Guard Independence and 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

Abstract. A method for detecting potential violations of integrity constraints of concurrent transactions running under snapshot isolation (SI) is presented. In contrast to methods for ensuring full serializability under snapshot isolation, violations of integrity constraints may be detected by examining certain read-write interaction of only two transactions at a time. The method, called *constraint-preserving snapshot isolation (CPSI)*, thus provides greater isolation than ordinary SI in that results do not violate any integrity constraints, while requiring substantially less overhead, and involving fewer false positives, than typical for enhancements to SI which guarantee full serializable isolation.

1 Introduction

Over the course of the past few decades, *snapshot isolation (SI)* has become one of the preferred modes of transaction isolation for concurrency control in database-management systems (DBMSs). In SI, each transaction operates on its own private copy of the database (its *snapshot*). To implement commit for concurrent transactions, the results of these individual snapshots must be integrated. If there is a write conflict; that is, if more than one transaction writes the same data object, then only one transaction is allowed to commit. The others must abort if they are not naturally terminated in some other way.

On the one hand, SI avoids many of the update anomalies associated with policies such as read uncommitted (RU) and read (latest) committed (RC), such as dirty and nonrepeatable reads, respectively [6, p. 61]. On the other hand, with the now widespread use of multiversion concurrency control (MVCC), it admits very efficient implementation, avoiding many of the performance bottlenecks associated with lock-based *rigorous two-phase locking (rigorous 2PL)* [4], more commonly called *strong strict two-phase locking (SS2PL)* nowadays. Nevertheless, it does allow certain types of undesirable behavior which do not occur under view serialization, such as read and write skew [2].

Because true view serializability [14, Sec. 2.4] is the gold standard for isolation of transactions, there has been substantial recent interest in extending SI to achieve such true serializablity, the idea being to achieve the desirable properties of true serializablity while exploiting the efficiency of SI. As a consequence, *serializable SI* (SSI), has been developed [8, 5]. In stark contrast to

SS2PL, SSI is an optimistic approach. On top of standard SI, it looks for *dangerous structures*, which are sequences of two consecutive read-write edges of concurrent transactions in the multiversion conflict graph for the transactions. If such a structure is found, one of the participating transactions is required to terminate without committing its results. The existence of a dangerous structure in the conflict graph is a necessary condition for the nonserializablity of a set of concurrent transactions under SI, but it is not a sufficient one. Thus, the SSI strategy is subject to false positives. To illustrate, let $n \geq 2$ be a natural number and let $\mathbf{E}_0$ be a database schema which includes n integer-valued data objects d_0, d_1, …, d_{n-1}. Let τ_i be the transaction which replaces the value of d_i with the current value of $d_{(i+1) \bmod n}$; i.e., which executes $d_i \leftarrow d_{(i+1) \bmod n}$. Running the set $\mathbf{T}_0 = \{\tau_i \mid 0 \leq i < n\}$ 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 $d_{(i+1) \bmod n}$, 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_i is run first and commits before any other transaction begins, then the old value of d_i will be overwritten before $\tau_{(i+1) \bmod n}$ is able to read it. Thus, $\mathbf{T}_0$ is not view serializable. Formally, there is a read-write dependency [8, Def. 2.2] (or *antidependency* [1, 4.4.2]) from $\tau_{i \bmod n}$ to $\tau_{(i+1) \bmod n}$ for data object $d_{(i+1) \bmod n}$; these dependencies are represented using the *multiversion serialization graph* or *multiversion conflict graph* as illustrated in Fig. 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_i is removed, then execution in the serial order $\tau_{i+1}\tau_{i+2}\ldots\tau_{n-1}\tau_0\tau_1\ldots\tau_{i-1}$ is equivalent to concurrent execution under SI. Thus, for any natural number n, there is a set $\mathbf{T}$ of n 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. Since a dangerous structure of SSI [5] involves at most three transactions, the approach must necessarily involve false positives. Nevertheless, benchmarks reported in [5] are impressive, but of course the transaction mix must be taken into account. 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 [15].

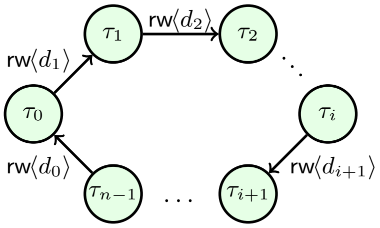

Fig. 1. An SI rw-conflict cycle of length n

In *Precisely SSI (PSSI)* [16], the entire multiversion conflict graph is constructed. This avoids virtually all false positives, but may involve a high overhead for transaction mixes involving long cycles, although reported benchmarks are favorable.

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 requires frequent aborts and/or waits is highly undesirable. Interactive business processes are one such domain, and it is in particular the context of cooperative transactions within that setting [13, 11] which motivated the work of this paper. The central notion is an augmentation of SI, named *constraint-preserving SI (CPSI)*, which ensures that all integrity constraints will be satisfied. It is strictly weaker than SSI, in that nonserializable behavior which does not result in constraint violation is not ruled out. On the other hand, CPSI involves only a relatively simple check of a property of the conflict graph which, at least under one definition, is both necessary and sufficient to guarantee constraint satisfaction; that is, it does not produce any false positives. To illustrate, let $\mathbf{E}_1$ be identical to $\mathbf{E}_0$, save that the constraint $\varphi_1 = \sum_{i=0}^{n-1} d_i > 100 \cdot n$ is enforced. In concrete terms, think of each d_i as the balance in a bank account, and the constraint requiring that the average of the balances must exceed 100. Let $\mathbf{T}_1 = \{\tau'_i \mid 0 \leq i \leq n-1\}$ be the set of transactions on $\mathbf{E}_1$ with τ'_i defined by $d_i \leftarrow d_i - 1$. This is a generalization of the write-skew example of [2]. In order to determine whether the update to d_i will preserve the constraint φ_1, it is necessary for τ'_i to read *every* other d'_j. This means that there is a read-write conflict between *any* two τ_{j_1}, τ_{j_2}, as illustrated in Fig. 2. As a specific example, let M_{10} be the database which has $d_0 = 101$ and $d_j = 100$ for $1 \leq j \leq n-1$. Then any single transaction from $\mathbf{E}_1$, run in isolation, preserves φ_1, while the concurrent execution under SI of any two distinct members of $\mathbf{E}_1$ does not. The main result of this paper is that this two-element characterization holds in general; if a set of transactions running under SI results in a constraint violation, then there is a two-element subset which has this property when run on some legal database. This may in fact be further refined if transaction reads are separated into those required to verify constraints (called *guard reads*) and those required for other reasons. If the multiversion conflict graph is free of two-element cycles consisting of read-write edges when only writes and guard reads are considered, then the transactions under consideration cannot cause a constraint violation when run concurrently under SI, regardless of the initial database to which the transactions are applied.

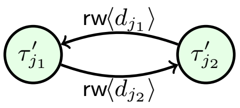

Fig. 2. An SI rw-conflict cycle of length 2

Although it may seem unacceptable to allow nonserializable results, this is in fact done all the time. For reasons of efficiency, lack of true serializability is routinely accepted with lower levels of isolation, such as RU and RC. 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.

2 Schemata, Views, and Updates

Although the ideas surrounding database schemata, views, and updates which are used in this paper should already be familiar to the reader, the specific notation and conventions which are used nevertheless need to be spelled out carefully. While similar in many aspects to those used in previous papers, such as [9], [10], and [12], the framework employed here also differs in substantial ways. In particular, database schemata, while still being modelled by their sets of states, are characterized by both their overall state sets (which need not satisfy the integrity constraints) and their legal states (which must satisfy the integrity constraints). Furthermore, for the order and lattice structure of views, the syntactic congruence (on all states, ignoring the integrity constraints) rather than the semantic congruence (on just the legal states, taking into account equivalences implied by the integrity constraints), is employed. Therefore, while a notation consistent with these earlier works has been used wherever possible, it seems best to provide a self-contained presentation, with an acknowledgment that much has been drawn from those previous works.

For concepts related to order and lattices, the reader is referred to [7] for further clarification of notions utilized in this paper.

Summary 2.1 (Database schemata and views). A *database schema* $\mathbf{D}$ is characterized by two sets: $\mathsf{DB}(\mathbf{D})$ is the collection of all databases (or *states*), while $\mathsf{LDB}(\mathbf{D})$ is the subset of $\mathsf{DB}(\mathbf{D})$ consisting of just the *legal* databases (or *legal states*); i.e., those which satisfy the integrity constraints of the schema. In describing examples, it may be useful to identify explicitly a set $\mathsf{Constr}(\mathbf{D})$ of *constraints*, with $\mathsf{LDB}(\mathbf{D})$ consisting of precisely those members of $\mathsf{DB}(\mathbf{D})$ which satisfy the elements of $\mathsf{Constr}(\mathbf{D})$. However, such an explicit representation of constraints is not essential to the theory.

Given database schemata $\mathbf{D}_1$ and $\mathbf{D}_2$, a *database morphism* $f : \mathbf{D}_1 \to \mathbf{D}_2$ is a function $f : \mathsf{DB}(\mathbf{D}_1) \to \mathsf{DB}(\mathbf{D}_2)$. The morphism f is *semantic* if for every $M \in \mathsf{LDB}(\mathbf{D}_1)$, $f(M) \in \mathsf{LDB}(\mathbf{D}_2)$; i.e., if it maps legal databases to legal databases. The morphism f is said to be *fully surjective* if it is semantic, the function $f : \mathsf{DB}(\mathbf{D}_1) \to \mathsf{DB}(\mathbf{D}_2)$ is surjective, and for every $M_2 \in \mathsf{LDB}(\mathbf{D}_2)$, there is an $M_1 \in \mathsf{LDB}(\mathbf{D}_1)$ with $f(M_1) = M_2$. In other words, it is fully surjective if it is surjective as a function on all databases and also when it is restricted to just the legal databases of both $\mathbf{D}_1$ and $\mathbf{D}_2$.

A *view* over the database schema $\mathbf{D}$ is a pair $\Gamma = (\mathbf{V}, \gamma)$ in which $\mathbf{V}$ is a database schema with $\gamma : \mathbf{D} \to \mathbf{V}$ a fully surjective morphism. The set of all views on $\mathbf{D}$ is denoted $\mathsf{Views}(\mathbf{D})$. Full surjectivity is a natural property. By its very nature, the states (resp. legal states) of a view are determined by the states (resp. legal states) of the main schema. The notation for states of schemata extends naturally to views. Given a view $\Gamma = (\mathbf{V}, \gamma)$, $\mathsf{DB}(\Gamma)$ and $\mathsf{LDB}(\Gamma)$ are alternate notation for $\mathsf{DB}(\mathbf{V})$ and $\mathsf{LDB}(\mathbf{V})$ respectively.

The *syntactic congruence* of Γ is $\mathsf{SynCongr}(\Gamma) = \{(M_1, M_2) \in \mathsf{DB}(\mathbf{D}) \times \mathsf{DB}(\mathbf{D}) \mid \gamma(M_1) = \gamma(M_2)\}$. The *syntactic preorder* $\preceq_{\mathbf{D}}$ on $\mathsf{Views}(\mathbf{D})$ is defined by $\Gamma_1 \preceq_{\mathbf{D}} \Gamma_2$ iff $\mathsf{SynCongr}(\Gamma_2) \subseteq \mathsf{SynCongr}(\Gamma_1)$. The intuitive idea behind this order is that if $\mathit{\Gamma}_1 \preceq_{\mathbf{D}} \mathit{\Gamma}_2$, then $\mathit{\Gamma}_2$ preserves at least as much information about the state of the main schema $\mathbf{D}$ as does $\mathit{\Gamma}_1$.

For $\mathit{\Gamma}_1 = (\mathbf{V}_1, \gamma_1)$ and $\mathit{\Gamma}_2 = (\mathbf{V}_2, \gamma_2)$, with $\mathit{\Gamma}_2 \preceq_{\mathbf{D}} \mathit{\Gamma}_1$, $\mathit{\Gamma}_2$ may be regarded as a view of $\mathit{\Gamma}_1$. More precisely, define the relative morphism $\lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle : \mathbf{V}_1 \to \mathbf{V}_2$ to be the unique function $\lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle : \mathsf{DB}(\mathbf{V}_1) \to \mathsf{DB}(\mathbf{V}_2)$ which satisfies $\lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle \circ \gamma_1 = \gamma_2$. See [9, Def. 2.3] for a elaboration of this concept.

The *identity view* $\mathbf{1_D}$ of $\mathbf{D}$ has schema is $\mathbf{D}$ and morphism the identity $\mathbf{D} \to \mathbf{D}$. Similarly, a *zero view* $\mathbf{0_D}$ of $\mathbf{D}$ has a schema which has only one database, with the view whose schema is a one element set, with the view morphism sending every element of $\mathsf{DB}(\mathbf{D})$ to the unique element of that set. It is immediate that for any $\mathit{\Gamma} \in \mathsf{Views}(\mathbf{D})$, $\mathbf{0_D} \preceq_{\mathbf{D}} \mathit{\Gamma} \preceq_{\mathbf{D}} \mathbf{1_D}$.

Summary 2.2 (Updates). An update on $\mathbf{D}$ is a pair $\langle M_1, M_2\rangle \in \mathsf{LDB}(\mathbf{D}) \times \mathsf{LDB}(\mathbf{D})$. M_1 is the current or old state before the update, and M_2 is the new state afterwards. Note that updates always transform legal states to legal states. The set of all updates on $\mathbf{D}$ is denoted $\mathsf{Updates}(\mathbf{D})$.

Updates are often identified by name; therefore, it is convenient to have a shorthand for its components. To this end, if $u \in \mathsf{Updates}(\mathbf{D})$, then write $u^{(1)}$ and $u^{(2)}$ for the values of the state before and after the update, respectively; i.e., $u = \langle u^{(1)}, u^{(2)}\rangle$. The *composition* $u_1 \circ u_2$ of two updates $u_1, u_2 \in \mathsf{Updates}(\mathbf{D})$ is just their composition in the sense of mathematical relations. More precisely, $u_1 \circ u_2 = \{(M_1, M_3) \mid (\exists M_2 \in \mathsf{LDB}(\mathbf{D}))((M_1, M_2) \in u_1 \wedge (M_2, M_3) \in u_2)\}$.

It will also prove useful to be able to select just those updates which apply to a specific state. For $N \in \mathsf{LDB}(\mathbf{D})$, define $\mathbf{u}_{|N} = \{u \in \mathbf{u} \mid u^{(1)} = N\}$.

It will sometimes be necessary to map updates from one view to a second, smaller one. Recall the relative morphism $\lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle : \mathbf{V}_1 \to \mathbf{V}_2$ defined in Summary 2.1 above. Then, for $u \in \mathsf{Updates}(\mathit{\Gamma}_1)$, define $\lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle(u) = \langle \lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle(u^{(1)}), \lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle(u^{(2)})\rangle$, and for $\mathbf{u} \subseteq \mathsf{Updates}(\mathit{\Gamma}_1)$, define $\lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle(\mathbf{u}) = \{\lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle(u) \mid u \in \mathbf{u}\}$. The set $\mathbf{u} \subseteq \mathsf{Updates}(\mathit{\Gamma}_1)$ is *a partial identity* on $\mathit{\Gamma}_2$ if $\lambda\langle \mathit{\Gamma}_1, \mathit{\Gamma}_2\rangle(\mathbf{u})$ is a subset of the identity relation on $\mathsf{LDB}(\mathbf{V}_2)$. In this case, it is said that $\mathbf{u}$ *holds* $\mathit{\Gamma}$ *constant.* The single update $u \in \mathsf{Updates}(\mathbf{V}_1)$ is *a partial identity* on $\mathit{\Gamma}_2$ if $\{u\}$ has this property.

Finally the notational conventions which permit view names to be used in lieu of their components is extended to updates. Specifically, for $\mathit{\Gamma} = (\mathbf{V}, \gamma)$, $\mathsf{Updates}(\mathit{\Gamma})$ will be used as an alternate notation for $\mathsf{Updates}(\mathbf{V})$.

Definition 2.3 (Algebras of updateable views). In an investigation of the interaction of transactions, it is central to be able to model their read-write interaction; for example, to express succinctly that transaction T_1 does or does not write some data object which T_2 reads. Transactions typically read and write compound data objects (e.g., sets of rows or tuples in the relational context) rather than just single primitive objects (e.g., single rows or tuples). In order to express such interaction, it is convenient to model compound data objects as being built up from simple ones in a systematic way. The natural mathematical structure for such a model is the Boolean algebra [7, p. 94]. It is assumed that the reader is familiar with that notion; only notation will be recalled here. In the Boolean algebra $\mathbf{L} = (L, \vee, \wedge, \overline{}, \top, \bot)$, L is the underlying set, $\vee$ and $\wedge$ are the join and meet operators, respectively, $\top$ and $\bot$ are the identity and zero elements, respectively, and $\overline{}$ is the complement operator, which is written as an overbar; i.e., the complement of x is $\overline{x}$. The join operation induces a partial order via $a \leq b$ iff $a \vee b = b$, called the *underlying partial order.* The order with equality excluded is denoted $<$. An *atom* $a \in L$ is a minimal element which is greater than $\bot$; i.e., $\bot < a$ and for no $b \in L$ is it the case that $\bot < b < a$. The set of all atoms of $\mathbf{L}$ is denoted $\mathsf{Atoms}_{\mathbf{L}}$. If $\mathbf{L}$ is finite, then every $a \in L$ has a unique representation as the join of atoms. In this case, define the *basis* of a to be $\mathsf{Basis}_{\mathbf{L}}\langle a \rangle = \{x \in \mathsf{Atoms}_{\mathbf{L}} \mid x \leq a\}$ [7, 5.5].

Now, given a database schema $\mathbf{D}$, an *algebra of updateable views* over $\mathbf{D}$ is a finite Boolean algebra $\boldsymbol{\mathcal{V}} = (\mathcal{V}, \vee, \wedge, \overline{}, \top, \bot)$ with $\mathcal{V} \subseteq \mathsf{Views}(\mathbf{D})$, whose underling partial order is the restriction of $\preceq_{\mathbf{D}}$ to $\mathcal{V}$. This requirement on the order structure has an important consequence. Given a view $\varGamma \in \mathcal{V}$, there is a natural correspondence between $\mathsf{DB}(\varGamma)$ and $\{\mathsf{DB}(\varGamma') \mid \varGamma' \in \mathsf{Basis}_{\mathcal{V}}\langle \varGamma \rangle\}$. Specifically, each $M \in \mathsf{DB}(\varGamma)$ has a unique representation as the set $\{\lambda\langle \varGamma, \varGamma' \rangle(M) \mid \varGamma' \in \mathsf{Basis}_{D}\langle \varGamma \rangle\}$. There is a bit of notational shorthand, based upon this observation, which will prove useful. If $\varGamma_1 = (\mathbf{V}_1, \gamma_1), \varGamma_2 = (\mathbf{V}_2, \gamma_2) \in \mathcal{V}$, with $\varGamma_1 \wedge \varGamma_2 = \bot$, and if $M_1 \in \mathsf{DB}(\varGamma_1)$ and $M_2 \in \mathsf{DB}(\varGamma_2)$, then $M_1 \vee M_2 \in \mathsf{DB}(\varGamma_1 \vee \varGamma_2)$ denotes the unique state with the representation $\{\lambda\langle \varGamma, \varGamma' \rangle(M) \mid \varGamma' \in \mathsf{Basis}_{D}\langle \varGamma_1 \rangle \cup \mathsf{Basis}_{D}\langle \varGamma_2 \rangle\}$.

The least element of $\boldsymbol{\mathcal{V}}$ is formally a zero view $\mathbf{0}_{\mathbf{D}}$, but will frequently be written as $\bot$.

Example 2.4 (The algebra of updateable views of $\mathbf{E}_0$ and $\mathbf{E}_1$). For the schema $\mathbf{E}_0$ introduced in Sec. 1, let Let $\varOmega_{d_i} = (\mathbf{W}_{d_i}, \omega_{d_i})$ be the view which retains just d_i, discarding all d_j for $j \neq i$. Thus, $\mathbf{W}_{d_i}$ contains just the data object d_i, while the view morphism $\omega_{d_i} : \mathbf{E}_0 \to \mathbf{W}_{d_i}$ retains just d_i from $D_{\mathbf{E}_0} = \{d_j \mid 0 \leq j \leq n-1\}$. $\{\varOmega_{d_j} \mid 0 \leq j \leq n-1\}$ forms the set $\mathsf{Atoms}_{\boldsymbol{\mathcal{V}}_{\mathbf{E}_0}}$ of atoms of the algebra $\boldsymbol{\mathcal{V}}_{\mathbf{E}_0} = (\mathcal{V}_{\mathbf{E}_0}, \vee, \wedge, \overline{}, \top, \bot)$ of updateable views associated with $\mathbf{E}_0$. Each element of $\mathcal{V}_{\mathbf{E}_0}$ is of the form $\bigvee_{j \in S} \varOmega_{d_j}$ for some $S \subseteq \{0, 1, \ldots, n-1\}$. In other words, the members of $\mathcal{V}_{\mathbf{E}_0}$ are in bijective correspondence with subsets of $D_{\mathbf{E}_0}$. The zero view corresponds to the empty set $\emptyset$, while the identity view corresponds to the entire set $D_{\mathbf{E}_0}$. Join and meet correspond to union and intersection on $\{d_j \mid 0 \leq j \leq n-1\}$, respectively. In short, $\boldsymbol{\mathcal{V}}_{\mathbf{E}_0} = (\mathcal{V}_{\mathbf{E}_0}, \vee, \wedge, \overline{}, \top, \bot)$ is isomorphic to the power-set algebra [7, 4.18(1)] of $D_{\mathbf{E}_0}$. The algebra for $\mathbf{E}_1$ is identical.

Definition 2.5 (The algebra of σ-views of a relational schema). It is instructive to illustrate the ideas of Definition 2.3 within a framework which recaptures common usage. Let $\mathbf{D}$ be a relational schema, and suppose that *row-level granularity* in relational DBMSs, in which the smallest grain of data access for a transaction is a single row (or tuple or atom), is employed. Let $\mathsf{GrAtoms}\langle \mathbf{D} \rangle$ denote the set of all ground atoms of $\mathbf{D}$ (tuples not involving variables, the values for columns/attributes are domain values only). In practice, $\mathsf{GrAtoms}\langle \mathbf{D} \rangle$ is always a finite set; this finiteness restriction is assumed to hold here as well.

Further, assume that tuples are tagged with the relations in which they occur, so that it is not necessary to identify relations explicitly in selections. For $t \in \mathsf{GrAtoms}\langle \mathbf{D} \rangle$, define $\varSigma_t = (\boldsymbol{\Upsilon}_t, \sigma_t)$ to be the view which selects t from the appropriate relation and discards everything else. $\mathsf{DB}(\boldsymbol{\Upsilon}_t) = \{\emptyset, \{t\}\}$; that is, there are only two states to the view schema, one representing that t is present in the main schema, and the other that it is not. On $\varSigma_t$, the only update operations which are possible are to delete t, to insert t, and to do nothing. The set of all such selections on a single ground tuple, $\varSigma\text{-}\mathsf{GrAtoms}\langle \mathbf{D} \rangle = \{\varSigma_t \mid t \in \mathsf{GrAtoms}\langle \mathbf{D} \rangle\}$, forms the set of atoms for the lattice of data objects.

Extending this to compound objects, if $S = \{t_1, t_2, \ldots, t_n\} \subseteq \mathsf{GrAtoms}\langle \mathbf{D} \rangle$, then $\varSigma_S = (\boldsymbol{\Upsilon}_S, \sigma_S)$ denotes the selection on all of S. The logical operation connecting the tuples is disjunction; all of the t_i's are selected. A longer but more descriptive representation might be $\varSigma_{\{t_1,t_2,\ldots,t_n\}} = (\boldsymbol{\Upsilon}_{\{t_1,t_2,\ldots,t_n\}}, \sigma_{t_1 \vee t_2 \vee \ldots \vee t_n})$. The point here is that the view $\varSigma_S$ is the join of a unique set of primitive selections, namely, $\{\varSigma_{t_i} \mid 1 \leq i \leq n\}$. As a degenerate but nevertheless very useful case, note also that $\varSigma_\emptyset$ is a zero view with $\emptyset$ as the only view state; no updates are possible on it.

Define $\varSigma\text{-}\mathsf{GrAtoms}\langle \mathbf{D} \rangle = \{\varSigma_t \mid t \in \mathsf{GrAtoms}\langle \mathbf{D} \rangle\}$, the set of all selections on a single ground tuple, and define $\varSigma\text{-}\mathsf{Views}\langle \mathbf{D} \rangle = \{\varSigma_S \mid S \subseteq \varSigma\text{-}\mathsf{GrAtoms}\langle \mathbf{D} \rangle\}$. Then $\varSigma\text{-}\mathsf{GrAtoms}\langle \mathbf{D} \rangle$ is the set of atoms of the lattice of updateable views whose elements are $\varSigma\text{-}\mathsf{Views}\langle \mathbf{D} \rangle$. Join, meet, and complement are given by union, intersection, and complement on the underlying sets of tuples: $\varSigma_{S_1} \vee \varSigma_{S_2} = \varSigma_{S_1 \cup S_2}$, $\varSigma_{S_1} \wedge \varSigma_{S_2} = \varSigma_{S_1 \cap S_2}$, and the complement $\overline{\varSigma_S}$ of $\varSigma_S$ is $\varSigma_{\mathsf{GrAtoms}\langle \mathbf{D} \rangle \setminus S}$.

Notation 2.6 (Notation for views). Views appear frequently in that which follows, and it is convenient to have a uniform convention for identifying constituent parts. If $\varGamma$ is the name of a view, possibly with subscripts or other annotations, then $\mathbf{V}$ and γ will be used to denote its schema and morphism, respectively with the same annotations. Thus, for the views $\varGamma$, $\varGamma'$, $\varGamma_1$, and $\overline{\varGamma_1}$, the full expansions are assumed to be $\varGamma = (\mathbf{V}, \gamma)$, $\varGamma' = (\mathbf{V}', \gamma')$, $\varGamma_1 = (\mathbf{V}_1, \gamma_1)$, and $\overline{\varGamma_1} = (\overline{\mathbf{V}}_1, \overline{\gamma_1})$.

Abstract views, as used in definitions and theorems, will always use the (possibly annotated) $\varGamma = (\mathbf{V}, \gamma)$ notation. For specific examples, an alternate notation, using $\varOmega = (\mathbf{W}, \omega)$, also with annotations, will be used. In this way, example views are always clearly distinguished from abstract ones. The same conventions apply; for example, the full expansion of $\overline{\varOmega_1'}$ is $\overline{\varOmega_1'} = (\overline{\mathbf{W}_1'}, \overline{\omega_1'})$.

Notation 2.7 (Some mathematical shorthand). 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 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 $f(x)$ is undefined on x.

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

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

3 Snapshot Isolation

In this section, an overview of snapshot isolation is presented and the concurrency issues surrounding it are summarized. It is assumed that the reader has a basic understanding of transactions, and in particular serializability, as is presented in [18], [3], and Chapters 14-15 of the textbook [17].

Summary 3.1 (The transaction model of snapshot isolation). 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.

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 transactions, and other transactions do not see its updates. When T finishes, its updates must be committed to the global database. Such commits are governed by the *first-committer wins (FCW)* rule. If any other transaction T' which is concurrent with T, and which has already committed has written a data object $\mathit{\Gamma}$ which T has also written, then T is not allowed to commit.

Summary 3.2 (Variations of SI in practice). In practice, things are not quite as simple as sketched in Summary 3.1 for at least two reasons. First of all, the rule for conflict resolution which is used in practice is most often that which is called *first updater wins (FUW)*. With FUW, if some other concurrent transaction T' writes a data object $\mathit{\Gamma}$ 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 aborted) or T' aborts (in which case T is allowed to continue). 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, FUW may be used in lieu of FCW with no loss of generality. It will be used here 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.

A second reason why the FCW model is somewhat idealistic is that in most implementations of SQL, primary-key and uniqueness constraints are enforced immediately, and unless checking is declared to be `deferrable` and then `deferred`, foreign-key constraints will also be enforced immediately. In this work, which is of a more foundational nature, this “implementation detail” will be ignored. In any case, for the purposes of this work, integrity constraints include not only the usual database dependencies (e.g., key and foreign-key dependencies), but also rules specified in triggers and possibly even application programs. The latter two are often of central importance in business processes.

4 Constraint Preservation and Its Characterization

Notation 4.1 (Notational conventions). Throughout this section, unless stated specifically to the contrary, take $\mathbf{D}$ to be a database schema and $\boldsymbol{\mathcal{V}} = (\mathcal{V}, \vee, \wedge, ^{-}, \top, \bot)$ a (finite Boolean) algebra of updateable views, as described in Summary 2.1 and Definition 2.3, respectively.

Definition 4.2 (Updateable objects). It is useful to combine an updateable view and the updates which may be applied to it into one package. Formally, an *updateable object* over $\boldsymbol{\mathcal{V}}$ is a pair $\langle \Gamma, \mathbf{u} \rangle$ with $\Gamma \in \mathcal{V}$ and $\mathbf{u} \subseteq \mathsf{Updates}(\Gamma)$. The set of all updateable objects over $\boldsymbol{\mathcal{V}}$ is denoted $\mathsf{UpdObj}(\boldsymbol{\mathcal{V}})$.

Call the updateable object $\langle \Gamma, \mathbf{u} \rangle$ *functional* if for any two $u_1, u_2 \in \mathbf{u}$, $u_1^{(1)} = u_2^{(1)}$ implies $u_1^{(2)} = u_2^{(2)}$. Thus, if $\langle \Gamma, \mathbf{u} \rangle$ is functional, there is at most one applicable update in $\mathbf{u}$ for each legal state of the associated view. The set of all functional updateable objects over $\boldsymbol{\mathcal{V}}$ is denoted $\mathsf{FUpdObj}(\boldsymbol{\mathcal{V}})$.

The *write view* of $\langle \Gamma, \mathbf{u} \rangle$ is the largest view $\Gamma^{(w)} \in \mathcal{V}$ with $\Gamma^{(w)} \preceq_{\mathbf{D}} \Gamma$ and the property that for some $u \in \mathbf{u}$, $\lambda\langle \Gamma, \Gamma^{(w)} \rangle (u^{(1)}) \neq \lambda\langle \Gamma, \Gamma^{(w)} \rangle (u^{(2)})$. The *read-only view* of $\langle \Gamma, \mathbf{u} \rangle$ is the largest view $\Gamma^{(r)} \in \mathcal{V}$ with $\Gamma^{(r)} \preceq_{\mathbf{D}} \Gamma$ and the property that for all $u \in \mathbf{u}$, $\lambda\langle \Gamma, \Gamma^{(r)} \rangle (u^{(1)}) = \lambda\langle \Gamma, \Gamma^{(r)} \rangle (u^{(2)})$. In other words, $\Gamma^{(r)}$ is the largest subview on which $\mathbf{u}$ is a partial identity. Clearly $\{\Gamma^{(r)}, \Gamma^{(w)}\}$ forms a decomposition of Γ into disjoint components; i.e., $\Gamma^{(r)} \wedge \Gamma^{(w)} = \bot$ and $\Gamma^{(r)} \vee \Gamma^{(w)} = \Gamma$.

For $\Gamma \in \mathcal{V}$ with $\Gamma' \preceq_{\mathbf{D}} \Gamma$, define the *projection* of $\langle \Gamma, \mathbf{u} \rangle$ onto Γ' to be the updateable object $\mathsf{Proj}_{\Gamma'}\langle\langle \Gamma, \mathbf{u} \rangle\rangle = \langle \Gamma', \lambda\langle \Gamma, \Gamma' \rangle(\mathbf{u}) \rangle$. In general, such a projection will not be functional, even if $\langle \Gamma, \mathbf{u} \rangle$ is. The projection $\mathsf{Proj}_{\Gamma^{(w)}}\langle\langle \Gamma, \mathbf{u} \rangle\rangle$ is called the *write object* of $\langle \Gamma, \mathbf{u} \rangle$.

The restriction of the write object of a functional updateable object to a single database $M \in \mathsf{LDB}(\mathbf{D})$ is, however, always functional, since there is at most one update in $\mathbf{u}$ which is applicable to M. Formally, define $\mathsf{Proj}_{\langle \Gamma' | M \rangle}\langle \Gamma, \mathbf{u} \rangle = \langle \Gamma', \lambda\langle \Gamma, \Gamma' \rangle(\mathbf{u}_{|M}) \rangle$.

Recall that $\mathbf{0_D}$ is the zero view on $\mathbf{D}$, and let $\mathbf{u_{0_D}}$ denote the set containing just one element, the unique (identity) update $u_{\mathbf{0_D}}$ on the singleton set $\mathsf{LDB}(\mathbf{0_D})$. $\langle \mathbf{0_D}, \mathbf{u_{0_D}} \rangle$, the unique updateable object of the zero view $\mathbf{0_D}$, is called the *zero object.* By itself, this updateable object is uninteresting, since no nontrivial updates are possible, but it will prove to be useful as a tool to assert succinctly that two views are disjoint (by asserting that their meet in the algebra $\boldsymbol{\mathcal{V}}$ of updateable views is the zero view).

Examples 4.3 (Read views and write views). The decomposition of the view of an updateable object into its write view and its read view is central, and deserves a closer look. To begin, consider again the schema $\mathbf{E}_0$ and the set $\mathbf{T}_0$ of transactions introduced in Sec. 1, with the corresponding algebra of updateable views described in Example 2.4. For the update family defined by τ_i; i.e., by $d_i \leftarrow d_{(i+1) \bmod n}$, the associated view is $\varOmega_{d_i} \vee \varOmega_{d_{(i+1) \bmod n}}$, with $\varOmega_{d_i}$ the write view and $\varOmega_{d_{(i+1) \bmod n}}$ the read-only view. The family of updates itself is $\boldsymbol{\upsilon}_{d_i} = \{\langle (n_1, n_2), (n_2, n_2) \rangle \mid n_1, n_2 \in \mathbb{Z}\}$, with a pair (n_1, n_2) representing the values for $(d_i, d_{(i+1) \bmod n})$. The updateable object is thus $\langle \varOmega_{d_i} \vee \varOmega_{d_{(i+1) \bmod n}}, \boldsymbol{\upsilon}_{d_i} \rangle$.

This simple introductory example does not cover all aspects of the framework. For a more comprehensive examination of the ideas, let $\mathbf{E}_2$ be a database schema which, for a fixed $q \in [3, \text{-}]$, includes three sets of data objects $\{x_1, x_2, \ldots, x_q\}$, $\{y_1, y_2, \ldots, y_q\}$, and $\{z_1, z_2, \ldots, z_q\}$, governed by the constraints $x_i + y_i \geq 500$ for each $i \in [1, q]$. Assume that each data object takes integer values, and in concordance with Definition 2.3, regard each x_i (resp. y_i) as a view $\varOmega_{x_i}$ (resp. $\varOmega_{y_i}$).

Fix $i \in [1, q]$, and let $\boldsymbol{\upsilon}_{x_i}$ be the update set on $\varOmega_{x_i}$ defined by $x_i \leftarrow x_i - 100$; more precisely, $\boldsymbol{\upsilon}_{x_i} = \{\langle n, n-100 \rangle \mid n \in \mathbb{Z}\}$. Define $\boldsymbol{\upsilon}_{y_i}$ on $\varOmega_{y_i}$ similarly by $y_i \leftarrow y_{i-1} - 100$. The pairs $\langle \varOmega_{x_i}, \boldsymbol{\upsilon}_{x_i} \rangle$ and $\langle \varOmega_{y_i}, \boldsymbol{\upsilon}_{y_i} \rangle$ then form the associated updateable objects. In these simple cases, $\varOmega_{x_i}^{(w)} = \varOmega_{x_i}$ and $\varOmega_{y_i}^{(w)} = \varOmega_{y_i}$, with $\varOmega_{x_i}^{(r)} = \varOmega_{y_i}^{(r)} = \bot$.

Next, consider the update operation $x_i \leftarrow x_i - z_i$ on the view $\varOmega_{x_i} \vee \varOmega_{z_i}$. The set of associated updates is $\boldsymbol{\upsilon}_{x_i z_i} = \{\langle (n_1, n_2), (n_1 - n_2, n_2) \rangle \mid n_1, n_2 \in \mathbb{Z}\}$, with n_1 and n_2 representing the values of x_i and y_i, respectively. Here $(\varOmega_{x_i} \vee \varOmega_{z_1})^{(w)} = \varOmega_{x_i}$ and $(\varOmega_{x_i} \vee \varOmega_{z_1})^{(r)} = \varOmega_{z_i}$. The projection $\mathsf{Proj}_{\varOmega_{x_i}}\langle\langle \varOmega_{x_i} \vee \varOmega_{z_i}, \boldsymbol{\upsilon}_{x_i z_i} \rangle\rangle = \mathbb{Z} \times \mathbb{Z}$, since any update is possible on x_i by choosing the appropriate value for z_i. More interesting is the projection based upon a particular state of the main schema. Let $M_{22} \in \mathsf{LDB}(\mathbf{E}_2)$ be any legal state with $x_i = 300$, $y_i = 300$, and $z_i = 100$. Then $\mathsf{Proj}_{\langle \varOmega_{x_i} \mid M_{22} \rangle}\langle\langle \varOmega_{x_i} \vee \varOmega_{z_i}, \boldsymbol{\upsilon}_{x_i z_i} \rangle\rangle = \{\langle n, n-100 \rangle \mid n \in \mathbb{Z}\}$, which is exactly $\boldsymbol{\upsilon}_{x_i}$. The difference is in how they are realized. With $\langle \varOmega_{x_i}, \boldsymbol{\upsilon}_{x_i} \rangle$, the parameter 100 is fixed in the update object itself, while in $\mathsf{Proj}_{\langle \varOmega_{x_i} \mid M_{22} \rangle}\langle\langle \varOmega_{x_i} \vee \varOmega_{z_i}, \boldsymbol{\upsilon}_{x_i z_i} \rangle\rangle$, the parameter z_i is bound to 100 after reading the value of z_i from the state M_{22}. This distinction will prove to be critical, since the parameter z_i is used only internally, by the transaction, to determine which update it is to apply. From the point of view of constraint satisfaction, it does not matter how that parameter was obtained; only the update itself matters.

A similar construction applies for the update $y_i \leftarrow y_i - z_i$ on $\varOmega_{y_i} \vee \varOmega_{z_i}$.

Definition 4.4 (Lifting for an updateable data object). A central feature of updates performed by transactions is that they are localized. If u is an update to be performed on data object $\varGamma$, then the changes are made only to the state of the data object $\varGamma$; the states of atomic data objects not included in $\varGamma$ remain fixed. The extension of a set $\mathbf{u}$ of updates to a larger environment, with the environment which is not part of $\varGamma$ held constant, is called a *lifting*. Formally, let $\langle \varGamma, \mathbf{u} \rangle \in \mathsf{FUpdObj}(\boldsymbol{\mathcal{V}})$, and let $\varGamma' \in \mathcal{V}$ with $\varGamma \wedge \varGamma' = \bot$. Define the *lifting* of $\langle \varGamma, \mathbf{u} \rangle$ to $\varGamma \vee \varGamma'$ (*with constant* $\varGamma'$) as

$$\begin{aligned} \mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u} \rangle\rangle = \{(M, u^{(2)} \vee \lambda\langle \varGamma \vee \varGamma', \varGamma' \rangle(M)) \mid \\ (M \in \mathsf{LDB}(\varGamma \vee \varGamma')) \wedge (u \in \mathbf{u}) \wedge \lambda\langle \varGamma \vee \varGamma', \varGamma \rangle(M) = u^{(1)}\} \end{aligned}$$

In parsing this definition, recall the notation shorthand for joining states of disjoint views introduced near the end of Definition 2.3.

Thus, $\mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$ is a relation on $\mathsf{LDB}(\varGamma \vee \varGamma') \times \mathsf{DB}(\varGamma \vee \varGamma')$. Since $\langle \varGamma, \mathbf{u}\rangle$ is functional, $\mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$ will be a function in the sense that for any $P \in \mathsf{LDB}(\varGamma \vee \varGamma')$, there is at most one $P' \in \mathsf{DB}(\varGamma \vee \varGamma')$ with $(P, P') \in \mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$. If such a P exists, it will be denoted $\mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle(P)$. In other words, $\mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$ may be regard as a partial function on $\mathsf{LDB}(\varGamma \vee \varGamma')$.

Define $\mathsf{Compat}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$ to be the set of all legal states of $\varGamma \vee \varGamma'$ which are sent to legal states when lifted to $\varGamma \vee \varGamma'$ using $\langle \varGamma, \mathbf{u}\rangle$. Formally, $\mathsf{Compat}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle = \{M \in \mathsf{LDB}(\varGamma \vee \varGamma') \mid \mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle(M){\downarrow} \in \mathsf{LDB}(\varGamma \vee \varGamma')\}$. This lifting is said to be *legal* (or *allowed*) for $P \in \mathsf{LDB}(\varGamma \vee \varGamma')$ if $P \in \mathsf{Compat}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$; otherwise it is *illegal* (or *disallowed*).

A special case occurs when $\varGamma' = \overline{\varGamma}$; i.e., for liftings to the entire main schema. Recalling that $\mathbf{1_D}$ is the identity view on $\mathbf{D}$, define $\mathsf{Lift}_{\mathbf{D}}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$ to be $\mathsf{Lift}_{\mathbf{1_D}}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$, and define $\mathsf{Compat}_{\mathbf{D}}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$ to be $\mathsf{Compat}_{\mathbf{1_D}}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$.

Liftings to $\mathbf{D}$ will be used to model the internal operation of database transactions, as described in Definition 4.6 below. The transaction must do something when the lifting is undefined or disallowed. The most useful solution is to have it perform the identity update; that is, to do nothing. Formally, for $\langle \varGamma, \mathbf{u}\rangle \in \mathsf{UpdObj}(\boldsymbol{\mathcal{V}})$ and $M \in \mathsf{LDB}(\mathbf{D})$, define

$$\begin{aligned}\mathsf{Lift}^{+}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle = {} & (\mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle \cap \mathsf{LDB}(\varGamma \vee \varGamma') \times \mathsf{LDB}(\varGamma \vee \varGamma')) \\ & \cup \{(M, M) \mid \mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle(M){\downarrow} \notin \mathsf{LDB}(\varGamma \vee \varGamma')\} \\ & \cup \{(M, M) \mid \mathsf{Lift}_{\varGamma \vee \varGamma'}\langle\langle \varGamma, \mathbf{u}\rangle\rangle(M){\uparrow}\}\end{aligned}$$

and define $\mathsf{Lift}^{+}_{\mathbf{D}}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$ to be $\mathsf{Lift}^{+}_{\mathbf{1_D}}\langle\langle \varGamma, \mathbf{u}\rangle\rangle$.

Examples 4.5 (Lifting). As a simple example of a lifting, return to the context of $\mathbf{E}_0$, as presented in Sec. 1, Example 2.4, and Examples 4.3. Let $i \in [0, n-1]$, let $i' = (i+1) \bmod n$, and let $J \subseteq [0, n-1]$ with $\varOmega'$ denoting $\bigvee_{j \in J} \varOmega_{d_j}$ and $\varOmega''$ denoting $\varOmega_{d_i} \vee \varOmega_{d_{i'}}$. Represent an $N \in \mathsf{DB}(\varOmega'' \vee \varOmega')$ as a tuple indexed by $\{i, i'\} \cup J$ in which the element indexed by j is the value of d_j. Then $\mathsf{Lift}_{\varOmega'' \vee \varOmega'}\langle\langle \varOmega'', \boldsymbol{\upsilon}_{d_i}\rangle\rangle$ consists of those pairs of $(\{i, i'\} \cup J)$-indexed tuples (N, N') for which $\pi_i(N') = \pi_{i'}(N)$, and which agree on all indices other than i. In other words, $\boldsymbol{\upsilon}_{d_i}$ is extended to component views of the form $\varOmega_{d_j}$ for $j \in J \setminus \{i, i'\}$ as the identity update.

To illustrate the interaction of lifting and constraints, in the context $\mathbf{E}_2$ of Examples 4.3, $\mathsf{Lift}_{\varOmega_{x_i} \vee \varOmega_{y_i}}\langle\langle \varOmega_{x_i}, \boldsymbol{\upsilon}_{x_i}\rangle\rangle = \{\langle (n_1, n_2), (n_1 - 100, n_2)\rangle \mid (n_1, n_2) \in \mathbb{Z} \times \mathbb{Z}\}$, with the tuple (n_1, n_2) representing the values (x_i, y_i). The lifting is allowed if $n_1 + n_2 - 100 \geq 500$, and disallowed otherwise.

Definition 4.6 (Black-box transactions and snapshot isolation). In a black-box model, the internal operations are hidden. Rather, just the interaction with the environment is modelled. A particularly simple version of a black-box model is appropriate for a theoretical study of snapshot isolation (SI). Indeed, under FCW, as described in Summary 3.1, the internal sequence of read and write operations of which a transaction is composed is not of interest. Rather, it is only the writes which are to be committed which are of relevance for modelling violations of integrity constraints. Thus, it is appropriate to regard a transaction under SI as a single update on an input database, taking that database as input at the beginning of the transaction and delivering an updated version at its end, a simplification which retains all necessary features for modelling conflicts.

More precisely, a *black-box transaction* T over $\mathcal{V}$ is represented by a pair $\langle \mathit{\Gamma}_T, \mathbf{u}_T\rangle \in \mathsf{FUpdObj}(\boldsymbol{\mathcal{V}})$. For an input state $M \in \mathsf{LDB}(\mathbf{D})$, the output state is $\mathsf{Lift}^{+}_{\mathbf{D}}\langle\langle \mathit{\Gamma}_T, \mathbf{u}_T\rangle\rangle(M)$. This defines a total operation on $\mathsf{LDB}(\mathbf{D})$; for any input state M, $\mathsf{Lift}^{+}_{\mathbf{D}}\langle\langle \mathit{\Gamma}_T, \mathbf{u}_T\rangle\rangle(M) \in \mathsf{LDB}(\mathbf{D})$ as well. The set of all black-box transactions over $\boldsymbol{\mathcal{V}}$ is denoted $\mathsf{BBTrans}_{\boldsymbol{\mathcal{V}}}$.

The notation $\langle \mathit{\Gamma}_T, u_T\rangle$ will be used throughout the rest of this paper to denote the update 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 $\langle \mathit{\Gamma}_{T'_i}, \mathbf{u}_{T'_i}\rangle$. On the other hand, update objects not associated with a transaction will never use subscripts involving T or τ.

Definition 4.7 (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 4.4, 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 $\mathsf{BBTrans}_{\boldsymbol{\mathcal{V}}}$. 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 partial order $<_{\mathbf{T}}$ on $\mathsf{SCSet}\langle T\rangle$ with the property that for each $T \in \mathbf{T}$, $T^s < T^c$. 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 $T \in \mathbf{T}$, $\mathsf{CSPred}_{<_{\mathbf{T}}}\langle T\rangle$ denotes the last transaction to commit before T starts, when it exists. Thus, $(\mathsf{CSPred}_{<_{\mathbf{T}}}\langle T\rangle)^c <_{\mathbf{T}} T^s$ and for no $T' \in \mathbf{T}$ is it the case that $(\mathsf{CSPred}_{<_{\mathbf{T}}}\langle T\rangle)^c <_{\mathbf{T}} T'^c <_{\mathbf{T}} T^c$.

Similarly, $\mathsf{CCPred}_{<_{\mathbf{T}}}\langle T\rangle$ denotes the last $T' \in \mathbf{T}$ which commits before T does, when it exists. Note that both $\mathsf{CSPred}_{<_{\mathbf{T}}}\langle -\rangle$ and $\mathsf{CCPred}_{<_{\mathbf{T}}}\langle -\rangle$ 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 and $\{T_1, T_2\}$ is said to form a *concurrent pair.*

A subset $\mathbf{S} \subseteq T$ is said to be *nonoverlapping* if for any $T_1, T_2 \in \mathbf{S}$, $\mathit{\Gamma}_1^{(w)} \wedge \mathit{\Gamma}_2^{(w)} = \bot$. In other words, the write views do not overlap. The schedule $<_{\mathbf{T}}$ is *nonoverlapping* if every concurrent pair $\{T_1, T_2\}$ is nonoverlapping. In the rest of this paper, schedules will always be taken to be nonoverlapping.

Definition 4.8 (Semantics of SI-schedules). In order to be able to model the interaction of transactions and 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 4.6.

Let $\mathbf{T}$ be a finite subset of $\mathsf{BBTrans}_{\boldsymbol{\mathcal{V}}}$ 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 initial state $M \in \mathsf{LDB}(\mathbf{D})$ for the entire schedule:

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

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

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

For each initial state $M \in \mathsf{LDB}(\mathbf{D})$, The semantics are defined in a formal way as follows:

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

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

$$\mathsf{AfterCmt}_{\langle <_{\mathbf{T}} : M \rangle}\langle T \rangle = \\ \mathsf{Lift}_{\mathbf{D}}\langle \mathsf{Proj}_{\langle \mathit{\Gamma}^{(w)} | \mathsf{InitSnap}_{\langle <_{\mathbf{T}} : M \rangle}\langle T \rangle \rangle}\langle \langle \mathit{\Gamma}, \mathbf{u} \rangle \rangle \rangle (\mathsf{BeforeCmt}_{\langle <_{\mathbf{T}} : M \rangle}\langle T \rangle)$$

Less formally, for an initial state M, $\mathsf{InitSnap}_{\langle <_{\mathbf{T}} : M \rangle}\langle T \rangle$ is the state of the global 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. $\mathsf{BeforeCmt}_{\langle <_{\mathbf{T}} : M \rangle}\langle T \rangle$ is the state of the global database just after the last commit operation which occurs before the commit operation of T. Finally, $\mathsf{AfterCmt}_{\langle <_{\mathbf{T}} : M \rangle}\langle T \rangle$ is the result of lifting, to $\mathsf{BeforeCmt}_{\langle <_{\mathbf{T}} : M \rangle}\langle T \rangle$, the projection of the update operation of T onto its write view.

It is important to note that it is only the update to the write view $\mathit{\Gamma}_T^{(w)}$, and not the entire update $\mathbf{u}_T$ to $\mathit{\Gamma}_T$, which is lifted upon commit. This is critical because the read view $\mathit{\Gamma}_T^{(r)}$ may have been updated by another concurrent transaction. For example, in the context of Examples 4.3, let $\tau_{x_i z_i}$ be the transaction whose update object is $\langle \mathit{\Omega}_{x_1} \vee \mathit{\Omega}_{z_i}, \boldsymbol{\upsilon}_{x_1 z_i} \rangle$. It is quite possible that another, concurrent transaction could write z_i after $\tau_{x_i z_i}$ begins but before it commits. In that case, lifting the entire update $\boldsymbol{\upsilon}_{x_iz_i}$ would not produce the correct result, since the transaction $\tau_{x_iz_i}$ does not change the value z_i, and should not restore its value to that when the transaction began. The correct approach, as defined above, is to lift only the projection of $\boldsymbol{\upsilon}_{x_iz_i}$ onto its write view, for the database state which $\tau_{x_iz_i}$ acquired for its snapshot. This uses the value of z_i at the beginning of $\tau_{x_iz_i}$, as required, to compute the changes on the write view defined by the update, and then commits only those changes. The value of z_i should not be written as part of the final update of $\tau_{x_iz_i}$.

Returning to the general context, call $<_{\mathbf{T}}$ *constraint preserving* if for every $M \in \mathsf{LDB}(\mathbf{D})$ and every $T \in \mathbf{T}$, $\mathsf{AfterCmt}_{\langle <_{\mathbf{T}}:M\rangle}\langle T\rangle \in \mathsf{LDB}(\mathbf{D})$. The goal is to show how to ensure that $<_{\mathbf{T}}$ has this property.

Definition 4.9 (Write-commuting pairs). The key abstract property to be used in guaranteeing constraint-preserving schedules is write commutativity, Roughly, two transactions T_1 and T_2 form a write-commuting pair if whenever they each may be executed concurrently on a given database state, then they may be executed serially as well, in either order. However, it is only the updates on their respective write views, and not the entire update of each transactions, which must obey this commutativity constraint. Thus, each transaction may overwrite the read-only view of the other with the two still forming a write-commuting pair.

Formally, call $\{T_1, T_2\} \subseteq \mathsf{BBTrans}_{\boldsymbol{\mathcal{V}}}$ a *write-commuting pair* if it is nonoverlapping and for any $M \in \mathsf{LDB}(\mathbf{D})$ and any $u_1 \in \mathsf{Proj}_{\langle \Gamma_1^{(w)}|M\rangle}\langle\langle \Gamma_1, \mathbf{u}_1\rangle\rangle$ and $u_2 \in \mathsf{Proj}_{\langle \Gamma_2^{(w)}|M\rangle}\langle\langle \Gamma_2, \mathbf{u}_2\rangle\rangle$, with both $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_1}^{(w)}, \{u_1\}\rangle\rangle(M){\downarrow} \in \mathsf{LDB}(\mathbf{D})$ and $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_2}^{(w)}, \{u_2\}\rangle\rangle(M){\downarrow} \in \mathsf{LDB}(\mathbf{D})$, it is the case that both $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_1}^{(w)}, \{u_1\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_2}^{(w)}, \{u_2\}\rangle\rangle(M){\downarrow}{\in} \mathsf{LDB}(\mathbf{D})$ and $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_2}^{(w)}, \{u_2\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_1}^{(w)}, \{u_1\}\rangle\rangle(M){\downarrow}{\in} \mathsf{LDB}(\mathbf{D})$.

A large class of write-commuting pairs will be identified in Proposition 4.19 below. For now, the key property to observe, which justifies the name, is that such pairs produce the same result in either order of composition, with the consequence (Theorem 4.11) that SI schedules are constraint preserving.

Observation 4.10 (Write-commuting pairs produce the same result in either order). *If $\{T_1, T_2\}$ is a write-commuting pair, $M \in \mathsf{LDB}(\mathbf{D})$, and $u_1 \in \mathsf{Proj}_{\langle \Gamma_1^{(w)}|M\rangle}\langle\langle \Gamma_1, \mathbf{u}_1\rangle\rangle$, $u_2 \in \mathsf{Proj}_{\langle \Gamma_2^{(w)}|M\rangle}\langle\langle \Gamma_2, \mathbf{u}_2\rangle\rangle$ with both of the liftings $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_1}^{(w)}, \{u_1\}\rangle\rangle(M){\downarrow}{\in} \mathsf{LDB}(\mathbf{D})$ and $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_2}^{(w)}, \{u_2\}\rangle\rangle(M){\downarrow}{\in} \mathsf{LDB}(\mathbf{D})$, then*

$$\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_1}^{(w)}, \{u_1\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_2}^{(w)}, \{u_2\}\rangle\rangle(M) = \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_2}^{(w)}, \{u_2\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma_{T_1}^{(w)}, \{u_1\}\rangle\rangle(M).$$

Proof. This is immediate, since the updates are nonoverlapping. As long as both are defined, they must be the same. □

Theorem 4.11 (Write-commuting concurrent pairs guarantee constraint-preserving SI-schedules). *Let $\mathbf{T}$ be a finite subset of $\mathsf{BBTrans}_{\boldsymbol{\mathcal{V}}}$, and let $<_{\mathbf{T}}$ be an SI-schedule for $\mathbf{T}$. If every concurrent pair of $<_{\mathbf{T}}$ is write commuting, then $<_{\mathbf{T}}$ is constraint preserving.*

Proof. The proof is by induction on the size of $\mathbf{T}$. For zero or one transaction, the result is immediate. For the inductive step, let $n \in \mathbb{N}$ and assume that the result is true whenever $\mathsf{Card}(\mathbf{T}) \leq n$. Then let $\mathsf{Card}(\mathbf{T}) = n+1$ (with $n \geq 1$), and let T_n and T_{n+1} be the n^{th} and $n+1^{\text{st}}$ transactions to commit in $<_{\mathbf{T}}$, respectively (that is, the penultimate and last transactions to commit). If $n \geq 2$, i.e., if $n+1 \geq 3$, let T_{n-1} be the transaction which commits just before T_n. Let $M \in \mathsf{LDB}(\mathbf{D})$ be the initial state for the schedule.

If T_{n+1} starts after T_n has committed; that is, the two transactions are not concurrent, then the result is immediate; there cannot be any constraint violation with serial transactions which operate correctly in isolation. So, assume that $\{T_n, T_{n+1}\}$ forms a concurrent pair. Let u_n and u_{n+1} be the updates which T_n and T_{n+1} perform on $\mathit{\Gamma}_n^{(w)}$ and $\mathit{\Gamma}_{n+1}^{(w)}$, respectively, and let $\mathbf{S}_{n-1} = \mathbf{T} \setminus \{T_n, T_{n+1}\}$, $\mathbf{S}_n = \mathbf{T} \setminus \{T_n\}$, and $\mathbf{S}_{n+1} = \mathbf{T} \setminus \{T_{n+1}\}$, with $<_{\mathbf{S}_{n-1}}$, $<_{\mathbf{S}_n}$, and $<_{\mathbf{S}_{n+1}}$ be the schedules obtained by restricting $<_{\mathbf{T}}$ to the transactions in $\mathbf{S}_{n-1}$, $\mathbf{S}_n$, and $\mathbf{S}_{n+1}$, respectively. Then by the inductive hypothesis, each of $<_{\mathbf{S}_{n-1}}$, $<_{\mathbf{S}_n}$, and $<_{\mathbf{S}_{n+1}}$ is constraint preserving. If at least one of $\mathsf{Lift}^{+}_{\mathbf{D}}\langle\langle \mathit{\Gamma}^{(w)}_{T_n}, \{u_n\}\rangle\rangle$ and $\mathsf{Lift}^{+}_{\mathbf{D}}\langle\langle \mathit{\Gamma}^{(w)}_{T_{n+1}}, \{u_{n+1}\}\rangle\rangle$ is the identity update (for example, if one of the liftings was not defined or did not result in a legal state), then the corresponding transaction may be removed from $<_{\mathbf{T}}$ to obtain one of $<_{\mathbf{S}_n}$ or $<_{\mathbf{S}_{n+1}}$, without any change in the semantics, since an identity transaction has no effect. In that case, the result follows from the inductive hypothesis. So, assume that both $\mathsf{Lift}_{\mathbf{D}}\langle\langle \mathit{\Gamma}^{(w)}_{T_n}, \{u_n\}\rangle\rangle(N)\!\downarrow$ and $\mathsf{Lift}_{\mathbf{D}}\langle\langle \mathit{\Gamma}^{(w)}_{T_{n+1}}, \{u_{n+1}\}\rangle\rangle(N)\!\downarrow$, with $N = \mathsf{AfterCmt}_{\langle <_{\mathbf{T}}:M\rangle}\langle T_{n-1}\rangle$ if $n \geq 2$ and $N = M$ if $n = 1$. Then $(\mathsf{Lift}_{\mathbf{D}}\langle\langle \mathit{\Gamma}^{(w)}_{T_n}, \{u_n\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \mathit{\Gamma}^{(w)}_{T_{n+1}}, \{u_{n+1}\}\rangle\rangle)(N)\!\downarrow \in \mathsf{LDB}(\mathbf{D})$, since by assumption $\{T_1, T_2\}$ forms a write commuting pair. However, it is easy to see that in that case the semantics of re-inserting T_n and T_{n+1} into $<_{\mathbf{S}}$ is just to perform that composed update, which establishes that $\mathsf{AfterCmt}_{\langle <_{\mathbf{T}}:M\rangle}\langle T_{n+1}\rangle \in \mathsf{LDB}(\mathbf{D})$, as required. □

Definition 4.12 (Guard views and guarded black-box transactions). The property of write commutativity is an abstract one. A useful, concrete class of transactions with that property may be obtained via the notion of a guard for an updateable object $\langle \mathit{\Gamma}, \mathbf{u}\rangle$. Such a guard is a view $\mathit{\Gamma}'$ with the property that if, for a given $M \in \mathsf{LDB}(\mathbf{D})$ and any $u \in \mathbf{u}$, the projection $\lambda\langle \mathit{\Gamma}, \mathit{\Gamma}^{(w)}\rangle(u)$ of u onto $\mathit{\Gamma}^{(w)}$ restricted to M may be lifted to $\mathbf{D}$ iff it may be lifted to $\mathit{\Gamma}^{(w)} \vee \mathit{\Gamma}'$. Thus, a guard view reduces the global test for lifting to all of $\mathbf{D}$ to the much more local test of lifting to just the write view and its guard. Formally, given $\langle \mathit{\Gamma}, \mathbf{u}\rangle \in \mathsf{FUpdObj}(\boldsymbol{\mathcal{V}})$, $\mathit{\Gamma}' \in \mathcal{V}$ is a *guard view for* $\langle \mathit{\Gamma}, \mathbf{u}\rangle$ if $\mathit{\Gamma}^{(w)} \wedge \mathit{\Gamma}' = \perp$ and for every $M \in \mathsf{LDB}(\mathbf{D})$, $M \in \mathsf{Compat}_{\mathbf{D}}\langle\langle \mathit{\Gamma}, \mathbf{u}\rangle\rangle$ iff $(\gamma^{(w)} \vee \gamma')(M) \in \mathsf{Compat}_{\mathit{\Gamma}^{(w)} \vee \mathit{\Gamma}'}\langle \mathsf{Proj}_{\langle \mathit{\Gamma}^{(w)} \vee \mathit{\Gamma}' | M\rangle}\langle\langle \mathit{\Gamma}, \mathbf{u}\rangle\rangle\rangle$. The set of all guard views for $\langle \mathit{\Gamma}, \mathbf{u}\rangle$ is denoted $\mathsf{Guards}_{\boldsymbol{\mathcal{V}}}\langle \mathit{\Gamma}, \mathbf{u}\rangle$.

A *guarded black-box transaction* is represented by a pair $\langle\langle \mathit{\Gamma}, \mathbf{u}\rangle, \mathit{\Gamma}'\rangle$ in which $\langle \mathit{\Gamma}, \mathbf{u}\rangle \in \mathsf{FUpdObj}(\mathcal{V})$ and $\mathit{\Gamma}' \in \mathcal{V}$ is a guard for $\langle \mathit{\Gamma}, \mathbf{u}\rangle$. It is convenient to have a notation for guarded black-box transactions which extends that of Definition 4.6. To that end, if T is such a transaction, then its guard will be denoted

$\mathit{\Gamma}^{(g)}_T$. Thus, T is represented by $\langle\langle \mathit{\Gamma}_T, \mathbf{u}_T\rangle, \mathit{\Gamma}^{(g)}_T\rangle$. The set of all guarded black-box transactions over $\mathcal{V}$ is denoted $\mathsf{GBBTrans}_{\boldsymbol{\mathcal{V}}}$.

Examples 4.13 (Guards). Returning to the context $\mathbf{E}_2$ of Examples 4.3, a guard for $\langle \mathit{\Omega}_{x_i}, \boldsymbol{\upsilon}_{x_i}\rangle$ is $\mathit{\Omega}_{y_i}$. Indeed, to verify that an update to x_i is legal, only the value of y_i need be checked; the state of the rest of the database is irrelevant. Similarly, a guard for $\langle \mathit{\Omega}_{y_i}, \boldsymbol{\upsilon}_{y_i}\rangle$ is $\mathit{\Omega}_{x_i}$. It is only the write view of an update, and not its read-only view, which affects the definition of a guard. Thus, a guard of $\langle \mathit{\Omega}_{x_i} \vee \mathit{\Omega}_{z_i}, \boldsymbol{\upsilon}_{x_i z_i}\rangle$ is $\mathit{\Omega}_{y_i}$, and a guard of $\langle \mathit{\Omega}_{y_i} \vee \mathit{\Omega}_{z_i}, \boldsymbol{\upsilon}_{y_i z_i}\rangle$ is $\mathit{\Omega}_{x_i}$.

The guard view need not be disjoint from the read-only view. For example, given the update set $\boldsymbol{\upsilon}_{x_i y_i}$ on $\mathit{\Omega}_{x_i} \vee \mathit{\Omega}_{y_i}$ defined by the update rule $x_i \leftarrow x_i - y_i$, the view $\mathit{\Omega}_{y_i}$ is a guard for $\langle \mathit{\Omega}_{x_i} \vee \mathit{\Omega}_{y_i}, \boldsymbol{\upsilon}_{x_i y_i}\rangle$. However, $\mathit{\Omega}^{(r)}_{x_i y_i} = \mathit{\Omega}_{y_i}$ as well, so the guard and the read-only view are the same in this case.

Observation 4.14 (Guards always exist). *Given $\langle \mathit{\Gamma}, \mathbf{u}\rangle \in \mathsf{UpdObj}(\mathcal{V})$, the complement $\overline{\mathit{\Gamma}^{(w)}}$ of the associated write view is always a guard view for $\mathit{\Gamma}$. Thus, every updateable object has a guard.* □

Definition 4.15 (Minimal and least guards). Let $\langle \mathit{\Gamma}, \mathbf{u}\rangle \in \mathsf{FUpdObj}(\boldsymbol{\mathcal{V}})$. Then $\mathit{\Gamma}' \in \mathsf{Guards}_{\boldsymbol{\mathcal{V}}}\langle \mathit{\Gamma}, \mathbf{u}\rangle$ is is a *minimal guard view* for $\langle \mathit{\Gamma}, \mathbf{u}\rangle$ if for any guard $\mathit{\Gamma}''$ for $\langle \mathit{\Gamma}, \mathbf{u}\rangle$, if $\mathit{\Gamma}'' \preceq_{\mathbf{D}} \mathit{\Gamma}'$, then $\mathit{\Gamma}'' = \mathit{\Gamma}'$. A unique minimal guard view is *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.

Example 4.16 (Least guards need not exist). While a minimal guard may always be chosen, it is not the case that least guards always exist. For example, if the constraint $y_i = z_i$ is added to the schema $\mathbf{E}_2$ of Examples 4.3, then both $\mathit{\Omega}_{y_i}$ and $\mathit{\Omega}_{z_i}$ are guards for $\langle \mathit{\Omega}_{x_i}, \boldsymbol{\upsilon}_{x_i}\rangle$. Of course, these are effectively the same. In general, it can be shown that the choice of a minimal guard does not matter. Roughly, the reason is that if a transaction T_1 writes a guard of another transaction T_2 in way which affects which updates T_2 may perform legally, then it must write all guards of T_2. Otherwise, some guards would allow updates which others would not, which is not consistent with the definition of guard.

Definition 4.17 (Independent and conflicting pairs of guarded transactions). Two transactions are guard independent if at least one does not write the guard of the other. In other words, they do not each have a read-write dependency on the other, with respect to reading the guard view only. More formally, the two element set $\{T_1, T_2\} \subseteq \mathsf{GBBTrans}_{\boldsymbol{\mathcal{V}}}$ forms a *guard-independent* pair if $\{T_1, T_2\}$ is nonoverlapping and at least one of $\mathit{\Gamma}^{(w)}_{T_2} \wedge \mathit{\Gamma}^{(g)}_{T_1} = \bot$ and $\mathit{\Gamma}^{(w)}_{T_1} \wedge \mathit{\Gamma}^{(g)}_{T_2} = \bot$ holds. A pair $\{T_1, T_2\}$ which is not guard independent is *guard-conflicting*, and T_1 and T_2 are then said to be *in guard conflict* with each other.

Examples 4.18 (Independent and conflicting pairs). Continuing with Examples 4.13, $\{\tau_{x_i}, \tau_{y_i}\}$ form a guard-conflicting pair, since each reads the guard of the other. On the other hand, for distinct i and j, τ_{x_i} and τ_{x_j} are guard independent, since neither writes the guard of the other. To illustrate the interesting middle ground, define $\tau_{x'_i}$ to be the transaction on Ω_{x_i} which implements the update rule $x_i \leftarrow x_i + 50$. Then $\{\tau_{x'_i}, \tau_{y_i}\}$ forms a guard independent pair, since the least guard of $\tau_{x'_i}$ is $\perp$; i.e., its update is always legal. As will be shown next, guard independence implies write commutativity, and this example illustrates the intuition behind this. Even though $\tau_{x'_i}$ writes the guard of τ_{y_i}, which is Ω_{x_i} itself, it does so in a "harmless" way. Because $\tau_{x'_i}$ does not read y_i, it cannot make any updates whose legality depends upon the value of y_i.

Proposition 4.19 (Guard independence $\Rightarrow$ write commutativity). *Every $\{T_1, T_2\} \in \mathsf{GBBTrans}_{\boldsymbol{\mathcal{V}}}$ which is guard independent forms a write-commuting pair.*

Proof. Let $\{T_1, T_2\} \subseteq \mathsf{GBBTrans}_{\boldsymbol{\mathcal{V}}}$. Without loss of generality, assume that $\Gamma^{(w)}_{T_2} \wedge \Gamma^{(g)}_{T_1} = \perp$. Let $M \in \mathsf{LDB}(\mathbf{D})$ and let any $u_1 \in \mathsf{Proj}_{\langle \Gamma_1^{(w)} | M\rangle}\langle\langle \Gamma_1, \mathbf{u}_1\rangle\rangle$, $u_2 \in \mathsf{Proj}_{\langle \Gamma_2^{(w)} | M\rangle}\langle\langle \Gamma_2, \mathbf{u}_2\rangle\rangle$, with both $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_1}, \{u_1\}\rangle\rangle(M)\downarrow \in \mathsf{LDB}(\mathbf{D})$ and $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_2}, \{u_2\}\rangle\rangle(M)\downarrow \in \mathsf{LDB}(\mathbf{D})$. It is immediate that $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_2}, \{u_2\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_1}, \{u_1\}\rangle\rangle(M)\downarrow\in \mathsf{LDB}(\mathbf{D})$, 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 legality of the update which T_1 performs.

For the opposite direction, first note that it is the case that $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_1}, \{u_1\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_2}, \{u_2\}\rangle\rangle(M)\downarrow\in \mathsf{DB}(\mathbf{D})$ since the write views are nonoverlapping; i.e., $\Gamma_1^{(w)} \wedge \Gamma_2^{(w)} = \perp$. The only question is whether the result is in $\mathsf{LDB}(\mathbf{D})$. However, again since $\Gamma_1^{(w)} \wedge \Gamma_2^{(w)} = \perp$, the two compositions must be identical; i.e., $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_1}, \{u_1\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_2}, \{u_2\}\rangle\rangle(M) = \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_2}, \{u_2\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_1}, \{u_1\}\rangle\rangle(M)$. This shows that the composition $\mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_1}, \{u_1\}\rangle\rangle \circ \mathsf{Lift}_{\mathbf{D}}\langle\langle \Gamma^{(w)}_{T_2}, \{u_2\}\rangle\rangle(M) \in \mathsf{LDB}(\mathbf{D})$, as required. Hence $\{T_1, T_2\}$ forms a write commuting pair. □

The main theorem of this paper may now be established.

Theorem 4.20 (Guard independence guarantees constraint preservation). *Let $\mathbf{T}$ be a finite subset of $\mathsf{GBBTrans}_{\boldsymbol{\mathcal{V}}}$, and let $<_{\mathbf{T}}$ be an SI-schedule for $\mathbf{T}$. If every concurrent pair of $<_{\mathbf{T}}$ is guard independent, then $<_{\mathbf{T}}$ is constraint preserving.*

Proof. The proof follows immediately from Theorem 4.11 and Proposition 4.19. □

Discussion 4.21 (Constraint-Preserving Snapshot Isolation (CPSI)). In an SI-schedule $<_{\mathbf{T}}$, there is an *rw-dependency* from T_1 to T_2, written $T_1 \xrightarrow{\mathsf{rw}} T_2$, if T_2 writes the read set of T_1. Within the context of the formalism of this paper, this translates to $\Gamma^{(w)}_{T_1} \wedge (\Gamma^{(r)}_{T_2} \vee \Gamma^{(g)}_{T_2}) \neq \perp$, since to operate correctly, a transaction T must read both its update view Γ_T (in order to know which update to execute) and its guard view $\Gamma^{(g)}_T$ (in order to determine whether that update is legal). In the implementation of SSI, as described in [5], the critical notion is the *dangerous structure*, which consists of two consecutive read-write dependencies of concurrent pairs in the conflict graph; 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. The absence of such a pair of dependencies is sufficient, but not necessary, for an SI-schedule to be serializable. Necessity requires that it be part of a cycle in the conflict graph. Working with this same model, the results of this paper show that for such a dangerous structure to lead to a constraint violation, it must be the case that $T_1 = T_3$. Thus, a much simpler test suffices if only constraint violation is to be flagged.

An approach with fewer false positives may be obtained by working only with guard reads. More precisely, say that there is a *gw-dependency* from T_1 to T_2 if T_2 writes the guard of T_1; i.e., $\mathit{\Gamma}_2^{(w)} \wedge \mathit{\Gamma}_{T_1}^{(g)} \neq \bot$, and write $T_1 \xrightarrow{\mathsf{gw}} T_2$ to denote this. Call $\{T_1, T_2\}$ a *dangerous gw-pair* 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. Theorem 4.20 guarantees that an SI-schedule will be constraint preserving in the absence of such pairs. This strategy, called *constraint-preserving snapshot isolation (CPSI)*, has false positives only to the extent that two transactions could each write the guard of the other without causing a constraint violation. This is of course possible, but the general assumption in transaction management is that the manager only knows which objects are read and written, not how they are written or how the writes are used. With that understanding, there would be no false positives, since there is always some update to the guard which would cause a constraint violation.

A correct implementation would of course require that the system be able to identify which reads of a transaction are to the guard. This could be done, for example, in a context of fixed transactions for business processes by having such guards known to the transaction manager.

Example 4.22 (Dangerous structures which do not result in constraint violations). To illustrate the ways that the multiversion conflict graph may contain dangerous structures yet be free of dangerous gw-pairs, return to the context of $\mathbf{E}_2$, as described in Examples 4.3, and consider three concurrent transactions: τ_1 operates on $\mathit{\Omega}_{x_1} \vee \mathit{\Omega}_{x_2}$ via the rule $x_1 \leftarrow x_1 - x_2$, τ_2 operates on $\mathit{\Omega}_{x_1} \vee \mathit{\Omega}_{x_2}$ via the rule $x_2 \leftarrow x_2 - x_1$, and τ_3 operates on $\mathit{\Omega}_{x_2} \vee \mathit{\Omega}_{y_1}$ via the rule $y_1 \leftarrow y_1 + |x_2|$. It is assumed that each transaction performs the given update if it would not result in a constraint violation (when run in isolation), and performs the identity update otherwise. The multiversion conflict graph for these three transactions is shown in Fig. 3. Note in particular that although τ_3 reads x_2, it uses only its absolute value in the computation of the new value for y_1, and so $\mathit{\Omega}_{x_2}$ is not in its guard.

Observe that this graph contains two cycles. The first, between τ_1 and τ_2, involves only rw-edges. The second, between τ_2 and τ_3, involves one gw-edge and one rw-edge. Although both of these cycles define dangerous structures, as do the sequences $\tau_1 \xrightarrow{\mathsf{rw}} \tau_2 \xrightarrow{\mathsf{gw}} \tau_3$ and $\tau_3 \xrightarrow{\mathsf{rw}} \tau_2 \xrightarrow{\mathsf{rw}} \tau_1$, none represents a dangerous gw-pair. Since a constraint violation can occur only if there is a (two-vertex) cycle consisting of gw-edges, no constraint violation is possible when running under SI, provided that each transaction individually respects all constraints. even though the result need not be serializable. Thus, while SSI and even PSSI would force at least one of these transactions to terminate, with CPSI all may run to completion.

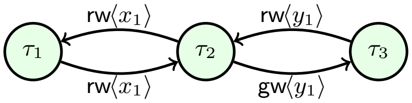

Fig. 3. An SI conflict graph with dangerous structures but no dangerous gw-pairs

5 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, and may be tested fully. without concern for false positives. 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.

Strategies for revising transactions: The motivation for this work arose from earlier studies on cooperative updates [13, 11]. The focus there is particularly upon interactive, long-running business processes in which abort and restart for transactions is not an 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 is 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 work on independence and overlap: In [10], 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.

References

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

2. H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 1–10, 1995.
3. P. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann, second edition, 2009.
4. Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Trans. Software Eng.*, 17(9):954–960, 1991.
5. M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), 2009.
6. C. J. Date. *A Guide to the SQL Standard*. Addison-Wesley, 1997. (with Hugh Darwen).
7. B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order.* Cambridge University Press, second edition, 2002.
8. A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.
9. S. J. Hegner. An order-based theory of updates for closed database views. *Ann. Math. Art. Intell.*, 40:63–125, 2004.
10. S. J. Hegner. A model of independence and overlap for transactions on database schemata. In B. Catania, M. Ivanović, and B. Thalheim, editors, *Advances in Databases and Information Systems, 14th East European Conference, ADBIS 2010, Novi Sad, Serbia, September 20-24, 2010, Proceedings*, volume 6295 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 2010.
11. S. J. Hegner. A simple model of negotiation for cooperative updates on database schema components. In Y. Kiyoki, T. Tokuda, A. Heimbürger, H. Jaakkola, and N. Yoshida., editors, *Frontiers in Artificial Intelligence and Applications XX11*, pages 154–173. IOS Press, 2011.
12. S. J. Hegner. Invariance properties of the constant-complement view-update strategy. In K.-D. Schewe and B. Thalheim, editors, *Semantics in Data and Knowledge Bases, 5th International Workshop, SDKB 2011, Zürich, Switzerland, July 3, 2011, Revised Selected Papers*, volume 7693 of *Lecture Notes in Computer Science*, pages 118–148. Springer-Verlag, 2013.
13. S. J. Hegner and P. Schmidt. Update support for database views via cooperation. In Y. Ioannis, B. Novikov, and B. Rachev, editors, *Advances in Databases and Information Systems, 11th East European Conference, ADBIS 2007, Varna, Bulgaria, September 29 - October 3, 2007, Proceedings*, volume 4690 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 2007.
14. C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.
15. D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endowment*, 5(12):1850–1861, 2012.
16. S. Revilak, P. E. O'Neil, and E. J. O'Neil. Precisely serializable snapshot isolation (PSSI). In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 482–493, 2011.
17. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw Hill, sixth edition, 2011.
18. G. Weikum and G. Vossen. *Transactional Information Systems.* Morgan Kaufmann, 2002.