

Optimal database locks for efficient integrity checking

Davide Martinenghi

Roskilde University, Computer Science Dept.
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: dm@ruc.dk

Abstract. In concurrent database systems, correctness of update transactions refers to the equivalent effects of the execution schedule and some serial schedule over the same set of transactions. Integrity constraints add further semantic requirements to the correctness of the database states reached upon the execution of update transactions. Several methods for efficient integrity checking and enforcing exist. We show in this paper how to apply one such method to automatically extend update transactions with locks and simplified consistency tests on the locked entities. All schedules produced in this way are conflict serializable and preserve consistency. For certain classes of databases we also guarantee that the amount of locked database entities is minimal.

1 Introduction

When an update transaction is executed on a database, it is important to ensure that the consistency of the database is preserved and that the transaction produces the desired result, i.e., its execution is not affected by the execution of other, possibly interleaved transactions. A common view in concurrent database systems is that a transaction executes correctly if it belongs to a schedule that is conflict serializable, i.e., equivalent to a schedule in which all the transactions are executed in series (not interleaved). Several strategies and protocols, such as two-phase locking and timestamp ordering, have been established that can dynamically enforce conflict serializability. Locking is the most common practice for concurrency control; however, maintaining locks is expensive and may limit the throughput of the database, as locks actually reduce the concurrent accesses to the resources.

Another aspect of correctness is determined by the semantic requirements expressed by integrity constraints. Integrity constraints are logical formulas that characterize the consistent states of a database. Integrity checking and maintenance are central issues, as without any guarantee of data consistency, the answers to queries become unreliable. Consistency checks often require polynomial time complexity with respect to the size of the database, which is usually too costly. We therefore need to simplify the integrity constraints into specialized checks that can be executed more efficiently at each update, employing the hypothesis that the initial database state was consistent. To optimize the run-time

performance, these tests should be generated at database design time and executed before potentially offensive updates, in order to avoid rollback operations completely.

The principle of simplification of integrity constraints has been long known and recognized [21], but the common practice in database applications is still based on *ad hoc* techniques. The two main approaches are triggers, at the database level, and hand-coding of tests, at the application level. By their procedural nature, both methods have major disadvantages, as they are prone to errors, require advanced programming skills and have little flexibility with respect to changes in the schema of the database. This motivates the need for automated simplification methods. Although a few, standard principles exist [3], none of them has prevailed in current database implementations. We briefly review in this paper a simplification procedure [6] based on syntactic transformations that produce simplified integrity constraints at design time; these are necessary and sufficient conditions for consistency that can be tested prior to the execution of updates. The procedure can be extended to integrity constraints containing aggregates and arithmetic built-ins [20] and can handle very general rule-based updates [1] that include additions, deletions and changes. For simplicity, we do not consider aggregates and arithmetic and we restrict our attention to tuple additions and deletions.

In this paper we focus on the interaction between integrity checking and locking policies. We present a method that uses the simplified integrity constraints not only to ensure consistency of each individual update transaction, but also to determine the database resources that need to be locked in order to guarantee the correctness of all legal schedules. Furthermore, this method minimizes the amount of database resources that need to be locked in order to have correctness, thus improving the global performance. For certain classes of databases, minimality can indeed be proved; in all other cases we can only achieve good approximations thereof. To the best of our knowledge, a similar automatic generation of locking conditions for a given set of transactions and a given set of integrity constraints has not been attempted before.

The paper is organized as follows. We review existing literature in the field in section 2. The simplification procedure is presented in section 3, while the results concerning its application to database locking are explained and exemplified in section 4. Concluding remarks are provided in section 5. Proofs are omitted or only sketched for reasons of space.

2 Related works

As mentioned, the principle of simplification of integrity constraints is essential for optimizations in database consistency checking. A central quality of any good approach to integrity checking is the ability to verify the consistency of a database to be updated *before* the execution of the transaction in question, so that inconsistent states are completely avoided. Several approaches to simplification do not comply with this requirement, e.g., [21, 17, 9, 12]. Other methods,

e.g., [14], provide pre-tests that, however, are not proven to be necessary conditions; in other words, if the tests fail, nothing can be concluded about the consistency. In [23] a powerful simplification procedure is presented that, however, does not allow more than one update action in a transaction to operate on the same relation. For these reasons, none of the above methods lends itself well to the context of concurrent database transactions. Most works in the field lack a characterization of what it means for a formula to be simplified. Our view is that a transformed integrity constraint, in order to qualify as “simplified”, must represent a minimum in some ordering that reflects the effort of actually evaluating it. We propose a simple ordering that minimizes the amount of database resources involved in the check, but, by the nature of the problem, no algorithm can reach a minimum in all cases. Most simplification methods do not allow recursion. The rare exceptions we are aware of (e.g., [17, 16, 4]) hardly provide useful results: when recursive rules are present, these methods typically output the same constraints as in the input set.

As for locking, the literature is rich in methods developed in the last three decades that guarantee safeness, i.e., the preservation of consistency. The most common protocol is two-phase locking [10], but others have been proposed [25], for example to better capture the requirements of long-lived transactions [24]. All these methods are based on the implicit assumption that a single transaction preserves consistency when executed alone; however, in general this is not the case, and in the following we shall remove this assumption and introduce consistency tests for each transaction. We refer in this paper to the schedule correctness principle of conflict serializability, but algorithms and methods exist also for the broader notion of view serializability (see [5] and the references therein). These criteria have been further extended to describe higher degrees of transaction parallelism, such as the update-in-place and the deferred-update semantics [18]. Once locks are introduced, a concurrent database system may incur in deadlocks, unless special prevention techniques are in place; see [11] for an overview.

3 A simplification procedure for integrity constraints

3.1 Preliminaries

We describe our proposal in a function-free first-order language, such as DATALOG with default negation (DATALOG[−]). We assume familiarity with the notions of *terms* (t, s, \dots), *variables* (x, y, \dots), *constants* (a, b, \dots), *predicates* (p, q, \dots), *atoms*, *literals*, *formulas* and (definite) *clauses*. We characterize a database as a set of non recursive clauses that we divide in three classes: the *extensional database* or set of *facts* (ground bodiless clauses), the *constraint theory* or set of *denials* (headless clauses), and the *intensional database* or set of *rules* (all other clauses) [13]. We disregard from now on the intensional database, as, without recursion, the constraint theory can be transformed in an equivalent one that does not contain any intensional predicates [3]. Therefore, by *database state* we

refer to the extensional part only. The truth value of a closed formula F , relative to a database state D , is defined as its evaluation in D 's standard model [22] and denoted $D(F)$. Logical equivalence between formulas is denoted by \equiv , entailment by \models , provability by \vdash . The notation \vec{t} indicates a sequence of terms t_1, \dots, t_n and the expressions $p(\vec{t})$, $\vec{s} \doteq \vec{t}$ and $\vec{s} \neq \vec{t}$ are defined accordingly.

Definition 1 (Consistency). *A database state D is consistent with a constraint theory Γ iff $D(\Gamma) = \text{true}$.*

We assume that all clauses are range restricted, as defined below.

Definition 2 (Range restriction). *A variable in a clause is range bound if it appears in a positive database literal in the body. A clause is range restricted if all variables in it are range bound.*

Subsumption is a syntactic principle, used in the simplification procedure, that has the semantic property that the subsuming formula entails the subsumed one. We give here a definition for denials.

Definition 3 (Subsumption). *Given two denials D_1, D_2 , D_1 subsumes D_2 iff there is a substitution σ such that each literal in $D_1\sigma$ occurs in D_2 .*

Example 1. The denial $\leftarrow p(x, y) \wedge q(y)$ subsumes $\leftarrow p(x, b) \wedge x \neq a \wedge q(b)$. \square

As mentioned, the simplification method we describe here can handle general forms of update, but, for ease of exposition, we limit our attention to sets of additions and deletions.

Definition 4 (Update). *An update $U = U^+ \cup U^-$ is a non-empty set of additions U^+ and deletions U^- . An addition is a ground atom and a deletion a negated ground atom. The reverse of an update U , denoted $\neg U$, contains the same elements as U but with the roles of additions and deletions interchanged. The additions and deletions of an update are required to be disjoint, i.e. $U^+ \cap \neg U^- = \emptyset$. The notation D^U , where D is a database state, is a shorthand for $(D \cup U^+) \setminus \neg U^-$.*

We also allow in updates special constants called *parameters* (written in boldface: $\mathbf{a}, \mathbf{b}, \dots$), i.e., placeholders for constants. In this way we can generalize updates into update *patterns* and simplify integrity constraints with respect to classes of updates, rather than specific updates. For example, the notation $\{p(\mathbf{a}), \neg q(\mathbf{a})\}$, where \mathbf{a} is a parameter, indicates the class of updates that add a tuple to the unary relation p and remove the same tuple from the unary relation q . We refer to [6] for precise definitions concerning parameters.

3.2 Semantic notions

We characterize the semantic correctness of simplification with the notion of weakest precondition [8, 15], i.e., a test that can be checked in the present state but indicating properties of the new state.

Definition 5 (Weakest precondition). Let Γ and Σ be constraint theories and U an update. Σ is a weakest precondition of Γ with respect to U whenever $D(\Sigma) = D^U(\Gamma)$ for any database state D .

The hypothesis that the constraint theory holds in the present state can be exploited to further optimize a weakest precondition.

Definition 6 (Conditional weakest precondition). Let Γ, Δ be constraint theories and U an update. A constraint theory Σ is a Δ -conditional weakest precondition (Δ -CWP) of Γ with respect to U whenever $D(\Sigma) = D^U(\Gamma)$ for any database state D consistent with Δ .

In definition 6, Δ will typically include Γ and perhaps further properties of the database that are trusted. All CWPs of the same constraint theory with respect to the same update are in an equivalence class called conditional equivalence.

Definition 7 (Conditional equivalence). Let $\Delta, \Gamma_1, \Gamma_2$ be constraint theories; then Γ_1 and Γ_2 are conditionally equivalent with respect to Δ , denoted $\Gamma_1 \stackrel{\Delta}{\equiv} \Gamma_2$, whenever $D(\Gamma_1) = D(\Gamma_2)$ for any database state D consistent with Δ .

To find a simplification means then to choose among all the Δ -conditionally equivalent CWPs according to an optimality criterion. Usually this criterion serves as an abstraction over actual computation times. For this purpose, we introduce the notion of resource set, i.e., a portion of the Herbrand base (the set of all ground atoms) that affects the semantics of a constraint theory: the smaller the resource set, the better the CWP.

Definition 8 (Resource set). A subset \mathcal{R} of the Herbrand base is a resource set for a constraint theory Γ whenever

$$D(\Gamma) = D'(\Gamma)$$

for any two database states D, D' such that $D \cap \mathcal{R} = D' \cap \mathcal{R}$. If, furthermore, Γ admits no other resource set $\mathcal{R}' \subset \mathcal{R}$, \mathcal{R} is a minimal resource set for Γ .

Proposition 1. For any constraint theory Γ there exists a unique minimal resource set.

Proof (Sketch). Suppose that there exist two minimal resource sets \mathcal{R}_1 and \mathcal{R}_2 such that $\mathcal{R}_1 \neq \mathcal{R}_2$. It can be shown that their intersection $\mathcal{R}_1 \cap \mathcal{R}_2$ is also a resource set and therefore they are not minimal, against the hypotheses. \square

We indicate the minimal resource set of a constraint theory Γ as $\mathcal{R}(\Gamma)$ and in the following we shall omit the word “minimal”. We define an optimality criterion that selects the CWPs with the smallest resource sets.

Definition 9 (Optimality). Given two constraint theories Δ and Σ , Σ is Δ -optimal if its resource set $\mathcal{R}(\Sigma)$ is such that there exists no other constraint theory $\Sigma' \stackrel{\Delta}{\equiv} \Sigma$ with resource set $\mathcal{R}(\Sigma') \subset \mathcal{R}(\Sigma)$.

Several different optimality criteria can be defined for constraint theories. For example, another natural choice is a syntactic order based on the number of literals: the optimal theories are those with the minimal count. Another possibility is to define a semantic order that selects a weakest constraint theory (when a theory Γ_1 holds in more states than another theory Γ_2 , we say that Γ_1 is weaker than Γ_2 and that Γ_2 is stronger than Γ_1). It is possible to find examples where the optimal constraint theories found according to these three criteria (minimal resource set, minimal number of literals, semantic weakness) differ. We stress, however, that, with respect to efficient query evaluation, each criterion can only approximate optimality, as the actual execution times depend on the database state, that is not available at the time of the simplification process. For example, a syntactically minimal query does not necessarily evaluate faster than an equivalent non-minimal query in all database states; the amount of computation required to answer a query can be reduced, for instance, by adding a join with a very small relation. In the remainder of the paper we stick to the criterion of definition 9; this choice will be justified in section 4.4.

3.3 Transformations

We now sketchily describe the transformations that compose the simplification procedure for integrity constraints of [6]. Given an input constraint theory Γ and an update U , the procedure returns a simplified theory to be tested prior to the execution of U to guarantee consistency with respect to Γ after the update.

Definition 10. *Let Γ be a constraint theory and U an update. The notation $\text{After}^U(\Gamma)$ refers to a copy of Γ in which all atoms of the form $p(\vec{t})$ have been simultaneously replaced by*

$$(p(\vec{t}) \wedge \vec{t} \neq \vec{b}_1 \wedge \dots \wedge \vec{t} \neq \vec{b}_m) \vee \vec{t} \doteq \vec{a}_1 \vee \dots \vee \vec{t} \doteq \vec{a}_n.$$

where $p(\vec{a}_1), \dots, p(\vec{a}_n)$ are all additions on p in U and $\neg p(\vec{b}_1), \dots, \neg p(\vec{b}_m)$ all deletions.

We also assume that the result of the transformation of definition 10 is always given as a set of denials, which can be produced by using, e.g., De Morgan's laws. The semantic correctness of After is expressed by the following property.

Theorem 1. *For any update U and constraint theory Γ , $\text{After}^U(\Gamma)$ is a weakest precondition of Γ with respect to U .*

After may contain redundancies, such as, e.g., $a \doteq a$. Furthermore, the fact that the original integrity constraints hold in the current database state can be exploited to further simplify. For this purpose, we define a transformation Optimize that simplifies the input theory Γ based on a given set of trusted hypotheses Δ . We describe an implementation [19] in terms of sound rewrite rules; each application of a rule produces a smaller theory. Optimize applies the rules as long as possible and thus removes from Γ every denial that is subsumed by another denial in Δ or Γ and all literals that are proved to be redundant.

Definition 11. Given two constraint theories Δ and Γ , $\text{Optimize}_\Delta(\Gamma)$ is the result of applying the following rewrite rules on Γ as long as possible. In the following, x is a variable, t is a term, A, B are (possibly empty) conjunctions of literals, L is a literal, σ a substitution, ϕ, ψ are denials, Γ' is a constraint theory.

$$\begin{aligned}
\{\leftarrow x \doteq t \wedge A\} \cup \Gamma' &\Rightarrow \{\leftarrow A\{x/t\}\} \cup \Gamma' \\
\{\leftarrow A \wedge B\} \cup \Gamma' &\Rightarrow \{\leftarrow A\} \cup \Gamma' \text{ if } A \vdash B \\
\{\phi\} \cup \Gamma' &\Rightarrow \Gamma' \text{ if } \exists \psi \in (\Gamma' \cup \Delta) : \psi \text{ subsumes } \phi \\
\{\leftarrow A\} \cup \Gamma' &\Rightarrow \Gamma' \text{ if } A \vdash \text{false} \\
\{(\leftarrow A \wedge \neg L \wedge B)\sigma\} \cup \Gamma' &\Rightarrow \{(\leftarrow A \wedge B)\sigma\} \cup \Gamma' \text{ if } (\Gamma \cup \Delta) \vdash \{\leftarrow A \wedge L\}
\end{aligned}$$

The last rule mimics resolution; to avoid termination problems, it can be implemented, e.g., with a data driven procedure that looks in the deductive closure of $(\Gamma \cup \Delta)$ containing denials whose size is not bigger than the biggest denial in Γ . The operators can now be combined to define a procedure for simplification of integrity constraints.

Definition 12. For two constraint theories Γ and Δ and an update U , we define

$$\text{Simp}_\Delta^U(\Gamma) = \text{Optimize}_\Delta(\text{After}^U(\Gamma)).$$

We observe that Simp preserves range restriction, as all the steps in After and Optimize do. We state now its correctness and refer to [6] for a proof.

Theorem 2. Simp terminates on any input and, for any constraint theories Γ, Δ and update U , $\text{Simp}_\Delta^U(\Gamma)$ is a Δ -CWP of Γ with respect to U .

In the following, $\text{Simp}^U(\Gamma)$ is a shorthand for $\text{Simp}_\Gamma^U(\Gamma)$.

Example 2. Consider a binary database relation f containing child-father entries and the following integrity constraint:

$$\Gamma = \{\leftarrow f(x, y) \wedge f(x, z) \wedge y \neq z\}$$

meaning that no child can have two different fathers. The simplification with respect to an update pattern of the form $U = \{f(\mathbf{a}, \mathbf{b})\}$, where \mathbf{a} and \mathbf{b} are parameters, is as follows.

$$\begin{aligned}
\text{After}^U(\{\phi\}) = \{ &\leftarrow f(x, y) \wedge f(x, z) \wedge y \neq z, \\
&\leftarrow f(x, y) \wedge x \doteq \mathbf{a} \wedge z \doteq \mathbf{b} \wedge y \neq z, \\
&\leftarrow x \doteq \mathbf{a} \wedge y \doteq \mathbf{b} \wedge f(x, z) \wedge y \neq z, \\
&\leftarrow x \doteq \mathbf{a} \wedge y \doteq \mathbf{b} \wedge x \doteq \mathbf{a} \wedge z \doteq \mathbf{b} \wedge y \neq z \}.
\end{aligned}$$

$$\text{Simp}^U(\{\phi\}) = \{\leftarrow f(\mathbf{a}, y) \wedge y \neq \mathbf{b}\}.$$

□

4 Locks on simplified integrity constraints

4.1 Transactions

The notion of transaction includes, in general, write as well as read operations on database elements. We identify a database element¹ with a ground atom of the Herbrand base, and a write operation adds or removes such an atom from the database state. A transaction is always concluded with either a **commit** or an **abort**; in the former case the executed write operations are finalized into the database state, in the latter they are cancelled. We omit, for simplicity, the indication of **abort** and **commit** operations in transactions and consider the execution of a transaction concluded and committed after its last operation.

Definition 13 (Transaction). *A transaction T is a finite sequence of operations $\langle T^1, \dots, T^n \rangle$ such that each step T^i is either a $\text{read}(e_i)$ or a $\text{write}(e_i)$ operation on a database entity e_i .*

A schedule is a sequence of the operations of one or more transactions.

Definition 14 (Schedule). *A schedule σ over a set of transactions \mathcal{T} is an ordering of the operations of all the transactions in \mathcal{T} which preserves the ordering of the operations of each transaction. A schedule is serial if, for any two transactions T' and T'' in \mathcal{T} , either all operations in T' occur before all operations of T'' in σ or conversely. A schedule which is not serial, for $|\mathcal{T}| > 1$, is an interleaved schedule.*

A schedule executes correctly with respect to concurrency of transactions if it corresponds to the execution of some serial schedule, according to definition 15 below.

Definition 15 (Conflict serializability). *Two operations in a transaction are conflicting iff they refer to the same database entity and at least one of them is a write operation. Two schedules are conflict equivalent iff they contain the same set of committed transactions and operations and every pair of conflicting operations is ordered in the same way in both schedules. A schedule is conflict serializable iff it is conflict equivalent to a serial schedule.*

Checking conflict serializability can be done in linear time by testing the acyclicity of the directed graph in which the nodes are the transactions in the schedule and the edges correspond to the order of conflicting operations in two different transactions. Conflict serializability is a characterization of the correctness of schedules, based on the implicit assumption that each transaction, executed alone, maps any consistent database state to another consistent state (the “correctness principle”, [11]). In section 4.2 we will explicitly enforce this property by checking a CWP of the database integrity constraints with respect to the update corresponding to the given transaction.

¹ We also use the terms *database entity* and *resource* to indicate the same notion.

4.2 Extended transactions

In the update language discussed in section 3, the updates consist of write operations on database entities, where the entities are the tuples of the database relations. These operations are known and do not depend on previous read operations. Furthermore, an update does not contain conflicting operations, as the sets of additions and deletions are disjoint. Therefore we can, for the moment, restrict our attention to write transactions, i.e., sequences of non-conflicting write operations. In order to be able to map back and forth between write transactions and updates, we also indicate for each write if it is an addition or a deletion (using a \neg sign on the database element).

Definition 16 (Write transaction). *For a given update U , any minimal sequence T of write operations on all the literals in U is a write transaction on U . Given a write transaction T , we indicate the corresponding update as \overline{T} .*

Example 3. Let $U = \{p(a), \neg q(a)\}$ be an update. Then the only possible write transactions on U are:

$$\begin{aligned} T_1 &= \langle \text{write}(p(a)), \text{write}(\neg q(a)) \rangle \\ T_2 &= \langle \text{write}(\neg q(a)), \text{write}(p(a)) \rangle. \end{aligned}$$

Conversely, $\overline{T_1} = \overline{T_2} = U$. □

In the following we shall indicate the i -th element of a sequence S with the notation S^i . For a given write transaction T of size n and a database state D , the notation D^T refers to the database state $((D^{\overline{T^1}})^{\overline{T^2}}) \dots^{\overline{T^n}}$. The same notation applies to schedules over write transactions. Clearly, $D^T = D^{\overline{T}}$ for any write transaction T , whereas for a schedule σ the notation D^σ is not allowed, as σ might contain conflicting operations.

Proposition 2. *Let T be a write transaction, Γ a constraint theory, D a database state consistent with Γ and let Σ be a CWP of Γ with respect to \overline{T} . Then $D^T(\Gamma) = \text{true}$ iff $D(\Sigma) = \text{true}$.*

Proposition 2 follows immediately from definition 6 and indicates that a write transaction executes correctly if and only if the database state satisfies the requirements imposed by the corresponding CWP. This leads to the following notion of simplified write transaction.

Definition 17 (Simplified write transaction). *Let T be a write transaction, Γ a constraint theory and D a database state. The simplified write transaction of T with respect to Γ and D is $\langle \rangle^2$ if $D(\Sigma) = \text{false}$ and T if $D(\Sigma) = \text{true}$, where Σ is a CWP of Γ with respect to \overline{T} .*

Obviously, the execution of a simplified write transaction is always correct.

² The empty sequence.

Corollary 1. *Let T be a write transaction and Γ a constraint theory. Then, for every database state D consistent with Γ , $D^{T_S}(\Gamma) = \text{true}$, where T_S is the simplified write transaction of T with respect to Γ and D .*

In order to determine a simplified write transaction, the database state must be accessed. We can therefore model the behavior of a scheduler that dynamically produces simplified write transactions by starting them with read operations corresponding to the database actions needed to evaluate the CWP. Checking a constraint theory Γ corresponds to reading all database elements that contribute to the evaluation of Γ , i.e., its resource set. The effort of evaluating a constraint theory can then be expressed by concatenating (\circ) the sequence of all needed read operations with the (simplified) write transaction. To simplify the notation, we indicate any minimal sequence of read operations on every element of a resource set \mathcal{R} as $\text{Read}(\mathcal{R})$; in section 4.3 we shall use a similar notation for lock ($\text{Lock}(\mathcal{R})$) and unlock ($\text{Unlock}(\mathcal{R})$) operations.

Definition 18 (Simplified read-write transaction). *For a constraint theory Γ , a write transaction T and a database state D , any transaction of the form*

$$T' = \text{Read}(\mathcal{R}(\Sigma)) \circ T_S$$

is a simplified read-write transaction of T with respect to Γ and D , where T_S is the simplified write transaction of T with respect to Γ and D , and Σ is a CWP of Γ with respect to \bar{T} . The execution of T' is legal iff T_S starts at D .

A schedule executes legally if all its transactions do. For a schedule σ containing read operations, we write D^σ as a shorthand for D^{σ_w} , where σ_w is the sequence that contains all the write operations as in σ and in the same order, but no read operation. Note that a legal schedule over simplified read-write transactions is not guaranteed to execute correctly.

Example 4. [2 continued] Let us consider the two following transactions:

$$\begin{aligned} T_1 &= \langle f(\text{bart}, \text{homer}) \rangle \\ T_2 &= \langle f(\text{bart}, \text{ned}) \rangle. \end{aligned}$$

Any schedule σ over T_1, T_2 yields an inconsistent database state, i.e., $D^\sigma(\Gamma) = \text{false}$ for any database state D , because *bart* would end up having (at least) two different fathers. We therefore consider schedules over the simplified read-write transactions corresponding to T_1 and T_2 . Suitable CWPs of Γ with respect to \bar{T}_i , $i = 1, 2$, are given by **Simp** by instantiating the parameters with the actual constants in the parametric simplification found in example 2:

$$\begin{aligned} \Sigma_1 &= \text{Simp}^{\bar{T}_1}(\{\Gamma\}) = \{\leftarrow f(\text{bart}, y) \wedge y \neq \text{homer}\} \\ \Sigma_2 &= \text{Simp}^{\bar{T}_2}(\{\Gamma\}) = \{\leftarrow f(\text{bart}, y) \wedge y \neq \text{ned}\}. \end{aligned}$$

In this way a schedule such as

$$\sigma_1 = \text{Read}(\mathcal{R}(\Sigma_1)) \circ \langle \text{write}(f(\text{bart}, \text{homer})) \rangle \circ \text{Read}(\mathcal{R}(\Sigma_2)) \circ \langle \text{write}(f(\text{bart}, \text{ned})) \rangle$$

would not be a legal schedule over simplified read-write transactions, because in the database state after the last Read, Σ_2 necessarily evaluates to *false*. However it is still possible to create legal schedules leading to an inconsistent final state. Consider, e.g., the following schedule:

$$\sigma_2 = \text{Read}(\mathcal{R}(\Sigma_1)) \circ \text{Read}(\mathcal{R}(\Sigma_2)) \circ \langle \text{write}(f(\text{bart}, \text{homer})), \text{write}(f(\text{bart}, \text{ned})) \rangle.$$

At the end of the $\text{Read}(\mathcal{R}(\Sigma_2))$ sequence, transaction T_1 has not yet performed the write operation on f , so Σ_2 can evaluate to *true*, but the final state will be inconsistent. \square

4.3 Locks

The problem pointed out in example 4 is due to the fact that some of the elements in the resource set of a CWP of a transaction T were modified by another transaction before T finished its execution. In order to prevent this situation we need to introduce locks. A lock is an operation of the form $\text{lock}(e)$ where e is a database element; the lock on e is released with the dual operation $\text{unlock}(e)$. A transaction containing lock and unlock operations is a *locked transaction*. A scheduler for locked transactions verifies that the transactions behave consistently with the locking policy, i.e., database elements are only accessed by transactions that have acquired the lock on them, no two transactions have a lock on the same element at the same time and all acquired locks are released. The two-phase locking (2PL) protocol requires that in every transaction all lock operations precede all unlock operations; all 2PL transactions are conflict serializable.

We can now extend the notion of simplified read-write transaction with locks on the set of resources that are read at the beginning of the transaction and on the elements to be written. With the notation $\mathcal{R}(U)$, U an update, we refer to the set of ground atoms occurring in U .

Definition 19 (Simplified locked transaction). *For a constraint theory Γ , a write transaction T and a database state D , any transaction of the form*

$$T' = \text{Lock}(\mathcal{R}(\Sigma)) \circ \text{Read}(\mathcal{R}(\Sigma)) \circ \text{Lock}(\mathcal{R}(\bar{T}_S) \setminus \mathcal{R}(\Sigma)) \circ T_S \circ \text{Unlock}(\mathcal{R}(\bar{T}_S) \cup \mathcal{R}(\Sigma))$$

is a simplified locked transaction of T with respect to Γ and D , where T_S is the simplified write transaction of T with respect to Γ and D , and Σ is a CWP of Γ with respect to \bar{T} . The execution of T' is legal iff T_S starts at D .

Any legal schedule over simplified locked transactions is guaranteed to be correct. For such a schedule σ , we write D^σ as a shorthand for D^{σ_w} , where σ_w is the sequence that contains all the write operations as in σ and in the same order, but no read, lock or unlock operation.

Theorem 3. *Let Γ be a constraint theory and D a database state such that $D(\Gamma) = \text{true}$. Given the write transactions T_1, \dots, T_n , let T'_i be the simplified locked transaction of T_i with respect to Γ and the state D_i reached after the last Lock in T'_i , for $1 \leq i \leq n$. Then any legal schedule σ over $\{T'_1, \dots, T'_n\}$ is conflict serializable and $D^\sigma(\Gamma) = \text{true}$.*

Proof. (Sketch) Conflict serializability follows from the fact that all simplified locked transactions are 2PL. Consistency of the final state is guaranteed by the level of isolation of each transaction T'_i : no other transaction can modify elements in the resource set of T'_i while T'_i executes. The only possible concurrent execution is limited to the elements to be written by T'_i before T'_i acquires the lock. However, once T'_i has the lock, these elements will be updated by T'_i . Therefore, after the last write in T'_i , the integrity constraints of Γ that are affected by T'_i will be satisfied, as the CWP of Γ with respect to \bar{T}_i is; hence the consistency of D^σ . \square

The locks we have considered for simplified locked transactions are *exclusive* locks; note, however, that if the locks in the Read sequence were *shared*, the kind of inconsistency shown in example 4 could still occur.

4.4 Minimality of locked resources

The result of theorem 3 describes a transaction system in which all possible schedules execute correctly. However, the database performance can vary dramatically, depending on the chosen schedule, on the database state and, in this case, on the waits due to the locks. For this reason, we observe that if the CWPs in use in the simplified locked transactions are minimal according to the resource set criterion, then the amount of locked database resources is also minimal; this maximizes the database throughput. Unfortunately, in general there cannot be any simplification procedure that is guaranteed to produce optimal results for all inputs - and this for any of the orderings described in section 3.2. The impossibility of a “perfect” simplification procedure stems from the undecidability of the query containment problem (QC), which can be reduced to the problem of simplifying a given theory to *true* (which is indeed minimal in any order). Conversely, for all contexts in which QC is decidable it is possible to devise a simplification procedure, based on QC, that always reaches a minimum. This can be done, e.g., by enumerating all theories that precede a given CWP, such as *After*, in the ordering and that are conditionally equivalent to it. The enumeration can be done in different ways, depending on the ordering; conditional equivalence can be checked by QC. A large body of research on QC has described classes of expressive languages in which the problem is decidable, such as monadic DATALOG queries and conjunctive two-way regular path queries [2]. For these classes, we can minimize the amount of locked resources if we assume an ideal simplification procedure Simp^+ based on the resource set ordering.

Theorem 4. *In any language in which QC is decidable, the amount of locked resources for the simplified locked transactions based on Simp^+ is minimal, i.e., for any schedule construction policy that locks fewer resources than in theorem 3 it is possible to define legal schedules that produce inconsistent database states.*

Proof (Sketch). Suppose that a transaction T_1 does not lock one of the entities in the resource set of its optimal CWP Σ_1 . Then it is possible to construct a transaction that modifies that unlocked entity in such a way that Σ_1 does not hold when T_1 starts writing, thus producing an inconsistent final state. \square

A simplification procedure based on enumeration is unlikely to be of any use, since impractically many CWP candidates will typically be generated; furthermore, it does not apply to the languages in which QC is undecidable. Note, however, that in the decidable classes for QC, **Simp** behaves ideally with respect to the ordering based on literal counting, and from the syntactic minimum we can efficiently generate an optimal result in the resource set sense. In all the other cases, **Simp** produces results that approximate the optimal CWP, but that can contain some redundancies. In particular, this may happen when cyclic patterns and recursive definitions are encoded in the integrity constraints, thus rendering the detection of redundancy harder. The simplified locked transactions resulting from **Simp** will, in these cases, indicate a little extra amount of resources to be locked, but will still be relevant for practical purposes.

5 Conclusion

We have shown a lock-based schedule construction policy that guarantees conflict serializability and correctness, which are essential qualities in any concurrent database system. For logical fragments in which the query containment problem is decidable, we have shown that an ideal simplification procedure for integrity constraints determines a minimal amount of database resources to be locked. The proposed implementation **Simp** is also optimal within these fragments, and produces results that are close to the optimal ones in the other cases.

We believe that this method can have a significant impact on current database practice. An immediate application is, e.g., the implementation of a database driver that, upon requests from an application, communicates with the database and transparently carries out the required lock operations and simplified checks. This results in major assets in terms of efficiency, such as simplified integrity checking and minimal amount of locks, all with a compiled approach, as the simplifications are generated at design time.

Although we discussed an update language that only admits tuple additions and deletions, we deem it possible to extend these results to transactions defined with more expressive rule-based update languages [1]; as for the simplification procedure, these updates were considered in [20]. Another direction of research includes the investigation of deadlocks, that can still occur in the present version of the method.

Acknowledgements The author wishes to thank Henning Christiansen for many helpful comments and suggestions.

References

1. Abiteboul, S., Hall, R., Vianu, V., *Foundations of Databases*. Addison-Wesley, Reading, Massachusetts (1995).
2. Calvanese, D., De Giacomo, G., Vardi, M. Decidable Containment of Recursive Queries. *ICDT 2003*: 330–345, LNCS 2572, Springer (2002).

3. Chakravarthy, U. S., Grant, J., Minker, J. Foundations of semantic query optimization for deductive databases. *Foundations of Deductive Databases and Logic Programming*: 243–273, Minker, J. (Ed.), Morgan Kaufmann (1988).
4. Chakravarthy, U., Grant, J., Minker, J. Logic-based approach to semantic query optimization. *ACM TODS* 15(2): 162–207, ACM Press (1990).
5. Cosmadakis, S., Ioannidou, K., Stergiou, S. View Serializable Updates of Concurrent Index Structures. *DBPL 2001*: 247-262, LNCS 2397, Springer (2001).
6. Christiansen, H., Martinenghi, D., Simplification of database integrity constraints revisited: A transformational approach. *LOPSTR 2003*, to appear in LNCS, Springer (2004).
7. Christiansen, H., Martinenghi, D. Simplification of integrity constraints for data integration. *FoIKS 2004*: 31–48, LNCS 2942, Springer (2004).
8. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall (1976).
9. Decker, H., Celma, M. A slick procedure for integrity checking in deductive databases. *ICLP 1994*: 456–469, MIT Press (1994).
10. Eswaran, K., Gray, J., Lorie, R., Traiger, I. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19(11): 624–633, ACM Press (1976).
11. Garcia-Molina, H., Ullman, J. D., Widom, J.: *Database Systems. The complete book*. Prentice-Hall (2002).
12. Grant, J., Minker, J. Integrity Constraints in Knowledge Based Systems. In *Knowledge Eng. II, Applications*: 1–25, Adeli, H. (Ed.), McGraw-Hill (1990).
13. Godfrey, P., Grant, J., Gryz, J., Minker, J. Integrity Constraints: Semantics and Applications. *Logics for Databases and Information System*: 265–306, Chomicki, J. Saake, G. (Eds.), Kluwer (1988).
14. Henschen, L., McCune, W., Naqvi, S. Compiling Constraint-Checking Programs from First-Order Formulas. *Advances in Database Theory 1982*, volume 2: 145–169, Gallaire, H., Nicolas, J.-M., Minker, J. (Eds.), Plenum Press (1984).
15. Hoare, C.A.R. (1969). An axiomatic basis for computer programming. *Commun. ACM* 12(10): 576–580, ACM Press (1969).
16. Leuschel, M., De Schreye, D. Creating Specialised Integrity Checks Through Partial Evaluation of Meta-Interpreters. *JLP* 36(2): 149–193 (1998).
17. Lloyd, J., Sonenberg, L., Topor, R. (1987). Integrity Constraint Checking in Stratified Databases. *JLP* 4(4): 331–343 (1987).
18. Lechtenbörger, J., Vossen, G. On Herbrand Semantics and Conflict Serializability of Read-Write Transactions. *PODS 2000*: 187-194, ACM (2000).
19. Martinenghi, D. A Simplification Procedure for Integrity Constraints. *World Wide Web*, <http://www.dat.ruc.dk/~dm/spic/index.html> (2003).
20. Martinenghi, D. Simplification of integrity constraints with aggregates and arithmetic built-ins. *FQAS 2004*, to appear (2004) in LNAI, Springer (2004).
21. Nicolas, J.-M. Logic for Improving Integrity Checking in Relational Data Bases., *Acta Inf.* 18: 227–253 (1982).
22. Nilsson, U., Małczyński, J. *Logic, Programming and Prolog* (2nd ed.), John Wiley & Sons Ltd. (1995).
23. Qian, X. An Effective Method for Integrity Constraint Simplification. *ICDE 1988*: 338–345, IEEE Computer Society (1988).
24. Salem, K., Garcia-Molina, H., Shands, J. Altruistic Locking. *ACM Trans. Database Syst.* 19(1): 117–165 (1994).
25. Yannakakis, M. A Theory of Safe Locking Policies in Database Systems. *Journal of ACM* 29(3): 718-740 (1982).