PD-Vol. 43, Computer Applications and Design Abstraction
ASME 1992

USING TRANSACTION IMPORT STATEMENTS FOR INTEGRITY CONTROL OPTIMIZATION

Stefan Böttcher*
Daimler Benz AG
Forschung und Technik Ulm
Ulm, Germany

Abstract

Integrity control is important in order to preserve correctness of the database, but it tends to be a major bottle-neck of database transaction processing. We present an optimization technique which reduces the number of integrity checks which a transaction has to perform. The presented optimization can be combined with optimizations suggested by other researchers. The key idea of our optimization is to prove that a given transaction can not violate a given integrity constraint. These proofs require to rewrite the integrity constraint and to compute those parts of a relation which are relevant for a possible violation of the integrity constraint. The paper focusses on the rewrite rules underlying the optimization.

Furthermore, we describe how the optimization can be integrated into the database system and how the database programming language can support compile time optimization. We use the database programming language DBPL and the DBPL database system which have been developed at the University of Frankfurt as an example. If a transaction procedure of a database application program describes in an IMPORT statement on which part of a relation the transaction procedure intends to write, then the presented optimization at compile time filters out a variety of integrity constraints which can not be violated by the transaction executed at run time. However, if subrelations accessed by a transaction can only be determined at run time, then the presented optimization is still useful to filter out integrity constraints which need not be checked, because they can not be violated.

*This work has been partially done at the University of Frankfurt and has been supported by the Deutsche Forschungsgemeinschaft under Grant-No SCHM350/3-1. A previous version of this paper has been written at IBM Deutschland GmbH, Scientific Center, Institute for Knowledge Based Systems, P.O.Box 80 08 80, D-7000 Stuttgart 80, West Germany.

1 Introduction

During the last decade optimization of integrity checking has been discussed mainly in two directions. On the one hand, the optimization of integrity checks at run time has been investigated on the basis of individual or multiple write operations, e.g. [SV87], [BDM88], [KSS87], [SK88], [PO90], [Nic82] and [BB82]. On the other hand, it has been suggested to modify the transaction code at compile time, i.e. to integrate additional queries for integrity checking, in order to guarantee that all integrity constraints are preserved by the transaction (c.f. e.g. [SS86] and [SK86]).

This paper proposes a third way of integrity control which is similar to ideas described in [Mot89]. Nevertheless, the proposal of this paper goes far beyond the approach of [Mot89] which does no optimization of integrity checks and allows to state only integrity constraints which can be expressed in a small subclass of tuple relational calculus. However, we present an optimization reducing the number of integrity checks and our optimization can be applied to arbitrary integrity constraints given as boolean tuple relational calculus queries.

We assume that the transaction programmer specifies in an IMPORT declaration for each transaction procedure which part of which database relation this transaction procedure wants to modify at most, as suggested by [SM90] and [MRS84]. [1] The basic idea of our approach is to use this import information for a proof that certain transaction procedures of an application program can never violate certain integrity constraints and that therefore the executed transactions need not check these integrity constraints.

Standard integrity control optimization techniques (e.g. [BDM88], [SV87], [BB82]) reduce the number of inte-

[1] We use the term *transaction procedure* for the piece of code of the application program which describes the algorithm for the transaction execution, whereas we use the term *transaction* for the actual execution sequence of the transaction procedure.

grity checks and reduce the query complexity of integrity checks. The number of integrity checks which have to be performed by a transaction is reduced e.g. by the following rules:

- A certain integrity check needs not be done by a transaction if no write operation of the transaction changes a relation which occurs in the integrity constraint.
- An insert operation on a relation can not violate an integrity constraint, if the relation occurs only positive [2] in that integrity constraint.
- A delete operation on a relation can not violate an integrity constraint, if the relation occurs only negative in that integrity constraint.

Compared to these approaches our approach is more selective, i.e. it further reduces the number of integrity constraints which have to be checked, because it finds out that additionally certain integrity checks need not be done by a transaction although the transaction executes write operations which modify relations which occur in the integrity constraint. Therefore, our approach can be combined with these integrity checking techniques in such a way that our approach is used as a filter, in order reduce the number of integrity checks, whereas the other approaches can be used for optimizing the remaining integrity checks.

Compared to the second group of approaches which compile the integrity constraints into transaction queries and augment the transaction code by these queries, our approach lets the transaction code unchanged and still allows to distinguish between integrity constraints and other transaction queries. Besides other advantages, this distinction between integrity checks and other queries has the advantage that the scheduler of the database system can allow for more parallelism between the transactions (c.f. [Böt90b]).

2 A motivating example

This section presents an example consisting of an integrity constraint and a transaction procedure in order to demonstrate how integrity checking can be optimized. The example integrity constraint is the following (all examples are written in the database programming language DBPL [SEM88], [SM90]):

IC1 "Employees who are older than 40 years earn at least 3000":

```
ALL e IN emp ( e.age > 40 ⇒ e.sal ≥ 3000 )
```

The integrity check can also be written as:

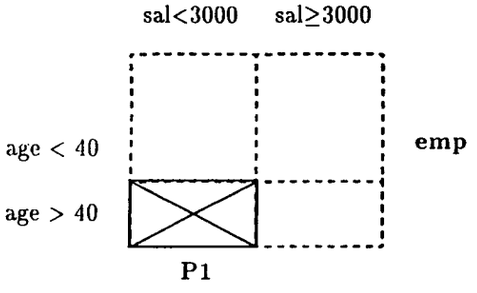
IC1' "There is no employee who is older than 40 years and earns less than 3000":

```
NOT SOME e IN emp
( e.age > 40 AND e.sal < 3000 )
```

The following diagramm shows the schema of the relation emp and the subrelation

```
P1 { EACH e IN emp
   ( e.age > 40 AND e.sal < 3000 ) }
```

of the relation emp which denotes the subset of those employees who are older than 40 years and earn less than 3000. Note that IC1' holds as long as the subrelation P1 is empty.



We say that the subrelation P1 *is relevant for* the integrity constraint IC1', because only insert operations into the subrelation P1 of emp may violate the integrity constraint IC1', however write operations on the rest of the relation emp can not violate IC1'. Therefore, IC1' needs not be checked for write operations operating only on the rest of the relation emp. Since IC1' is equivalent to IC1, P1 is also the relevant subrelation of emp for IC1.

The example transaction procedure, called New_Salaries_for_young_employees, computes new salaries for young employees, i.e. for those employees which are at most 30 years old. The transactions executed for this transaction procedure perform write operations on *only a subrelation* of the relation emp. In DBPL, this access restriction is written down in the IMPORT statement of the transaction procedure of a database application program:

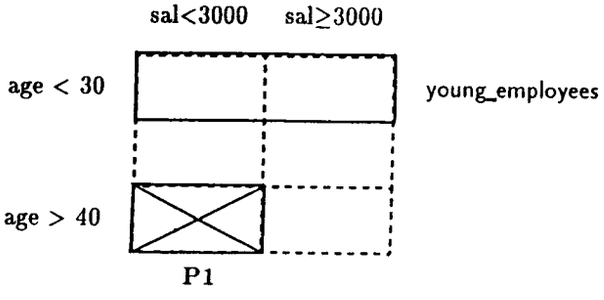
[2] A relation is said to occur only positive (negative) in an integrity constraint, if it occurs only existential (universal) quantified in the prenex normal form of the integrity constraint.

```
TRANSACTION New_Salaries_for_young_employees ;

IMPORT WRITE young_employees IS
      { EACH e IN emp ( e.age ≤ 30 ) }

BEGIN
      FOR EACH ye IN young_employees DO
      ye.sal := compute_sal( ye ) ;
END ;
```

The following diagram shows the subrelation young_employees, i.e. that part of the relation emp, which is accessed by an execution of the transaction procedure New_Salaries_for_young_employees.



Additionally, the diagram shows that the subrelation **young_employees** which is accessed by the transaction ***does not overlap*** with the subrelation P1 of emp that is relevant for the integrity constraint IC1'. This can also be seen from the formulas for P1 and for **young_employees**: the intersection of the subrelation **young_employees** and the subrelation P1 which is relevant for IC1' is empty, i.e.

```
    { EACH e IN emp ( e.age ≤ 30 ) }
∩   { EACH e IN emp ( e.age > 40 AND e.sal < 3000 ) }

=   { EACH e IN emp ( e.age ≤ 30 AND e.age > 40
                      AND e.sal < 3000 ) }
=   { } .
```

because the subformula

```
e.age ≤ 30 AND e.age > 40
```

is insatisfiable (independent of the contents of the database). Since the subrelations P1 and young_employees do not overlap, write operations on the subrelation **young_employees** can not change the contents of P1. This is the reason why the integrity constraint can not be violated by the transaction.

Note that the test whether or not the subrelations overlap can be done completely on intensional formulas, i.e. independent of the contents of the database. Therefore, this overlapping test can be done much faster than an integrity check which queries the database (c.f. section 4.3). Furthermore, this overlapping test can be done at compile time, if the subrelations accessed by the transactions are known at compile time, i.e. if the IMPORT statement of the transaction procedure does not contain any parameters (c.f. [SEM88] for a detailed description of transaction IMPORT statemnets).

Of course, if a transaction imports a subrelation of the relation emp that overlaps the subrelation P1 which is relevant for IC1', then this transaction has to check the integrity constraint IC1' or an optimized formula equivalent to this constraint. How this integrity check can be further optimized is already described in the papers cited above, e.g. in [SV87].

In summary, a transaction can only violate the integrity constraint IC1', if it inserts an element into the subrelation which is relevant for IC1'. The IMPORT information which the transaction programmer has to specify in the head of each DBPL transaction procedure provides an additional filter which indicates that certain integrity constraints need not be checked by the transactions of this transaction procedure. This approach may substantially reduce the number of integrity constraints to be checked and thereby reduces the transaction run time.

3 Subrelations relevant for integrity constraints

In this section we generalize the optimization shown for the given example. For this purpose, we show how to compute relevant subrelations for arbitrary integrity constraints given in tuple relational calculus. Note however that the presented results are not restricted to relational databases but can be applied for object-oriented databases based on attribute inheritance too, because both integrity constraints and imported subrelations can be propagated to descendent classes as described in [Böt90a]. Furthermore, we show how to compute from the import statements of arbitrary transactions whether or not the transactions can violate a given integrity constraint. The general case is more complicated than the previous example, because integrity constraints may contain an arbitrary number of existential and universal quantified relations, hence for each of these relations the appropriate subrelations have to be computed.

We explain both steps of the general case using a second example. We consider the integrity constraint

IC2 "Every department employs at least one employee":

```
ALL d IN dept SOME e IN emp ( e.dnr = d.nr )
```

and a transaction procedure T2 accessing departments which employ at least one young employee

```
TRANSACTION
      Modify_Departments_with_young_employees ;

IMPORT WRITE departments_with_young_employees IS
      { EACH d IN dept ( SOME e IN emp
          ( e.age < 30 AND e.dnr = d.nr ) ) } ;

BEGIN
      ...
END ;
```

3.1 The normal form of an integrity constraint

The basic idea of our approach is to transform each integrity constraint into an equivalent normal form and to determine the relevant subrelations for the integrity constraint from its normal form. The goal of the normalization can be regarded as rewriting the given integrity constraint for each relation variable vi which occurs in the constraint and is bound to a database relation Ri The rewriting shall proceed until it can be directly seen from the normal form which part of the database relation Ri has to be read in order to instanciate the relation variable vi such that the formula in normal form can be evaluated.

The normalization algorithm consists of the following steps (the steps are applied to the formula of the integrity constraint from left to right whereever possible):

1. Eliminate universal quantifiers

   As we have seen from the example IC1 is equivalent to IC1'. The general rewrite rule for integrity constraints containing universal quantifiers is:

   ```
   ALL v IN R ( F(v,...) )
   → NOT SOME v IN R ( NOT F(v,...) )
   ```

   where R stands for an arbitrary relation valued expression. This means R may be a database relation or can be any other relation valued expression.

2. Eliminate projections

   ```
   SOME t IN { < projectionlist > R } ( F(t,...) )
   → SOME t IN { R } ( F(t,...) )
   ```

   We assume that attribute names are unique.

3. Eliminate unions

   ```
   SOME v IN { R1 ∪ ... ∪ Rn } ( F(v,...) )
   →
   SOME v IN { R1 } ( F(v,...) )    OR
   ...                              OR
   SOME v IN { Rn } ( F(v,...) )
   ```

4. Eliminate joins and carthesian products

   ```
   SOME v IN {  EACH v1 IN R1
                ...
                EACH vn IN Rn
                ( F1(v1,...,vn) ) } F2(v,...)
   →
   SOME v1 IN R1 ... SOME vn IN Rn
   ( F1(v1,...,vn) AND F2(v1,...,vn,...) )
   ```

   Within this transformation, the variable v has to be replaced in every occurence of v.*attribute* in the subformula F2 by that variable vi for which *attribute* is an attribute of Ri. This variable vi is unique, if the attribute names are unique, as we assumed.

A set of similar transformation steps has been suggested by [Klu83] in order to compute the relevant subrelations for queries expressed in a subset of the relational algebra which includes intersection but does not include set difference.

In our example, the normalization algorithm uses rewrite rule 1 in order to transform the integrity constraint IC2 into

IC2' "There is no department which does not employ an employee":

```
NOT SOME d IN dept (
      NOT SOME e IN emp ( e.dnr = d.nr ) ) .
```

The normal form N of an arbitrary integrity constraint does neither contain any universal quantifiers nor projections nor unions nor joins nor carthesian products (i.e. it does not contain any of the symbols ALL, <projectionlist>, ∪ or EACH); the normal form N is only constructed from atomic comparisons using the boolean operators AND, OR and NOT and quantifications of the form SOME vk IN Rk where vk is a relation variable bound to a database relation Rk.

3.2 Determining relevant subrelations

In the following formulas, we use the notation [NOT] in order to indicate that at this place in the formula there may be or may not be a boolean operator NOT.

After normalization, we determine for each relation variable vi which occurs in an arbitrary formula N given in normal form and which is bound to a database relation Ri: Which subrelation of Ri is relevant for the instanciation of vi in this formula, i.e. from which subrelation of Ri have the instanciations of vi to be taken in order to evaluate the formula N? Note that this step is done separately for each relation variable occurring in the normal form N of the given integrity constraint.

Depending on the relation variable vi which is bound by a quantification [NOT] SOME vi IN Ri to the database relation Ri, we rewrite the formula N in order to get an equivalent formula Ni : We move the boolean operators AND and OR as far as possible to the 'right end' of the formula and move the negated or non-negated quantifications

[NOT] SOME vk IN Rk

as far as possible to the 'left'. The rewrite rules applied here are the same as the rules for transforming a formula into an equivalent prenex normal form. By applying these rules, we get a formula of the kind

Ni [NOT] SOME v1 IN R1 ... [NOT] SOME vi IN Ri
[NOT] SOME vi+1 IN Ri+1 ... [NOT] SOME vn IN
Rn [NOT] (M(v1,...,vn))

Sometimes it is possible to move the quantification [NOT] SOME vi IN Ri which binds vi to Ri further to the left of the formula without changing the truth value of the formula. More general: If it is possible to rewrite the formula N into several equivalent formulas Ni with different prefixes, then, for the following, we choose a formula Ni in which the quantification

[NOT] SOME vi IN Ri

occurs at the leftmost point in the prefix

[NOT] SOME v1 IN R1 ... [NOT] SOME vn IN Rn [NOT]

of the prenex normal form.

Note that Ni is equivalent to N and therefore also equivalent to the formula of the given integrity constraint.

In order to keep the following simple, we use the term (F(v1,...,vi)) as a short notation for the rest which occurs after Ri in the formula Ni, i.e., F(v1,...,vi) is a short notation for

[NOT] SOME vi+1 IN Ri+1 ... [NOT] SOME vn IN Rn
[NOT] (M(v1,...,vn))

What follows determines for the special quantification [NOT] SOME vi IN Ri in the formula Ni which subrelation of Ri has to be read in order to instanciate vi such that the equivalent formula Ni can be evaluated. Note that only this subrelation of Ri is relevant for the integrity check, i.e. it is not necessary to read the complete relation Ri. In other words, it is sufficient to read that subrelation of Ri which may satisfy the subformula

... SOME vi IN Ri (F(v1,...,v_{i-1},vi)) ,

whatever the values of v1,...,v_{i-1} may be. Whenever the evaluation of this formula comes to the point where the variable vi has to be instanciated, then the variables v1,...,v_{i-1} have already been instanciated. Therefore, it is sufficient to read the following subrelation of Ri:

PRi { EACH vi IN Ri (SOME v1 IN R1 ...
SOME v_{i-1} IN R_{i-1} (F(v1,...,vi))) } .

This means that only the subrelation PRi of the relation Ri is relevant for the given integrity constraint: only this subrelation has to be protected from write operations.

In order to compute all relevant subrelations for a given integrity constraint, this step is done for every relation variable vi occurring in the equivalent normalized integrity constraint N. If the formula N contains different relation variables $v_{i1},...,v_{ik}$ which are bound to the same database relation Ri, then the relevant subrelation of Ri is the union of the subrelations which are sufficient to instanciate the relation variables $v_{i1},...,v_{ik}$.

Let us return to our example of the integrity constraint IC2' and compute that subrelation of the relation dept which is relevant for IC2'. The subrelation of the relation dept which is relevant for IC2' is

P2 { EACH d IN dept (NOT SOME e IN emp
(e.dnr = d.nr)) } .

A similar computation yields the subrelation of the relation emp which is relevant for IC2'. This subrelation is
{ EACH e IN emp (SOME d IN dept (e.dnr = d.nr)) }

3.3 The overlap test

Now, we can compute whether or not the transactions of transaction procedure T2 can violate the integrity

constraint IC2'. Transactions of T2 can access all departments which employ a young employee

```
{ EACH d IN dept SOME e IN emp
        ( e.age < 30 AND e.dnr = d.nr ) } .
```

The approach is to check whether or not the subrelation P2 of the relation dept which is relevant for IC2' and the subrelation accessed by transactions of T2 overlap, i.e. whether or not the intersection of both subrelations is empty in every database state. Therefore, the input to this overlap test is:

```
    { EACH d IN dept ( NOT SOME e IN emp
      ( e.dnr = d.nr ) ) }
∩   { EACH d IN dept ( SOME e IN emp
      ( e.age < 30 AND e.dnr = d.nr ) ) }
=   { } ?
```

The theorem prover of the DBPL system which is described in section 4.3 finds out very efficiently that the intersection of both subrelations is empty in every database state, because the first subrelation (the one which is relevant for IC2') describes those departments which employ *no* employee, whereas the second subrelation describes a subset of those departments which employee *at least one* employee.

Therefore, it has been proved that no transaction of transaction procedure T2 needs to check integrity constraint IC2. Note that this optimization is not found by other integrity check optimization approaches (e.g. [BDM88], [SV87]). These approaches would suggest to check IC2 or an optimized version of it, because a transaction T2 modifies a part of the relation dept which occurs in IC2.

4 Practical system integration

This section shows how to integrate the presented optimization into a database system and uses the DBPL database system which was developed at the University of Frankfurt as an example. We outline the system integration of compile time optimization as well as the system integration of run time optimization.

Both ways of integrating the optimization into the database system have the following in common. During the compilation of the database schema, the integrity subsystem compiles the integrity constraints and computes for each integrity constraint IC and for each relation R_{IC} occurring in the integrity constraint IC, which subrelation of the relation R_{IC} is relevant (i.e. has to be protected) for preserving the integrity constraint IC.

4.1 System integration of the compile time optimization

During the compilation of a transaction procedure, the DBPL compiler analyses the IMPORT statements used in the transaction procedure. The compiler checks for each subrelation imported in a transaction procedure and for each subrelation which is relevant for an integrity constraint whether or not they overlap (c.f. section 4.3). If the subrelations overlap, then the transaction may or may not violate the integrity constraint, and therefore the transaction has to check the integrity constraint. However, if the subrelations imported by the transaction procedure do not overlap with the subrelations relevant for the integrity constraint, then it is written into the code that the transaction needs not check the integrity constraint, e.g. by excluding the integrity constraint from the list of constraints which have to be checked by this transaction. This optimization becomes effective at run time, because the transaction has to perform fewer integrity checks.

The run time support needed for this compile time optimization is to control that the transaction does not violate the import declarations of its transaction procedure. However, this control is done once per transaction and serves many purposes including not only integrity control, but also correct query evaluation and concurrency control. If a transaction procedure is programmed wrong such that the transaction violates its import restrictions, i.e. the corresponding transaction accesses a part of a relation which the transaction procedure did not import, then the transaction is aborted by the DBPL database management system. In other words, each transaction which commits must follow the import declarations given in its transaction procedure. Therefore, it is correct to use the import declarations of a transaction procedure in order to prove that a transaction can not violate an integrity constraint.

However, if an import statement of a transaction procedure contains parameters which can only be evaluated at run time, then, of course, the overlap test has to be delayed until run time and the compiler produces the appropriate code. This overlap test at run time is still more efficient than integrity checking for set oriented write operations (c.f. section 4.3).

4.2 System integration of runtime optimization

The suggested optimization can be used at runtime, additionally or alternatively to the compile time optimization. The run time optimization does neither need

the explicit declaration of an import statement in the transaction procedures nor a run time check whether or not a transaction obeyes the import statements of its transaction procedure. Instead the subrelations accessed by the write operations of a transaction can be derived from the code of the transaction procedure in order to apply the optimization at run time. For set-oriented write operations, the written subrelation can be directly determined from the granularity of the operation. This can be done completely within the database system at run time. However, for a sequence of tuple oriented write operations embedded in a FOR EACH loop as given in the first example of section 2, it is necessary that the compiler provides additional information to the database system about the subrelation which is accessed by the write operations. In the given example, the compiler has to provide the information that the FOR EACH loop ranges only over the young_employees, i.e. over the subrelation

{ EACH e IN emp (e.age $\leq$ 30) }

of the database relation emp. Note however, that this is exactly the subrelation imported by the transaction.

4.3 The theorem prover

The DBPL database system uses a theorem prover in order to check whether or not two subrelations overlap. Because the theorem proving algorithm is already described in [BJS86], we summarize in the following only some basic properties of the DBPL system theorem prover.

The theorem prover is incomplete but efficient [3], i.e. it terminates in a time $O(n^3)$ with one of two answers:

1. The subrelations do not overlap.
2. The subrelations may or may not overlap.

In the first case, it is proved that the given transaction procedure can not violate the given integrity constraint. In the second case, this has not been proved, i.e. either the subrelations overlap or the theorem prover could not find out in the given time whether or not the subrelations overlap. In this case, the integrity constraint is checked by the transaction in order to be correct.

Note that the theorem prover is constructed in such a way that it can check in nearly all cases occurring in practice whether or not two subrelations overlap. Therefore, it is much better to use this efficient, incomplete theorem prover instead of a complete theorem prover which may run much longer in the worst case in order to prove that an integrity constraint can not be violated be a transaction.

[3] On a MicroVax II, the theorem prover needs a few milliseconds in order to check which integrity constraints can be violated by a transaction and which can not.

5 Summary and Conclusions

We have presented an optimization technique which reduces the number of integrity checks which are needed at run time. The optimization is based on the following ideas. First, the integrity subsystem of the database schema compiler computes those subrelations which are relevant for an integrity constraint using a normalization technique based on rewriting. Second, the database system uses the information which part of a database relation a given transaction procedure wants to access at most. We suggest that this should be specified in IMPORT statements of a transaction procedure. A transaction can violate an integrity constraint, only if a subrelation which is relevant for the integrity constraint overlaps with a subrelation imported by the corresponding transaction procedure. This overlap test can be done by a fast theorem prover which may be incomplete. The overlap test can be done at compile time, if the import statement of the transaction procedure is completely instanciated, i.e. the import statement of the transaction procedure does not contain any parameters which are evaluated at run time. But even if the import statement of the transaction procedure contains parameters which are evaluated at run time, this approach may be usefully applied at run time, since the proof for the overlap test is typically much faster than an integrity check.

Furthermore, if the optimization shall only be used at run time for costly set-oriented write operations, then the subrelations accessed by the set-oriented write operations of a transaction can be determined at run time and need not be stated explicitly in an import statement of a transaction procedure.

We have outlined the rewrite rules which compute for arbitrary integrity constraints written in the tuple relational calculus of DBPL which subrelations are relevant. However, the optimization is not limited to relational database systems and can be easily applied to object-oriented database systems which are based on attribute inheritance. Finally, the presented optimization is compatible with other optimization techniques reducing the query complexity of integrity checks, i.e. both kinds of optimization can be naturally integrated into a single

database system. For these reasons, the proposed filtering method reducing the number of needed integrity checks seems to be an important optimization for database integrity constraints.

Acknowledgement

I would like to thank J.W. Schmidt and the DBPL group who developed the programming language DBPL and the DBPL database system the concepts of which have contributed much to the ideas presented in this paper.

References

[BB82] P.A. Bernstein and B. Blaustein. Fast methods for testing quantified relational calculus assertions. In *ACM SIGMOD International Conference*, 1982.

[BDM88] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proceedings of the 1st International Conference on Extending Database Technology*, pages 488–505, Venice, Italy, 1988.

[BJS86] S. Böttcher, M. Jarke, and J.W. Schmidt. Adaptive predicate managers in database systems. In *Proc. of the 12th International Conference on VLDB*, Kyoto, 1986.

[Böt90a] S. Böttcher. Attribute inheritance implemented on top of a relational database system. In *Proc. 6th International Conference on Data Engineering*, Los Angeles, California, USA, 1990. IEEE.

[Böt90b] S. Böttcher. Improving the concurrency of integrity checks and write operations. In *Proc. 3rd International Conference on Database Theory, ICDT-90*, Paris, Dec. 1990. Springer-Verlag.

[Klu83] A. Klug. Locking expressions for increased database concurrency. *Journal of the ACM*, 30(1):36–54, 1983.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity checking in deductive databases. In *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, Great Britain, 1987.

[Mot89] A. Motro. Using integrity constraints to provide intensional answers to relational queries. In Peter M.G. Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, August 1989.

[MRS84] M. Mall, M. Reimer, and J.W. Schmidt. Data selection, sharing and access control in a relational scenario. In M.L. Brodie, J.L. Myopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*. Springer-Verlag, 1984.

[Nic82] J.M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica*, 18:227–253, 1982.

[PO90] J.A. Pastor and A. Olive. Integrity constraints checking in deductive databases. In *Proceedings of the 2nd International Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, 1990.

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

[SK86] A. Shepherd and L. Kerschberg. Constraint management in expert database systems. In *Proceedings of the 1st International Conference on Expert Database Systems*. Benjamin Cummings, 1986.

[SK88] F. Sadri and R. Kowalski. A theorem proving approach to database integrity. In *Foundations of deductive databases and logic programming*. Morgan Kaufmann, Los Altos, 1988.

[SM90] J.W. Schmidt and F. Matthes. DBPL Language and System Manual. In. document, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1990.

[SS86] T. Sheard and D. Stemple. Automatic verification of database transaction safety. Coins Technical Report 86-30, University of Amherst, 1986.

[SV87] E. Simon and P. Valduriez. Design and analysis of a relational integrity subsystem. Technical Report 015-87, MCC, 1987.