Transaction management with integrity checking

Davide Martinenghi and Henning Christiansen

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

Abstract. Database integrity constraints, understood as logical conditions that must hold for any database state, are not fully supported by current database technology. It is typically up to the database designer and application programmer to enforce integrity via triggers or tests at the application level, which are difficult to maintain and error prone. Two important aspects must be taken care of. 1. It is too time consuming to check integrity constraints from scratch after each update, so simplified checks before each update should be used relying on the assumption that the current state is consistent. 2. In concurrent database systems, besides the traditional correctness criterion, the execution schedule must ensure that the different transactions can overlap in time without destroying the consistency requirements tested by other, concurrent transactions. We show in this paper how to apply a method for incremental integrity checking to automatically extend update transactions with locks and simplified consistency tests on the locked elements. All schedules produced in this way are conflict serializable and preserve consistency in an optimized way.

1 Introduction

When an update transaction is executed on a database, it is important to ensure that database consistency 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 concurrency of the accesses to the resources. Another aspect of correctness is determined by the semantic requirements expressed by integrity constraints (ICs). ICs are logical formulas that characterize the consistent states of a database. Integrity maintenance is a central issue, as without any guarantee of data consistency, answers to queries become unreliable. A full check of consistency often requires polynomial time wrt the size of the database, which is usually too costly. We therefore need to simplify the ICs into specialized checks that can be executed more efficiently at each update, employing the hypothesis that the initial database state was consistent. To optimize 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. This principle, known as *simplification* of ICs, has been long known and recognized [14], but *ad hoc* techniques are still prevalent. 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 wrt changes in the database schema. This suggests a need for automated simplification methods. Although a few, standard principles exist [2], none of them has prevailed in current database systems. In this paper we refer to a simplification procedure [3] based on transformations that produce simplified ICs already at design time; these are necessary and sufficient conditions for consistency that can be tested prior to the execution of updates. Due to space constraints, we restrict updates to tuple additions and deletions and disregard aggregates in ICs. We focus on the interaction between integrity checking and locking policies; we present a method that uses the simplified ICs not only to ensure consistency of each individual update transaction, but also to determine the minimal amount of database resources to be locked in order to guarantee the correctness of all legal schedules.

The paper extends the results presented in [12] and is organized as follows. We review existing literature in the field in section 2. Simplification of ICs is introduced in section 3, while the results concerning its application to database locking are explained and exemplified in section 4. Further discussion on the applicability of the method and concluding remarks are provided in section 5.

2 Related works

Simplification of ICs is an essential optimization principle for database consistency checking. We emphasize, furthermore, the importance of identifying a violation to be introduced by an update before it is executed, so that inconsistent states are completely avoided. Several approaches to simplification do not comply with this requirement, e.g., [14, 11, 5, 8]. Other methods, e.g., [10], provide pretests that, however, are not proven to be necessary conditions; in other words, if the tests fail, nothing can be concluded about consistency. Integrity checking is often regarded as an instance of materialized view maintenance: integrity constraints are defined as views that must always remain empty for the database to be consistent. These approaches use update propagation techniques to perform the task in an incremental way. However, they typically require "immediate" view maintenance (i.e., after each step in a transaction, which is more costly) or, if "deferred" maintenance is allowed, then additional overhead is needed (e.g., to avoid the *state bug*). We refer to [9] for a survey of these methods. Triggers' reactive behavior has been used since [1] for integrity enforcement without, however, semantically optimizing the triggering condition. Semantic optimization of triggers was introduced in [4], where, however, only single updates are allowed. For these reasons, none of the above methods is well-suited for concurrent database transactions. As for locking, the literature is rich in methods aimed at the correctness of concurrently executed transactions. The most common protocol is two-phase locking, but others have been proposed, e.g., [16]. All these methods depend on the implicit assumption that each single transaction preserves consistency when executed alone [6], which, thus, entrusts this responsibility to the designer of the transactions. We present, instead, an integrated approach to automatically obtain this combined with locking, so that the designer need only concentrate on the declarative specification of integrity constraints.

3 A simplification procedure for integrity constraints

3.1 Preliminaries

We describe our proposal in the function-free first-order language DATALOG extended with default negation (DATALOG$^\neg$) and assume familiarity with the notions of *terms* $(t, s, \ldots)$, *variables* $(x, y, \ldots)$, *constants* $(a, b, \ldots)$, *predicates* $(p, q, \ldots)$, *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*, the *constraint theory* (CT) or set of ICs, and the *intensional database* or set of *rules* [7]. We can disregard the intensional database, as, without recursion, the CT and database can be transformed in equivalent ones that do not contain any intensional predicates [2]. By *database state* we refer to the extensional part only. We further assume that every clause is *range restricted*, i.e., each variable in it appears in a positive database literal in the body. The truth value of a closed formula F, relative to a database state D, is defined as its evaluation in D's standard model [15] and denoted $D(F)$.

Definition 1 (Consistency). *A database state D is* consistent *with a CT Γ iff $D(\Gamma) = true$.*

The simplification method we describe here can handle general forms of update, but, for reasons of space, we limit our attention to sets of additions and deletions.

Definition 2 (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 $(D \cup U^+) \setminus \neg U^-$.*

Updates can also contain *parameters* (written in boldface: $\mathbf{a}$, $\mathbf{b}$, ...), that are placeholders for constants. In this way we can generalize updates into update *patterns* and simplify ICs for classes of updates, rather than specific updates. For example, the notation $\{p(\mathbf{a}), \neg q(\mathbf{a})\}$, $\mathbf{a}$ a parameter, indicates the class of updates that add a tuple to the relation p and remove the same tuple from the relation q. We refer to [3] for further discussion about parameters.

3.2 Semantic notions

We characterize the semantic correctness of simplification with the notion of conditional weakest precondition, i.e., a test that can be checked in the present state but indicating properties of the new state, further optimized with the hypothesis that the present state is consistent.

Definition 3 (Conditional weakest precondition). *Let Γ, Δ be CTs and U an update. A CT Σ is a Δ-*conditional weakest precondition *(Δ-CWP) of Γ wrt U whenever $D(\Sigma) = D^U(\Gamma)$ for any database state D consistent with Δ.*

In definition 3, Δ will typically include Γ and perhaps further properties of the database that are trusted. All CWPs of the same CT wrt the same update are in an equivalence class called conditional equivalence.

Definition 4 (Conditional equivalence). *Let Δ, Γ_1, Γ_2 be CTs; then Γ_1 and Γ_2 are conditionally equivalent wrt Δ, denoted $\Gamma_1 \overset{\Delta}{\equiv} \Gamma_2$, 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 that serves as an abstraction over actual computation times. We then introduce the notion of resource set, i.e., a portion of the Herbrand base $\mathcal{B}$ (the set of all ground atoms) that affects the semantics of a CT: the smaller the resource set, the better the CWP.

Definition 5 (Resource set). *A subset $\mathcal{R}$ of the Herbrand base is a* resource set *for a CT 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 CT there exists a unique minimal resource set.*

We indicate the minimal resource set of a CT Γ as $\mathcal{R}(\Gamma)$ and in the following we shall omit the word "minimal". For example, the resource set $\mathcal{R}(\{\leftarrow f(c,y) \wedge y \neq d\})$ is $\{f(x,y) \mid x = c \wedge y \neq d\}$. For an update U, the notation $\mathcal{R}(U)$ refers to the set of ground atoms occurring in U.

A CWP is independent of the atoms of the update upon which it is calculated.

Proposition 2. *Let Γ, Δ be CTs, U an update and Σ a Δ-CWP of Γ wrt U. Then $\mathcal{R}(\Sigma) \cap \mathcal{R}(U) = \emptyset$.*

We define an optimality criterion that selects the CWPs with the smallest resource sets.

Definition 6 (Optimality). *Given two CTs Δ and Σ, Σ is Δ-*optimal *if there exists no other CT $\Sigma' \overset{\Delta}{\equiv} \Sigma$ such that $\mathcal{R}(\Sigma') \subset \mathcal{R}(\Sigma)$.*

Different optimality criteria can be defined for CTs. For example, another natural choice is a syntactic order based on the number of literals: the optimal theories are those with the minimal count. It is possible to find examples where the optimal CTs found according to these criteria (minimal resource set, minimal number of literals) differ. However, it should be made clear that they can serve only as approximative comparisons of execution times, which may vary highly in different database state. ICs are simplified as to provide best choices that apply to *any* database state, and we must rely on the query optimizers embedded in standard RDBMSs to take into account state information such as the size of each relation. 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 6, as this proves useful for locking purposes (see section 4.3).

Syntactic notions, such as subsumption, can be used to define a simplification procedure that, for an input CT Γ and update U, produces a CWP of Γ wrt U, indicated as $\mathsf{Simp}^U(\Gamma)$; we refer to the implementation given in [3].

Example 1. Consider a database relation f containing child-father entries and the CT $\Gamma = \{\leftarrow f(x,y) \land f(x,z) \land y \neq z\}$, meaning that no child can have two different fathers. The simplification wrt the update pattern $U = \{f(\mathbf{a},\mathbf{b})\}$, where $\mathbf{a}$ and $\mathbf{b}$ are parameters, is $\mathsf{Simp}^U(\Gamma) = \{\leftarrow f(\mathbf{a},y) \land y \neq \mathbf{b}\}$, which indicates that the added child $\mathbf{a}$ cannot already have a father different from $\mathbf{b}$.

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 element (or *resource*) 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 7 (Transaction). *A* transaction T *is a finite sequence of operations* $\langle T^1, \ldots, T^n \rangle$ *such that each step* T^i *is either a* $\mathsf{read}(e_i)$ *or a* $\mathsf{write}(e_i)$ *operation on a database element* e_i*.*

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

Definition 8 (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 wrt concurrency of transactions if it corresponds to the execution of some serial schedule, according to definition 9 below.

Definition 9 (Conflict serializability). *Two operations in a transaction are* conflicting *iff they refer to the same database element 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.

4.2 Extended transactions

In the update language discussed in section 3, the updates consist of write operations on database elements, where the elements 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 10 (Write transaction). *For a given update U, any sequence T, of minimal length, 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 2. The possible write transactions on an update $U = \{p(a), \neg q(a)\}$ are $T_1 = \langle \mathsf{write}(p(a)), \mathsf{write}(\neg q(a))\rangle$ and $T_2 = \langle \mathsf{write}(\neg q(a)), \mathsf{write}(p(a))\rangle$. 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 $(\ldots((D^{\{T^1\}})^{\{T^2\}})^{\cdots})^{\{T^n\}}$. The same notation applies to schedules over write transactions. Clearly, $D^T = D^{\overline{T}}$ for any write transaction T, while for a schedule σ the notation $D^{\overline{\sigma}}$ is not allowed, as σ might contain conflicting operations, i.e., σ cannot be mapped to an update $\overline{\sigma}$.

Proposition 3. *Let T be a write transaction, Γ a CT, D a database state consistent with Γ and let Σ be a CWP of Γ wrt $\overline{T}$. Then $D^T(\Gamma) = true$ iff $D(\Sigma) = true$.*

Proposition 3 indicates that a write transaction executes correctly if and only if the database state satisfies the corresponding CWP. This leads to the following notion of simplified write transaction.

Definition 11 (Simplified write transaction). *Let T be a write transaction, Γ a CT and D a database state. The* simplified write transaction *of T wrt Γ and D is $\langle\rangle$*[1] *if $D(\Sigma) = false$ and T if $D(\Sigma) = true$, where Σ is a CWP of Γ wrt $\overline{T}$.*

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

Corollary 1. *Let T be a write transaction and Γ a CT. Then, for every database state D consistent with Γ, $D^{T_S}(\Gamma) = true$, where T_S is the simplified write transaction of T wrt 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 CT Γ corresponds to reading all database elements that contribute to the evaluation of Γ, i.e., its resource set. The effort of evaluating a CT 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 $\mathrm{Read}(\mathcal{R})$; in section 4.3 we shall use a similar notation for lock ($\mathrm{Lock}(\mathcal{R})$) and unlock ($\mathrm{Unlock}(\mathcal{R})$) operations.

Definition 12 (Simplified read-write transaction). *For a CT Γ, a write transaction T and a database state D, any transaction of the form*

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

is a simplified read-write transaction *of T wrt Γ and D, where T_S is the simplified write transaction of T wrt Γ and D, and Σ is a CWP of Γ wrt $\overline{T}$. 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 3. [1 continued] Let us consider the transactions $T_1 = \langle f(bart, homer)\rangle$ and $T_2 = \langle f(bart, ned)\rangle$. Any schedule σ over T_1, T_2 yields an inconsistent database state, i.e., $D^\sigma(\Gamma) = false$ for any 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 wrt $\overline{T}_i$, $i = 1, 2$, are given by Simp by instantiating the parameters with the actual constants in the parametric simplification found in example 1:

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

[1] The empty sequence.

In this way a schedule such as

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

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:

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

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

4.3 Locks

The problem pointed out in example 3 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 $\mathsf{lock}(e)$ where e is a database element; the lock on e is released with the dual operation $\mathsf{unlock}(e)$. 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.

Definition 13 (Simplified locked transaction). *For a CT Γ, a write transaction T and a database state D, any transaction of the form*

$$T' = \mathrm{Lock}(\mathcal{R}(\Sigma)) \circ \mathrm{Read}(\mathcal{R}(\Sigma)) \circ \mathrm{Lock}(\mathcal{R}(\overline{T}_S) \circ T_S \circ \mathrm{Unlock}(\mathcal{R}(\overline{T}_S) \cup \mathcal{R}(\Sigma))$$

is a simplified locked transaction *of T wrt Γ and D, where T_S is the simplified write transaction of T wrt Γ and D, and Σ is a CWP of Γ wrt $\overline{T}$. 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 1. *Let Γ be a CT and D a database state such that $D(\Gamma) = true$. Given the write transactions $T_1, \ldots, T_n$, let T_i' be the simplified locked transaction of T_i wrt 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 $\{T_1', \ldots, T_n'\}$ is conflict serializable and $D^\sigma(\Gamma) = true$.*

The result of theorem 1 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 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 minimized, which increases the database throughput. It is not possible to obtain a minimal simplification *in all cases*, as query containment (which is in general undecidable) and simplification can be reduced to one another. Anyhow, any good approximation of an ideal simplification procedure, i.e., one which might occasionally contain some redundancy, will be useful as a component of an architecture for transaction management based on our principles. For example, the procedure of [3] produced optimal results for non-recursive databases in all examples we tested.

5 Discussion

The proposed approach describes a schedule construction policy that guarantees conflict serializability and correctness, which are essential qualities in any concurrent database system. There are several evident improvements wrt previous approaches. Firstly, it complies with the semantics of deferred integrity checking (i.e., ICs do *not* have to hold in intermediate transaction states), as opposed to what transaction transformation techniques for view maintenance typically offer. Secondly, it features an early detection of inconsistency — before the execution of the update and without simulating the updated state — that allows one to avoid executions of illegal transactions and subsequent rollbacks. This requires the introduction of locks, whose amount is, however, minimized and, in the case of well designed transactions, often null. Consider, e.g., a referential IC and an update that inserts both the referencing and the referenced tuple: no lock and no check are needed with our approach, whereas an (immediate) view maintenance approach would require two checks. Experiments were run with the help of a prototype of the simplification procedure [13], and implemented as stored procedures, on the ICs described in [17], that include rather large sets of complex ICs. An initial comparison with the simplification method of [17][2], has shown that we can save up to 75% of the execution time upon illegal transactions, whereas the performance is similar when the transactions are legal. We also stress that complex ICs, such as arbitrary functional dependencies, beyond key and foreign key constraints, are necessary to capture the semantics of complex scenarios[3].

The presented approach can be further refined in a number of ways. For example, we have implicitly assumed *exclusive* locks so far; however, a higher degree of concurrency is obtained, without affecting theorem 1, by using *shared* locks on $\mathcal{R}(\Sigma)$ and *update* locks on $\mathcal{R}(\overline{T}_S)$ in definition 13. Besides, to achieve serializability, a so-called *predicate lock* (PL) needs to be used on the condition given by the simplified ICs; as is well-known, PLs require a high computational

[2] Shown by its authors to perform better than previous approaches, such as [11].
[3] However, in some cases, they may be a symptom of a badly designed schema.

effort and, implementation-wise, are approximated as *index locks*. To this end, performance is highly improved by adding an index for every argument position occupied by an update parameter in a database literal in the simplified ICs. Finally, we note that, although the introduced locks may determine deadlocks, all standard deadlock detection and prevention techniques are still applicable. For instance, an arbitrary deadlocked transaction T can be restarted after unlocking its resources and granting them to the other deadlocked transactions.

References

1. S. Ceri and J. Widom. Deriving production rules for constraint maintainance. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases*, pages 566–577. Morgan Kaufmann, 1990.
2. U. S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kaufmann, 1988.
3. H. Christiansen and D. Martinenghi. Simplification of database integrity constraints revisited: A transformational approach. In M. Bruynooghe, editor, *LOPSTR'03*, volume 3018 of *LNCS*, pages 178–197. Springer, 2004.
4. H. Decker. Translating advanced integrity checking technology to sql. In *Database integrity: challenges and solutions*, pages 203–249. Idea Group Publishing, 2002.
5. H. Decker and M. Celma. a slick procedure for integrity checking in deductive databases. In P. Van Hentenryck, editor,*ICLP '94*, pages 456–469.MIT Press,1994.
6. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems. The complete book.* Prentice-Hall, 2002.
7. P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. In *Logics for Databases and Information Systems*, pages 265–306. Kluwer, 1998.
8. J. Grant and J. Minker. Integrity constraints in knowledge based systems. In H. Adeli, editor, *Knowl. Eng. Vol II, Applications*, pages 1–25. McGraw-Hill, 1990.
9. A. Gupta and I. S. Mumick (eds.). *Materialized Views. Techniques, Implementations, and Applications.* MIT Press, 1999.
10. L. Henschen, W. McCune, and S. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *ADT'88*, volume 2, pages 145–169. Plenum Press, New York, 1984.
11. J. W. Lloyd, L. Sonenberg, and R. W. Topor. Integrity constraint checking in stratified databases. *JLP*, 4(4):331–343, 1987.
12. D. Martinenghi. Optimal database locks for efficient integrity checking. In *ADBIS (Local Proceedings)*, pages 64–77, 2004.
13. D. Martinenghi. A simplification procedure for integrity constraints. http://www.dat.ruc.dk/ dm/spic/index.html, 2004.
14. J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.
15. U. Nilsson and J. Małuzyński. *Logic, Programming and Prolog (2nd ed.).* John Wiley & Sons Ltd, 1995.
16. K. Salem, H. Garcia-Molina, and J. Shands. Altruistic locking. *ACM Trans. Database Syst.*, 19(1):117–165, 1994.
17. R. Seljée and H. C. M. de Swart. Three types of redundancy in integrity checking: An optimal solution. *Data & Knowledge Engineering*, 30(2):135–151, 1999.