsf{WSet}\langle \mathsf{Trim}_M\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle\rangle}$. In words, the updateable object $\langle \mathbf{c}, \mathbf{u} \rangle$ is first trimmed to reflect just the updates which apply to M, and then it is restricted to just the write set of those updates. Clearly, $\mathsf{WTrim}_M\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle$ is always a full write object. In addition, if $\langle \mathbf{c}, \mathbf{u} \rangle$ is functional and complete, then $\mathsf{WUpd}\langle\langle \mathbf{c}, \mathbf{u} \rangle\rangle$ consists of exactly one update. The set of all full write objects over $\mathbf{D}$ is denoted $\mathsf{FWObjs}\langle \mathbf{D} \rangle$.

Write trimming is central to defining the action of a transaction under SI. Given a transaction T whose updateable object is $\langle \mathbf{D}, \mathcal{U}_T \rangle$ and whose initial snapshot is $M \in \mathsf{LDB}(\mathbf{D})$, the update which it performs is represented by the (single) update of the updateable object $\mathsf{WTrim}_M\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle$. All data objects not included in $\mathsf{WTrim}_M\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle$ are left unchanged.

Examples 3.15 (Write sets and write trimming) Return to the transaction τ_{31}, introduced in Discussion 3.13, and consider also the update representation of Examples 3.10. Then

$$\mathsf{WTrim}_{M_{311}}\langle\langle \mathbf{E}_3, \mathfrak{U}_{\tau_{31}}\rangle\rangle = \langle \mathsf{Domain}\langle S_{31}\rangle, \mathsf{Upd}_{[\![\mathbf{E}_3|\mathsf{Domain}\langle S_{31}\rangle]\!]}\langle S_{31}\rangle\rangle$$

with $S_{31} = \{300 \overset{x_{c_1}}{\rightsquigarrow} 210, 300 \overset{x_{c_2}}{\rightsquigarrow} 260\}$ and $\mathsf{Domain}\langle S_{31}\rangle = \{x_{c_1}, x_{c_2}\}$. On the other hand,

$$\mathsf{WTrim}_{M_{312}}\langle\langle \mathbf{E}_3, \mathfrak{U}_{\tau_{31}}\rangle\rangle = \langle \emptyset, \mathsf{Upd}_{[\![\mathbf{E}_3|\emptyset]\!]}\langle\emptyset\rangle\rangle$$

since no writes are performed by τ_{31} with initial state M_{312}. Since $M_{313} \notin \mathsf{ELDB}(\mathbf{E}_3)$, the value of $\mathsf{WTrim}_{M_{312}}\langle\langle \mathbf{E}_3, \mathfrak{U}_{\tau_{31}}\rangle\rangle$ is not fixed in the formal model, but rather depends upon the implementation.

Definition 3.16 (Liftings on updateable objects) The idea of extending an operation on a local context to a larger one is termed *lifting*; all the states of those simple data objects which are not part of the context remain fixed. More precisely, let $\langle \mathbf{c}, \mathbf{u}\rangle$ be an updateable object on $\mathbf{D}$, and let $\mathbf{x} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$ with $\mathbf{c} \subseteq \mathbf{x}$. The *lifting* of $\langle \mathbf{c}, \mathbf{u}\rangle$ from $[\![\mathbf{D}|\mathbf{c}]\!]$ to $[\![\mathbf{D}|\mathbf{x}]\!]$, also called just the *lifting* of $\mathbf{u}$ from $[\![\mathbf{D}|\mathbf{c}]\!]$ to $[\![\mathbf{D}|\mathbf{x}]\!]$, and denoted $\mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$, is the set of all $v \in \mathsf{SynUpdates}([\![\mathbf{D}|\mathbf{x}]\!])$ with the following two properties:

(ls-i) v agrees with some $u \in \mathbf{u}$ on $\mathbf{c}$: $\quad v_{|\mathbf{c}} \in \mathbf{u}$.

(ls-ii) v is the identity on all data objects of $\mathbf{x}$ which are not in $\mathbf{c}$: ${v^{(1)}}_{|\mathbf{x}\setminus\mathbf{c}} = {v^{(2)}}_{|\mathbf{x}\setminus\mathbf{c}}$.

Every update $u \in \mathbf{u}$ is embedded in some $v \in \mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ in the precise sense that $v_{|\mathbf{c}} = u$. Thus, no update in $\mathbf{u}$ is "lost" in the lifting process. Indeed, given $u \in \mathbf{u}$, $u^{(1)} \in \mathsf{LDB}\langle\mathbf{D}|\mathbf{c}\rangle$ by definition of syntactic update. So, by the definition of $\mathsf{LDB}\langle\mathbf{D}|\mathbf{c}\rangle$, there is some $M_1 \in \mathsf{LDB}(\mathbf{D})$ with $(M_1)_{|\mathbf{c}} = u^{(1)}$. Define $M_2 \in \mathsf{DB}(\mathbf{D})$ by $(M_2)_{|\mathbf{c}} = u^{(2)}$ and $(M_2)_{|\mathsf{DObj}\langle\mathbf{D}\rangle\setminus\mathbf{c}} = (M_1)_{|\mathsf{DObj}\langle\mathbf{D}\rangle\setminus\mathbf{c}}$. Then $\langle M_1, M_2\rangle_{|\mathbf{x}} = u$, and so $\langle M_1, M_2\rangle_{|\mathbf{x}} \in \mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$.

If $\langle \mathbf{c}, \mathbf{u}\rangle$ is functional, then so too is $\mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$. This case is important enough to warrant its own notation. Specifically, if $\langle \mathbf{c}, \mathbf{u}\rangle$ is functional, then $\mathsf{FLift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle : \mathsf{LDB}\langle\mathbf{D}|\mathbf{x}\rangle \to \mathsf{DB}(\mathbf{x})$ is the partial function defined by $M \mapsto M'$ if $\langle M, M'\rangle \in \mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ and is undefined otherwise. Note in particular that if $\mathbf{u}$ consists of a single update, then $\langle \mathbf{c}, \mathbf{u}\rangle$ is trivially functional, and so $\mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ is functional as well.

Because liftings to all of $\mathbf{D}$ occur often, it is convenient to introduce a simplified notation for them; $\mathsf{Lift}_{[\![\mathbf{D}|\mathsf{DObj}\langle\mathbf{D}\rangle]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ is abbreviated to $\mathsf{Lift}_{\mathbf{D}}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$, and $\mathsf{FLift}_{[\![\mathbf{D}|\mathsf{DObj}\langle\mathbf{D}\rangle]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ is abbreviated to $\mathsf{FLift}_{\mathbf{D}}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$.

Examples 3.17 (Lifting) Return to the transaction τ_{31}, introduced in Discussion 3.13 and continued in Examples 3.15. Let $\mathbf{y} \subseteq \mathsf{DObj}\langle\mathbf{E}_3\rangle$ with $\{x_{c_1}, x_{c_2}\} \subseteq \mathbf{y}$. Then

$$\mathsf{Lift}_{[\![\mathbf{E}_3|\mathbf{y}]\!]}\langle\mathsf{WTrim}_{M_{311}}\langle\langle \mathbf{E}_3, \mathfrak{U}_{\tau_{31}}\rangle\rangle\rangle = \langle \mathbf{y}, \mathsf{Upd}_{[\![\mathbf{E}_3|\mathbf{y}]\!]}\langle\{x_{c_1} \leftarrow 210,\ x_{c_2} \leftarrow 260\}\rangle\rangle$$

Lifting to all of $\mathbf{E}_3$ is of central importance. In particular,

$$\mathsf{Lift}_{\mathbf{D}}\langle\mathsf{WTrim}_{M_{311}}\langle\langle \mathbf{E}_3, \mathfrak{U}_{\tau_{31}}\rangle\rangle\rangle = \langle \mathbf{D}, \mathsf{Upd}_{\mathbf{D}}\langle\{x_{c_1} \leftarrow 210,\ x_{c_2} \leftarrow 260\}\rangle\rangle$$

Discussion 3.18 (The contexts of a transaction) A transaction T typically operates on only a small part of the database, sometimes called its *database context* [15, Sec. 19.2.1]. At the most fundamental level, there is the distinction between the *write context*, consisting of those $x \in \mathsf{DObj}\langle\mathbf{D}\rangle$ which the transaction actually writes, and the *read context*, consisting of those $x \in \mathsf{DObj}\langle\mathbf{D}\rangle$ which must be read in order to execute the transaction.

The write context is just another name for the write set, as defined in Definition 3.14.

Regarding the read context, there is furthermore an important subclassification, into the *grounding context* and the *integrity context.*[7] This idea has already been discussed in the introduction, but it is important enough to be considered in more detail here. Roughly speaking, the reads of data objects in the grounding context are for the purpose of determining which updates to apply, while reads of data objects in the integrity context are used to determine whether or not the grounded update will satisfy the extended integrity constraints. This distinction is crucial under constraint-preserving SI because while concurrent updates to data objects in the grounding context have no effect on the correctness, concurrent updates to the integrity context very much do. A read of the integrity context is called an *integrity read* while a read of the grounding context is called a *grounding read.*

These ideas are best illustrated by example. In the setting of $\mathbf{E}_3$ of Examples 3.5, fix $c \in \mathsf{States}\langle w\rangle$ and consider the transaction τ_{33} for which is defined by the assignment $x_c \leftarrow x_c - z_c$. In order to execute this assignment correctly, two distinct reads must be made. First of all, x_c and z_c must be read in order to determine the update to be considered. For example, if $x_c = 300$ and $z_c = 100$, then the *grounded update* becomes $x_c \leftarrow 200$. Thus, the grounding context is $\{x_c, z_c\}$. Second, if $z_c > 0$, then the integrity context is $\{y_c\}$, since y_c must be read in order to determine whether the constraint $x_c + y_c \geq 500$ will be satisfied after the update. If $z_c \leq 0$, then the update cannot possibly result in a violation of the integrity constraint, and so the integrity context is $\emptyset$. The write context is $\{x_c\}$ provided that $z_c \neq 0$ and the resulting update would not violate the integrity constraints.

As illustrated in the above example, these contexts may vary, depending upon the initial state used to determine the grounding. For a more complex example, consider the transaction τ_{34} defined by the following statement:

$$\mathsf{if}\ (x_{c_2} < 300) \wedge (x_{c_1} > 300)\ \mathsf{then}\ \{x_{c_1} \leftarrow x_{c_1} - z_{c_1}\}\ \mathsf{else}\ \{x_{c_2} \leftarrow x_{c_2} - z_{c_2}\}\ \mathsf{endif}$$

Clearly, the write context, and also the integrity context, depend upon which branch of the conditional is taken, provided that $z_{c_i} > 0$ for the appropriate $i \in \{1, 2\}$ and the resulting update is x-legal. However, even more economy is possible. If the conditional is evaluated left to right, and $x_{c_2} \geq 300$ for the snapshot state, then it is not necessary to evaluate the second condition, since the conjunction will be false regardless of the value of x_{c_1}. The actual transaction may or may not evaluate the second conjunct and hence x_{c_1}. If so, since the statement $x_{c_2} \leftarrow x_{c_2} - z_{c_2}$ does not involve x_{c_1}, that data object might not be read at all. The goal is to model real transactions, as much as possible, so no position on how conditionals should be evaluated is taken in this paper. While the write context (qua write set) has already been defined formally in Definition 3.14, and a refined notion of integrity context will be formalized

[7] This distinction is also made in [17, Sec. 4], where reads of the integrity context are termed *integrity reads.* Since that paper deals exclusively with the maintenance of internal integrity constraints (see Summary 2.4), its details will not be considered further here.

in Definition 5.8, there is no need to formalize further the notion of grounding context. As remarked above, it is the distinction between the grounding context and the integrity context, and not the formalization of the former, which is of primary importance. The importance of this distinction will become apparent in Sec. 5.

Notation 3.19 (Conventions for transactions) The notation $\langle \mathbf{D}, \mathcal{U}_T \rangle$ will be used throughout the rest of this paper to denote the updateable object which underlies the transaction T. No confusion should result, because transaction names will always take the form of T or τ, possibly with a prime and/or subscript. Thus, for example, the update object associated with T'_i is $\langle \mathbf{D}, \mathcal{U}_{T'_i} \rangle$. On the other hand, updateable objects not associated with a transaction will never use subscripts involving T or τ.

4 A Formal Model of Concurrency and Snapshot Isolation

The purpose of this section is to provide a formal model of snapshot isolation (hereafter *SI*), at a level of detail appropriate for the extension which is developed in Sec. 5.

Definition 4.1 (Schedules of transactions under SI) The usual model of execution for a transaction T employs a start time $t_{\mathsf{Start}}\langle T \rangle$ and an end time $t_{\mathsf{End}}\langle T \rangle$. Concurrency properties are then defined in terms of these parameters. Specifically, two transactions T_1 and T_2 run *serially* if $t_{\mathsf{End}}\langle T_1 \rangle < t_{\mathsf{Start}}\langle T_2 \rangle$ or $t_{\mathsf{End}}\langle T_2 \rangle < t_{\mathsf{Start}}\langle T_1 \rangle$, and they run *concurrently* otherwise. In the theory presented here, the end time of a transaction is its commit time. As explained in Definition 3.12, a transaction which fails for some reason is modelled as executing the identity update.

The actual times do not matter; rather, it is only their ordering relative to each other which is of interest in terms of behavior. To this end, rather than working with explicit timestamps, an order-based representation will be employed. Let $\mathbf{T}$ be a finite subset of $\mathsf{BBTrans}_{\mathbf{D}}$. Define $\mathsf{SCSet}\langle \mathbf{T} \rangle = \{T^s \mid T \in \mathbf{T}\} \cup \{T^c \mid T \in \mathbf{T}\}$, in which T^s and T^c represent the relative start and commit times of transaction T, respectively. A *SI-schedule* on $\mathbf{T}$ is given by a total order $\leq_{\mathbf{T}}$ on $\mathsf{SCSet}\langle T \rangle$ with the property that for each $T \in \mathbf{T}$, $T^s <_{\mathbf{T}} T^c$. (As is common practice, $x <_{\mathbf{T}} y$ denotes that $x \leq_{\mathbf{T}} y$ but $x \neq y$.) It is important to understand that T^s and T^c are just symbols; the representation is only for the relative times; no numerical values are specified. In translating from a representation with explicit timestamps, $T_1^s <_{\mathbf{T}} T_2^s$ iff $t_{\mathsf{Start}}\langle T_1 \rangle < t_{\mathsf{Start}}\langle T_2 \rangle$, $T_1^c <_{\mathbf{T}} T_2^c$ iff $t_{\mathsf{End}}\langle T_1 \rangle < t_{\mathsf{End}}\langle T_2 \rangle$, $T_1^s <_{\mathbf{T}} T_2^c$ iff $t_{\mathsf{Start}}\langle T_1 \rangle < t_{\mathsf{End}}\langle T_2 \rangle$, and $T_1^c <_{\mathbf{T}} T_2^s$ iff $t_{\mathsf{End}}\langle T_1 \rangle < t_{\mathsf{Start}}\langle T_2 \rangle$.

For any $T \in \mathbf{T}$, $\mathsf{CSPred}_{\leq_{\mathbf{T}}}\langle T \rangle$ denotes the last transaction to commit before T starts, when it exists. Thus, $(\mathsf{CSPred}_{\leq_{\mathbf{T}}}\langle T \rangle)^c <_{\mathbf{T}} T^s$ and for no $T' \in \mathbf{T}$ is it the case that $(\mathsf{CSPred}_{\leq_{\mathbf{T}}}\langle T \rangle)^c <_{\mathbf{T}} T'^c <_{\mathbf{T}} T^s$. Similarly, $\mathsf{CCPred}_{\leq_{\mathbf{T}}}\langle T \rangle$ denotes the last $T' \in \mathbf{T}$ which commits before T does, when it exists. Note that both $\mathsf{CSPred}_{\leq_{\mathbf{T}}}\langle - \rangle$ and $\mathsf{CCPred}_{\leq_{\mathbf{T}}}\langle - \rangle$ are partial functions, since some transactions will not have the required predecessors.

For $T_1, T_2 \in \mathbf{T}$, T_1 *serially precedes* T_2 if $T_1^c <_{\mathbf{T}} T_2^s$. If neither T_1 serially precedes T_2 nor T_2 serially precedes T_1, then T_1 and T_2 *execute concurrently* and $\{T_1, T_2\}$ is said to form a *concurrent pair.*

Definition 4.2 (Formal semantics of SI-schedules) In order to be able to model the interaction of transactions, as well as to characterize constraint-preserving properties, it is necessary to have a formal model of the semantics of an SI-schedule; that is, to have a way of representing the overall behavior of the execution of a schedule of transactions, given the semantics of each individual transaction as described in Definition 3.12.

Let $\mathbf{T}$ be a finite subset of $\mathsf{BBTrans}_{\mathbf{D}}$ and let $<_{\mathbf{T}}$ be an SI-schedule for $\mathbf{T}$. For the execution of $<_{\mathbf{T}}$, three states in $\mathsf{LDB}(\mathbf{D})$ are defined for each transaction $T \in \mathbf{T}$ and each possible initial state $M \in \mathsf{LDB}(\mathbf{D})$ for the entire schedule:

$\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$: The initial state of the stable database which transaction T reads at the beginning of its execution. In other words, it is the initial snapshot of T.

$\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$: The state of the stable database immediately before T commits.

$\mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$: The state of the stable database immediately after T commits.

In the above, the *stable database* means the global database which is common to all transactions, and does not include any local modifications made by transactions to local copies before they commit.

For each possible initial state $M \in \mathsf{LDB}(\mathbf{D})$, the semantics of SI are defined formally as follows:

$$\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle = \begin{cases} \mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle \mathsf{CSPred}_{\leq_{\mathbf{T}}}\langle T \rangle \rangle & \text{if } \mathsf{CSPred}_{\leq_{\mathbf{T}}}\langle T \rangle \downarrow \\ M & \text{otherwise} \end{cases}$$

$$\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle = \begin{cases} \mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle \mathsf{CCPred}_{\leq_{\mathbf{T}}}\langle T \rangle \rangle & \text{if } \mathsf{CCPred}_{\leq_{\mathbf{T}}}\langle T \rangle \downarrow \\ M & \text{otherwise} \end{cases}$$

$$\mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle =$$
$$\begin{cases} \mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle}\langle\langle \mathbf{D}, \mathfrak{U}_T \rangle\rangle\rangle(\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle) \\ \qquad \text{if } (\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle)_{|\mathsf{WSet}\langle \mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle \rangle\langle \mathbf{D}, \mathfrak{U}_T \rangle} = \\ \qquad\qquad (\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle)_{|\mathsf{WSet}\langle \mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle \rangle\langle \mathbf{D}, \mathfrak{U}_T \rangle} \\ \mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle\langle \mathbf{D}, \mathfrak{U}_T \rangle \qquad \text{otherwise} \end{cases}$$

Less formally, for an initial state M, $\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$ is the state of the stable database just after the last commit operation which occurs before T starts, or the initial state M in the case that no such commit operation has occurred. $\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$ is the state of the stable database just after the last commit operation which occurs before the commit operation of T. It is just M if no previous commit has occurred. Finally, $\mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$ is the result of lifting, to $\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$, the trimming of the initial state of T onto its write view, provided that no transaction which runs concurrently with T has already written a data object which T writes.

It is important to note that it is only the update to the write trim $\mathsf{WTrim}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle}\langle\langle \mathbf{D}, \mathfrak{U}_T \rangle\rangle$, and not the entire update in $\mathfrak{U}_T$, which is lifted upon commit. This is critical because the initial snapshot $\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$ may have been updated by another concurrent transaction. Changes to data objects which occurred after the transaction T began do not affect the write which T performs upon commit. In the terminology of

Discussion 3.18, the grounding context, which determines which update to perform, is determined when the transaction begins, not when it commits. In ordinary SI, the integrity context is not involved.

Definition 4.3 (Constraint preservation) It is important to remember that although the goal of this work is to model and characterize constraint-preserving SI, the above model recaptures only ordinary SI. In particular, as long as the initial snapshot for a transaction is legal, so too will be the database after it commits. Preservation of x-legality requires further effort, and is the focus of the remainder of this paper. Formally, for $M \in \mathsf{ELDB}(\mathbf{D})$, and continuing with the context of Definition 4.2, call $\leq_{\mathbf{T}}$ *constraint preserving for (initial state) M* if for every $T \in \mathbf{T}$, $\mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle \in \mathsf{ELDB}(\mathbf{D})$. Thus, *constraint preservation*, as used in the remainder of this paper, entails preservation of both internal and external constraints, as defined in Definition 3.3. Of course, internal constraints are enforced automatically by the DBMS, so the main task is to show how to ensure that $\leq_{\mathbf{T}}$ preserves external constraints.

Notation 4.4 (Notational convention) Throughout the remainder of this paper, unless explicitly stated to the contrary, take $\mathbf{T}$ to be a finite subset of $\mathsf{BBTrans}_{\mathbf{D}}$ and $\leq_{\mathbf{T}}$ an SI-schedule for $\mathbf{T}$.

5 Constraint Preservation and Basic CPSI

The main results of this paper are developed and presented in this section. Specifically, the notion of constraint-preserving snapshot isolation (CPSI) is developed. There are three distinct stages. First, a general theory of *write independence* is developed, which lays the foundation for guaranteeing constraint preservation while examining the interaction of two (concurrent) transactions at a time. Next, a means of guaranteeing write commutativity via *guards*, a representation of the integrity context of transactions, is developed. Finally, a sketch of how real systems might be augmented to use these ideas is presented.

Discussion 5.1 (Motivation for write independence) Consider two transactions T_1 and T_2 which execute concurrently. Write independence asserts that for any state $M \in \mathsf{ELDB}(\mathbf{D})$ (regarded as the initial snapshot) for which both transactions can be applied individually while preserving x-legality, they may be run concurrently while also preserving x-legality. As defined, write independence is a very local property; it applies only to two concurrent transactions T_1 and T_2 which commit one after the other. However, if all pairs of concurrent transactions of $\leq_{\mathbf{T}}$ have this property, then their commit order may be changed at will, without affecting the final result. This ability to alter the commit order is crucial in the proof by induction of Theorem 5.6, the main result which establishes that the entire schedule $\leq_{\mathbf{T}}$ is then constraint preserving for the given initial state.

In the definitions which develop this idea of write independence, the crucial point to keep in mind is that the initial snapshot $\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$ of the transaction T is used only to determine which updates T will perform. The updates themselves are applied to $\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$, the state of the stable database just before T commits. Thus, in the terminology of Discussion 3.18, writes by other, concurrent transactions to the grounding context have no impact upon correctness. It is only concurrent writes to the integrity context of T which must be considered when evaluating whether or not the update of T may lead to a state which is not x-legal. This observation motivates working with the write trim $\mathsf{WTrim}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle}\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle$ of a transaction as its update set, applied to $\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$, and not to $\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$.

Definition 5.2 (State assignment) Let $\mathbf{T}'$ be a finite subset of $\mathsf{BBTrans}_{\mathbf{D}}$. A *state assignment* for $\mathbf{T}'$ is a function $\iota : \mathbf{T}' \to \mathsf{LDB}(\mathbf{D})$. Ultimately, such an assignment will be used, for $\mathbf{T}' = \mathbf{T}$, to identify the initial snapshot $\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$ for a transaction $T \in \mathbf{T}$. However, For the moment, it will prove easier to work with a state assignment in its general form, with $\mathbf{T}'$ not required to be all of $\mathbf{T}$ (or even a subset of $\mathbf{T}$ for that matter).

To continue, for each $T \in \mathbf{T}'$, observe that $\mathsf{WTrim}_{\iota(T)}\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle$ is singleton; i.e., $\mathfrak{U}_T$ consists of exactly one update, since the updateable object $\langle \mathbf{D}, \mathfrak{U}_T\rangle$ of a transaction T is always functional. Thus, the partial function $\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T)}\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle\rangle : \mathsf{LDB}(\mathbf{D}) \to \mathsf{DB}(\mathbf{D})$ is well defined. Using this observation and given $M \in \mathsf{ELDB}(\mathbf{D})$, call the assignment ι *extendedly legal* (or *x-legal*) for M if $\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T)}\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle\rangle(M){\downarrow} \in \mathsf{ELDB}(\mathbf{D})$ for every $T \in \mathbf{T}'$.

The state assignment ι is *nonoverlapping* if $\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle \cap \mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle = \emptyset$ for every $T_1, T_2 \in \mathbf{T}'$ with $T_1 \neq T_2$. This recaptures exactly the condition that as concurrent transactions, T_1 and T_2 must not write a common data object, which is a fundamental property of SI, as described in Summary 2.1. Note that for any $T \in \mathbf{T}'$, $\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T)}\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle\rangle = \mathsf{WSet}\langle \mathsf{WTrim}_{\iota(T)}\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle\rangle$. In other words, the write set of an ordinary trim is the same as the write set of a write trim. Thus, the above condition may also be expressed as $\mathsf{WSet}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle \cap \mathsf{WSet}\langle \mathsf{WTrim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle = \emptyset$.

The following lemma establishes that nonoverlapping transactions may executed in either order, with the same result. It does not establish, by itself, that this result will be legal. Note in addition that the starting state M need only be legal; it need not be x-legal.

Lemma 5.3 (Commutativity under nonoverlap) *Let* $\{T_1, T_2\} \subseteq \mathsf{BBTrans}_{\mathbf{D}}$ *be a pair of distinct transactions, let* $M \in \mathsf{LDB}(\mathbf{D})$*, and let* $\iota : \{T_1, T_2\} \to \mathsf{LDB}(\mathbf{D})$ *be a nonoverlapping state assignment. Then the following equality always holds.*

$$\begin{aligned}(5.3)\quad &(\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle \circ \mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle)(M){\downarrow}\\ &\quad = (\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle \circ \mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle)(M){\downarrow} \in \mathsf{DB}(\mathbf{D})\end{aligned}$$

Proof For convenience, let M' denote $\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle(M)$. Since $\{T_1, T_2\}$ is nonoverlapping, $\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle \cap \mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle = \emptyset$, and so the databases M and M' agree on $\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle$; i.e., $M_{|\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle} = M'_{|\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle}$. This implies in particular that M' is in the domain of $\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle$. Indeed, $M' \in \mathsf{LDB}(\mathbf{D})$ by assumption, and so $\langle M', M''\rangle \in \mathsf{Lift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle$, with M'' the database which agrees with $\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle(M)$ on $\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle$ and with M' everywhere else. Thus, the expression of the first line of Formula (5.3) is defined.

The situation is analogous for the rôles of T_1 and T_2 exchanged. Thus, in Formula (5.3) above, the result in each of the second line is defined as well.

It remains to show that these two compositions are equal when applied to M. Since $\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_i)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_i}\rangle\rangle\rangle \cap \mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle = \emptyset$; i.e., the two updates act on disjoint data objects, neither update overwrites the other, and so they may be applied in either order. □

Definition 5.4 (Write-independent pairs) Write independence adds to the conditions established in Lemma 5.3 by requiring that the state resulting from either composition be x-legal in case the input state M has that property. More precisely, let $\{T_1, T_2\} \subseteq \mathsf{BBTrans}_{\mathbf{D}}$ be a pair of distinct transactions. Call a state assignment $\iota : \{T_1, T_2\} \to \mathsf{LDB}(\mathbf{D})$ *write independent* if it is nonoverlapping and, for every $M \in \mathsf{ELDB}(\mathbf{D})$ for which ι is x-legal, it is the case that the following two (equivalent) conditions are satisfied.

(5.4-a) $\quad (\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle \circ \mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle)(M){\downarrow}\in \mathsf{ELDB}(\mathbf{D})$

(5.4-b) $\quad (\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_1}\rangle\rangle\rangle \circ \mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathfrak{U}_{T_2}\rangle\rangle\rangle)(M){\downarrow}\in \mathsf{ELDB}(\mathbf{D})$

The equivalence of Formulas (5.4-a) and (5.4-b) follows immediately from Lemma 5.3.

It should be noted that a write-independent pair is called a *write-commuting pair* in [13].

Definition 5.5 (State assignment under SI) Let $M \in \mathsf{LDB}(\mathbf{D})$. Working within the definitions associated with the semantics of SI, as formalized in Definition 4.2, the *state assignment* for $\leq_{\mathbf{T}}$ with initial state M is the function $\mathsf{StAssign}_{\langle \leq_{\mathbf{T}} : M\rangle} : \mathbf{T} \to \mathsf{LDB}(\mathbf{D})$ given on elements by $T \mapsto \mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$.

For $\mathbf{T}' \subseteq \mathbf{T}$, $\mathsf{StAssign}^{|\mathbf{T}'}_{\langle \leq_{\mathbf{T}} : M\rangle} : \mathbf{T}' \to \mathsf{LDB}(\mathbf{D})$ is the function $\mathsf{StAssign}_{\langle \leq_{\mathbf{T}} : M\rangle}$ restricted to $\mathbf{T}'$. In particular, for $T_i, T_j \in \mathbf{T}$, $\mathsf{StAssign}^{|\{T_i,T_j\}}_{\langle \leq_{\mathbf{T}} : M\rangle}$ is the function $\mathsf{StAssign}_{\langle \leq_{\mathbf{T}} : M\rangle}$ restricted to $\{T_i, T_j\}$.

The main abstract result of this paper — that write independence implies constraint preservation under SI — may now be established.

Theorem 5.6 (Write independence ⇒ constraint-preserving SI-schedules) *Let* $M \in \mathsf{ELDB}(\mathbf{D})$. *If* $\mathsf{StAssign}^{|\{T,T'\}}_{\langle \leq_{\mathbf{T}} : M\rangle}$ *is write independent for every concurrent pair* $\{T, T'\}$ *of* $\mathbf{T}$, *then* $\leq_{\mathbf{T}}$ *is constraint preserving for initial state* M.

Proof The proof is by induction on the size of $\mathbf{T}$. Let T_i represent the i^{th} transaction which commits; for n transactions, the commit order is therefore $T_1, T_2, \ldots, T_{i-1}, T_i, T_{i+1}, \ldots, T_{n-1}, T_n$.

The basis step of the induction, for zero transactions or one transaction, is trivial.

For the inductive step, let $n \in [1, \text{-}]$ and assume that the result is true whenever $\mathsf{Card}(\mathbf{T}) \leq n$. Consider the case that $\mathbf{T}$ now consists of $n+1$ transactions, with commit order $T_1, T_2, \ldots, T_{i-1}$, $T_i, T_{i+1}, \ldots, T_{n-1}, T_n, T_{n+1}$.

If T_{n+1} starts after T_n has committed; that is, the two transactions are not concurrent, then the result is immediate, since the final state $\mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T_{n+1}\rangle$ is just the result of running T_{n+1} on input state $\mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T_n\rangle$. There cannot be any constraint violation, extended or otherwise, with serial transactions which operate correctly in isolation.

So, assume that $\{T_n, T_{n+1}\}$ forms a concurrent pair. Define $\mathbf{T}_{\overline{n+1}} = \mathbf{T} \setminus \{T_{n+1}\}$ and $\mathbf{T}_{\overline{n}} = \mathbf{T} \setminus \{T_n\}$, with $\leq_{\mathbf{T}_{\overline{n+1}}}$ and $\leq_{\mathbf{T}_{\overline{n}}}$ the schedules obtained by restricting $\leq_{\mathbf{T}}$ to the transactions in $\mathbf{T}_{\overline{n+1}}$, and $\mathbf{T}_{\overline{n}}$, respectively. Then by the inductive hypothesis, each of $\leq_{\mathbf{T}_{\overline{n+1}}}$ and $\leq_{\mathbf{T}_{\overline{n}}}$ is constraint preserving for initial state M. Letting $N \in \mathsf{ELDB}(\mathbf{D})$ be given by $N = M$ if $n = 2$ and $N = \mathsf{AfterCmt}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T_{n-1}\rangle$ if $n \geq 3$, this implies in particular that

$\mathsf{FLift}_{\mathbf{D}}\langle\mathsf{WTrim}_{\iota(T_n)}\langle\langle\mathbf{D},\mathfrak{U}_{T_n}\rangle\rangle\rangle(N) \in \mathsf{ELDB}(\mathbf{D})$ as well as $\mathsf{FLift}_{\mathbf{D}}\langle\mathsf{WTrim}_{\iota(T_{n+1})}\langle\langle\mathbf{D},\mathfrak{U}_{T_{n+1}}\rangle\rangle\rangle(N) \in \mathsf{ELDB}(\mathbf{D})$. Thus, in view of Definition 5.4, the following holds.

$$(\mathsf{FLift}_{\mathbf{D}}\langle\mathsf{WTrim}_{\iota(T_n)}\langle\langle\mathbf{D},\mathfrak{U}_{T_n}\rangle\rangle\rangle \circ \mathsf{FLift}_{\mathbf{D}}\langle\mathsf{WTrim}_{\iota(T_{n+1})}\langle\langle\mathbf{D},\mathfrak{U}_{T_{n+1}}\rangle\rangle\rangle)(N)\!\downarrow\in \mathsf{ELDB}(\mathbf{D})$$

However, the database state which results from the above composition applied to N is just that obtained by running $\leq_{\mathbf{T}}$, which establishes that it is constraint preserving, as required.

As a point of clarification, in the above, while it is assumed that $\{T_n, T_{n+1}\}$ forms a concurrent pair, if T_{n-1} exists, it need not be the case that $\{T_{n-1}, T_{n+1}\}$ and $\{T_{n-1}, T_n\}$ are concurrent; it does not matter. □

Definition 5.7 (Guards for singleton full write objects) The property of write independence is an abstract one, not directly applicable to the implementation of constraint preservation under SI. The notion of a guard provides a much more concrete and useful condition of this abstract notion, as it is based upon the integrity context of a transaction (see Discussion 3.18). A guard reduces the global test for lifting to all of $\mathbf{D}$ to the much more local test of lifting to just the context of the update plus its guard. Before presenting the definition for transactions (in Definition 5.8 below), the more general concept of a guard object is developed.

Let $\langle\mathbf{c},\{u\}\rangle$ be a singleton full write object over $\mathbf{D}$; that is, a full write object with just one update. A *guard object* for $\langle\mathbf{c},\{u\}\rangle$ is a $\mathbf{y} \in \mathsf{DObj}\langle\mathbf{D}\rangle$ which satisfies the following two properties.

(go-i) $\mathbf{y} \cap \mathbf{c} = \emptyset$.

(go-ii) For every $M \in \mathsf{ELDB}(\mathbf{D})$ with $M_{|\mathbf{c}} = u^{(1)}$,
$\mathsf{FLift}_{\mathbf{D}}\langle\langle\mathbf{c},\{u\}\rangle\rangle(M)\!\downarrow \in \mathsf{ELDB}(\mathbf{D}) \Leftrightarrow \mathsf{FLift}_{[\![\mathbf{D}|\mathbf{y}\cup\mathbf{c}]\!]}\langle\langle\mathbf{c},\{u\}\rangle\rangle(M_{|\mathbf{y}\cup\mathbf{c}})\!\downarrow \in \mathsf{ELDB}([\![\mathbf{D}|\mathbf{y}\cup\mathbf{c}]\!])$.

Condition (go-i) simply states that the guard object does not overlap the write set. Condition (go-ii) requires that the lifting of u to the entire schema $\mathbf{D}$ is extendedly legal iff the lifting to just $\mathbf{y}\cup\mathbf{c}$ has that property. In the context of a transaction, in order to determine whether the update u may be applied without a constraint violation, it suffices to read the data object of the guard object $\mathbf{y}$.

Definition 5.8 (Guard functions and guarded black-box transactions) Now let T be a transaction. Applying Definition 5.7 with $N \in \mathsf{ELDB}(\mathbf{D})$ and $\langle\mathbf{c},\{u\}\rangle = \mathsf{WTrim}_N\langle\langle\mathbf{D},\mathfrak{U}_T\rangle\rangle$ yields the basis for a concrete representation of write commutativity. Note that $\mathsf{WTrim}_N\langle\langle\mathbf{D},\mathfrak{U}_T\rangle\rangle$ is a singleton functional full-write object by construction, so the context of that definition applies.

Formally, a *guard function* for T is a function $g : \mathsf{LDB}(\mathbf{D}) \to \mathsf{SubObj}\langle\mathbf{D}\rangle$ with the property that for any $N \in \mathsf{ELDB}(\mathbf{D})$ the following two conditions are met.

(g-i) $g(N)$ is a guard object for $\mathsf{WTrim}_N\langle\langle\mathbf{D},\mathfrak{U}_T\rangle\rangle$.

(g-ii) $(\forall N' \in \mathsf{LDB}(\mathbf{D}))((\mathsf{WTrim}_N\langle\langle\mathbf{D},\mathfrak{U}_T\rangle\rangle = \mathsf{WTrim}_{N'}\langle\langle\mathbf{D},\mathfrak{U}_T\rangle\rangle) \Rightarrow (g(N) = g(N')))$.

Given $N \in \mathsf{ELDB}(\mathbf{D})$, $g(N)$ is called the *guard object* for N.

The core idea is that the update of the transaction will not result in a constraint violation iff that update when restricted to the complex data object consisting of the guard plus the write set of the update does not result in a constraint violation. In other words, the test for correctness on all of $\mathsf{DObj}\langle \mathbf{D}\rangle$ may be reduced to a test on just the guard object plus the objects to be updated.

Condition (g-ii) requires that the guard object depend only upon the associated grounded update, and not further upon the input state. In other words, if two initial states produce the same ground update, then the guard objects for those two states are the same as well. It is important to keep in mind that the guard object $g(N)$ for a database $N \in \mathsf{ELDB}(\mathbf{D})$ must be valid for applications of the update $\mathsf{WTrim}_N\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle$ to all $N' \in \mathsf{ELDB}(\mathbf{D})$ to which it is applicable, and not just those databases to which T applies it. See Definition 6.2 for a further discussion of this issue.

It is useful to observe that (go-i) of Definition 5.7 translates to the following:

(go-i′) $g(N) \cap \mathsf{WSet}\langle \mathsf{Trim}_N\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle\rangle = \emptyset$.

In the context of $\leq_{\mathbf{T}}$ with initial state $M \in \mathsf{ELDB}(\mathbf{D})$ and $T \in \mathbf{T}$, the utility of the notion of guard is immediate — if the guard object $g(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)$ is not written by any concurrent transaction, then as long as the write update $\mathsf{Trim}_M\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle$ is legal for its initial snapshot $\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$, it will also be legal when the transaction writes its update to $\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$, since $\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$ and $\mathsf{BeforeCmt}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$ must agree on the guard. Actually, a stronger condition will be established in Theorem 5.17 — to ensure constraint preservation, it will be shown to suffice that for two concurrent transactions T_1 and T_2, at least one not write the guard object of the other. It is not necessary that neither write the guard object of the other.

The set of all guard functions for T is denoted $\mathsf{Guards}_{\mathbf{D}}\langle T\rangle$.

It is convenient to have a notation for black-box transactions involving guards which extends that of Definition 3.12. To that end, a *guarded black-box transaction* T is represented by a pair $\langle\langle \mathbf{D}, \mathcal{U}_T\rangle, \mathcal{G}_T\rangle$ in which $\langle \mathbf{D}, \mathcal{U}_T\rangle \in \mathsf{FUpdObj}(\mathbf{D})$ and $\mathcal{G}_T \in \mathsf{Guards}_{\mathbf{D}}\langle T\rangle$, with T represented by $\langle \mathbf{D}, \mathcal{U}_T\rangle$ in the latter.

The set of all guarded black-box transactions over $\mathbf{D}$ is denoted $\mathsf{GBBTrans}_{\mathbf{D}}$.

A guard function g for T is *static* if g is a constant function; that is, if $g(M_1) = g(M_2)$ for all $M_1, M_2 \in \mathsf{LDB}(\mathbf{D})$. Otherwise, it is *dynamic*. In [13], guards are static by definition. The use of dynamic guards in this paper constitutes a major generalization over [13], and is addressed in more detail in Discussion 5.25.

Notation 5.9 (Notational convention) From now on, unless stated explicitly to the contrary, augment Notation 4.4 so that $\mathbf{T}$ is taken to be a finite subset of $\mathsf{GBBTrans}_{\mathbf{D}}$, and not just of $\mathsf{BBTrans}_{\mathbf{D}}$. In other words, assume that every transaction in $\mathbf{T}$ has a guard function associated with it. Furthermore, as explained in Definition 5.8 above, the guard function of $T \in \mathbf{T}$ will be denoted $\mathcal{G}_T$.

Guards always exist. Indeed, the set of all data objects not in the write set form a guard, as formalized by the following.

Observation 5.10 (Guards always exist) *For any $T \in \mathsf{BBTrans}_{\mathbf{D}}$, the function $g_{\max} : \mathsf{LDB}(\mathbf{D}) \to \mathsf{SubObj}\langle \mathbf{D}\rangle$ defined on elements by $M \mapsto \mathsf{DObj}\langle \mathbf{D}\rangle \setminus \mathsf{WSet}\langle \mathsf{Trim}_M\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle\rangle$ is a guard for T.* $\square$

Examples 5.11 (Guards) First, fix $c \in \mathsf{States}\langle w\rangle$ and consider the transaction τ_{35} on $\mathbf{D}$, defined by the rule $x_c \leftarrow x_c - z_c$, as in Discussion 3.18. A simple static guard function for τ_{35} is the constant function $g'_{35} : \mathsf{LDB}(\mathbf{D}) \to \mathsf{DObj}\langle\mathbf{E}_3\rangle$ given on elements by $M \mapsto \{y_c\}$. Indeed, at most x_c is changed by this transaction, and the only constraint involving x_c is $x_c + y_c \geq 500$. Thus, the lifting set $g'_{35}(M) \cup \mathsf{WSet}\langle\mathsf{Trim}_M\langle\langle\mathbf{E}_3, \mathfrak{U}_{\tau_{35}}\rangle\rangle\rangle$ of (g-ii) of Definition 5.8 is $\{x_c, y_c\}$. Since x_c is written if the transaction does anything at all, and since by (go-i′) of Definition 5.8, the write set and the guard set are always disjoint, g'_{35} suffices as a guard.

It is, however, possible to do better when the guard is allowed to depend upon the initial snapshot M. The natural guard function for τ_{35} is $g_{35} : \mathsf{LDB}(\mathbf{E}_3) \to \mathsf{DObj}\langle\mathbf{E}_3\rangle$ given on elements as follows.

$$M \mapsto \begin{cases} \{y_c\} & \text{if } (M(z_c) > 0) \wedge (M(x_c) + M(y_c) - M(z_c) \geq 500) \\ \emptyset & \text{otherwise} \end{cases}$$

To understand this guard function, it is convenient to break the process of the transaction into steps. First, it evaluates $M(x_c)$, $M(y_c)$, and $M(z_c)$ to obtain the simple ground update $M(x_c) \overset{x_c}{\rightsquigarrow} M(x_c) - M(z_c)$. This is just a representation of the unique update in $\mathsf{WTrim}_M\langle\langle\mathbf{D}, \mathfrak{U}_{T_{35c}}\rangle\rangle$.

If $M \in \mathsf{ELDB}(\mathbf{D})$, then the update will be constraint preserving provided that $M(x_c) + M(y_c) - M(z_c) \geq 500$ holds, and the values of x_c and y_c are not changed by any other transaction. It is important here to think of $M(x_c)$, $M(y_c)$, and $M(z_c)$ as numbers, as fixed values, not variables. Now if $M(z_c) > 0$, then $M(x_c) \overset{x_c}{\rightsquigarrow} M(x_c) - M(z_c)$ could result in a violation of the constraint $x_c + y_c \geq 500$, even if $M(x_c) + M(y_c) - M(z_c) \geq 500$ holds, in the case that another transaction T were to modify the value of x_c or y_c. Arguing as above, y_c serves as the suitable data object for the guard in this case; T must not write y_c. Of course, T will not write x_c either, since distinct, concurrent transactions never write the same data object under SI.

If $M(z_c) < 0$, then $\mathsf{FLift}_{\mathbf{E_3}}\langle\mathsf{WTrim}_M\langle\langle\mathbf{E}_3, \mathfrak{U}_{T_{35c}}\rangle\rangle\rangle \in \mathsf{ELDB}(\mathbf{E}_3)$ must always hold, since adding a positive value to x_c cannot result in a constraint violation if y_c is held constant and $x_c + y_c \geq 500$ holds for the original values. It might seem that y_c must still be included in the guard, since another transaction could reduce its value by more than $M(z_c)$, but in this case, it is the guard of the other transaction which will flag the problem. This will become clear after the results below are presented.

If $M(z_c) = 0$, $\emptyset$ is also a guard, since the resulting update is the identity, which is always x-legal.

Finally, for $M \in \mathsf{LDB}(\mathbf{E}_3) \setminus \mathsf{ELDB}(\mathbf{E}_3)$, the value of the guard does not matter (since the initial state is not x-legal in that case).

Static guards need not exist. Indeed, again fixing $c \in \mathsf{States}\langle w\rangle$, consider the transaction τ_{36} on $\mathbf{E}_3$ defined by the following rule.

$$\mathsf{if}\ (x_c > 300)\ \mathsf{then}\ \{x_c \leftarrow x_c - z_c\}\ \mathsf{else}\ \{y_c \leftarrow y_c - z_c\}\ \mathsf{endif}$$

Arguing as above, if $M(z_c) > 0$ and $M(x_c) + M(y_c) - M(z_c) \geq 500$, and the then part of the conditional is executed so x_c is written, then y_c must be part of the guard object. On the other hand, if the else part is executed and y_c is written, then x_c must be part of the guard object. Thus, the smallest static guard is given by $M \mapsto \{x_c, y_c\}$. In view of (go-i′) of Definition 5.8, the guard object and the write object must be disjoint, and this is clearly impossible, since x_c and y_c are written in some cases.

The non-existence of static guard functions is not a fundamental one, but rather one which is enforced since dynamic guard functions are employed in this paper. First of all, if static guards are preferred for some reason, an alternate definition which drops condition (go-i′) may be used. Condition (go-i′) is enforced largely for bookkeeping purposes, to make it easier to separate read-only data objects from objects which are written. If condition (go-i′) is dropped, the theory will still be correct, but reads may then overlap with writes, making the theory and presentation less tidy. A more detailed explanation of how to manage this is not presented here, because as elaborated in Discussion 5.25, dynamic guard functions admit increased concurrency while also providing implementation advantages.

Definition 5.12 (Minimal and least guards) It is always possible to add additional simple data objects to a guard object, as long as those added objects are not written by the transaction. Indeed, the result of Observation 5.10 shows that *maximal* guard objects always exist. Far more useful is the notion of a *minimal* guard object. For $T \in \mathsf{BBTrans}_{\mathbf{D}}$, a *minimal guard function* for T is a $g \in \mathsf{Guards}_{\mathbf{D}}\langle T\rangle$ with the property that for any $g' \in \mathsf{Guards}_{\mathbf{D}}\langle\langle \mathbf{D}, \mathbf{u}\rangle\rangle$ and any $M \in \mathsf{LDB}(\mathbf{D})$, the inclusion $g'(M) \subseteq g(M)$ implies the equality $g'(M) = g(M)$. A unique minimal guard view is called *least*.

It is always desirable to choose a minimal guard, because it will be the independence of the guard view of one transaction from the write view of another which will prove to be the critical property in characterizing schedules which are constraint preserving.

Examples 5.13 (Minimal guards) Returning to the context of Examples 5.11, g_{35c} is a minimal guard, while the static guard function g'_{35c} is not.

Discussion 5.14 (Existence of least guards) Although they are clearly a desirable feature, it is easy to show that least guard functions need not exist. Let $\mathbf{E}_4$ be the database schema with $\mathsf{DObj}\langle \mathbf{E}_4\rangle = \{x, y, z\}$. Assume that while x takes integer state values, y and z take ordered pairs of integers as state values. Write y_1 (resp. y_2) for the first (resp. second) integer in the state of y. Thus, if $y = (3, 4)$, $y_1 = 3$ and $y_2 = 4$. Define z_1 and z_2 similarly for z. Assume that the entire schema is governed by the constraints $y_1 = z_1$ and $x + y_1 \geq 500$. Consider the transaction τ_{41} given by the rule $x \leftarrow x - 100$. There are two minimal guard functions for τ_{41}. Arguing as in Examples 5.11, the first is given by

$$M \mapsto \begin{cases} \{y\} & \text{if } M(x) + M(y_1) - 100 \geq 500) \\ \emptyset & \text{otherwise} \end{cases}$$

while the second is identical, save for that y is replaced by z. The simple data objects y and z are incomparable; each contains further information. There is nothing in the model which provides a way to prefer one over the other. The fields y_1 and z_1 are not simple data objects; rather, they are parts of simple data objects which cannot be decomposed. Therefore, neither can be chosen as a guard object by itself. Thus, there is no least guard function.

This is not a serious problem, since as will be argued in Discussion 5.25 below, guards are typically determined dynamically by trigger code, and not fixed beforehand mathematically. Nevertheless, in the general case, the theory cannot always identify an optimal guard, if optimal is interpreted as least.

Definition 5.15 (Independent and conflicting pairs of guarded transactions) Returning to the general case, it is time to use the notion of guard function to obtain a concrete characterization of write independence. To this end, call two transactions guard independent for a given $M \in \mathsf{ELDB}(\mathbf{D})$ if at least one does not write the guard of the other. More formally, let $\{T_1, T_2\} \subseteq \mathsf{GBBTrans}_{\mathbf{D}}$. A state assignment $\iota : \{T_1, T_2\} \to \mathsf{ELDB}(\mathbf{D})$ is *guard independent* if it is nonoverlapping and at least one of the following two conditions holds.

(5.15-a) $$\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathcal{U}_{T_1}\rangle\rangle\rangle \cap \mathcal{G}_{T_2}(\iota(T_2)) = \emptyset$$

(5.15-b) $$\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathcal{U}_{T_2}\rangle\rangle\rangle \cap \mathcal{G}_{T_1}(\iota(T_1)) = \emptyset$$

As established next, guard independence is sufficient to ensure write independence.

Proposition 5.16 (Guard independence ⇒ write Independence) *Let $\{T_1, T_2\} \in \mathsf{GBBTrans}_{\mathbf{D}}$ and let $\iota : \{T_1, T_2\} \to \mathsf{ELDB}(\mathbf{D})$ be an x-legal state assignment for $\{T_1, T_2\}$. If ι is guard independent, then it is write independent.*

Proof Without loss of generality, assume that Formula (5.15-b) holds. It is immediate that Formula (5.4-a) is satisfied, since $\{T_1, T_2\}$ is nonoverlapping and T_2 does not write the guard of T_1, so the update which T_2 performs does not affect the x-legality of the update which T_1 performs. □

The main theorem of this paper may now be established.

Theorem 5.17 (Guard independence guarantees constraint preservation) *Let $M \in \mathsf{ELDB}(\mathbf{D})$. If $\mathsf{StAssign}^{|\{T,T'\}}_{\langle \leq_{\mathbf{T}} : M\rangle}$ is guard independent for every concurrent pair $\{T, T'\}$ of $\mathbf{T}$, then $\leq_{\mathbf{T}}$ is constraint preserving for initial state M.*

Proof The proof follows immediately from Theorem 5.6 and Proposition 5.16. □

Examples 5.18 (Guard independence) Return to the context of $\mathbf{E}_3$, as well as the transaction τ_{35} defined in Discussion 3.18 via the rule $x_c \leftarrow x_c - z_c$, and elaborated for guards in Examples 5.11. Let τ_{37} be the transaction defined by the rule $y_c \leftarrow y_c - 100$. Arguing as in Examples 5.11 for τ_{35}, a minimal guard function for τ_{37} is $g_{\tau_{37}} : \mathsf{LDB}(\mathbf{E}_3) \to \mathsf{DObj}\langle \mathbf{E}_3\rangle$ given on elements as follows.

$$M \mapsto \begin{cases} \{x_c\} & \text{if } M(x_c) + M(y_c) - 100 \geq 500 \\ \emptyset & \text{otherwise} \end{cases}$$

Since it is possible to choose $M \in \mathsf{ELDB}(\mathbf{E}_3)$ such that the guard object of τ_{35} is $\{y_c\}$ while writing x_c, and the guard object of τ_{37} is $\{x_c\}$ while writing y_c, it follows that these two transactions are not guard independent. More precisely, let $M_{33} \in \mathsf{ELDB}(\mathbf{E}_3)$ with the property that $M_{33}(x_c) = M_{33}(y_c) = 300$ and $M_{33}(z_c) = 50$. Then each of $\mathsf{WSet}\langle \mathsf{Trim}_{M_{33}}\langle\langle \mathbf{E}_3, \mathcal{U}_{\tau_{35}}\rangle\rangle\rangle = \{x_c\}$, $\mathcal{G}_{\tau_{35}}(M_{33}) = \{y_c\}$, $\mathsf{WSet}\langle \mathsf{Trim}_{M_{33}}\langle\langle \mathbf{E}_3, \mathcal{U}_{\tau_{37}}\rangle\rangle\rangle = \{y_c\}$, and $\mathcal{G}_{\tau_{37}}(M_{33}) = \{x_c\}$ holds, whence neither (5.15-a) nor (5.15-b) is satisfied. These two transactions cannot be run concurrently without risking a constraint violation. This is none other than the classical write-skew example of [2]. The use of this notion of conflict in CPSI is developed further in Definition 5.20 and Theorem 5.21.

Next, let τ_{38} be the transaction defined by the rule $y_c \leftarrow y_c + 25$. Arguing in a manner similar to that for the case of $M(z_c) > 0$ for τ_{35} in Examples 5.11, it follows that the constant function $g_{\tau_{38}} : \mathsf{LDB}(\mathbf{E}_3) \to \mathsf{DObj}\langle \mathbf{E}_3\rangle$ given on elements by $M \mapsto \emptyset$ is a guard function for τ_{38}.

Indeed, adding a positive value to y_c can never, by itself, result in a constraint violation. Thus, $\{\tau_{35}, \tau_{38}\}$ forms a guard-independent pair, and so the two transactions may be run concurrently.

Now it is clear that τ_{35} may commit before τ_{38} without risking a constraint violation, since τ_{35} does not write the (static) guard of τ_{38}, that guard being the constant function which maps every $M \in \mathsf{ELDB}(\mathbf{E}_3)$ to $\emptyset$. More interesting is that τ_{38} may commit before τ_{35}, even though it does write the guard of τ_{35}. Intuitively, think of τ_{38} as performing a "harmless" write of the guard of τ_{35}. Since τ_{35} increases the value of y_c, it can only make things "better" with respect to the constraint $x_c - y_c \geq 500$. If that constraint were satisfied before adding 25 to y_c, surely it will be satisfied afterwards as well. Of course, that these two transaction may commit in either order without a constraint violation is guaranteed by write independence, but it is nevertheless interesting to see how that abstract property manifests itself in a concrete example.

Guard objects need not be simple. Consider the transaction τ_{39} on $\mathbf{E}_3$ given, for fixed $c_1, c_2 \in \mathsf{States}\langle w \rangle$, by $S_{39} = \{x_{c_1} \leftarrow x_{c_1} - z_{c_1}, x_{c_2} \leftarrow x_{c_2} - z_{c_2}\}$. The two assignments are independent in terms of constraints; the x-legality of one does not depend in any way upon the x-legality of the other. In effect, τ_{39} consists of two independent subtransactions. A guard function may be obtained by replacing c with c_1, x_c with x_{c_1}, and y_c with y_{c_1}, and then independently c with c_2, x_c with x_{c_2} and y_c with y_{c_2}, all in in τ_{35} above. The guard object will be one of $\emptyset$, $\{y_1\}$, $\{y_{c_2}\}$, and $\{y_{c_1}, y_{c_2}\}$. The details are omitted. The point is that the guard is determined *dynamically*, based upon the initial snapshot, and not statically, with one guard object for all cases.

As a final example, again in the context of $\mathbf{E}_3$, let τ_{3a} be the transaction defined by the assignment $x_{c_1} \leftarrow x_{c_1} - x_{c_2}$, and let $\tau_{3a'}$ be the transaction defined by the assignment $x_{c_2} \leftarrow x_{c_2} - x_{c_1}$. Arguing as above, a minimal guard for τ_{3a} is $g_{\tau_{3a}} : \mathsf{LDB}(\mathbf{E}_3) \to \mathsf{DObj}\langle \mathbf{E}_3 \rangle$ given on elements as follows.

$$M \mapsto \begin{cases} \{y_{c_1}\} & \text{if } (M(x_{c_2}) > 0) \wedge (M(x_{c_1}) + M(y_{c_1}) - M(x_{c_2}) \geq 500) \\ \emptyset & \text{otherwise} \end{cases}$$

A guard function $g_{3a'}$ for $\tau_{3a'}$ is obtained by exchanging c_1 and c_2 in the above. Clearly, $\{\tau_{3a}, \tau_{3a'}\}$ forms a guard-independent pair, since neither transaction writes any y_{c_i}. Nevertheless, for certain cases of M, each transaction does write a data object which the other reads. More precisely, for any $M \in \mathsf{ELDB}(\mathbf{D})$ with $M(x_{c_1}) = M(x_{c_2}) = 300$ and $M(y_{c_1}) = M(y_{c_2}) = 600$, τ_{3a} writes x_{c_1} and $\tau_{3a'}$ writes x_{c_2}. However, in the terminology of Discussion 3.18, the reads are grounding reads, not integrity reads. Writing of the values of those read data objects by a concurrent transaction has no effect upon whether the final result will be x-legal or not. This illustrates clearly the importance of distinguishing grounding reads from integrity reads.

Ordinary SSI makes no such distinction; this issue will now be examined in more detail.

Summary 5.19 (A short summary of serializable SI) In order to make a proper comparison between CPSI and SSI, it is necessary to begin by providing a brief summary of the latter. Although some ideas of SSI have already been sketched in Sec. 1, there are other, important aspects which were not needed in the overview presented there. The summary here is nevertheless very terse, touching only upon those aspects necessary to contrast SSI with CPSI. For details of the theory underlying SSI, the reader is referred to the papers [9] and [5].

An understanding of SSI must necessarily begin with knowledge of the *direct serialization graph* [1], or *DSG* for short, associated with a schedule $\leq_{\mathbf{T}}$ and an initial state $M \in \mathsf{ELDB}(\mathbf{D})$.

In that graph, which is denoted $\mathsf{DSG}\langle \leq_{\mathbf{T}} : M \rangle$ in this paper, the vertices are transactions. There are three types of edges. There is a *read-write edge*, or *rw-edge*, from T_1 to T_2, denoted $T_1 \xrightarrow{\mathsf{rw}} T_2$, if T_1 reads some data object x for which T_2 is the next writer. Similarly, there is a *write-write edge*, or *ww-edge*, from T_1 to T_2, denoted $T_1 \xrightarrow{\mathsf{ww}} T_2$ if T_1 writes some data object x and T_2 is the next writer of x. Finally, there is *write-read edge*, or *wr-edge*, from T_1 to T_2, denoted $T_1 \xrightarrow{\mathsf{wr}} T_2$, if T_1 writes some data object x and T_2 subsequently reads the version of x which T_1 wrote. Observe that ww-edges and wr-edges can never connect concurrent transactions under SI, while the transactions connected by an rw-edge may be, but need not be, concurrent. The relationships indicated by edges in $\mathsf{DSG}\langle \leq_{\mathbf{T}} : M \rangle$ are called *dependencies*. So, for example, if there is a rw-edge from T_1 to T_2, then it is said that there is *rw-dependency* from T_1 to T_2 for $\leq_{\mathbf{T}}$ with initial state M. The notions of *ww-dependency* and *wr-dependency* are defined similarly.

In the implementation of SSI, as described in [5], the critical notion is the *dangerous structure*, which consists of two consecutive read-write edges between concurrent pairs in the DSG *which occur in a cycle*; that is, two dependencies of the form $T_1 \xrightarrow{\mathsf{rw}} T_2$ and $T_2 \xrightarrow{\mathsf{rw}} T_3$ with $\{T_1, T_2\}$ and $\{T_2, T_3\}$ concurrent pairs. In a dangerous structure, T_1 and T_3 may be, but need not be, the same transaction. A sufficient condition for an SI schedule to be serializable is that the DSG be free of dangerous structures [9, Thm. 3.1].

Unfortunately, in order to determine whether a sequence of two consecutive rw-edges lies within a cycle of the DSG, it may be necessary to build a large part of, if not all of, that graph. As illustrated by the example surrounding $\mathbf{E}_0$ of Sec. 1, in the worst case, it may be necessary to construct the entire graph. Therefore, in the version of SSI described in [5], a simpler test is used. Specifically, define a *potential dangerous structure* to be the same as a dangerous structure, save that it need not lie in a cycle of the DSG. SSI does not permit potential dangerous structures, regardless of whether or not they are true dangerous structures. This results in a much simpler test, involving at most three transactions a a time, at the expense of a greater number of false positives.[8] If a potential dangerous structure is found, at least one of the participating transactions must not be allowed to commit.

Definition 5.20 (Guard-write dependencies, the GDSG, and guard-write pairs) Let $M \in \mathsf{ELDB}(\mathbf{D})$. Say that there is a *gw-dependency* from T_1 to T_2 for $\leq_{\mathbf{T}}$ with initial state $M \in \mathsf{ELDB}(\mathbf{D})$ if T_2 writes the guard of T_1; that is, if the following inequality holds.

$$\mathsf{WSet}\langle \mathsf{Trim}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T_2 \rangle} \langle\langle \mathbf{D}, \mathcal{U}_{T_2} \rangle\rangle\rangle \cap \mathcal{G}_{T_1}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T_1 \rangle) \neq \emptyset \tag{5.20}$$

Observe that the above is essentially the negation of Formula (5.15-b).

The guard-augmented DSG extends the DSG by adding gw-edges. Formally, the *guard-augmented DSG* or *GDSG* associated with a schedule $\leq_{\mathbf{T}}$ and an initial state $M \in \mathsf{ELDB}(\mathbf{D})$ has all of the edges of $\mathsf{DSG}\langle \leq_{\mathbf{T}} : G \rangle$, together with a *gw-edge* from T_1 to T_2 precisely in the case that T_2 writes the guard of T_1, that is, if inequality (5.20) is satisfied. The GDSG for $\leq_{\mathbf{T}}$ for initial state M is denoted $\mathsf{GDSG}\langle \leq_{\mathbf{T}} : M \rangle$.

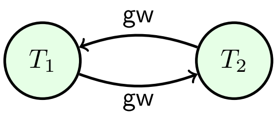


Figure 5.1: A guard-write pair

Call $\{T_1, T_2\}$ a *guard-write pair*, or *gw-pair*, in $\mathsf{GDSG}\langle \leq_{\mathbf{T}} : M \rangle$ if it forms a concurrent pair for which both $T_1 \xrightarrow{\mathsf{gw}} T_2$ and $T_2 \xrightarrow{\mathsf{gw}} T_1$ hold. In other words, a gw-pair is a loop in the GDSG consisting of exactly two gw-edges, as illustrated in Fig. 5.1.

[8]Actually, this test may be refined even further; T_3, the last transaction in the chain $T_1 \xrightarrow{\mathsf{rw}} T_2 \xrightarrow{\mathsf{rw}} T_3$, must be the first transaction to commit. See [9, proof of Thm. 2.1] for details.

It is more compact to represent a guard-write pair using a single edge with double arrows, as shown in Fig. 5.2, and also in Fig. 1.4. However, it is often desirable to show not only the dependency, but also the data objects involved. In that case, a two-arrow representation, as illustrated in Fig. 5.5 below, is more appropriate. Both notations will be used in that which follows.

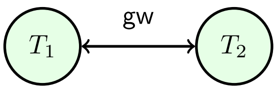


Figure 5.2: Alternate notation for a guard-write pair

The strengthened result for detecting conflicts which may result in constraint violation may be expressed as follows.

Theorem 5.21 (Freedom from gw-pairs ⇒ constraint preservation) *Let* $M \in \mathsf{ELDB}(\mathbf{D})$*. If* $\mathsf{GDSG}\langle \leq_{\mathbf{T}} : M\rangle$ *is free of gw-pairs, then* $\leq_{\mathbf{T}}$ *is constraint preserving for initial state* M*.*

Proof This is essentially a reformulation of Theorem 5.17, and follows immediately from that theorem. □

Examples 5.22 (The GDSG) Consider three mutually concurrent transactions on the schema $\mathbf{E}_3$ of Examples 3.5 for $c_1, c_2 \in \mathsf{States}\langle x\rangle$. The transaction τ_{3b} is defined via the rule $x_{c_1} \leftarrow x_{c_1} - 0.2 * x_{c_2}$, τ_{3c} is defined via the rule $x_{c_2} \leftarrow x_{c_2} - 0.2 * x_{c_1}$, and τ_{3d} is defined via the rule $y_{c_1} \leftarrow y_{c_1} + 0.2 * |x_{c_2}|$. The GDSG conflict graph for these three transactions is shown in Fig. 5.3, for initial state $M_{34} \in \mathsf{ELDB}(\mathbf{D})$ defined by $M_{34}(x_{c_1}) = M_{34}(x_{c_2}) = M_{34}(y_{c_1}) = M_{34}(y_{c_2}) = 300$. The values of the other members of $\mathsf{DObj}\langle \mathbf{E}_3\rangle$ are irrelevant. For convenience, the data objects underlying the rw- and gw-dependencies are shown as well. Note in particular that although τ_{3b} reads x_{c_2}, it does so only as a grounding read, and similarly for τ_{3c} reading x_{c_1} and τ_{3d} reading x_{c_2}. Because these reads are not used to check integrity constraints, they cannot contribute to non-preservation of a constraint. Observe also that τ_{3d} uses only the absolute value of x_{c_2} in the computation of the new value for y_1, and so x_{c_1} will not lie in a minimal guard object. Indeed, τ_{3d} has the a uniform guard defined on elements by $g_{3d} : M \mapsto \emptyset$. Adding a nonnegative value to y_{c_1} can never result in a constraint violation. Although the GDSG contains cycles, in view of Theorem 5.21, the schedule is constraint-preserving for M_{34}, since it is free of gw-pairs. Indeed, it is not difficult to see that the schedule is constraint-preserving for any $M \in \mathsf{ELDB}(\mathbf{E}_3)$. Of course, the associated schedule is potentially non-serializable, since the GDSG contains a dangerous structure, and if full view serialization is needed, the techniques of SSI must be applied.

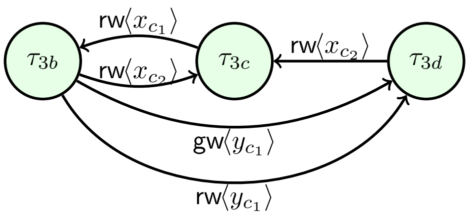
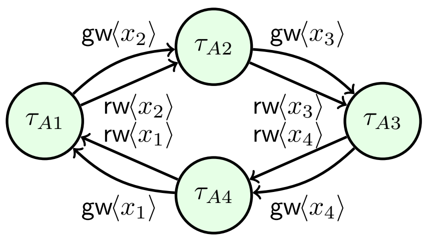

$\mathsf{rw}\langle x_{c_1}\rangle$
$\mathsf{rw}\langle x_{c_2}\rangle$
$\mathsf{rw}\langle x_{c_2}\rangle$
τ_{3b}
τ_{3c}
τ_{3d}
$\mathsf{gw}\langle y_{c_1}\rangle$
$\mathsf{rw}\langle y_{c_1}\rangle$

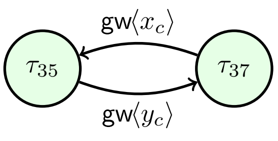
Figure 5.3: A GDSG with dangerous structures but no gw-pairs

Cycles of length greater than two do not represent gw-dependencies. To illustrate, let $\mathbf{E}_A$ be the schema with $\mathsf{DObj}\langle \mathbf{E}_1\rangle = \{x_1, x_2, x_3, x_4\}$, each taking integer values, and governed by the four constraints $x_1 + x_2 \geq 500$, $x_2 + x_3 \leq 500$, $x_3 + x_4 \geq 500$, and $x_4 + x_1 \leq 500$, There are four transactions. τ_{A1} executes the update $x_1 \leftarrow x_1 - 100$, τ_{A2} executes the update $x_2 \leftarrow x_2 + 100$, τ_{A3} executes the update $x_3 \leftarrow x_3 - 100$, and τ_{A4} executes the update $x_4 \leftarrow x_4 + 100$. τ_{A1} has the constant function $M \mapsto \{x_2\}$ as guard. Note in particular that the data object x_4 is not in the guard of τ_{A1} since the constraint $x_4 + x_1 \leq 500$ can never be violated by a reduction in the value of x_1. Similarly, τ_{A2} has the constant function $M \mapsto \{x_3\}$ as guard, τ_{A3} has the constant function $M \mapsto \{x_4\}$ as guard, and τ_{A4} has the constant function $M \mapsto \{x_1\}$ as guard.

If these four transactions are run concurrently, the associated GDSG for any initial state is shown in Fig. 5.4. Even though there is a loop consisting of four gw-edges, there are no gw-loops, and no constraint violation is possible. Note further that the same loop is connected by four rw-edges, so SSI would flag this as potentially nonserializable. In fact, any subset of three of these transactions form the vertices of a potential dangerous structure, and would be flagged as potentially nonserializable.


$\mathsf{gw}\langle x_2\rangle$
τ_{A2}
$\mathsf{gw}\langle x_3\rangle$
τ_{A1}
$\mathsf{rw}\langle x_2\rangle$
$\mathsf{rw}\langle x_1\rangle$
$\mathsf{rw}\langle x_3\rangle$
$\mathsf{rw}\langle x_4\rangle$
τ_{A3}
τ_{A4}
$\mathsf{gw}\langle x_1\rangle$
$\mathsf{gw}\langle x_4\rangle$
Figure 5.4: A gw-cycle not containing any gw-pairs

To obtain a simple example which involves a guard-write pair, return to the the pair $\{\tau_{35}, \tau_{37}\}$ of Examples 5.18, defined by the rules $x_c \leftarrow x_c - z_c$ and $y_c \leftarrow y_c - 100$. For this pair run concurrently, the GDSG for M_{33} has a gw-pair, as illustrated in Fig. 5.5. Thus, the two cannot be run concurrently without risking constraint violation.


$\mathsf{gw}\langle x_c\rangle$
τ_{35}
τ_{37}
$\mathsf{gw}\langle y_c\rangle$
Figure 5.5: A gw-pair for $\{\tau_{35}, \tau_{37}\}$

Discussion 5.23 (Comparison to approaches which convert reads to writes) In [9, p. 497], two related alternatives to addressing problems of constraint violation under SI are suggested. The first is to report integrity reads as writes. For a concrete example, return to the context of $\mathbf{E}_3$ introduced in Examples 3.5. To enforce a constraint of the form $x_c + y_c \geq 500$ when x_c is written by a transaction T, it is considered, for the purposes of concurrency control, to have written both x_c and y_c, thus eliminating the possibility of write skew. An advantage of this approach is that ordinary SI may be used to ensure constraint satisfaction, but at the expense of reduced concurrency in comparison to CPSI. For example, the guard-independent pair $\{\tau_{35}, \tau_{38}\}$ of Examples 5.18 would not be allowed to run concurrently under such a strategy.

A second strategy is to materialize the constraint $x_c + y_c \geq 500$ by using a third data object, say v_c, which is constrained to be the sum of x_c and y_c; i.e., $v_c = x_c + y_c$. Then, any update to either x_c or y_c will also update v_c. The net effect is the same, and the limitation is the same, as in the first approach noted above. In short, these approaches results in strictly more false positives than does CPSI.

Discussion 5.24 (Constraint-preserving snapshot isolation (CPSI)) The above theorem and examples form the basis for *constraint-preserving snapshot isolation (CPSI)*. The strategy for CPSI is similar to that for SSI. If a gw-pair is discovered, then further action must ensure that a constraint violation does not occur. Various implementation alternatives are discussed in Discussion 5.25 below.

Discussion 5.25 (Implementation of CPSI) The question remains as to how an existing DBMS could be modified to support CPSI. To answer that question with authority and thoroughness would require much more investigation, as well as the implementation of CPSI in various ways, followed by performance studies. Therefore, this discussion is limited to the identification of the most important points to consider in any such implementation, as well as the main alternatives.

Any such strategy will be based upon Theorem 5.21, with the presence of one or more gw-pairs in the GDSG a signal that constraint satisfaction cannot be guaranteed, so that further action must be taken. It is important to understand that a guard does not, by definition, consist of data objects which a transaction actually reads. Rather, it consists of a set of data objects which are sufficient to protect, from updates by concurrent transactions, in order to guarantee that the transaction does not violate any integrity constraints. There are thus two tasks which must be carried out. First, the guard object which is associated with the ground update to be performed by the transaction must be identified. Second, the guards of concurrent transactions must be used to identify gw-conflicts. and the the system must act upon such conflicts.

For the first task, to identify the necessary guard objects, there are two main strategies. One is to have each transaction take the responsibility to identify its guard, which has the obvious disadvantage that even one transaction which computes a guard incorrectly could lead to false negatives, clearly an undesirable situation. The second alternative, which avoids this drawback, is to use a service common to all transactions. A natural way to implement extended integrity constraints is via triggers, and in that case, the guards may be computed within the triggers themselves, for which there are two strategies. The first is to look up the guard in a guard database, or else to to compute it from some rules. This strategy is particularly attractive in situations in which there is a relatively small number of parametric transactions which are executed frequently. Some simple examples for business processes involving travel are given in [14] and [12]. A second strategy is to extract the guard dynamically, from the actual reads of the transactions. An example will help illustrate. Consider again the transaction τ_{35}, defined by the rule $x_c \leftarrow x_c - z_c$ as elaborated in Examples 5.11. The main transaction code will evaluate z_c (as well as x_c) in order to identify the associated ground update. For example, if $x_c = 300$ and $z_c = 100$, then the ground update $300 \overset{x_c}{\rightsquigarrow} 200$ will be computed by the transaction code, and it is this ground update which will be checked by the trigger. The trigger need not and in any reasonable implementation will not read z_c. However, it must read y_c in order to determine whether or not the constraint $x_c + y_c \geq 500$ would be satisfied after the update is performed. In other words, in addition to x_c, it will read only y_c.

It must be noted, however, that in order to obtain an optimal guard object, some additional care may need to be taken in writing the trigger. For example, suppose that $z_c = -100$ instead of $z_c = 100$ in the initial snapshot. The associated ground update will then be $300 \overset{x_c}{\rightsquigarrow} 400$. Assuming that the initial snapshot satisfies the extended constraint $x_c + y_c \geq 500$, the result after that update will also satisfy this constraint; the trigger function need not read the value of y_c in order to determine this. However, the trigger code for updates to x_c must utilize the knowledge that the constraint $x_c + y_c \geq 500$ need not be checked if the value of x_c is increased. Of course, if it does check y_c in all cases, correctness will not be affected, although unnecessary gw-pairs may occur, resulting in false positives which may limit concurrency.

There is one detail — a transaction may perform several simple updates, each supported by its own trigger. In this case, the guard object for the entire update performed by the transaction is just the union of the guard objects for each component update. Thus, the guard object for the entire transaction will be the union of the guard objects obtained from each individual trigger.

It should finally be noted that it is possible to use a form of lazy guard evaluation, in which the entire guard set need not be read in all circumstances. In such cases, the full guard may need to be determined in other ways. See Discussion 6.4 for additional information.

For the second task — to identify and act upon gw-conflicts, there are three high-level strategies. The first is to service gw-conflicts in the same way as potential violations to internal integrity constraints. As noted in Summary 2.4, the latter are enforced immediately, during the lifetime of the transaction. The snapshot of a transaction is modified, by the system, whenever a concurrent transaction performs an update which affects an internal integrity constraint. This happens immediately, before either transaction commits. In principle, such a strategy could also be used for extended constraints. However, there are at least two issues which limit this possibility. First and foremost, it would require major modifications to the DBMS. Second, that strategy compromises one of the main selling points of SI — that writers do not block readers (and conversely). Employing immediate constraint maintenance on a scale which also includes extended constraints (which often involve large sets of data objects) could easily compromise the performance of the system beyond acceptable levels.

The second approach is at the other extreme — to treat gw-conflicts in much the same way that concurrent-write conflicts are handled in ordinary SI. An analog of either FCW or FUW could be used. With the analog of FUW, gw-conflicts would be handled at commit time, a transaction would not be allowed to commit if it participates in a gw-conflict and the other transaction in the pair has already committed. With the analog of FCW, a transaction would be blocked as soon as it declares a read or write which would result in a gw-conflict with a concurrent transaction. It would not be allowed to continue unless the other transaction is terminated for some reason.

For interactive transactions, there is a third possibility which is far preferable and particularly well-suited to the binary nature of gw-conflicts. Since gw-conflicts are binary in nature, two transactions which have a conflict may be able to negotiate in order to see whether a constraint violation will actually result, and, if so, to modify their updates to avoid such a violation. Of course, in the worst case, it may be that every transaction has a conflict with every other concurrent transaction, but that is unlikely to occur in practice. It is, in any case, an approach which deserves further development, as suggested in Sec. 7.

6 Enhancements to CPSI

As sketched in the introduction, there are certain situations in which CPSI can produce false positives which SSI does not, and conversely. The purpose of this section is twofold. First, to highlight why these false positives may occur, a formal model of *conditional guard functions*, which generalize the ordinary guard functions developed in Sec. 5, is developed and then illustrated with a number of examples. The goal is to provide some insight into how difficult it would be to implement extended integrity constraints, guaranteeing correctness, without reading the entire guard.

The second topic of this section is to examine the idea of combining CPSI with SSI or one of its variants, in order to provide an isolation level which guarantees constraint satisfaction while resulting in fewer false positives than either strategy alone. Central to this is a constraint-only version of SSI, called Constraint-Only SSI, or CSSI, which operates much as does SSI, but which ignores ground reads and so flags only cycles which may result in constraint violation.

Definition 6.1 (Conditional guards for full write objects) To recapture the guard phenomena surrounding the schema $\mathbf{E}'_1$, as sketched in Sec. 1, it is convenient to begin by extending Definition 5.7 to a conditional case. Specifically, let $\langle \mathbf{c}, \{u\}\rangle$ be a singleton full write object over $\mathbf{D}$. A *conditional guard object* for $\langle \mathbf{c}, \{u\}\rangle$ is a pair $\langle \mathbf{y}, \mathbf{P}\rangle$ in which $\mathbf{y} \in \mathsf{DObj}\langle \mathbf{D}\rangle$ and $\mathbf{P} \subseteq \mathsf{ELDB}(\llbracket \mathbf{D}|\mathbf{y}\rrbracket)$, satisfying the following property.

(pgo-ii) For every $M \in \mathsf{ELDB}(\mathbf{D})$ with $M_{|\mathbf{c}} = u^{(1)}$ and $M_{|\mathbf{y}} \in \mathbf{P}$,
$\mathsf{FLift}_{\llbracket \mathbf{D}|\mathbf{y}\cup\mathbf{c}\rrbracket}\langle\langle \mathbf{c}, \{u\}\rangle\rangle(M_{|\mathbf{y}\cup\mathbf{c}}){\downarrow} \in \mathsf{ELDB}(\llbracket \mathbf{D}|\mathbf{y}\cup\mathbf{c}\rrbracket) \Rightarrow \mathsf{FLift}_{\mathbf{D}}\langle\langle \mathbf{c}, \{u\}\rangle\rangle(M){\downarrow} \in \mathsf{ELDB}(\mathbf{D})$

In contrast to the definition of guard object in Definition 5.7, there is no condition corresponding to (go-i); i.e., it need not be the case that $\mathbf{y} \cap \mathbf{c} = \emptyset$. As will be illustrated in Examples 6.3, it is sometimes necessary to have the condition depend upon the objects to be updated as well as the read-only guard set. The set $\mathsf{ROGuard}\langle\langle \mathbf{y}, \mathbf{P}\rangle\rangle = \mathbf{y} \setminus \mathbf{c}$ is called the *read-only guard* of $\langle \mathbf{y}, \mathbf{P}\rangle$.

Condition (pgo-ii) expresses that correctness of the update is only guaranteed if the restriction of the input database to $\mathbf{y}$ matches some database in $\mathbf{P}$, and the update restricted to the write set plus the guard object is x-legal. Note that the implication in (pgo-ii) is only in one direction, in contrast to that of (go-ii) of Definition 5.7, which is bidirectional. Compared to a full guard object, it potentially reads less and thus guarantees correctness in fewer cases.

If $\mathbf{y}$ is a guard object (in the sense of Definition 5.7) and $\mathbf{P} = \mathsf{ELDB}(\llbracket \mathbf{D}|\mathbf{y}\cup\mathbf{c}\rrbracket)$, then the conditional guard behaves exactly as the guard object y. Thus, a conditional guard object generalizes an ordinary guard object.

The set of all conditional guard objects for $\langle \mathbf{c}, \{u\}\rangle$ is denoted $\mathsf{CondGd}\langle\langle \mathbf{c}, \{u\}\rangle\rangle$, with $\mathsf{CondGd}\langle \mathbf{D}\rangle$, the set of conditional guard objects for $\mathbf{D}$, defined to be $\bigcup\{\mathsf{CondGd}\langle\langle \mathbf{c}, \{u\}\rangle\rangle \mid \langle \mathbf{c}, \{u\}\rangle \in \mathsf{FWObjs}\langle \mathbf{D}\rangle\}$. In $\langle \mathbf{y}, \mathbf{P}\rangle$, y is called the *object* and $\mathbf{P}$ is called the *condition set.*

It is convenient to have a notation for extracting the components of a conditional guard object. If $\mathbf{o}$ is a conditional guard object, then $\mathsf{DObject}\langle \mathbf{o}\rangle$ denotes the object and $\mathsf{CondSet}\langle \mathbf{o}\rangle$ denotes the condition set. More concretely, if $\mathbf{o} = \langle \mathbf{y}, \mathbf{P}\rangle$, then $\mathsf{DObject}\langle \mathbf{o}\rangle = \mathbf{y}$ and $\mathsf{CondSet}\langle \mathbf{o}\rangle = \mathbf{P}$. If $y = \emptyset$ and $\mathbf{P} = \{\phi_{\mathsf{DB}}\}$, then $\langle \mathbf{y}, \mathbf{P}\rangle$ is called *trivial*; otherwise, it is *nontrivial.* Thus, the *trivial conditional guard object* is $\langle \emptyset, \{\phi_{\mathsf{DB}}\}\rangle$. (See Definition 3.2 for ϕ_{DB}.)

Definition 6.2 (Conditional guard functions) Definition 5.8 is extended to the conditional case as follows. Let T be a transaction. A *conditional guard pair* for T is a pair $\langle g, \Pi\rangle$ in which $g : \mathsf{LDB}(\mathbf{D}) \to \mathsf{CondGd}\langle \mathbf{D}\rangle$ is a function and Π is a partition on $\mathsf{ELDB}(\mathbf{D})$ such that the following conditions hold for any $N \in \mathsf{ELDB}(\mathbf{D})$.

(cg-i) $g(N)$ is a conditional guard object for $\mathsf{WTrim}_N\langle\langle \mathbf{D}, \mathfrak{U}_T\rangle\rangle$.

(cg-ii) If $N' \in \mathsf{ELDB}(\mathbf{D})$ with $(N, N') \in \Pi$; i.e., if N and N' are equivalent under Π, then $g(N) = g(N')$.

There are two forms of condition which are present in a conditional guard pair which are absent in an ordinary guard function. First of all, the guard for a given state $N \in \mathsf{ELDB}(\mathbf{D})$ is conditional in the sense of Definition 6.1. Second, the same ground update may have different conditional guards, depending upon the state. This is in stark contrast to an ordinary guard function, in the sense of Definition 5.8.

Examples 6.3 (Conditional guards) These ideas are best illustrated via example. Consider again the schema $\mathbf{E}'_1$ of Sec. 1, and consider a typical transaction, say τ^d_{1ij}, which applies the update $d_i \leftarrow d_{ij} - e_{ij}$ conditionally, just in case $d_{ij} + d_{i(j+1) \bmod n_1} - e_{ij} \geq 1000$. Recall also the applicable constraint φ^d_{1i}, given by $\sum_{j=0}^{n_1-1} d_{ij} \geq 1000$, and that all data values are nonnegative integers. The conditional guard pair $\langle g^d_{1ij}, \Pi^2_{1ij}\rangle$ in which $g^d_{1ij} : \mathsf{ELDB}(\mathbf{E}'_1) \to \mathsf{CondGd}\langle\mathbf{E}'_1\rangle$ for τ^d_{1ij} is defined on elements by

$$M \mapsto \begin{cases} \langle\{d_{ij}, d_{i(j+1) \bmod n_1|}\}, \mathbf{P}^d_{1'ij}\rangle & \text{if } M(d_{ij} - e_{ij}) < 1000 \\ \langle\emptyset, \{\phi_{\mathsf{DB}}\}\rangle & \text{if } M(d_{ij} - e_{ij}) \geq 1000 \end{cases}$$

with

$$\mathbf{P}^d_{1'ij} = \{N \in \mathsf{ELDB}([\![\mathbf{D}|\{d_{ij}, d_{i(j+1) \bmod n_1}\}]\!]) \mid d_{ij} + d_{i(j+1) \bmod n_1} - e_{ij} \geq 1000\}$$

and Π^2_{1ij} is the partition of $\mathsf{ELDB}(\mathbf{D})$ which has two blocks, one one in which the states have $d_{ij} - e_{ij} < 1000$, and a second in which $d_{ij} - e_{ij} \geq 1000$. In the second line of the definition of g^d_{1ij} on elements, $\langle\emptyset, \{\phi_{\mathsf{DB}}\}\rangle$ states that no membership check is made, since the empty data object $\emptyset$ has just one possible database, ϕ_{DB}. Thus, the update is always performed if $M(d_{ij} - e_{ij}) \geq 1000$. Note also that d_{ij} lies in both the update set $\{d_{ij}\}$ and the guard set $\{d_{ij}, d_{i(j+1) \bmod n_1|}\}$ in the first case; these sets cannot be disjoint for this transaction.

In the example above, there is only one nontrivial guard object. However, there may be several. Assume that $n_1 \geq 3$, and let $\ddot{\tau}^d_{1ij}$ be the transaction which has two cases. If $e_{i(j+1) \bmod n_1} \geq 99$, then it applies the update $d_{ij} \leftarrow d_{ij} - e_{ij}$ conditionally, just in case $d_{ij} + d_{i(j+1) \bmod n_1} - e_{ij} \geq 1000$. On the other hand, if $e_{i(j+1) \bmod n_1} < 99$, it applies the same update $d_{ij} \leftarrow d_{ij} - e_{ij}$ conditionally, but this time just in case $d_{ij} + d_{i(j-1) \bmod n_1} - e_{ij} \geq 1000$. Here the same ground update is tested with different conditions, depending upon the value of $e_{i(j+1) \bmod n_1}$. The conditional guard pair $\langle \ddot{g}^d_{1ij}, \Pi^3_{1ij}\rangle$ in which $\ddot{g}^d_{1ij} : \mathsf{ELDB}(\mathbf{E}'_1) \to \mathsf{CondGd}\langle\mathbf{E}'_1\rangle$ for $\ddot{\tau}^d_{1ij}$ is defined on elements by

$$M \mapsto \begin{cases} \langle\{d_{ij}.d_{i(j+1) \bmod n_1|}\}, \mathbf{P}^{d\geq}_{1'ij}\rangle & \text{if } M(d_{ij} - e_{ij}) < 1000 \wedge M(e_{i(j+1) \bmod n_1}) \geq 99 \\ \langle\{d_{ij}, d_{i(j-1) \bmod n_1|}\}, \mathbf{P}^{d<}_{1'ij}\rangle & \text{if } M(d_{ij} - e_{ij}) < 1000 \wedge M(e_{i(j+1) \bmod n_1}) < 99 \\ \langle\emptyset, \{\phi_{\mathsf{DB}}\}\rangle & \text{if } M(d_{ij} - e_{ij}) \geq 1000 \end{cases}$$

with

$$\begin{aligned} \mathbf{P}^{d\geq}_{1'ij} &= \{N \in \mathsf{ELDB}([\![\mathbf{D}|\{d_{ij}, d_{i(j+1) \bmod n_1}\}]\!]) \mid d_{ij} + d_{i(j+1) \bmod n_1} - e_{ij} \geq 1000\} \\ \mathbf{P}^{d<}_{1'ij} &= \{N \in \mathsf{ELDB}([\![\mathbf{D}|\{d_{ij}, d_{i(j-1) \bmod n_1}\}]\!]) \mid d_{ij} + d_{i(j-1) \bmod n_1} - e_{ij} \geq 1000\} \end{aligned}$$

and Π^3_{1ij} the partition which divides $\mathsf{ELDB}(\mathbf{E}_1)$ into three blocks, one in which the states have $d_{ij} - e_{ij} < 1000$ and $e_{i(j+1) \bmod n_1} \geq 99$, a second in which $d_{ij} - e_{ij} < 1000$ and $e_{i(j+1) \bmod n_1} < 99$, and a third in which $d_{ij} - e_{ij} \geq 1000$. Note in particular that the same ground update may be applied with two distinct conditional guard functions. If $e_{i(j+1) \bmod n_1} \geq 99$, then the element $d_{i(j+1) \bmod n_1}$ to the right (modulo n_1) of d_{ij} is used to validate the constraint conditionally. On the other hand, if $e_{i(j+1) \bmod n_1} < 99$, then the element $d_{i(j-1) \bmod n_1}$ to the left (modulo n_1) is used.

The second example in particular illustrates just how complex conditional guard pairs can become. This leads to significant implementation issues, as are discussed next.

Discussion 6.4 (Implementation of conditional guards and lazy evaluation of ordinary guards) As described in Discussion 5.25, it is highly desirable to have a central authority manage the enforcement of integrity constraints, even extended constraints. The above examples show that this will be a very difficult task when conditional guards are involved. Triggers, the natural tool for managing guards, operate on ground updates. A trigger is not aware of how the proposed ground update was obtained; it only knows that there is a request to perform it, and it must perform a full check to make certain that no constraints will be violated. Thus, the only feasible way to support conditional guards is to allow the transaction itself to manage the entire process of integrity enforcement. As noted in Discussion 5.25, this carries a large risk in that even one faulty transaction will corrupt the entire database. Therefore, it must be accepted that if enforcement of extended integrity constraints is a priority, then conditional guards are not a viable option. It is crucial to keep in mind that excluding conditional guards would not exclude any transactions; rather, it would require the use of ordinary guards, which may involve a higher rate of false positives.

Even if conditional guards are not used, it is possible for a trigger to avoid reading the entire guard in some cases. Consider again the schema $\mathbf{E}'_1$ of the introduction, together with the transaction τ^d_{ij} for fixed i and j, which applies the update $d_{ij} \leftarrow d_{ij} - e_{ij}$ provided no constraint violation will result. The constraint φ^d_{1i}, given by $\sum_{j=0}^{n_1-1} d_{ij} \geq 1000$, yields that a guard object for any instance of this transaction with $d_{ij} - e_{ij} < 1000$ is $\{d_{ik} \mid (0 \leq k \leq n_1) \wedge (k \neq j)\}$. However, a trigger which implements this guard need not read all of it in all cases. A further constraint on $\mathbf{E}'_1$ is that all data objects have nonnegative values. Therefore, if the code of the trigger reads the elements of $\{d_{ik} \mid (0 \leq k \leq n_1) \wedge (k \neq j)\}$ one at a time and maintains a running sum of $d_{ij} - e_{ij}$ plus the elements already read, it may stop as soon as that sum reaches 1000. Whether this will actually improve performance is questionable, since block reads are generally more efficient than individual, one-at-a-time reads, but it does show that such *lazy evaluation* could be used to limit the read set. This is significant if the strategy sketched in Discussion 5.25, which obtains the guard set from the actual reads of the trigger, is used. In this case, such a strategy will underreport the trigger set, so the guard set must be determined in another way.

Definition 6.5 (CPSI+SSI) As illustrated via Examples 5.22, there are schedules for which SSI reports possible nonserializability while CDSG reports that no constraint violation is possible. On the other hand, as illustrated in Sec. 1 using $\mathbf{E}'_1$ and in particular the situation illustrated in Fig. 1.4, CPSI may report a possible constraint violation for a schedule which SSI reports to be serializable. Thus, each strategy can produce false positives which the other does not. It is easy to conceive of a strategy which runs both tests, and which will be called *CPSI+SSI*. This strategy will have a false positive only if both CPSI and SSI both report positive falsely, indicating possibility of constraint violation (CPSI) or of nonserializability (SSI), when neither is actually the case. It has the advantage that since implementations of SSI already exist, if CPSI is implemented in such a system, and if CPSI reports a possible constraint violation, then that report by CPSI may safely be disregarded if SSI reports serializability. Similarly, if SSI reports nonserializability while CPSI reports no constraint violation, then the transactions may be allowed to complete. This would require at most a straightforward modification of a system which supports CPSI and which also has SSI built in, to run the SSI test but to allow the constraint manager to decide how to use its result.

It is possible to reduce the occurrence of false positives by modifying SSI so that it it does not enforce full serializability, but rather only possible constraint violation. This idea is explored next.

Definition 6.6 (The CDSG) Recall from Discussion 3.18 the distinction between the integrity context and the grounding context. For the purpose of detecting constraint violation, as opposed to full serializability, it is convenient to work with a variant of the DSG which includes only integrity reads, while excluding grounding reads which are not integrity reads. Formally, let $\leq_{\mathbf{T}}$ be a schedule and let $M \in \mathsf{ELDB}(\mathbf{D})$. The associated *constraint DSG*, or *CDSG* for short, denoted $\mathsf{CDSG}\langle \leq_{\mathbf{T}} : M \rangle$, is the same as the DSG, save for that only integrity reads (i.e., reads of the integrity context) are included. To be more precise, begin by identifying three sets of data objects for a transaction T operating on initial snapshot M. $\mathsf{InRdSet}\langle T, \leq_{\mathbf{T}} : M \rangle$ is the set of all integrity reads which transaction T performs, $\mathsf{GrRdSet}\langle T, \leq_{\mathbf{T}} : M \rangle$ the set of all grounding reads which it performs, and $\mathsf{WrSet}\langle T, \leq_{\mathbf{T}} : M \rangle$ is the set of all writes which it performs, all within the context of $\leq_{\mathbf{T}}$ for initial state M. Then there is a rw-edge $T_1 \xrightarrow{\mathsf{rw}} T_2$ in $\mathsf{CDSG}\langle \leq_{\mathbf{T}} : M \rangle$ iff there is the same edge in $\mathsf{DSG}\langle \leq_{\mathbf{T}} : M \rangle$, with the further property that there is an $x \in \mathsf{InRdSet}\langle M, \leq_{\mathbf{T}} : T_1 \rangle \cap \mathsf{WrSet}\langle M, \leq_{\mathbf{T}} : T_2 \rangle$. Similarly, there is a wr-edge $T_1 \xrightarrow{\mathsf{wr}} T_2$ in $\mathsf{CDSG}\langle \leq_{\mathbf{T}} : M \rangle$ iff there is the same edge in $\mathsf{DSG}\langle \leq_{\mathbf{T}} : M \rangle$, with the further property that there is an $x \in \mathsf{WrSet}\langle M, \leq_{\mathbf{T}} : T_1 \rangle \cap \mathsf{InRdSet}\langle M, \leq_{\mathbf{T}} : T_2 \rangle$. The ww-edges of $\mathsf{CDSG}\langle \leq_{\mathbf{T}} : M \rangle$ are exactly those of $\mathsf{DSG}\langle \leq_{\mathbf{T}} : M \rangle$. Keep in mind that it need not be the case that $\mathsf{InRdSet}\langle T, \leq_{\mathbf{T}} : M \rangle \cap \mathsf{GrRdSet}\langle T, \leq_{\mathbf{T}} : M \rangle = \emptyset$. Integrity reads which also happen to be grounding reads are included in $\mathsf{CDSG}\langle \leq_{\mathbf{T}} : M \rangle$.

It is important to understand that an rw-edge of $\mathsf{CDSG}\langle \leq_{\mathbf{T}} : M \rangle$ is not necessarily the same as a gw-edge of $\mathsf{GDSG}\langle \leq_{\mathbf{T}} : M \rangle$. The rw- and wr-edges in $\mathsf{CDSG}\langle \leq_{\mathbf{T}} : M \rangle$ represent actual reads by the transaction. On the other hand, gw-edges in $\mathsf{GDSG}\langle \leq_{\mathbf{T}} : M \rangle$ represent guards, which are properties of ground updates. In Fig. 1.4, the actual reads are all integrity reads, illustrating that this set may be much smaller than the guard set, although, as sketched in Discussion 6.4, such an implementation may be difficult and not worth the additional cost.

Definition 6.7 (CSSI) *Constraint-only SSI*, or *CSSI* for short, is the strategy which operates exactly as SSI (see Summary 5.19), except that it works with the CDSG instead of the full DSG. It flags a conflict precisely in the case that the CDSG has a potential dangerous structure.

It will next be shown that CSSI works.

Definition 6.8 (The grounding of a DSG) Let $T \in \mathbf{T}$ and let $N \in \mathsf{LDB}(\mathbf{D})$. The *grounding* of T with respect to N, denoted $\mathsf{GndTr}_N\langle T \rangle$, is obtained by hardwiring the ground update of T to be that obtained by using the initial snapshot N. For example, in the context of τ_{33} on $\mathbf{E}_3$, defined by $x_c \leftarrow x_c - z_c$ as in Discussion 3.18, if $x_c = 300$, $y_c = 300$, and $z_c = 100$ in N, then the grounding of T with respect to N hardwires the update to be $x_c \leftarrow 200$. On the other hand, if $x_c = 300$, $y_c = 300$, and $z_c = 200$ in N, then the grounding of then the grounding of T with respect to N hardwires the update to no change, since the constraint $x_c + y_c \geq 500$ would be violated.

Now let $\leq_{\mathbf{T}}$ be a schedule of the set $\mathbf{T}$ of transactions and let $M \in \mathsf{ELDB}(\mathbf{D})$. The *grounding* of $\leq_{\mathbf{T}}$ for M, denoted $\leq_{\mathbf{T}M}$, is obtained from $\leq_{\mathbf{T}}$ by replacing each $T \in \mathbf{T}$ with $\mathsf{GndTr}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle}\langle T \rangle$, with $\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle$ the initial snapshot which T sees in $\leq_{\mathbf{T}}$ for initial database M to $\leq_{\mathbf{T}}$, as developed in Definition 4.2.

The *grounding* of $\mathsf{DSG}\langle \leq_{\mathbf{T}} : M \rangle$ for M, denoted $\mathsf{GndDSG}\langle \leq_{\mathbf{T}} : M \rangle$, is $\mathsf{DSG}\langle \leq_{\mathbf{T}:M} : M \rangle$. Technically, $\mathsf{DSG}\langle \leq_{\mathbf{T}:M} : M \rangle$ is a schedule on the set $\{\mathsf{GndTr}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle}\langle T \rangle \mid T \in \mathbf{T}\}$ of grounded transactions. However, since this notation is very cumbersome, when no confusion can result, $\mathsf{GndTr}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M \rangle}\langle T \rangle}\langle T \rangle$ will be represented by just T in the graph $\mathsf{GndDSG}\langle \leq_{\mathbf{T}} : M \rangle$, so that it becomes a graph with vertices in $\mathbf{T}$, just as is $\mathsf{DSG}\langle \leq_{\mathbf{T}} : M \rangle$. The difference is that $\mathsf{GndDSG}\langle \leq_{\mathbf{T}} : M \rangle$ has had all grounding reads excised, with the equivalent ground updates hardwired into the corresponding transactions.

Lemma 6.9 (CDSG cycle free implies constraint preserving) *Let $M \in \mathsf{ELDB}(\mathbf{D})$. If $\mathsf{GndDSG}\langle \leq_{\mathbf{T}} : M \rangle$ is free of cycles, then $\leq_{\mathbf{T}}$ is constraint preserving for initial state M.*

Proof Using exactly the same argument which establishes that $\leq_{\mathbf{T}}$ is serializable if $\mathsf{DSG}\langle \leq_{\mathbf{T}} : M \rangle$ is free of cycles, it follows that if $\mathsf{GndDSG}\langle \leq_{\mathbf{T}} : M \rangle$ is free of cycles, then $\leq_{\mathbf{T}:M}$ is serializable [1, Sec. 5.3]). Now, from initial state $M \in \mathsf{ELDB}(\mathbf{D})$, $\leq_{\mathbf{T}}$ and $\leq_{\mathbf{T}:M}$ produce exactly the same final state, since each transaction produces exactly the same output. Thus, since $\leq_{\mathbf{T}:M}$ is serializable for initial state M, it is a fortiori constraint preserving, whence $\leq_{\mathbf{T}}$ is constraint preserving for initial state M as well. $\square$

Theorem 6.10 (CSSI is constraint preserving) *For any $M \in \mathsf{ELDB}(\mathbf{D})$, if $\mathsf{GndDSG}\langle \leq_{\mathbf{T}} : M \rangle$ is free of potential dangerous structures, then $\leq_{\mathbf{T}}$ is constraint preserving for initial state M.*

Proof The argument is identical to that which shows that SSI works. If $\mathsf{GndDSG}\langle \leq_{\mathbf{T}} : M \rangle$ does not contain a potential dangerous structure, then it must be cycle free. (See Summary 5.19 and the references there for further information.) An invocation of Lemma 6.9 completes the proof. $\square$

Definition 6.11 (CSSI+CPSI) CSSI may be combined with CPSI to obtain a hybrid, which will be called *CPSI+CSSI.* The idea is analogous to that of CPSI+SSI — CPSI and CSSI are run in parallel, and as long as either one reports that the schedule is constraint preserving, the computation is allowed to continue. Since both CSSI (Theorem 6.10) and CPSI (Theorem 5.21) have been shown to be correct, it follows immediately that CSSI+CPSI is also correct.

Examples 6.12 (CPSI+SSI and CPSI+CSSI) It is instructive to illustrate the kind of situations which one but not the other of CPSI and CSSI flags as a false positive. First of all, in the context of $\mathbf{E}_A$, any three of the four transactions described in Examples 5.22 and presented visually in Fig. 5.4, form the basis for a potential dangerous structure in $\mathsf{GDSG}\langle \leq_{\mathbf{T}} : M \rangle$ for any $M \in \mathsf{ELDB}(\mathbf{E}_A)$, while all four together form a true cycle in that graph. On the other hand, no pair of these transactions forms a gw-pair, so CPSI certifies it to be constraint preserving for any initial state.

For an example which has perhaps a better ground in modelling financial transactions, return to the schema $\mathbf{E}_3$ of Examples 3.5, and consider three transactions. The transaction τ_{3e} has the update defined by the single assignment $x_1 \leftarrow x_1 - 50$, which may be thought of as a withdrawal from account x_1. The transaction τ_{3f} has the update defined by the two assignments $x_2 \leftarrow x_2 - 50$ and $y_1 \leftarrow y_1 + 50$, which may be thought of as a transfer from account x_2 to account y_1. Finally, the transaction τ_{3g} has the update defined by the two assignments $x_3 \leftarrow x_3 - 10$ and $y_2 \leftarrow y_2 + 10$, which may be thought of as a transfer from account x_3 to account y_2. Then $\tau_{3e} \xrightarrow{\mathsf{rw}} \tau_{3f} \xrightarrow{\mathsf{rw}} \tau_{3g}$ forms a potential dangerous structure in $\mathsf{CDSG}\langle \leq_{\mathbf{T}} : M\rangle$ for any $M \in \mathsf{ELDB}(\mathbf{D})$ for which these updates do not create any constraint violations when the transactions are run in isolation, yet no pair of these transactions forms a gw-pair. The problems is that CSSI does not distinguish rw-pairs involving "benign" updates, such as deposits, which cannot lead to a constraint violation, from more general writes. It is true, for example, that τ_{3f} writes the guard of τ_{3e}, but this write is guaranteed not to cause a constraint violation, so the definition of independence (see Definition 5.20) does not flag it as a problem.

The schema $\mathbf{E}'_1$, introduced in Sec. 1, provides the context for situations which CSSI flags as constraint preserving while CPSI produces false positives. For any fixed i with $0 \leq i \leq m_1 - 1$, any pair $\{\tau^d_{1ij_1}, \tau^d_{1ij_2}\}$ with $j_1 \neq j_2$ forms a gw-pair. Yet, the only potential dangerous structures are of the form $\tau^d_{1ij} \xrightarrow{\mathsf{rw}} \tau^d_{1i(j+1)\text{mod } n_3} \xrightarrow{\mathsf{rw}} \tau^d_{1i(j+2)\text{mod } n_3}$. In this case, CSSI does not flag any false positives which CPSI does not also flag.

Discussion 6.13 (Possible improvements to CPSI+CSSI) In CPSI+CSSI, the tests of each of the two components are independent of one another. One possible improvement would be to find a characterization which employs both simultaneously. It might be conjectured that for a schedule to fail constraint preservation, there must be a potential dangerous structure in which both concurrent pairs are also gw-pairs. Unfortunately, it is easy to show that this need not be the case. As a specific example, return once again to the schema $\mathbf{E}_3$ of Examples 3.5, and consider the three transactions τ_{3e} defined by $x_1 \leftarrow x_1 - 50$, τ_{3h} defined by $y_1 \leftarrow y_1 + 50$, and τ_{3r} defined by $y_1 \leftarrow y_1 - 100$, with $\{\tau_{3r}, \tau_{3e}\}$ and $\{\tau_{3e}, \tau_{3h}\}$ each concurrent pairs, as illustrated in Fig. 6.1. The arrangement $\tau_{3r} \xrightarrow{\mathsf{rw}} \tau_{3e} \xrightarrow{\mathsf{rw}} \tau_{3h}$ forms a potential dangerous structure in the CDSG, yet $\{\tau_{3e}, \tau_{3h}\}$ does not form a gw-pair.

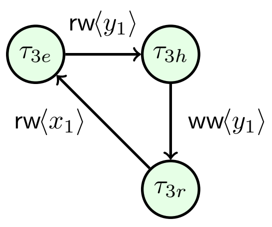

$\mathsf{rw}\langle y_1\rangle$
τ_{3e}
τ_{3h}
$\mathsf{rw}\langle x_1\rangle$
$\mathsf{ww}\langle y_1\rangle$
τ_{3r}

Figure 6.1: A potential dangerous structure containing only one gw-pair

On the other hand, it does appear that in any dangerous structure $T_1 \xrightarrow{\mathsf{rw}} T_2 \xrightarrow{\mathsf{rw}} T_3$, the first concurrent pair $\{T_1, T_2\}$ must also be a gw-pair. However, the proof requires the lengthy development of a number of properties of the DSG, and is beyond the scope of this paper, so for the time being, this is left as an unproven conjecture. Nevertheless, such a result would be very useful in limiting the number of false positives, and so increasing the attractiveness of a strategy which combines CPSI with a variant of SSI.

7 Conclusions and Further Directions

A method for identifying conflicts leading to violations of integrity constraints in transactions whose concurrency is governed by snapshot isolation has been presented. In contrast to methods for ensuring full serializability, the method of identification involves only pairs of transactions. It promises to have application in settings in which aborting and or delaying the execution of transactions is not a viable option.

There are several key areas for further work on this subject.

Prototype implementation: It would definitely be useful to implement a prototype of CPSI. This would be done using an open-source system such as PostgreSQL or MariaDB by modifying the source code to capture the reads performed by triggers and then feeding that information to the transaction manager. It would be particularly advantageous to modify a system which already supports SSI, since that would facilitate implementation of CPSI+SSI and even CPSI+CSSI.

Strategies for revising transactions: The motivation for this work arose from earlier studies on cooperative updates [14, 12]. The focus there is particularly upon interactive, long-running business processes in which abort and restart for transactions is not a viable option. Rather, the best strategy in such settings would seem to be to identify methods for cooperative revision of updates in the case of conflict. The current work constitutes a substantial step in that direction, in that the conflicts which are considered are between pairs of transactions, rather than large sets. The goal of exploiting the current work in that context is a subject for further study.

Integration with SSI and its variants: As suggested in Sec. 6, it may be fruitful to combine CPSI with SSI or CSSI, in order to obtain a strategy for constraint preservation which is superior to either one alone. This will require, in particular, the solution of some theoretical problems, as sketched in Discussion 6.13.

Integration with work on independence and overlap: In [11], the foundations for a theory of structured data objects for transactions is developed. These structured objects have both writeable parts and read-only parts, with the read-only parts allowed to overlap, even for writeable objects. As that work was also motivated by work on cooperative updates, an integration of those results with the ideas of this paper would likely prove a fruitful area for study.

Acknowledgments: M. Andrea Rodríguez, as well as the anonymous referees, made valuable comments on earlier drafts which have led to the improvement of this version.

References

[1] Adya, A., Liskov, B., O'Neil, P.E.: Generalized isolation level definitions. In: D.B. Lomet, G. Weikum (eds.) Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000, pp. 67–78 (2000)

[2] Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995, pp. 1–10 (1995)

[3] Bernstein, P., Newcomer, E.: Principles of Transaction Processing, second edn. Morgan Kaufmann (2009)

[4] Breitbart, Y., Georgakopoulos, D., Rusinkiewicz, M., Silberschatz, A.: On rigorous transaction scheduling. IEEE Trans. Software Eng. **17**(9), 954–960 (1991)

[5] Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. ACM Trans. Database Syst. **34**(4) (2009)

[6] Date, C.J.: A Guide to the SQL Standard. Addison-Wesley (1997). (with Hugh Darwen)

[7] Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, sixth edn. Addison Wesley (2011)

[8] Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Comm. ACM **19**(11), 624–633 (1976)

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

[10] Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983)

[11] Hegner, S.J.: A model of independence and overlap for transactions on database schemata. In: B. Catania, M. Ivanović, B. Thalheim (eds.) Advances in Databases and Information Systems, 14th East European Conference, ADBIS 2010, Novi Sad, Serbia, September 20-24, 2010, Proceedings, *Lecture Notes in Computer Science*, vol. 6295, pp. 209–223. Springer-Verlag (2010)

[12] Hegner, S.J.: A simple model of negotiation for cooperative updates on database schema components. In: Y. Kiyoki, T. Tokuda, A. Heimbürger, H. Jaakkola, N. Yoshida. (eds.) Frontiers in Artificial Intelligence and Applications XX11, pp. 154–173. IOS Press (2011)

[13] Hegner, S.J.: Guard independence and constraint-preserving snapshot isolation. In: C. Bierle, C. Meghini (eds.) Foundations of Information and Knowledge Systems: Eighth International Symposium, FoIKS 2014, Bordeaux, France, March 3-7, 2014, Proceedings, *Lecture Notes in Computer Science*, vol. 8367, pp. 231–250. Springer-Verlag (2014)

[14] Hegner, S.J., Schmidt, P.: Update support for database views via cooperation. In: Y. Ioannis, B. Novikov, B. Rachev (eds.) Advances in Databases and Information Systems, 11th East European Conference, ADBIS 2007, Varna, Bulgaria, September 29 - October 3, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4690, pp. 98–113. Springer-Verlag (2007)

[15] Kifer, M., Bernstein, A., Lewis, P.M.: Database Systems: An Application-Oriented Approach, second edn. Addison-Wesley (2006)

[16] Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Systems **6**(2), 213–226 (1981)

[17] Lin, Y., Kemme, B., Jiménez-Peris, R., Patiño-Martínez, M., Armendáriz-Iñigo, J.E.: Snapshot isolation and integrity constraints in replicated databases. ACM Trans. Database Systems **34**(2) (2009)

[18] Papadimitriou, C.: The Theory of Database Concurrency Control. Computer Science Press (1986)

[19] Ports, D.R.K., Grittner, K.: Serializable snapshot isolation in PostgreSQL. Proc. VLDB Endowment **5**(12), 1850–1861 (2012)

[20] Revilak, S., O'Neil, P.E., O'Neil, E.J.: Precisely serializable snapshot isolation (PSSI). In: Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, pp. 482–493 (2011)

[21] Rosenkrantz, D.J., Stearns, R.E., Lewis II, P.M.: Consistency and serializability in concurrent database systems. SIAM J. Comput. **13**(3), 508–530 (1984)

[22] Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, sixth edn. McGraw Hill (2011)

[23] Weikum, G., Vossen, G.: Transactional Information Systems. Morgan Kaufmann (2002)