Adaptive Predicate Managers in Database Systems

Stefan Böttcher, Matthias Jarke and Joachim W. Schmidt

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32, D-6000 Frankfurt/Main 11
West Germany

Abstract

Relational databases use predicates for a large variety of different functions, some leading to database search, others being handled by theorem proving. First we demonstrate that the theorem proving applications have very similar basic requirements for predicate management but differ in their need for efficiency. Second we present dedicated deduction methods for DBMS that employ a concept of "doubtfuls" in order to allow trade-offs between deductive completeness and efficiency. A new algorithm for testing the satisfiability of predicates with universally quantified variables is described and shown to offer advantages over general-purpose theorem provers for many database applications.

1. Introduction

Predicate logic and the relational model of data have been closely related since the original papers by Codd ([Codd70],[Codd72]). Predicates have been used for query languages (tuple and domain relational calculus) as well as for expressing integrity constraints, access rights, guard conditions in distributed databases, and view definitions (deduction rules). Recently, much interest has been directed towards the study of predicatively oriented constraint managers [Shepherd86] and deduction components [Brachman86] in such systems.

When charged with the evaluation of a predicate, a DBMS is faced with a choice: It can either search the database extension or prove the predicate using schema information alone. Proof methods have the advantage of avoiding secondary storage access but the potential drawback of possibly non-terminating proofs. As part of an effort to realize a general predicative approach to database management in the DBPL project at Frankfurt University, we attempted to develop a predicate manager that offers theorem proving capabilities tailored to the requirements of DBMS. To achieve this goal, two propositions must be established. First, a single variety of theorem provers is shown to be sufficient for all DBMS predicate management tasks. This allows the development of a separate predicate management component for an extensible DBMS architecture, similar to the one proposed in [Carey85]. Second, the predicate management tasks need different trade-offs between generality and efficiency depending on their usage frequency and response time requirements.

The paper is divided into two major sections. Section 2 demonstrates, how DBMS predicate management tasks can be solved by a separate predicate manager. In order to allow a clean trade-off between completeness and efficiency, a concept of "doubtfuls" is introduced and shown to apply uniformly to all DBMS predicate management tasks. This section concludes that predicate managers must be both extensible (i.e., allow the later addition of more powerful or more efficient deduction components) and adaptive (i.e., offer a dynamic choice of deduction components depending on the affordable costs).

In section 3, two dedicated predicate testers for relational calculus predicates are presented. Both theorem provers described here have been implemented as part of the DBPL enhanced relational DBMS at Johann Wolfgang Goethe - University and are currently used as a basis for predicate locking, access control, and semantic query simplification. The first one is a slight extension of a fast and relatively simple test for existential conjunctive queries proposed by [Rosenkrantz80], whereas the second one is a new proof procedure which takes into account the quantification of tuple variables, especially universal quantification. In this respect, the results of the present paper complement [Jarke83], where extensional query evaluation procedures for quantified queries were presented. The proposed algorithm is based on the automatic generation of counterexamples. It is reasonably efficient, as long as the number of universal quantifiers in an expression is small. The paper concludes by comparing the dedicated database predicate testers proposed here to general-purpose theorem provers, in particular to those based on resolution [Robinson 65], with respect to the different DBMS application requirements identified in section 2.

2. Roles of a Predicate Manager

This section reviews predicate management requirements of the major components of a relational database system. It focuses on theorem proving applications and does not discuss predicate handling tasks that must be solved by query processing. A concept of "doubtfuls" is then introduced to substitute for incomplete proofs.

Throughout this paper, predicates are represented in the tuple relational calculus of DBPL ([Mall84], [Edelmann84]), a database programming language which extends Pascal/R [Schmidt77] by concepts for modularization, access abstraction, and predicative multiuser transaction handling. For simplicity of exposition, it is assumed that predicates are in disjunctive prenex normal form (*DPNF*) and all relations occurring in the predicate are non empty. The necessary terminology is established by the following example (see [Jarke83] for formal definitions and transformation procedures) :

$$\text{ALL x IN Rel}_1 \ \text{SOME t IN Rel}_2 \ ((x.a_1 < 5) \text{ OR } (x.a_1 = t.a_1) \text{ AND } (t.a_2 \neq 0))$$

is a predicate in DPNF with the prefix

$$\text{ALL x IN Rel}_1 \ \text{SOME t IN Rel}_2$$

and the matrix

$$(x.a_1 < 5) \text{ OR } (x.a_1 = t.a_1) \text{ AND } (t.a_2 \neq 0)$$

containing the conjunctions

$$(x.a_1 < 5) \quad \text{and}$$

$$(x.a_1 = t.a_1) \text{ AND } (t.a_2 \neq 0) .$$

The following is the *antiprenex form* of the same predicate

$$\text{ALL x IN Rel}_1 \ ((x.a_1 < 5) \text{ OR SOME t IN Rel}_2 ((x.a_1 = t.a_1) \text{ AND } (t.a_2 \neq 0)))$$

Intuitively, prenex normal form moves quantifiers as much to the left as possible, whereas antiprenex form moves them as far to the right as possible.

2.1 Applications of a predicate manager

There is a large number of useful roles predicates can play in database systems. The following is a list of applications amenable to proof procedures rather than to database search. It extends a similar list given by [Munz79] :

- access control,
- synchronizing transactions by predicate locking,
- semantic query simplification,
- reusing previous query results,
- data partitioning and querying in distributed databases,
- consistency and redundancy of integrity constraints.

In order to show similarities and differences, we briefly study the role of predicates in each of these functions.

(1) To enforce access controls, the predicate manager must check whether the data read by the user query are a subset of the data for which the user has access rights. This requires the decomposition of the query into a set of query-relevant data sets, each of which accesses a single relation restricted by a predicate [Böttcher85]. For each such data set, it has to be demonstrated that it is contained in the data set defined by the user's access rights:

$$\{ \text{EACH t IN Rel} : P_{query}(t) \} \subseteq$$

$$\{ \text{EACH t IN Rel} : P_{rights}(t) \}$$

Note that this proof-oriented technique differs from the runtime query modification technique proposed by [Stonebraker75].

(2) To synchronize transactions by predicate locking [Eswaran76, Klug83], it must be shown that for any transaction writing on a set of data defined by a predicate P2 over relation Rel, there is no other transaction running concurrently which accesses an intersecting data set defined by P1 over Rel:

$$\{ \text{EACH t IN Rel} : P_1(t) \} \cap$$

$$\{ \text{EACH t IN Rel} : P_2(t) \} = \{ \}$$

(3) A query submitted to the DBMS can be simplified, if some part of the query predicate is either unsatisfiable or redundant (i.e., implied by some other part) [Aho79]. Similarly a query can be simplified, if it contradicts or is implied by an integrity constraint [Chakravarthy86, Jarke86].

(4) The result of a previous query can be reused to evaluate a new query more efficiently if the new query result can be shown to be a subset of the previous result [Finkelstein82]. Note, that this requires the same predicate test as access control. An access path can be seen as a shorthand for a set of previous query results [Jarke85]; thus, the same predicate test can also be used for checking applicability of access paths.

(5) Query processing in a distributed database has to determine database fragments to be searched to answer a given query. A database fragment defined by predicate P1 over relation Rel can be excluded from the search if it is disjoint with the predicate P2 over Rel appearing in the query [Ceri84, Munz79, Ullman82].

(6) An integrity constraint must be consistent with existing constraints. Inconsistency is determined by a test for unsatisfiability of the expression

$$IC_1 \text{ AND } IC_2 \text{ AND} \ldots \text{AND } IC_{new}$$

Note, that this test is only sufficient if performed at database design time; otherwise, an extensional search for database tuples that violate the new constraint will have to be added [Bowen82, Kitakami84].

Conversely, redundant integrity constraints should be removed from a set of existing constraints. Testing redundancy requires for each integrity constraint ICi one test for implication of

$$IC_1 \text{ AND} \ldots \text{AND } IC_{i-1} \text{ AND}$$

$$IC_{i+1} \text{ AND} \ldots \text{AND } IC_n \rightarrow IC_i .$$

Collecting all requirements, a predicate manager has to perform tests of the following kinds:

(1) Does

$$\{ \text{EACH t IN Rel} : P_1(t) \} \cap \{ \text{EACH t IN Rel} : P_2(t) \} = \{ \}$$

hold ? (for synchronizing transactions and for queries in distributed databases)

(2) Does $\{ \text{EACH t IN Rel} : P(t) \} = \{ \}$ hold ? (for query simplification)

(3) Does

$\{ \text{EACH } t \text{ IN Rel} : P_1(t) \} \subseteq \{ \text{EACH } t \text{ IN Rel} : P_2(t) \}$

hold ? (for access control and to reuse previous query results)

(4) Is $P_1(t_1,...,t_n) \rightarrow P_2(t_1,...,t_n)$ valid ?
(for rejecting unnecessary integrity constraints and for query simplification)

(5) Is $P(t_1,...,t_n)$ valid ? (for query simplification)

(6) Is $P(t_1,...,t_n)$ unsatisfiable ? (for consistency test of integrity constraints and for query simplification)

All these required tests can be standardized into the same theorem proving task by the following reduction steps.

(1) To reduce requirement 1 to a test of kind 2, let
$P(t) := P_1(t)$ AND $P_2(t)$.
Then we get

$\{ \text{EACH } t \text{ IN Rel} : P_1(t) \} \cap \{ \text{EACH } t \text{ IN Rel} : P_2(t) \} =$

$\{ \text{EACH } t \text{ IN Rel} : P(t) \}$.

(2) Requirement 2 can be reduced to a test of kind 6 :

$\{ \text{EACH } t \text{ IN Rel} : P(t) \} = \{ \}$,

iff

SOME t IN Rel ($P(t)$) is unsatisfiable .

(3) To reduce 3 to 6, let $P(t) := P_1(t)$ AND NOT $P_2(t)$.
Then

$\{ \text{EACH } t \text{ IN Rel} : P_1(t) \} \subseteq \{ \text{EACH } t \text{ IN Rel} : P_2(t) \}$,

iff

SOME t IN Rel ($P(t)$) is unsatisfiable .

(4) To reduce 4 to 6 let
$P(t_1,...,t_n) := P_1(t_1,...,t_n)$ AND NOT $P_2(t_1,...,t_n)$.

Then

$P_1(t_1,...,t_n) \rightarrow P_2(t_1,...,t_n)$

iff

$P(t_1,...,t_n)$ is unsatisfiable .

(5) $P(t_1,...,t_n)$ is valid, iff
NOT $P(t_1,...,t_n)$ is unsatisfiable.

2.2 Incomplete Provers

Supporting database system tasks by a predicate manager requires a sufficiently fast rather than a complete theorem prover. For example, the time for optimizing a query should not exceed the savings through the optimization. Therefore, instead of performing an exhaustive search, the predicate manager may finish with the output *doubtful* : In the time given the predicate manager was unable to determine whether a predicate was satisfiable or not. In such cases, the predicate manager will assume that the predicate is satisfiable.

In order to show that a single predicate management component is capable of dealing with all the above applications, we have to demonstrate that the interpretation of "doubtfuls" as "satisfiables" always yields reasonable decisions. If this were not the case, a more complex theorem-prover for double-sided correctness would be required, that would generate three-valued output (satisfiable, non-satisfiable, doubtful). Fortunately, the following discussion shows that such a theorem prover is not needed for the above applications :

(1) To guarantee data security access is granted only if provably permitted. If in doubt, a predicate manager has to reject a query. Confronted with the rejection a user could then split his query into a sequence of simpler ones and possibly get an answer.

(2) Consider synchronizing transactions. In the doubtful case transactions must be serialized. The same is required, if the predicate manager finds out, that the predicate defining the data set used in common is satisfiable.

(3) In the doubtful case no query simplification should be performed. Recall that in the case of redundancy, this is tested by evaluating the implication $P \rightarrow Q$. If it does not hold, this corresponds to the fact that P AND NOT Q is satisfiable.

(4) Consider reusing previous query results. The case, that a predicate is satisfiable indicates that some error could occur when an existing query result (or access path) is reused; the doubtful case requires the same treatment.

(5) In doubt, a data collection in a distributed database has to be searched for tuples matching a query. Exactly the same has to be done, if the predicate manager finds out, that the intersection of query relevant information and the data collection is satisfiable.

(6) As long as a set of integrity constraints cannot be proved to be unsatisfiable, we assume them to be consistent and treat them as satisfiable. Similarly, as long as a predicate manager cannot prove an integrity constraint redundant, it has to be retained in the database schema - no matter, whether this proof failed because the integrity constraint is non-redundant or because of doubt.

2.3 The Need for Adaptive Predicate Managers

From the above discussion it might seem that a single theorem prover with doubtfuls would be sufficient for any of the above predicate management tasks in database systems. However, the tasks differ in the typical complexity of predicates to be tested as well as in their performance requirements. For efficiency reasons, any theorem prover is complete only for a certain class of predicates and allows more and more doubtfuls as predicate complexity grows beyond this class. (This assumes that predicates of greater complexity are simplified before submission to the theorem prover.) Furthermore, performance (especially worst-case performance) degrades as the class covered by a predicate tester is expanded.

One criterion distinguishing the applications is the precision required for predicate tests. Query optimization, for example, just requires unnecessary query processor work if too many doubtfuls occur. Preventing a user from access has to be handled much more carefully, since a high degree of doubtfuls may deny rightful requests.

The performance requirements also differ between rare applications, such as testing consistency and redundancy of integrity constarints, more frequent compile time applications such as query simplification using integrity constarints, and very frequent runtime applications.

The last criterion is the complexity of predicates, depending on the expressive power of language constructs offered to, and used by, the user. In particular, operations involving a large number of predicates (e.g., testing redundancy of integrity constarints) lead to very complex satisfiability tests; fortunately, those applications -- as observed above -- also allow for more testing time.

Therefore, an adaptive database system should offer a choice of predicate testers, suitable for predicates of different complexity and adapted to the needs of the specific data definition and manipulation languages of the system. The extensible database system concept allows the subsequent addition of additional testers, as new interfaces are added to the system. Where performance requirements are stringent, predicates can be simplified and submitted to a tester of lower complexity.

In the DBPL system [Eckhardt85], the relational calculus orientation of the languages requires two main levels of testing. So-called matrixtests only test the satisfiability of the matrix of a relational calculus expression. It is easily shown [Rosenkrantz80, Chandra82] that matrix tests are sufficient for the frequent case of predicates which are conjunctive and do not contain universally quantified variables in their prefix. (Incidentally, this is also the class of SQL queries without set operations and negation.)

However, a special property of the DBPL system is that it also attempts to support quantified predicates (queries as well as constraints) efficiently [Jarke83]. In particular, many integrity constraints (key constraints and referential integrity, among others) rely on the use of universally quantified variables. For example a key constraint on relation Rel can be expressed as :

$$\text{ALL } t_1 \text{ IN Rel ALL } t_2 \text{ IN Rel} \quad ((t_1.\text{key} = t_2.\text{key}) \rightarrow (t_1 = t_2))$$

and a referential integrity constraint between relation Rel_1 and Rel_2 is expressed as :

$$\text{ALL } t_1 \text{ IN Rel}_1 \text{ SOME } t_2 \text{ IN Rel}_2 \quad (t_1.\text{foreignkey} = t_2.\text{key})$$

Consequently, the DBPL predicate manager has been augmented by a second tester which uses a "counterexample" strategy to test satisfiability of quantified expressions. If there are not too many universal quantifiers, the efficiency of this tester is comparable to that of the matrixtest, although its worst-case complexity is exponential in the maximum number of universally quantified variables over a particular relation. The tester has also proven reasonably efficient (taking .5 to a few seconds on a VAX-750) for relatively small sets of integrity constraints to be checked for consistency or redundancy. For very complex theorem-proving tasks (e.g., consistency of large rule bases) it may be necessary to add a third theorem prover, for example, based on the resolution principle.

In the following section, both DBPL predicate testers are described and compared to general-purpose theorem provers from AI with respect to their usefulness in database applications.

3. Theorem Provers in the DBPL Predicate Manager

3.1 Matrixtests

A test examining the satisfiability of a matrix of a predicate given in DPNF is called ***matrixtest***. Each tuple variable of the matrix has to be SOME quantified (called an ***existential variable***). The matrixtest is based on the following principle:

> A matrix in disjunctive normal form is satisfiable, iff at least one of its conjunctions is satisfiable.

The satisfiability of predicates of this important subclass is decidable [Ackermann54]. However, if each of the comparison operators $\neq$, $=$, $\geq$, $\leq$, $>$ and $<$ is allowed, then already testing the satisfiability of a conjunction becomes NP-hard [Hunt79]. To manage problems of this complexity Garey and Johnson [Garey79] propose to design a partial solution solving a class of frequent cases in polynomial time.

Rosenkrantz and Hunt [Rosenkrantz80] propose an algorithm testing in polynomial time the satisfiability of a conjunction containing no $\neq$ comparison operators.
To reduce the number of doubtful cases we extend this algorithm by testing most of the practical relevant conjunctions containing $\neq$ comparison operators.

The algorithm proposed by [Rosenkrantz80] *aligns* all comparisons using the following rules :

$a = b$ --> $(a \leq b)$ AND $(b \leq a)$

$c < d$ --> $c \leq d + -1$

$e \geq f$ --> $f \leq e$

$g > h$ --> $h \leq g + -1$

$x \leq$ constant --> $x \leq 0$ + constant

constant $\leq x$ + offset -->
$0 \leq x$ + (offset - constant)

For each aligned conjunction a graph is constructed : The constant 0 and each pair t.ai occurring in a comparison correspond to nodes and each aligned comparison corresponds to a directed edge valued by the offset of the aligned comparison.

If a graph contains a cycle of edges with a negative sum of values, the conjunction is unsatisfiable. Searching for cycles by drawing transitive edges has a complexity of O((number of comparisons)^3).

To include $\neq$ comparisons into the test, the predicate manager at first sorts the comparisons of each conjunction such that $\neq$ comparisons are processed last. Only if the graph for a conjunction contains no negative valued edge from one node to itself, the following additional test is performed :

> A comparison $a \neq b$ is incompatible with the graph, and the conjunction tested is unsatisfiable, if the graph contains paths from a to b and from b to a with value 0.
> Similarly, a comparison $c \neq$ constant is incompatible with the graph, if the graph contains edges $c \leq 0$ + constant and $0 \leq c$ - constant.

Otherwise, the conjunction is assumed to be satisfiable.

This assumption can be erroneous, e.g., for the conjunction "are there three different integer numbers between 10 and 13" :

> 10<a AND a<13 AND 10<b AND b<13 AND 10<c AND c<13 AND a≠b AND a≠c AND b≠c (a,b,c:INTEGER).

Such predicates have little practical relevance; even if they occur, searching the database may be faster than an extensive predicate test.

In connection with semantic query simplification, a similar algorithm was implemented as part of the Prolog database controller described in [Jarke86]. That algorithm also includes tableau techniques [Aho79] which handle some special cases more efficiently than the general matrix test. In the DBPL implementation the matrixtest has been extended to other attribute types such as REAL and TEXT, possibly in combination [Böttcher85].

3.2 Testing arbitrary quantified relational calculus expressions

As stated earlier, most integrity constraints and many other predicates in DBPL contain quantified variables. Therefore this subsection extends the matrixtest to a general predicate test. However, in contrast to general-purpose theorem provers, this test is tailored to the needs of database systems in that it is very fast for predicates with few universally quantified tuple variables over the same relation, which constitute the majority of database predicates to be tested.

For the frequent case that the predicate contains only existential variables the matrixtest is sufficient. Hence we consider tests for predicates containing ALL quantified tuple variables (*universal variables*). The key idea is, that such a predicate is unsatisfiable if by substituting constants for the universal variables we find a counterexample. We then analyze various ways to construct counterexamples. Finally, we develop an algorithm which systematically constructs potential counterexamples. Combining this algorithm with the matrixtest yields a test for arbitrary predicates. This test is then illustrated by a comprehensive example.

3.2.1 The theorem proving principle

To show the unsatisfiability of a predicate P, the tester tries to construct a counterexample by substituting some constant ci for each universal variable xi of P. This yields a predicate P' which is a logical consequence of P, i.e. P → P'. P' is testable by the matrixtest, since it has no universal variables. If the matrixtest shows P' to be unsatisfiable, i.e. P' → FALSE, we have

$$P \rightarrow P' \rightarrow \text{FALSE},$$

hence P is unsatisfiable.

As an example, consider the following predicate P_1 :

> SOME t IN R ALL x IN R
> (($t.a_1 \neq x.a_1$) AND ($x.a_2 < 20$))

The substitution x:=t applied to predicate P1 yields Predicate P1' :

> SOME t IN R
> (($t.a_1 \neq t.a_1$) AND ($t.a_2 < 20$))

The matrixtest proves that predicate P1' is unsatisfiable, hence predicate P1 is unsatisfiable. Intuitively, since there is **one** (representative) tuple t in R for which the matrix is unsatisfiable, it cannot be satisfiable for **all** elements of R. Thus, we have constructed the desired counterexample.

3.2.2 Compatible tuple substitutions

In general, there are several essentially different ways to construct potential counterexamples, using different variable substitutions. However, not all of these substitutions actually preserve correctness of the predicate test. In this subsection, we first show how to enumerate potential counterexamples, and then how to eliminate illegal ones.

To enumerate potential counterexamples, we first introduce two kinds of *precedences* between tuple variables : scope precedences (<) given by the predicate structure, and substitution precedences (<--) for each substitution made by the predicate tester.

Two tuple variables t_1, t_2 of a predicate P are in *scope precedence* ($t_1<t_2$), if in every equivalent predicate AP in antiprenex form, t_2 is defined in the scope of t_1. For example, predicate P_1 has the scope precedence t < x, since one cannot change the sequence of t and x in the prefix without changing the meaning of the predicate.

Substitution precedences are caused by so-called tuple substitutions: A tuple substitution of a predicate is a function of the form

$$\{ t_1 \leftarrow x_1 , \ldots, t_n \leftarrow x_n \} ,$$

which maps each universal variable x_i either to an existential variable of the same relation or to a sort constant cR of the range relation R of x_i. For example the predicate P1 has two possible tuple substitutions

$$T_1 = \{ t \leftarrow x \} \quad \text{and} \quad T_2 = \{ cR \leftarrow x \} .$$

Remarks:

(1) Substitution with cR, an arbitrary element of the relation R, leads to unification of the remaining universal variables.

(2) There cannot be two different sort constants of the same relation because there are predicates which are satisfiable only by relations with a single element.

(3) This restriction to one sort constant cR per relation R does not lead to loss of completeness [Böttcher85]. A similar restriction of substitutions is the Herbrand Universe [Chang73].

Tuple substitutions may be incompatible with the scope precedences of the tuple variables i.e., they do not preserve a logical implication from P to the simplified predicate P'. For example, compare the following predicate with P_1 :

Predicate P_2 :

> ALL x IN R SOME t IN R
> (($t.a_1 \neq x.a_1$) AND ($x.a_2 < 20$))

The substitution x:=t (t<--x) is allowed for P_1 but not for P_2 because in P_2, t is not known at the moment x is fixed by substitution. Note that, the scope rules for P_2 determine a precedence $x < t$ which, together with the precedence required by the substitution t <-- x (t before x), would require a cyclic precedence x < t <-- x .

To generalize, we define a ***prefix graph***. The tuple variables and the sort constants are the nodes, and the scope and substitution precedences are the edges of this graph. If the prefix graph is acyclic, we call the tuple substitution ***compatible*** with the scope precedence. Only compatible tuple substitutions can be used to construct counterexamples. For example, we cannot construct a counterexample for predicate P_2 although one existed for P1. Consequently, P_1 is unsatisfiable but P_2 is assumed to be satisfiable.

Frequently, the number of possible substitutions is reduced substantially by the compatibility rule. For example, predicate P_3

ALL x IN R   ALL y IN R   SOME t IN R   (...)

determines the scope precedences { x < t , y < t } . The tuple substitutions T_i of predicate P_3 are :

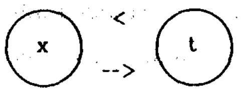
T_1 = { t <-- x , t <-- y }

T_2 = { t <-- x , cR <-- y }

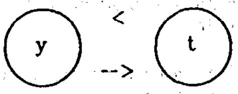
T_3 = { cR <-- x , t <-- y }

T_4 = { cR <-- x , cR <-- y }

Together with the scope precedences the tuple substitutions T_1 and T_2 induce the following cycle in the prefix graph :



The tuple substitutions T_1 and T_3 induce the cycle :



Hence only tuple substitution T_4 is compatible.

If a predicate obtained by a compatible tuple substitution is unsatisfiable, the original predicate is unsatisfiable, because a feasible counterexample has been constructed. Together with the matrixtest, this reasoning yields the ***theorem***:

> ***A predicate in DPNF is unsatisfiable, if there is a cycle free prefix graph such that (after substitution) every conjunction graph contains a negative valued cycle.***

Searching for cycles in the prefix graph can be performed by the same algorithm, as used for conjunctions in the matrixtest, if we treat each precedence (< or <--) as a comparison containing the < operator.

Using this result, we obtain the following ***algorithm*** for testing an arbitrary predicate containing universal variables:

```
FOR EACH tuple substitution DO
   IF  the tuple substitution is compatible  THEN
      IF  the matrixtest yields that the substituted
          predicate is unsatisfiable  THEN
          RETURN( the predicate tested is unsatisfiable )
      END ;
   END ;
END ;

RETURN( the predicate tested is
            assumed to be satisfiable ) ;
```

To illustrate the interaction of the concepts introduced in this section, we now present a comprehensive example:

In a database containing relations employees and papers the following integrity constraints are submitted to a consistency test :

I_1: There exists a paper with the author number 100

SOME p_1 IN PAP (p_1.author = 100)

I_2: All authors are employees (referential integrity)

ALL p_2 IN PAP SOME e_2 IN EMP (p_2.author = e_2.enr)

I_3: For every employee there exists a paper, of which he is not an author

ALL e_3 IN EMP SOME p_3 IN PAP (p_3.author $\neq$ e_3.enr)

I_4: All employees have an employee number less than 50

ALL e_4 IN EMP (e_4.enr < 50) .

These integrity constraints determine the scope precedences

$p_2 < e_2$ and

$e_3 < p_3$

The conjunction of the integrity constraints leads to the DPNF predicate :

SOME p_1 IN PAP   ALL p_2 IN PAP   SOME e_2 IN EMP
ALL e_3 IN EMP   SOME p_3 IN PAP   ALL e_4 IN EMP
((p_1.author = 100)   AND
(p_2.author = e_2.enr) AND
(p_3.author $\neq$ e_3.enr) AND
(e_4.enr < 50))

Among others, the following tuple substitutions are considered :

{ p_3 <-- p_2 , e_2 <-- e_3 , cEMP <-- e_4 }

{ cEMP <-- e_3 , cPAP <-- p_2 , cEMP <-- e_4 }

{ p_1 <-- p_2 , cEMP <-- e_3 , e_2 <-- e_4 }

The first tuple substitution yields the substitution precedences p_3 <-- p_2 and e_2 <-- e_3. Together with the scope precedences this leads to the following cycle in the prefix graph :

$$p_3 \leftarrow p_2 < e_2 \leftarrow e_3 < p_3 .$$

Thus, the first tuple substitution is incompatible : Although this substitution leads to an unsatisfiable matrix, this is not sufficient to prove the integrity constraints to be inconsistent.

The second tuple substitution is compatible but yields a predicate with a satisfiable matrix :

SOME p_1 IN PAP SOME e_2 IN EMP SOME p_3 IN PAP
((p_1.author = 100) AND
(cPAP.author = e_2.enr) AND
(p_3.author ≠ cEMP.enr) AND
(cEMP.enr < 50))

The third tuple substitution is compatible and yields a predicate with an unsatisfiable matrix :

SOME p_1 IN PAP SOME e_2 IN EMP SOME p_3 IN PAP
((p_1.author = 100) AND
(p_1.author = e_2.enr) AND
(p_3.author ≠ cEMP.enr) AND
(e_2.enr < 50))

The matrix is unsatisfiable because it contains

$$100 = p_1.\text{author} = e_2.\text{enr} < 50 .$$

Constructing this counterexample is sufficient to prove that the integrity constraints are inconsistent.

3.2.3 Evaluation Costs

The number of tuple substitutions | T | of a predicate and thus the maximum number of matrixtests depends primarily on the number of universal and existential variables per relation :

$$| T | = \prod_{\text{Rel1}}^{\text{Reln}} (\text{SOMEr}+1)^{\text{ALLr}}$$

SOMEr = number of existential variables bound to relation r,

ALLr = number of universal variables bound to relation r.

For the above example there are twelve tuple substitutions. This number can be reduced by first combining all ICs beginning with a universal quantifier over the same relation : in the example there are only six tuple substitutions if the the third and the forth integrity constraint are combined into one by identifying e3 with e4. In compile and runtime applications, there will be rarely more than one variable for a given range relation. Thus, although the worst case complexity of this algorithm is exponential, its costs are typically quite acceptable.

The number of compatible tuple substitutions is even smaller. Using a backtracking algorithm to produce compatible tuple substitutions [Böttcher86] further decreases the number of produced tuple substitutions.

The time spent on constructing the prefix graph is at most O((number of quantifiers + number of relations $)^3$). Hence the bottleneck of runtime applications is usually the matrixtest, consuming a time of at most O((number of comparisons $)^3$).

3.3 Comparison with general purpose theorem provers

The goal of well known theorem provers ([Andrews81], [Bibel81], [Bläsius81], [Hsiang83], [Kowalski75]) is the analysis of arbitrary first order predicates. In general these theorem provers offer the problem of termination : Since the first order predicate calculus is undecidable [Kleene71], for every complete theorem proving algorithm there are some predicates for which it doesn't terminate. Hence it is impossible to calculate, after how many steps at most the algorithm will terminate. In contrast most database applications must estimate execution time in advance. If we wish to limit the number of deduction steps the question arises : Which limitation of which deduction steps is adequate ?

Most special purpose theorem provers focus on reasoning with equality alone (e.g. Prolog). A ≤-resolution principle proposed by Bledsoe and Hines [Bledsoe80] is based upon techniques called variable elimination, splitting and chaining. However, the key idea, restriction of chaining, is not applicable to database predicates, hence the search space remains unrestricted and a long testing must be expected.

Considering most of the applications of the predicate manager, another weakness of most of the well known theorem provers is, that they do not integrate comparisons of variables with natural or real numbers, and texts values. Treating these comparisons as axioms is much too expensive.

To demonstrate the usefulness of different predicate managers we map the application requirements to the following three kinds of predicates to be tested :

(1) Predicates containing no universal variables. For these applications the matrixtest is sufficient.

(2) Predicates containing few universal variables. For these applications the extended matrixtest is sufficient.

(3) Predicates containing many universal variables. For these applications more powerful theorem provers are required, e.g. general purpose theorem provers.

Throughout the discussion, it is assumed (realistically, we believe) that user queries will contain no or few universal variables per relation.

(1) Access rights are usually expressed as simple selections (no quantifier) or at most by referential constraints (one quantifier). Thus, predicates of type 1 or 2 have to be proven.

(2) In principle, precision predicate lock tests might require several universal quantifiers over the same relation. However, this will happen only if there is heavy concurrent access to the same relation and if this access is not via the same logical access path (in which case the range relation is easily partitioned). Thus, we expect this case to be rare; if it does occur, predicates must be simplified so that tests of type 1 and 2 become applicable. General proof procedures must be considered infeasible for synchronization of short transactions.

(3) Query simplification is only useful if fast proof procedures are available. Predicate testers of type 1 and 2 can be used as a syntactic simplification tool which may be useful only in conjunction with semantic simplification [Jarke86, Chakravarthy86] or in combination with other fast special-purpose provers, such as tableau techniques [Aho79].

(4) Query results will be retained, or access paths will be supported, only if they save expensive operations and correspond to relatively simple queries. Typical examples are single or multiple attribute indexes (no quantifiers) or join indexes (one quantifier). Thus, tests of type 1 or 2 are typically sufficient.

(5) For distributed databases, the same argument as for access rights holds if data partitioning is by predicates at all.

(6) Tests for consistency and redundancy of integrity constraints may involve many quantifiers over the same relations if there are many integrity constraints per relation. Note, however, that universal variables can often be combined, as shown in the example of section 3.2.2. Formalizing this idea [Böttcher86] presents an algorithm that reduces the number of matrixtests by testing only so-called "maximal tuple substitutions". It is also shown that testing only maximal tuple substitutions is equivalent to testing all tuple substitutions. Nevertheless, the question of whether a type 2 tester, one of the AI provers, or another special-purpose tester is preferable for checking consistency and redundancy of integrity constraints, must remain open until further practical experimentation.

In summary, then, it turns out that the two predicate testers presented in this paper are efficient and sufficient for most database applications, with the possible exception of query simplification (where they must be combined with other methods) and consistency / redundancy checks for constraints (where further research is needed to establish the best methods). However the latter task is usually performed only at database design or restructuring time, where efficiency is of less concern.

4. Conclusion

There are two major conclusions to be drawn from this research. First, it was demonstrated that the same philosophy of predicate testing with doubtfuls applies to all of the major theorem proving applications in relational DBMS. Second, it was shown that specialized theorem provers taking explicitly into account the trade-off between number of doubtfuls (completeness) and execution speed may offer advantages over general purpose methods for most time-critical DBMS applications.

The matrixtest by [Rosenkrantz80], itself based on searching cycles, has been extended by a prefixtest for general predicates with universally quantified variables, also searching cycles. This extended matrixtest proves a predicate to be unsatisfiable, if it has a cycle free prefix graph such that every conjunction graph has a negative valued cycle. It is intended to integrate this algorithm with the tableau-oriented semantic query optimizer of [Jarke86] which is also graph based and allows efficient handling of specialized constraint classes such as functional dependencies and referential integrity constraints.

As the cost estimates show, the proposed tester is suitable for the frequently occurring predicates containing few tuple variables per relation. This is confirmed by experience with the implementation.

In further work, we plan to augment the predicate management system by faster and simpler tests (e.g., dropping analysis of inequalities), and by more powerful tests, taking into account, e.g., deduction rules with recursion. In this way, we want to use the extensible database architecture to provide fully adaptive predicate management. The predicate manager will also be used to provide an interface of the DBPL system to a multi-level logic-based design environment for data-intensive applications.

Acknowledgment

This work was supported by the Deutsche Forschungsgemeinschaft under grant no. SCHM450/3-1 (principal investigator: J.W.Schmidt).

References

[Ackermann54]
Ackermann, W.: *Solvable Cases of the Decision Problem.* Amsterdam, North Holland, 1954.

[Aho79]
Aho, A.V., Sagiv, Y., Ullman, J.D.: Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems, 4*, 4. pp. 435-454.

[Andrews81]
Andrews, P.B.: Theorem Proving via General Matings. *JACM 28*, 2, 1981. pp. 193-214.

[Bibel81]
Bibel, W.: On Matrices with Connections. *JACM 28*, 4, 1981. pp. 633-645.

[Bläsius81]
Bläsius, K., Eisinger, N., Siekmann, J., Smolka, G., Herold, A., Walther, C.: The Markgraph Karl Refutation Procedure. *Proceedings 7th IJCAI*, Vancouver, 1981.

[Bledsoe80]
Bledsoe, W.W., Hines, L.M.: Variable Elimination and Chaining in a Resolution-based Prover for Inequalities. *5th Conference on Automated Deduction*, Les Arcs, Juli 1980.

[Böttcher85]
Böttcher, S.: *Ein Testverfahren für Datenbankprädikate.* Report No. 114, Universität Hamburg, Fachbereich Informatik, 1985.

[Böttcher86]
Böttcher, S.: Ein Beweisverfahren für Datenbankprädikate. In Stoyan, H. (Ed.): *GWAI-85, 9th German Workshop on Artificial Intelligence*, Dassel/Solling, September 1985, Informatik Fachberichte 118. Berlin, Springer, 1986. pp.164-175.

[Bowen82]
Bowen, K.A., Kowalski, R.A.: Amalgamating language and metalanguage in logic programming. In Clark, K.L., Taernlund, S.A. (eds.): *Logic Programming.* Academic Press, 1982. pp. 153-172.

[Brachman86]
Brachman, Levesque, H.J.: What Makes a Knowledge Base Knowledgeable - A View of Databases from the Knowledge Level. In [Kerschberg86].

[Carey85]
Carey, M.J., Dewitt, D.J.: Extensible database systems. In Mylopoulos, J., Brodie, M.L. (eds.): *On Knowledge Base Management Systems*. Springer, 1985.

[Ceri84]
Ceri, S., Pelagatti G.: *Distributed Databases: Principles and Systems*. McGraw Hill, 1984.

[Chakravarthy86]
Chakravarthy, U.S., Fishman, D., Minker, J.: Semantic Query Optimization in Expert Database Systems. In [Kerschberg86].

[Chandra82]
Chandra, A.K., Harel, P.: Structure and complexity of relational queries. *Journal of Computing System Sciences*, *25*, 1982, pp. 99-128.

[Chang73]
Chang, C., Lee, R.C.: *Symbolic Logic and Mechanical Theorem Proving*. New York [u.a.], Academic Press, 1973.

[Codd70]
Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. *CACM*, *13*, 6, 1970, pp. 377-387.

[Codd72]
Codd, E.F.: Relational Completeness of Data Base Sublanguages. *Courant Computer Science Symposia 6*, pp. 65-101.

[Eckhardt85]
Eckhardt, H., Edelmann, J., Koch, J., Mall, M., Schmidt, J.W.: *Draft Report on the Database Programming Language DBPL*. Internal Memo, University of Frankfurt, 1985.

[Edelmann84]
Edelmann, J.: *Die Datenbankprogrammiersprache DBPL: Eine Beschreibung ausgewählter Sprachkonstrukte und deren Implementation*. Diploma thesis, Universität Hamburg, Fachbereich Informatik, 1984.

[Eswaran76]
Eswaran, K.P., Gray, J.N., Lorie, R.A. Traiger, I.L.: The Notions of Consistency ad Predicate Locks in a Database System. *CACM*, *19*, 11, 1976, pp. 624-633.

[Finkelstein82]
Finkelstein, S.: Common expression analysis in database applications. *Proceedings ACM-SIGMOD International Conference*, Orlando, June 1982. pp. 127-133.

[Garey79]
Garey, M.R., Johnson, D.S.: *Computers and Intractability*. Bell Telephon Laboratories, 1979.

[Hsiang83]
Hsiang, J., Dershowitz, N.: Rewrite Methods for Clausal and Non-Clausal Theorem Proving. *Automata, Languages and Programming, 10th Colloquium*, Barcelona, July 1983, Berlin [u.a.], Springer 1983.

[Hunt79]
Hunt, H.B., Rosenkrantz, D.J.: The Complexity of Testing Predicate Locks. *Proceedings ACM-SIGMOD International Conference on Management of Data*, May 1979. pp. 127-133.

[Jarke83]
Jarke, M., Koch, J.: Range Nesting: A Fast Method to Evaluate Quantified Queries. *Proceedings ACM-SIGMOD International Conference on Management of Data*, San Jose, June 1983. pp. 198-206.

[Jarke85]
Jarke, M.: Common Subexpression Isolation in Multiple Query Optimization. In Kim, W., Reiner, D.S., Batory, D.S.(Eds.): *Query Processing in Database Systems*, Springer, 1985.

[Jarke86]
Jarke, M.: External semantic query simplification : a graph-based algorithm and its implementation in Prolog. In [Kerschberg86].

[Kerschberg86]
Kerschberg, L. (Ed.): *Expert Database Systems*. Benjamin Cummings, 1986.

[Kitakami84]
Kitakami, H., Kunifuji, S., Miyachi, T., Furukawa, K.: A Methodology for Implementation of a Knowledge Acquisition System. *Procedings International Symposium on Logic Programming*, Atlantic City, NJ, 1984, pp. 131-142.

[Kleene71]
Kleene, S.C.: *Introduction to Metamathematics*. Wolthers-Noordhoff, North Holland, 1971.

[Klug83]
Klug, A.: Locking Expressions for Increased Database Concurrency. *JACM 30*, 1, 1983, pp. 36-54.

[Kowalski75]
Kowalski, R.: A Proof Procedure Using Connection Graphs. *JACM 22*, 4, 1975. pp. 572-595.

[Mall84]
Mall, M., Reimer, M., Schmidt, J.W.: Data Selection, Sharing and Access Control in a Relational Scenario. In Brodie, M.L., Mylopoulos, J.L., Schmidt, J.W. (Eds.): *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*. Berlin [u.a.], Springer, 1984.

[Munz79]
Munz, R., Schneider, H.J., Steyer, F.: Application of Sub-Predicate Tests in Database Systems. *Proc. of 5th International Conference on Very Large Data Bases*, Rio de Janeiro, October 1979.

[Rosenkrantz80]
Rosenkrantz, D.J., Hunt, H.B.: Processing Conjunctive Predicates and Queries. *Proc. of 6th International Conference on Very Large Data Bases*, Montreal, October 1980. pp.64-74.

[Schmidt77]
Schmidt, J.W.: Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems 2*, 3, 1977, pp. 247-261.

[Shepherd86]
Shepherd, A., Kerschberg, L.: Constraint Management in Expert Database Systems. In [Kerschberg86].

[Stonebraker75]
Stonebraker, M.: Implementation of Integrity Constraints and Views by Query Modification. In *Proc. ACM-SIGMOD Conference*, San Jose, pp. 65-77.

[Ullman82]
Ullman, J.D.: *Principles of Database Systems*. 2nd Ed., Computer Science Press, Rockville, 1982.