1988

DEDUCTIVE EXTENSION OF A RELATIONAL DATABASE SYSTEM

Nicolai PreiB
Institut fi Angewandte Informatik und Formale Beschreibungsverfahren Universitiit Karlsruhe*

DEDUCTIVE EXTENSION OF A RELATIONAL DATABASE SYSTEM

Nicolai Preiß
Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
Universität Karlsruhe

ABSTRACT

Logic based knowledge processing systems such as PROLOG based expert systems have shown obvious drawbacks in performing conventional database tasks. Knowledge processing by deduction on a large set of given facts can be better performed by a *deductive database system* based on Horn logic and relational database theory.

A concept is presented to extend an existing relational database system to make feasible the deduction of intensional data from a given extensional database. The deductive extension provides an *extended view mechanism* and the *integration of integrity constraints* and leads to an *enhanced query mechanism*. Thus, the conventional database becomes more expressive, shows a higher degree of consistency, and is evaluated more efficiently.

1. INTRODUCTION

Knowledge based systems such as expert systems (XPSs) or rule based systems use logical statements, generally first order formulas, to derive conclusions or at least to support the derivation process. A very popular approach to building such systems is the logic programming language PROLOG. But PROLOG based XPSs showed obvious drawbacks in performing database tasks and so *expert database systems* emerged as a combination of a PROLOG based XPS and a relational database system (DBS). The coupling of PROLOG with a conventional DBS represents a first, simple approach whereas a more advanced solution is feasible coupling PROLOG with a *deductive DBS* (Preiß 1988).

A *deductive DBS* consists of a set of relations and an inference mechanism to derive new relations. This is motivated by the most interesting fact that some inferences of XPSs may be substituted by queries of DBSs which generally provide the results much faster. Therefore, it is recommended to move as much as possible from the XPS down to the DBS (Smith 1986) requiring a deductive database component for an efficient solution.

This paper focuses on the development of such a deductive DBS realized as an extension of a conventional DBS. The extensions are based on the fact that logic provides a basis for relational databases, especially for expressing queries and for the definition of views and integrity constraints (Brodie and Jarke 1986). Therefore, our extensions are aiming at an *extended view mechanism* and the *integration of integrity constraints* (ICs) resulting in an *enhanced query mechanism*.

Contrary to other approaches in the field of deductive databases, no separate component with logic programming techniques is applied to provide deduction but the *DBS itself*. Moreover, unlike Postgres (Stonebraker and Rowe 1986), for example, we preserve the conventional relational database environment, especially the ease of use.

To provide deduction we will extend the standard database language SQL, specifically the data dictionary (DD), in a simple and "natural" way. The resulting extended view management including recursion and semantic ICs makes conventional databases more *expressive* and less space consuming (derivation of virtual relations). It enhances the *degree of consistency* (specification of semantic restrictions) and *efficiency* (semantic query optimization). Additionally, the independent deductive DBS may be used to provide an efficient database interface for knowledge processing systems based on logical rules. As far as we know, no other system provides such a *comprehensive deductive DBS* while preserving a uniform relational database environment.

The paper is organized as follows: section 2 shows the formal background of deduction in DBSs. Our concept of a *deductive relational database system* is presented in the section 3. We are concerned with derivation rules, i.e., extended view definitions, and especially with integrity constraints. Finally, a general overview and an illustrative example of the deductive query evaluation are given in section 4. The paper ends with some concluding remarks about our ideas, objectives, and further research.

2. DEDUCTION IN RELATIONAL DATABASE SYSTEMS

In a *deductive database system* new facts may be derived from the existing ones (Gallaire, Minker and Nicolas 1984). We exclude general function symbols as arguments in order to have finite and explicit answers to queries. Also, as often done in the context of databases, to formally represent facts, deductive laws, and integrity constraints, we use formulas in the form of *Horn clauses* which preclude the derivation of positive literals from negative ones:

$$A1(...), A2(...), ..., An(...) \rightarrow B(...) \qquad \{1\}$$

While it is well understood what a *fact* is (conclusion without conditional part), there is no final answer to the database design question, whether to consider a general definite Horn clause as a deductive law (*derivation rule*) or as an *integrity constraint*. However, in most cases the heuristic holds that if the clause is not intended to derive new facts but to restrict existing base or virtual relations then it represents an integrity constraint (see section 3.2).

In detail, the deductive relational database formally consists of (Preiß 1987):

- a restricted Horn language:
  - -- *range restricted* Horn clauses (all variables the right side must appear on the left),
  - -- at most *linear recursive* Horn clauses (at most one Ai is mutually recursive to B),
- a theory with:
  - -- axioms1 (*elementary facts*),
  - -- axioms2 (*derivation rules*),
- a set of *integrity constraints*,
- a metarule: negation as finite failure (which applies to the well-known closed world assumption (CWA) in conventional DBSs).

The facts are often referred to as extensional database (EDB) whereas the derivation rules and integrity constraints are called intensional database (IDB).

This approach of a deductive database enables us to incorporate *more real world knowledge* into the relational database, to enhance the expressiveness of the database language, and to treat the query evaluation and integrity preserving in a uniform manner. The deductive database is more complete (linear recursion) and less space consuming than conventional databases. Furthermore, it can be used to support PROLOG-based XPSs efficiently (Preiß 1988).

3. DEDUCTIVE RELATIONAL DATABASE SYSTEM

In dealing with Personal Computer applications, the relational database system Datenbank-Pascal, now called INOVIS-X86, was developed at our institute (Karszt 1984). A three level architecture (external, conceptual, and internal level), a data dictionary, a transaction management (recovery and concurrency in LANs), simple integrity preservation, a programming interface (PASCAL, C, FORTRAN, COBOL), and an SQL interface represent the most important features resulting in a powerful DBS for *conventional database applications* (up to 110 MB of database size and more).

Further developments led to a *deductive extension of the existing DBS* (Preiß 1987). This comprises an extended view mechanism (derivation rules including recursive views and relations derived by database procedures) and an advanced management of integrity constraints. The deductive DBS provides a *simple database language* (SQL), *high functionality* (conventional applications, simple inferences, database support for expert systems), *high performance* (active DD, simple concepts of relational databases), and *simple realization* (few extensions of the existing DBS). Logic is applied as a formal background only, not as a mechanism to provide deduction.

Although our proposals refer to the DBS INOVIS-X86, the concept may be applied to any other DBS that offers a DD, a programming interface, and an SQL interface. Mainly, the *database language* must be extended for the definition of possibly recursive derivation rules (DRs) and integrity constraints (ICs) and for the formulation of corresponding queries. Accordingly, the *data dictionary* must be extended to store the new types of declarations. Finally, a deductive component is added as a front end to the DBMS transforming the deductive query into a sequence of conventional query expressions evaluable by the conventional DBMS. Note that this approach of a deductive extension requires only few modifications on the user level. Thus, the database environment applied in conventional data processing is preserved.

3.1 Derivation Rules

3.1.1 Formulation of Derivation Rules

Logical Expressions

Generally, in the context of databases, a logical formula such as a database rule in PROLOG is considered as a derivation rule representing the definition of a virtual relation or *view*. As described in section 2, in our deductive DBS the logical formulation of such a definition is restricted to definite, range restricted, function free, and at most linear recursive Horn clauses. In addition, there are some pure syntactical restrictions bringing the logical formulas closer to the SQL expressions and enhancing their

readability. Moreover, these restrictions preserve the DBMS from the renaming overhead during query evaluation as it is performed in PROLOG.

In detail, a logical formula as a derivation rule (or integrity constraint) must obey the following syntactical restrictions if used in the database context:

A logical formula consists of

- n-ary *relational predicates* representing base relations and virtual relations including database procedure relations,
- binary *evaluable predicates* (<, >, ≤, ≥, =, <>) representing join and selection conditions and assignments (the '='-predicate is used to assign a constant or an argument of a relational predicate Ai of the formula {1} (see section 2) to an argument of the relational predicate B).

The evaluable predicates must be used to express the join and selection conditions. Join definitions by means of equal arguments in relational predicates, or selection definitions by means of constants as arguments in relational predicates are not allowed. That is, all of the constants appear in evaluable predicates (aggregate and arithmetic functions are considered as special constants) and, therefore, only variables -- i.e., attribute names -- are allowed as arguments of relational predicates (see Example 1).

The naming convention for relational predicates obeys the *unique name assumption* (UNA), that is, two relational predicates with the same name refer to the same relation. If the argument list does not represent the full attribute set, then a projection is specified. But such a projection must not appear as the conclusion of a derivation rule because such a definition contradicts the UNA.

If the same relational predicate is defined more than once, it gets its own attribute list. If the same relational predicate occurs several times within a formula, the occurrences are indexed making feasible the identification of arguments by prevailing predicate names (see Example 1). Finally, every deductive definition must be based on existing definitions and must not define a hybrid relational predicate, i.e., one that belongs to the EDB.

Example 1:

Let the relation "EL_PRO (Mach, In, Out)" contain the elementary production steps on machine "Mach" with input "In" and output "Out". Then, the view "PRODUCT (Mach, In, Out)" containing productions performed in two steps on the same machine is defined in terms of the relation EL_PRO as follows:

```
EL_PRO1 (Mach, In, Out), EL_PRO2 (Mach, In, Out),
- (EL_PRO1.Mach, EL_PRO2Mach), - (EL_PRO1.Out, EL_PRO2.In)
      → PRODUCT (EL_PRO1.Mach, EL_PRO1.In, EL_PRO2.Out).
```

An important issue in deductive databases is the introduction of *recursion* as a means to define and process specific views. This is a very interesting research area in which the *least fixpoint operator* (Aho and Ullman 1979; Bayer 1985; Bancilhon and Ramakrishnan 1986) represents the most popular approach transforming the recursion into a special kind of iteration. However, although this approach provides a very powerful view mechanism, it lacks the possibility to stop the iteration before the least fixpoint is obtained. This is required, for example, if we want to know the productions with a special number of elementary production steps on the same machine.

Therefore, to specify recursive definitions, the approach of the least fixpoint is applied with a slightly modified *implication operator*: the desired number of iteration steps is added as an index to the implication symbol (the default value is one and the least fixpoint iteration is marked by "+" -- see Example 2). We note that every recursive definition requires an initial (EDB or IDB) relation and, in the case of an indirect recursion, a sequence of dependent view definitions.

Example 2:

If we are looking for productions with at most two steps performed on the same machine then the view PRODUCT of example 1 is defined as follows:

```
EL_PRO (Mach, In, Out)
      →+ PRODUCT (Mach, In, Out) .
PRODUCT (Mach, In, Out), EL_PRO (Mach, In, Out)
- (PRODUCT.Mach, EL_PRO.Mach), - (PRODUCT.Out, EL_PRO.In)
      →+ PRODUCT (PRODUCT.Mach, PRODUCT.In, EL_PRO.Out).
```

In PROLOG these rules represent a view that contains all productions possible on one machine. However, if such a fixpoint is desired in our deductive DBS, the implication symbol "→+" must be applied in the second formula.

Besides recursion, another important issue in our deductive DBS is the introduction of a *model manager* that enhances both the expressiveness of view definitions and the integrity preserving capabilities. As mentioned above, relational predicates may be used to represent *database procedure relations*, i.e., special views that are derived by database procedures (PASCAL programs with database constructs). Although this feature allows an enormous flexibility it must be noted that the DBMS has no influence on the processes of the database procedure and thus cannot guarantee the correct derivation.

SQL Expressions

With these issues about the formulation of DRs in mind, we have to consider the modifications on the user level, i.e., the *SQL environment*. In detail, the management of a logical database as a deductive relational database requires the following extensions to SQL (we refer to the proposals of DIN [1987]):

- As done by Bayer (1985) the *UNION-operator* is introduced in the <view definition> statement to allow several subdefinitions of a virtual relation or view (see Example 3):

  CREATE VIEW <table name> [(<view column list>)]
  AS <query specification> [{UNION <query specification>}...]

- The recursive definition requires an extension because of our special treatment of the implication operator. For that, the key word SELECT is changed to *SELECTi* according to the indexing of the implication symbol above. Of course, SELECT as the default value of SELECTi is used as usual (see Example 3).

- A *database procedure relation* is defined as a special view that can only be described by an algorithmic procedure and therefore is not expressible in Horn logic or SQL (the table name corresponds to the procedure name):

  CREATE VIEW* <table name>
  (<table element> [{,<table element>}...])

Example 3:

The view PRODUCT of example 2 is expressed in SQL as

```
CREATE VIEW PRODUCT
    AS SELECT *
        FROM EL_PRO
        UNION
        SELECT PRODUCT.Mach, PRODUCT.In, EL_PRO.Out
{respectively:  SELECT+ PRODUCT.Mach, PRODUCT.In,EL_PRO.Out}
        FROM PRODUCT, EL_PRO
        WHERE PRODUCT.Mach = EL_PRO.Mach AND
            PRODUCT.Out = EL_PRO.In
```

Note that the FROM part of an SQL expression always refers to whole relations whereas the predicates in the body of a logical rule also refer to projections. That is, in the case of an SQL expression, the DBMS has to perform an optimizing attribute selection (i.e., automated projections).

3.1.2 Management of Derivation Rules

So far, in INOVIS-X86 information about the database has been stored in the *data dictionary*, a graphical representation of the external, conceptual, and internal schemes. This semantic net in main memory showed an excellent runtime behavior in updating and accessing the DD. This is exactly the point in evaluating deductive queries by some sort of *rule-goal graph* (Ullman 1985). Therefore, the DRs (and ICs) are stored in the DD representing some sort of complete rule-goal graph of the intensional database which we call the *derivation graph*. This DD graph is better suited for the management of relational definitions than the rule-goal graph of Ullman (1985), which is too complex for practical application (2^n nodes for an n-ary relation instead of one). Moreover, from the DD graph a query specific derivation graph is easy to obtain (see section 4.2).

In detail, a derivation rule is stored in the DD in compiled form, i.e., as a graphical representation of its logical formula (see Figure 1). If a relational predicate is defined more than once, the *union operator* (U) combines the subrelations. The *operator node* (OP) contains the evaluable predicates (joins and selections) referring to the involved relational predicates. Note that in this way *common subexpressions* can easily be recognized by multiple references to a relational predicate (base, virtual, or database procedure relation). An *attribute list* (AL) is used to represent the respective projection determined by the occurrences of attributes in the SQL expression, as well as attribute assignments (see section 3.1.1), e.g., renamings.

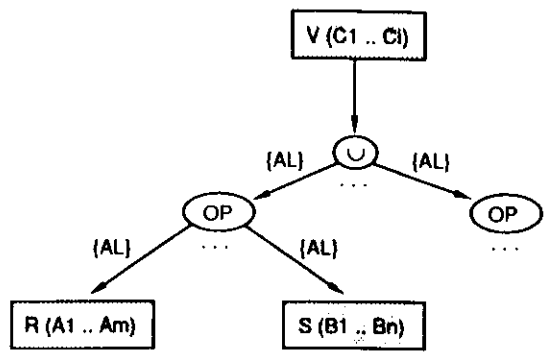

Figure 1. DD Description of a Derivation Rule (View Definition)

3.2 Integrity Constraints

An integrity constraint (IC) is an abstraction of a logical restriction that objects in the database must obey. Therefore, the existence of an *IC manager* is an essential presupposition for the consistency of a database. Derivation rules increase the number of facts retrievable from the database while integrity constraints reduce the number of facts that can be stored and retrieved. To achieve this, ICs must be treated as special Horn formulas (*special semantics*, see section 3.2.1) because a general Horn clause does not restrict anything.

The IC manager of the deductive DBS provides

- the formulation of database state dependent ICs,
- the consideration of at least the well-known semantic ICs, and
- the management and the application of ICs including database procedures.

Moreover, in the context of ICs we are not only concerned with *IC enforcement* but also with *semantic query optimization*. Semantic query optimization is the relatively new approach of utilizing ICs to simplify user specified queries by reduction of search space or identification of redundant join and selection clauses.

3.2.1 Formulation of Integrity Constraints

Usually, integrity checking in deductive databases is considered in the context of logical databases and theorem proving. However, our approach of a deductive DBS uses logic only as the formal background (Preiß 1987). Although the IDB is separated from the EDB, both are treated in the *same language environment*, namely the standard database language SQL. In the DD, the ICs are loosely coupled to the EDB relations and DRs enabling the deductive DBMS to select the profitable ones at query evaluation time. The formulation of integrity constraints obeys the following rules:

- The syntactical form of an IC corresponds to that of a DR:

  A1 (...), A2 (...), ..., An (...) → B (...) {1}

  The Ai and B represent relational and evaluable predicates.

- The only *syntactical differences to DRs* (definite Horn clauses) are that B may be an evaluable predicate, or missing, and that a projection may appear in the conclusion. Note that a relational predicate B requires an existing base or virtual relation B, otherwise the IC restriction for B makes no sense.

- On the user level of the deductive DBS, an IC is defined in SQL. In addition to the limited possibilities of standard SQL to specify ICs (DIN 1987) an *independent <integrity constraint definition> statement* similar to the view definition is introduced in order to capture the intended constraints:

  CREATE IC
  AS <query specification> {2}

  Note that contrary to the view definition and according to the *Principle of Empty Derivation* (see below) no <table name> part is specified. That is, in SQL we are able to distinguish DRs and ICs -- i.e., their semantics -- while this is not the case in Horn logic.

This kind of IC specification is a great step forward in dealing with database restrictions because it extends the scope of applicable ICs, applies query processing to integrity checking, allows semantic query optimization, and supports a *uniform user interface (SQL)*.

The proposed standard of SQL (DIN 1987) offers only limited possibilities for the declaration of integrity constraints which are specific to attributes of base relations (*domain, primary key, foreign key, unique, not null*). Our extensions comprise some of the *well-known semantic constraints* (e.g., general functional dependencies), *inter-relational constraints* (e.g., inclusion dependencies), *view constraints*, and *constraints through database procedures* that are all based on the "PRINCIPLE OF EMPTY DERIVATION":

- The *Principle of Empty Derivation* states that an IC is to consider as a query that must not derive any result tuple (see section 3.2.2), otherwise the database is inconsistent. This principle allows the formulation of ICs in two ways:

  - In the usual case of a restrictive IC, the conclusion of formula {1} is missing (Kowalski, Sadri and Soper 1987). In this case, the query is represented by the conditional part of formula {1} that is specified in the <query specification> part of the syntax rule {2}.

  - If a relational predicate is specified in the conclusion of formula {1}, then the IC checks some kind of completeness. Again the query is represented by the conditional part of formula {1} but this time a query result according to the conclusion is expected. In that case, the difference of the query result minus the (projection of the) base or virtual relation B must be empty. The difference is expressed in the <query specification> part of syntax rule {2} by a NOT EXISTS statement. (In Horn logic, this semantics is not explicitly expressible because it requires negation in the rule body.)

3.2.2 Semantic Integrity Constraints

In this section, the scope of our ICs comprises the *well-known semantic integrity constraints*, specifically those which are referred to as reasonable in real world databases. A good survey is given in the papers of Fagin (1981) and Fagin and Vardi (1984) in which the interested reader can find further, more "exotic" ICs.

As traditional dependencies -- often referred to in the context of schema design, less often in the context of query evaluation -- we consider *functional dependencies, multivalued dependencies* as a special kind of join dependencies, and *inclusion dependencies*. Furthermore, we are concerned with *domain constraints* and *implication constraints*. In the following, these ICs are presented in their logical form (Horn clauses) and as SQL expressions according to the user interface of the deductive DBS.

A FUNCTIONAL DEPENDENCY "A → B" with A, B subsets of U_R (attribute set of the relation R) is represented by m conjunctively connected Horn clauses (A = (A1 ... An), B = (B1 ... Bm), i = 1 ... m):

R1 (A1...An, B1...Bm), R2 (A1...An, B1...Bm),
= (R1.A1, R2.A1),..., = (R1.An, R2.An) → = (R1.Bi, R2.Bi)

or equivalently:

R1 (A1...An, B1...Bm), R2 (A1...An, B1...Bm),
= (R1.A1, R2.A1),..., = (R1.An, R2.An), <> (R1.Bi, R2.Bi) →.

This Horn formula is declared in SQL as follows:

```
CREATE IC
    AS SELECT *
        FROM R R1, R R2
        WHERE R1.A1 = R2.A1 AND...AND R1.An = R2.An
            AND (R1.B1 <> R2.B1 OR...OR R1.Bm <> R2.Bm).
```

Note that the <unique specification> of standard SQL offers a short form to specify special functional dependencies, namely *key dependencies*.

A JOIN DEPENDENCY "∞ [X1 ... Xk]" with X1 u ... u Xk = U_R is a generalization of the multivalued dependency and is represented by the following Horn clause (note that we do not want the expression to be regarded as a recursive definition):

R (X1),..., R (Xk) → R (U_R).

We consider the special case of a MULTIVALUED DEPENDENCY "A →> B" with A, B subsets of U_R and C = U_R \ AB :

R1 (A1...An, B1...Bm), R2 (A1...An, C1...Ck),
= (R1.A1, R2.A1),..., = (R1.An, R2.An) →
R (R1.A1...R1.An, B1...Bm, C1...Ck).

This does not represent a recursive definition and leads to the following SQL expression (remember that we only use projections):

```
CREATE IC
    AS SELECT R1.A1...R1.An, R1.B1...R1.Bm, R2.C1..R2.Ck
        FROM R R1, R R2
        WHERE R1.A1 = R2.A1 AND ... AND R1.An = R2.An
            AND NOT EXISTS (SELECT *
                            FROM R
                            WHERE R1.A1 = R.A1 AND ...
                                ... AND R2.Ck = R.Ck).
```

An INCLUSION DEPENDENCY "R (A) subset of S (B)" with A subset of U_R and B subset of U_S (R and S may be the same relation) is represented by the Horn clause

R (A1...An), = (A1, B1),..., = (An, Bn) → S (B1...Bn).

The corresponding IC declaration in SQL is

```
CREATE IC
    AS SELECT A1...An
        FROM R
        WHERE NOT EXISTS (SELECT *
                          FROM S
                          WHERE A1 = B1 AND ... AND An = Bn).
```

Note that the <referential constraint definition> of standard SQL offers the possibility to specify special inclusion dependencies, namely imports of *foreign keys*.

A DOMAIN CONSTRAINT "Ai op C" with Ai ∈ U_R, op ∈ {<, >, ≤, ≥, =, <>}, and C as a constant is represented by the Horn clause

R (Ai) → op (Ai, C) or equivalently R (Ai), ¬op (Ai, C) →.

The corresponding IC declaration in SQL is

```
CREATE IC
    AS SELECT *
        FROM R
        WHERE Ai ¬op C.
```

Note that the <column definition> of standard SQL also offers the possibility to specify such ICs.

An IMPLICATION CONSTRAINT "A1 op1 C1 → A2 op2 C2" (relational) respectively "A1 op1 C1, A2 op2 B1 → B2 op3 C2" (inter-relational) with Ai ∈ U_R, Bj ∈ U_S, opk ∈ {<, >, ≤, ≥, =, <>}, and Cn as constants is represented by the Horn clause

R (A1, A2), op1 (A1, C1) → op2 (A2, C2)

or equivalently

R (A1, A2), op1 (A1, C1), ¬op2 (A2, C2) →,

respectively

R (A1, A2), S (B1, B2), op1 (A1, C1), op2 (A2, B1)
→ op3 (B2, C2)

or equivalently

R (A1, A2), S (B1, B2), op1 (A1, C1), op2 (A2, B1),
¬op3 (B2, C2) →.

The corresponding IC declaration in SQL is

```
CREATE IC
    AS SELECT *
        FROM R
        WHERE A1 op1 C1 AND A2 ¬op2 C2
```

respectively

```
CREATE IC
    AS SELECT *
        FROM R, S
        WHERE A1 op1 C1 AND A2 op2 B1 AND B2 ¬op3 C2.
```

Of course, the well-known semantic integrity constraints are not the only ones expressible in our restricted Horn language. We will show by a concluding example that some *arbitrary ICs* can also be handled, e.g., some sort of *tuple dependency* (see Example 4).

Example 4:

Assume that it must be guaranteed that for every elementary production step on the machine "assembly-5" there is

a following production step on the machine "assembly-6". For this, the following integrity constraint must be defined:

```
EL_PRO1 (Mach, Out), = (EL_PRO1.Mach, "assembly-5"),
= (Out, In), = ("assembly-6", EL_PRO2.Mach)
    → EL_PRO2 (Mach, In).
```

This IC is expressed in SQL as

```
CREATE IC
 AS SELECT *
    FROM EL_PRO EL_PRO1
    WHERE Mach = "assembly-5" AND
        NOT EXISTS (SELECT *
                FROM EL_PRO EL_PRO2
                WHERE EL_PRO1.Out = EL_PRO2.In AND
                    EL_PRO2.Mach = "assembly-6").
```

3.2.3 Management of Integrity Constraints

As with all of the other information about the database scheme, the ICs are stored in the *data dictionary*. Because of the syntactical similarities between DRs and ICs, an integrity constraint can be described like a derivation rule (graphical representation of its logical formula -- see Figure 2). The union operator is replaced by a *difference operator* (\) according to our interpretation of an IC (Principle of Empty Derivation). If a relational predicate occurs as the conclusion of an IC, then the difference operator is applied to point to the corresponding base relation (BR), specifically virtual relation (VR), and to the conditional part of the IC definition (see Figure 2).

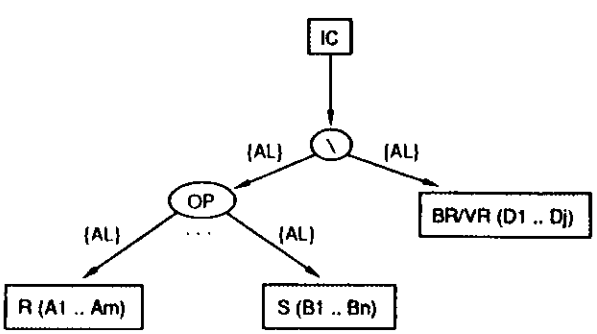

Figure 2. DD Description of an Integrity Constraint

The uniform treatment of DRs and ICs allows a homogeneous graphical representation of the database scheme and, thus, the *data dictionary* comprises four groups of definitions connected as shown in Figure 3.

to \ from	base relations	d.p. relations	views	ICs
base relations	÷	÷	DP	DP
d.p. relations	÷	÷	DP	DP
views	LR	LR	DP / LR	DP
ICs	LR	LR	LR	÷

Figure 3. Pointers in the DD of the Deductive DBS

The *direct pointers* (DP) refer to the components of the DRs and ICs enabling the deductive DBMS to extract the query specific derivation graph from the DD (see section 4.2) in a very short time period. On the other hand, the *lists of references* (LR) represent the backward chaining that enables the DBMS to perform fast and complete update operations on the DD graph.

This structure of the DD allows the *twofold* application of ICs in the query context: on the one hand, an IC represents a *query expression* to verify database consistency, and on the other hand, ICs can be used to *simplify the derivation process* of a query evaluation (not of an IC evaluation). In both fields, the deductive DBS provides a variety of ICs (see section 3.2.2) and a trigger mechanism for database procedures. Note that the application of *database procedures* offers a very flexible possibility of declaring arbitrary constraints. For example, the definition

```
CREATE IC
   AS SELECT *
      FROM <database procedure name>
```

triggers a PASCAL program that might generate tuples contradicting some *arbitrary semantic integrity constraints*.

4. DEDUCTIVE QUERY EVALUATION

4.1 General Overview

The idea of using an IC in *semantic query simplification* is a relatively new field of research especially in the context of deductive databases. Our approach *incorporates the relevant IC definitions* into the query specific derivation graph during query evaluation. That is, from the ICs of the involved relations, those chosen restrict the query predicates by reduction of search space or identification of redundant predicates (see section 4.2). Because of the pointer structures in the DD, a *matching IC* -- i.e., all conditional predicates of the IC are applicable to the conditional predicates of the (sub)query -- is found in short time.

A detailed description of the *deductive processing* goes beyond the scope of this paper. Therefore, an illustrative example is presented to give an idea about the use and operation of our deductive DBS. The *principle steps of deduction* are introduced in advance.

First, the GOAL rule of the query expression is temporarily integrated into the DD graph. The GOAL subgraph is then extracted including those ICs that exclusively concern the involved relations. The leaves of the resulting *hierarchical derivation graph* comprise base relations and database procedure relations. Cycles in the derivation graph represent recursively defined virtual relations. Common subrelations are marked by multiple references.

In a second step, a bottom up evaluation of the derivation graph generates a *(sequence of sub)query expression(s)* which can be processed by the conventional DBMS to determine the result of the overall query. For that, the virtual relations in the derivation graph are resolved, i.e., the upper operator node and the lower one are combined taking into consideration the restrictions of matching ICs. Recursively defined views, common subrelations, virtual relations with a UNION node, and those involved in aggregate functions are not resolved but selections are pushed through -- except in the last case.

The amalgamation produces a simplified GOAL subgraph consisting, besides base and database procedure relations, of the non-resolved virtual relations. Of these, the lowest is specified first, then the upper one, and so on. Finally, the statement for the overall query can be expressed using the *intermediate results* of the subexpressions. Note that only the recursive evaluation requires deductive support for the conventional DBMS applying Semi-Naive evaluation in the regular case and an enhanced version of the method of Henschen and Naqvi (1984) otherwise.

4.2 Illustrative Example

To show the principle of our deductive query evaluation, we consider the following formal specification of a sample "manufacturer" database (DRs and ICs):

The elementary production steps are performed in two factories A and B:

```
EL_PRO (In, Out)  :- EL_PRO_A (Mach, In, Out).
EL_PRO (In, Out)  :- EL_PRO_B (Mach, In, Out).
```

There is a regulation that factory A must not use any silicon:

```
:- EL_PRO_A (Mach, "silicon", Out).
```

The multi-step productions are derived as follows:

```
PRODUCT (In, Out)     :- EL_PRO (In, Out).
PRODUCT (In, Out)     :- EL_PRO (In, I/O), PRODUCT (I/O, Out).
```

A last derivation rule defines the two-step productions with the second step performed in factory B:

```
SEC_PRO_B (In, Out)   :- EL_PRO (In, I/O),
                         EL_PRO_B (Mach, I/O, Out).
```

To use this logical database as a *deductive relational database*, we specify the IDB as follows:

```
EL_PRO_A (Mach-A, In-A, Out-A), = (In-A, In), = (Out-A, Out)
                              → EL_PRO (In, Out).
EL_PRO_B (Mach-B, In-B, Out-B), = (In-B, In), = (Out-B, Out)
                              → EL_PRO (In, Out).
EL_PRO_B (Mach-B, In-B, Out-B), = (In-B, "silicon")
                              →
          EL_PRO (In, Out) → PRODUCT (In, Out).
EL_PRO (In, Out), PRODUCT (In, Out), = (EL_PRO.Out, PRODUCT.In),
                              →+ PRODUCT (EL_PRO.In, PRODUCT.Out).
EL_PRO (In, Out), EL_PRO_B (Mach-B, In-B, Out-B), = (Out, In-B)
                              → SEC_PRO_B (In, Out-B).
```

This IDB is defined in SQL:

```
CREATE VIEW EL_PRO (In, Out)
AS SELECT In-A, Out-A
    FROM EL_PRO_A
    UNION
    SELECT In-B, Out-B
    FROM EL_PRO_B

CREATE VIEW PRODUCT
AS SELECT *
    FROM EL_PRO
    UNION
    SELECT+ EL_PRO.In, PRODUCT.Out
    FROM EL_PRO, PRODUCT
    WHERE EL_PRO.Out = PRODUCT.In

CREATE IC
AS SELECT *
    FROM EL_PRO_B
    WHERE In-B = "silicon"

CREATE VIEW SEC_PRO_B
AS SELECT In, Out-B
    FROM EL_PRO, EL_PRO_B
    WHERE Out = In-B
```

The SQL definitions lead to the *data dictionary* shown in figure 4 (for reasons of clarity only direct pointers are listed).

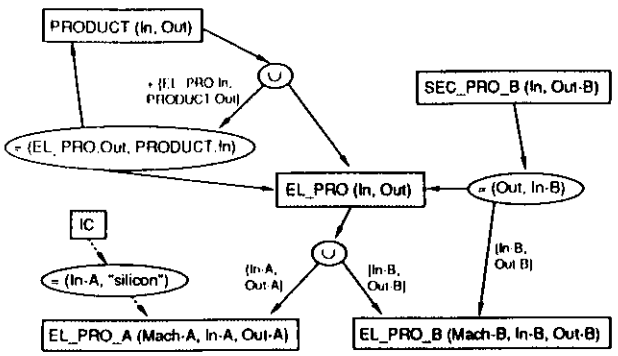

Figure 4. DD Graph of Sample Application

Obviously, from this DD graph a *query specific derivation graph* to find the two-step productions of SEC_PRO_B based on silicon is easy to extract (Figure 5).

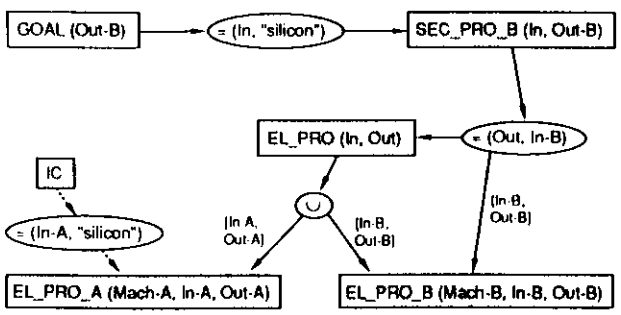

Figure 5. Derivation Graph of Sample Application

The virtual relation SEC_PRO_B is now resolved, i.e., the upper operator node and the lower one are united. Furthermore, the selection predicate can be pushed into the virtual relation EL_PRO according to Figure 6.

The IC condition contradicts the selection condition of relation EL_PRO_A which therefore can be removed. The virtual relation EL_PRO can now be resolved by renaming the corresponding identifiers. This yields the final graph of Figure 7.

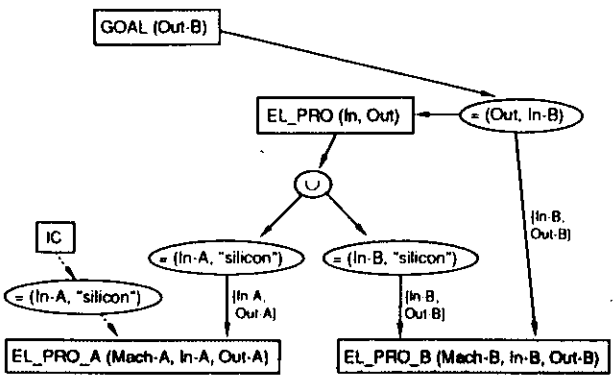

Figure 6. Simplified Derivation Graph of Sample Application

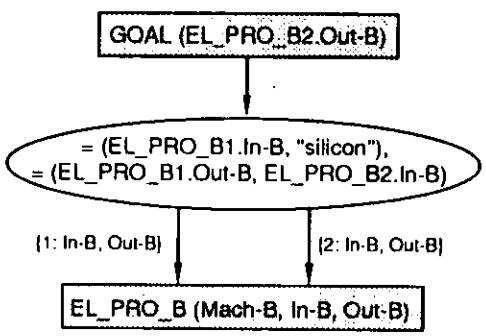

Figure 7. Final Derivation Graph of Sample Application

Consequently, our deductive component performed a *transformation* from the original query

```
SELECT Out
FROM SEC_PRO_B
WHERE In = "silicon"
```

to the conventional query expression:

```
SELECT EL_PRO_B2.Out-B
FROM EL_PRO_B EL_PRO_B1, EL_PRO_B EL_PRO_B2
WHERE  EL_PRO_B1.In-B = "silicon" AND
       EL_PRO_B1.Out-B = EL_PRO_B2.In-B
```

5. CONCLUSION

When processing knowledge by deduction on a large set of given facts, a *deductive relational database system* performs better than a logic based system, e.g., a PROLOG based XPS. This paper presented a unique concept for the construction of such a DBS; namely, the deductive extension of a relational database system. The deductive extension enhances the expressiveness, the degree of consistency, and the efficiency of evaluation of conventional databases. Consequently, the deductive DBS is able to perform conventional database tasks as well as simple XPS inferencing and therefore may be used to support PROLOG based XPSs efficiently.

While preserving the conventional relational database environment, we provided a comprehensive concept for the deductive extension including an *extended view mechanism* (derivation rules with recursion and database procedure relations) and an *advanced management of integrity constraints*. Contrary to other approaches, deduction is performed by the DBMS itself while logic is merely used as the formal background. So, on the user level, the deductive database is specified through SQL expressions; that means, the EDB and IDB (DRs + ICs) are treated in a uniform manner.

DRs and ICs are stored in the data dictionary as graphical representations of their logical formulas (range restricted, function free, at most linear recursive Horn clauses). From this DD a *query specific derivation graph* is easy to extract which may be used to transform a deductive query into a sequence of preoptimized conventional query expressions evaluable by the conventional DBMS.

Future developments and research efforts will be dealing with formal issues concerning the deductive database. On the one side, the *deductive evaluation* and the kind of *relational completeness* provided by the extended SQL must be described formally. Moreover, it is an open question when to check an IC and how to guarantee consistency of the IC system itself. On the other side, we are concerned with the precise analogies between logic programming and deductive relational database management. These may be used to determine an *automatic substitution* of specific logical inferences in a knowledge processing system based on logical rules by more efficient query evaluations in the deductive DBS.

6. ACKNOWLEDGEMENT

I would like to thank W. Stucky and F. Schönthaler for the valuable discussions and remarks that helped to make this paper possible.

7. REFERENCES

Aho, A. V., and Ullman, J. D. "Universality of Data Retrieval Languages." In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, 1979, pp. 110-120.

Bancilhon, F., and Ramakrishnan, R. "An Amateur's Introduction to Recursive Query Processing Strategies." In A. Nori (ed.), *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Cambridge, Massachusetts, 1986, pp. 16-52.

Bayer, R. "Database Technology for Expert Systems." In W. Brauer and B. Radig (eds.), *Wissensbasierte Systeme*, GI-Kongreß, München 1985. Heidelberg: Springer-Verlag, 1985, pp. 1-16.

Brodie, M. L., and Jarke, M. "On Integrating Logic Programming and Databases." In L. Kerschberg (ed.), *Proceedings from the First International Workshop on Expert Database Systems*, 1984. Menlo Park, CA: Benjamin/Cummings Publishing, 1986, pp. 191-207.

DIN. Deutsches Institut für Normung "Datenbanksprache SQL." (identical to ISO/DIS 9075, edition 1986), Berlin, 1987.

Fagin, R. "A Normal Form for Relational Databases that is Based on Domains and Keys." *ACM Transactions on Database Systems*, Vol. 6, September 1981, pp. 387-415.

Fagin, R., and Vardi, M. Y. "The Theory of Data Dependencies -- A Survey." *IBM Research Laboratory*, San Jose, California, 1984.

Gallaire, H.; Minker, J.; and Nicolas, J. M. "Logic and Databases: A Deductive Approach." *ACM Computing Surveys*, Vol. 16, June 1984, pp. 153-185.

Henschen, L. J., and Naqvi, S. A. "On Compiling Queries in Recursive First-Order Databases." *Journal of the ACM*, Vol. 31, January 1984, pp. 47-85.

Karszt, J. "Datenbank-Pascal: Ein ausbaubares Datenbanksystem nach einem Entity-Relationship-Model für Personal-Computer-Anwendungen." Dissertation, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 1984.

Kowalski, R.; Sadri, F.; and Soper, P. "Integrity Checking in Deductive Databases." In P. Hammersley (ed.), *Proceedings of the Thirteenth Conference on Very Large Databases*, Brighton, England, 1987, pp. 61-69.

Preiß, N. "Data Based Knowledge Processing." In A. Heuer (ed.), *Proceedings of the Workshop on Relational Databases and their Extensions*, Lessach, Austria, 1987, pp. 71-105.

Preiß, N. "PROLOG-X86: Coupling Prolog with a Relational Database System." To Appear in Proceedings of the Second Workshop on Relational Databases and their Extensions, Lessach, Austria, 1988.

Smith, J. M. "Expert Database Systems: A Database Perspective." In L. Kerschberg (ed.), *Proceedings from the First International Workshop on Expert Database Systems, 1984*. Menlo Park, CA: Benjamin/Cummings Publishing, pp. 3-14.

Stonebraker, M., and Rowe, L. A. "The Design of Postgres." In C. Zaniolo (ed.), *Proceedings of ACM Conference on Management of Data*, Washington, D.C., 1986, pp. 340-355.

Ullman, J. D. "Implementation of Logical Query Languages for Databases." *ACM Transactions on Database Systems*, Vol. 10, September 1985, pp. 289-321.