LOGIC DATA BASES VS DEDUCTIVE DATA BASES

Working Paper to be presented at the

Logic Programming Workshop 1983

ALBUFEIRA - PORTUGAL

Hervé GALLAIRE
Laboratoires de MARCOUSSIS
Centre de Recherches de la **C.G.E.**
Route de Nozay
91460 MARCOUSSIS - FRANCE

ABSTRACT

The area of data bases is the area of Computer science most likely to be invested by a new methodology -should one say a new technology- based on logic programming. This survey investigates various approaches to the merging of these two worlds, trying to straighten out the advantages, problems and applications of each of them.

INTRODUCTION

The area of data bases is one more area of computer science subject to being taken over by a new methodology -should one say a new technology- based on logic programming. This paper surveys the various approaches to merging these two fields, depending on viewpoints adopted for one's problem analysis ; the logic data base field starts from logic and tries to enhance it with data base assets, be they data access techniques or data base features ; on the converse deductive data bases are built from existing data base systems by enhancing them with deductive, and other, capabilities. These two viewpoints, although yielding different systems and being interesting for different types of applications and goals, are rather complementary and share many common problems. The paper concludes that enough of the theoretical aspects of the deal are well-known and that it is time now for practical applications as well as theoretical improvements.

The paper is divided into five sections. The first section presents the four approaches to linking data bases and logic ; the first three to be described in sections 2 through 4 adopt the logic viewpoint and culminate into full blown logic database ; section 5 presents the deductive database approach.

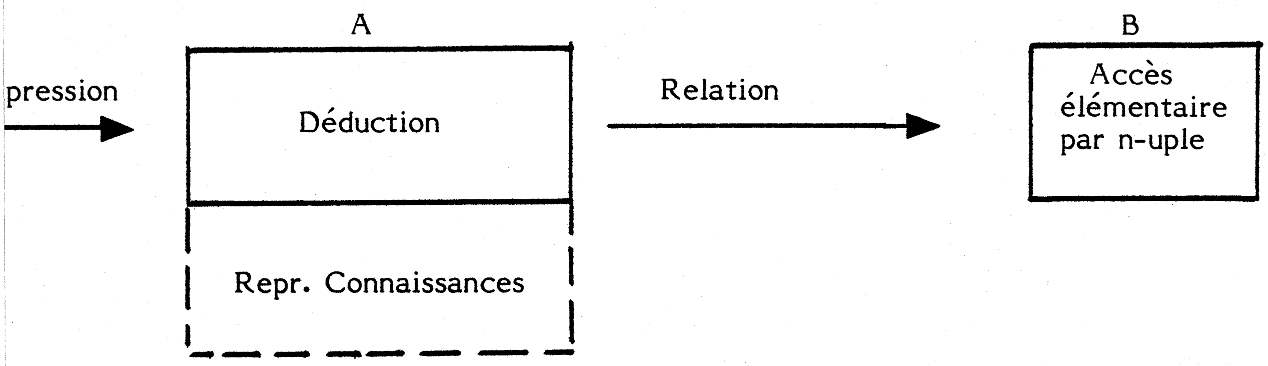
One should take note that the modeling power of logic databases will not be discussed in this overview because the paper does not deal at all with work on knowledge representation formalisms.

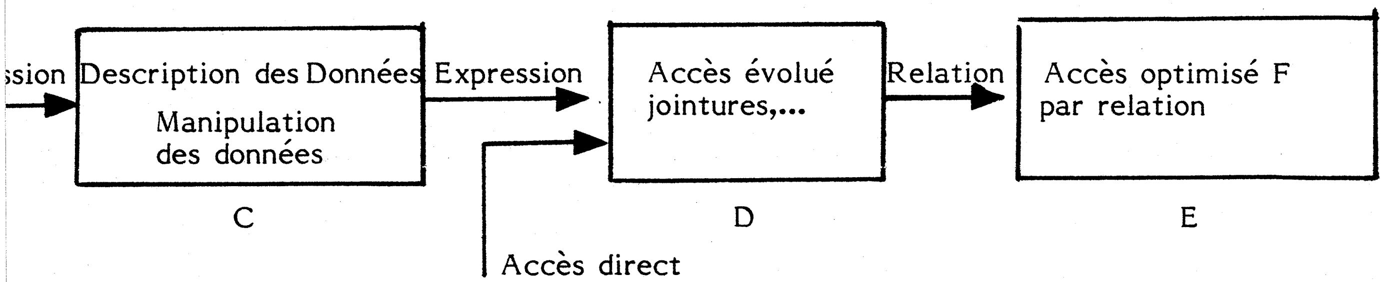
Section 1 : Logic programming - Data Bases

The first to realize the potential of logic programming for data bases was probably C.GREEN (1) who, although he did not know about logic programming which did not exist at that time, described how to connect logic-based Question-Answering system to data Base systems. Since that time various papers, books and workshops dedicated to that subject have brought up the subject (2, 3, 4, 5, 6) without fully clarifying the relationships between the two fields.

In order to study this relationship closely, we first decompose a logic programming system and a data base system into their respective components. A logic programming system, PROLOG being the most well known example of them, is made of a deductive component (A) and of a rudimentary access component (B) which provides the deductive component with individual tuples ; the query to A may be a relational expresssion (usually a negative clause in PROLOG) ; the interface between A and B is a relation.

A database system is made of a data description and data manipulation component (C), a data access expression optimizer (D), and a data access component (E) ; query relational expressions are submitted to (C) or to (D) ; interface between (C) and (D) is a relational expression, usually of the relational algebra ; interface between (D) and (E) is at the relation level, bringing back full sets of tuples instead of individual tuples as (B).





With such decompositions in mind, four types of connections can easily be thought of :

PROLOG+ : Some relations are defined as being managed by a mechanism of type E, thus giving a system made of :

$A \Leftrightarrow (B+E)$

PROLOGDB : PROLOG formulas can be considered as being a full fledged query language for a database access system (D-E) ; thus we obtain :

$A \Leftrightarrow (D+E)$

Logic Data Base : This is the natural extension of the previous two approaches where one builds above or aside PROLOG a true data base system, with a description and manipulation language, including capabilities for integrity constraints expressions etc... Although it is not necessary to include capabilities of type D or E they will be included if only for performance reasons ; thus we obtain :

$C \Leftrightarrow A \Leftrightarrow (D+E)$

with $C \Leftrightarrow A$ as a minimum system.

Deductive Database : The goal here is to provide extensions to conventional database systems which have well-known limitations, if only for the query languages which need to be embedded into foreign programming languages. The systems so obtained are of the type :

$A \Leftrightarrow C \Leftrightarrow D \Leftrightarrow E$

or even $C' \Leftrightarrow A \Leftrightarrow C \Leftrightarrow D \Leftrightarrow E$

when one combines this deductive database approach with the logic database one. Logic covers various aspects that the query language covers inadequately: views, optimizing techniques, theoretical understanding of important problems such as incomplete information handling,...

Section 2 : PROLOG+

It is known that PROLOG-like access to individual data is not well-suited to relations which would be stored in secondary memory, due to the fact that data is requested one at a time. Also it is known that even in primary memory there are ways to index data which make it faster to retrieve (e.g indexing on specific fields of a relation rather than sequential access on its name). On the contrary database systems are very much concerned with the efficiency of data retrieval. PROLOG+ systems are nothing more than systems in which some relations have been declared as database relations or DB-relations and handled by a DB-like access mechanism (indexing, B*-tree, multiple hashing,...). Such systems have already been built ; PROLOG-like access is simulated for the DB relations by buffering the set of tuples retrieved in one operation, and giving PROLOG one tuple at a time from this buffer. See eg (7). It is clear that this approach is an easy way to enhance PROLOG, and for some well defined large applications of PROLOG, worth implementing. It is surprising that no such a large scale application has been reported up to now.

Section 3 : PROLOG BD

Recall the configuration of such systems :

A $\Leftrightarrow$ (D+E)

Such a system can be seen as a PROLOG+ system in which instead of interfacing with the DB system at the relation level, one interfaces at the resolvent level : given DB-relations, given other relations (called PROLOG relations or P-relations to distinguish them from DB-relations), given a PROLOG program including clauses mixing P-relations and DB-relations, one would like to optimize access to P-expressions i.e. to expressions containing P-relations only, rather than to evaluate each P-relation when, in the deductive part of PROLOG, it becomes the leftmost literal of the resolvent (as done in PROLOG+). There are several ways to do this which are examined below. First let see why one would want to do such global retrieval as opposed to an individual, relation-based retrieval ; among the possible reasons one which is most appealing is that it is known that DB systems behave more efficiently than virtual memory systems, that they have quite efficient optimizers, that they offer set-operators which can be very much optimized and even executed through specific hardware (the database machines).

The connection sketched above is in principle easy to imagine. A major initial decision to be made is how much control over the evaluation process is left to the programmer ; in other words the decision is to be made whether the programmer can decide (i.e can tell the system) when a (sub-) expression is to be sent to the database system, how much data is to be brought back, etc.

Making such a possibility explicit in the hands of the programmer requires an extension of the logic language, namely that a set of system predicates be added which allows to express information about retrieval, insertion, deletion, etc., thus making a "data sublanguage" out of PROLOG by extending it. Such an explicit control has been defined and advocated in (8) ; it could be a basis of some of the 5G languages. One could perhaps also adapt to DB the technique of (9). These approaches are certainly worth experimenting, but we believe it is not easy : it is certainly not a simple matter to find logic programmers knowledgeable enough to make the right decisions about these retrieval expressions. Nevertheless it is the one which, in the short term, could prove the most effective ; one should bear in mind, though, that some DB researchers express concern about optimization problems and believe that DB access optimizing is a formidable task that needs much processing power, which is sometimes counter-intuitive, and which is usually better carried out by general programs.

If the responsability of the decision is to be taken by the system and not by the programmer, it remains two basic roads. The first is the compilation technique in which one translates an initial request into a DB-expression which is then sent to the DB-system ; thus there is a clear cut separation between deduction (generation of an evaluable expression) and access. The second technique is the interpretation one, in which both processes are intermixed.

Compilation

This technique has been widely studied (10) and has led to several implementations and approaches depending on the complexity of the logic program.

Case 1

There is no recursive axiom in the program for defining P-relations in terms of DB-relations.

This case is without difficulties. There are two ways to deal with it. One can modify the logic interpreter so that it delays evaluation of DB-relations until the resolvent involves DB-relations only ; this might perhaps be done by using Geler (Freeze) predicate from PROLOG II. Alternatively one could write a translator which acts as a meta-interpreter as done in (11, 12).

Case 2

There exist recursive axioms in the program ; an example of such a case would be a transitive closure relation supposed to be a P-relation and defined in terms of itself (hence the recursivity) and a DB-relation. Whereas in case 1 all that was to be done was a macro-expansion, one is now confronted to a true program generation problem ; at least in principle two classes of solutions have been studied :

<u>pseudo-compilation</u> : This is an extension of case 1, i.e the recursive program is not translated into an iterative one, or into an evaluable formula ; rather it generates

a sequence of evaluable formulas each corresponding to an alternative solution used on backtracking when the logic interpreter, or the user asks for additional solutions. An example extracted from (7) follows :

given ancestor(X,Y) ← parent(X,Y)

ancestor(X,Y) ← ancestor(X,Z), ancestor(Z,Y)

and a query ← ancestor(X,Y)

the system will generate the following evaluable formulas :

[edb(parent,X,Y)] then

[edb(parent,X,Z), edb(parent,Z,Y)] then

[edb(parent,X,Z), edb(parent,Z,Z1), edb(parent,Z1,Y)]

...

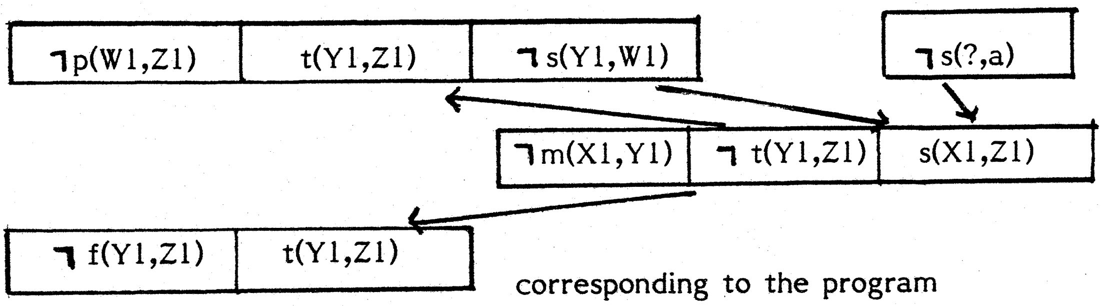
where edb(parent,-,-) is a relation evaluable by the DB ; such formulas evaluation can be optimized. Other examples show that additional capabilities (one should notice the example used a non-trivial recursion) such as negation and mixed relations can be handled too. A mixed relation is a relation defined by a program which includes assertions i.e positive litterals as well as conditionals (general axioms as above).

Although such systems are, in principle, simple enough, their drawback is a redundancy which is obvious from the example above : consecutive formulas share common literals which will be evaluated several times ; getting rid of this redundancy at the deductive system level amounts to a true compilation (see next) ; getting rid of it at the DB level is not a classical operation of such systems.

True compilation : It is possible to generate truly iterative programs involving purely evaluable DB-relations starting from recursive logic programs including both P-relations and DB-relations. Several techniques have been proposed (13, 14, 15).

In (13) recursive programs of the regular type (in the formal language sense) only can be handled ; it is not surprising that such a class of programs can be translated into iterative programs, as this is well known from automata theory. In (14) various extensions to the regular programs are given, without reaching the full power of logic programs.

(15) describes the most general approach as of to-day, it is based on connection graphs, a well-known technique (16) ; the basic idea is to generate a program which is a loop around the cycle(s) in the connection graph, collecting all DB-relations involved in this process until the exit of the loop. A simple example is in order (15) : given the following connection graph and a query s(?,a).



corresponding to the program

s(X1,Z1) ← m(X1,Y1), t(Y1,Z1)

t(Y1,Z1) ← s(Y1,W1), p(W1,Z1)

t(Y1,Z1) ← f(Y1,Z1)

with p,m,f DB-relations, the program to be generated goes along the loop collecting p-tuples, each of them driving an inner evaluation loop of m-tuples and f-tuples as can be seen by looking at the successive evaluable formulas :

m(?,Y1), f(Y1,a)

m(?,Y2), m(Y2,Y1), f(Y1,W2), p(W2,a) (r2)

m(?,Y3), m(Y3,Y2), m(Y2,Y1), f(Y1,W2), p(W2,W3), p(W3,a) (r3)

....

The program is :

```
Z1 = a
edb(p,W2,Z1) ; edb(m,X,Y1) ; edb(f,Y1,Z1) ; print(X)
enqueue(Q,W2)     values of W2 will drive an outer loop
foo = m(X2,Y2), m(Y2,Y1), f(Y1,W2)  to be evaluated, starting from
                                    f for each value of W2
i = 2
while (Q ≠ empty) do
   while (Q1≠empty) do W2=Deque(Q1) ; edb(foo) ; print(Xi) ; od
     does what was expected, see(r2) above
   Q1 = Q

   Q = empty
   while (Q1 ≠ empty) do W3 = Deque(Q1) ; edb(p(W2,W3) ;
                          enqueue(Q,W2) ; od
     collects now values for W2 as in r3 above
   replace m(Xi,Yi) by m(X_{i+1},Y_{i+1}), m(Y_{i+1},Yi) in foo ; i=i+1 ;
                      prepare for a new outer loop
od
```

This program is, on the surface, satisfactory ; the authors state that its only limitation is due to the fact that the form of the initial query must be known (here s(?,a)). There may be another difficulty which is that, in order for the program to stop, the enqueue operation is not a mere "push" : it must check that the value has not been pushed i.e. enqueued before ; this may be a practical limitation of the system.

Another approach, without any of these limitations maybe under way (17) but not enough is known about it at this time.

Interpretation

These techniques intermix deduction and evaluation steps ; in fact what was described in section 2 for PROLOG+ was already an interpretation. Other schemes have been presented, starting from the idea that unification done tuple at a time was not precisely adapted to systems in which DB-relations were handled ; such a case was argued in MRPPS (4) where the concept of Π-unification was developed. A more systematic study in terms of PROLOG implementation is described in (11) where the basic idea is the following : rather than storing at each node of the proof tree the whole set of unifications (as a table), it is possible either to store a unification set only at the root and to store at each node the computation rules which will allow to compute their new unification sets from their parent node, or to store unification sets at the leaves and at each node the information which allows to compute their unification set from their descendent nodes. Examples are described in (11) although a complete implementation of PROLOG based on this has not been realized.

Such techniques would be interesting for parallel PROLOG implementations.

Section 4 : Logic DB

This approach is the most natural one for all those who believe logic programming to be a universal programming language. Their arguments are strong, we adhere to them basically. A database, as seen by KOWALSKI (6,18) is a collection of HORN clauses including functions if one wishesto (already an extension of conventional DB), atop of which it suffices to build DB functionalities.

A starting point is that PROLOG, including its set-of extension is relationally complete, i.e. can express all queries expressed in relational algebra, the common base language to all relational DB systems (with operators such as union, projection, join, division,...) ; such a result although interesting is well-known since CODD results on equivalence between relational calculus (i.e. logic) and relational algebra. Of course the set-of construct gives all that is needed to express aggregation constructs, averages,...

However, PROLOG and logic provide more than a conventional query language because the expressive power of logic programming is at least that of least fixed points, an example of which being transitive closures :

lfp(R,R*) can be expressed as a simple PROLOG program computing the least fixed point R* for any PROLOG relation R. In specific cases, it is simpler to compute the closure directly, as in

ancestor(X,Y) ← parent(X,Y)

ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y)

Some limitations of the approach should nevertheless be phrased :

- It must clearly be connected to a DB system as described in Section 3 if only for efficiency problems ; this is clear for example in (19) where a set of queries to a DB system expressed in PROLOG had to be optimized before being sent to the DB system ; although one could argue that one of the major difficulties (duplicates) in the answers came from the PROLOG evaluation scheme itself, not from logic, this is still a problem to be faced in general.

- HORN clauses, if relationally complete, are not sufficient to express naturally all queries that one would like to ask using logic itself (6,18) : find all suppliers supplying all pieces needed for project "a".

  Such a query involves conditionals within conditionals ; this is translated into negation within the body of a clause and is not properly handled by PROLOG unless specific attention is paid.

- Integrity constraints, time-constraints involves additional mechanisms which resemble plan-generation techniques ; non-monotonic reasoning is also necessary ; possible solutions are presented in (6,18,20).

Some realizations have been reported along these lines, eg (21,22). The first one is a PROLOG implementation of QBE, while the other is a description of a system where PROLOG is an intermediate language target for a QBE external language as well as an SQL external language and a relational algebra external language. In the PROLOG implementation of QBE (21) it is shown how to simply take into account integrity constraints on inserts and deletes using a technique which was also used in (23), the catchall clause. That logic database approach is typically an approach which is closest to Artificial Intelligence, at least to the theorem proving part of Artificial Intelligence if not to the knowledge representation one. Systems built in that perspective include (4, 24). Powerful non-HORN theorem provers can be used, plan-generation techniques can be expressed.

Section 5 : Deductive Data Bases

The bias introduced in developing deductive database systems is that DB systems can be enhanced by adding to conventional retrieval capabilities of data explicitly introduced, that of retrieval through deduction mechanisms using general laws. This extension, introduced at first purely for retrieval purposes turns out to have many more facets which are briefly examined.

Conventional DB's manipulate facts only (the tuples of the relations). The general laws they use are so-called integrity constraints (IC), used to validate updates of facts. All queries are evaluated with respect to facts only.

In deductive DB's general laws can be partitioned in two sets : IC's and deductive rules (DR). Queries are then evaluated with respect to facts and DR's. But IC's will also need to be evaluated with respect to facts and DR's. This makes it more difficult of course to check IC's which thus require deductive capabilities. Deductive databases (DDB's) are made of a collection of solutions to various problems whose conventional solutions in DB's have to be adapted in this new context. To understand these new solutions, old problems and solutions must first be reviewed.

Conventional DB's enforce implicit assumptions for retrieval :

- Closed world assumption (3, 36) : all facts not known to be true, i.e not stored as tuples, are false

  $\neg R(a1,\ldots,an)$ iff $< a1,\ldots,an > \notin R$

- Unique names : elements with different names are different

  $\forall\, b,c \quad b \neq c$

- domain closure : there are no other elements than those stored in the DB.

The first two hypotheses combined allow negation evaluation (recall that NOT is an operator in relational algebra). The third one allows evaluation of queries such as $\forall\, xP(x)$,... It could be dispensed of if one restricted the allowable queries to meaningful subsets of the syntactically correct queries, thus reducing to range-restricted queries.

$\forall x(Q(x) \rightarrow P(x))$ is evaluable without hypothesis(3)

while $\forall xP(x)$ is not.

These conventional assumptions have to have counterparts in any formalized view of conventional DB's. After this formalization is done, it is possible to extend it to DDB's. Two formal views of conventional DB's have been studied (25, 5) : a model-theoretic view (MTV) and a proof-theoretic (PTV) one. Without going into details, the MTV assumes that the set of facts is an interpretation E, a model, of a theory made of IC's and that query evaluation is done in E, abiding to the above three assumptions. Although such a view deals with problems such as query evaluation and optimization, choice of conceptual schemas, etc... it does generalize to DDB's and incomplete information problems. The PTV sees a conventional DB as a first-order theory T plus a set of closed formulas, the IC's. The theory T is made of facts (positive HORN formulas) and a set of particularization axioms. These particularization axioms (Domain closure, Uniqueness of names, completion, equality) are the formal translation of the above three assumptions. The DB is still not a DDB but deduction could be used to handle T ; this may be unwise and in any implementation this is likely to be dealt with at a metalevel, i.e integrated to the query algorithm. Nevertheless PTV is very useful in terms of the generalizations it suggests :

- DDB's which are obtained via a third class of axioms, the deductive rules (DR) mentioned earlier.

- DB's which allow disjunctive information, leading to incomplete information (5, 26, 27).

DDB's are subject to new problems, in that the axioms introduced in T may be inconsistent with some general deductive laws ; it is well known that such is the case between disjunctive axioms and those (in T) accounting for CWA.

$$\neg R(al,\ldots,an) \text{ iff } \{T,DR\} \nvdash R(al,\ldots,an)$$

cannot be accepted as such :

$$\left.\begin{array}{ll} Cat(X) \rightarrow Black(X) \cup White(X) & (DR) \\ Cat(Felix) \leftarrow & \\ \text{axioms in T} & \end{array}\right\} \begin{array}{l} \nvdash Black(felix) \text{ hence } \neg Black(Felix) \\ \nvdash White(felix) \text{ hence } \neg White(Felix) \end{array}$$

These two informations are contradictory with the unique DR. Solutions to handle this are partially known (5, 26, 27) and consist either in restricting general laws (DR) to regular clauses with adequate axioms T' instead of T, or in dealing with incomplete information systems.

It must be emphasized again that this theoretical view (regular clauses + axioms T') is not to be implemented as such ; again, implementation goes through some meta-rules rather than using T' axioms ; for instance negation as failure (33) and range-restricted formulas (35).

There are two ways to exploit a DDB. Most of the systems realized today use the deductive approach where data is actually deduced when needed. In the generative approach (28), deductive rules are used as generative rules : each time data is entered, all information derivable from it, or with its help, is derived and generated (stored in the DB) ; of course supressing data becomes a non-trivial process, akin to Truth Maintenance Systems in AI since generation is similar to forward system in AI. The generation task appears to be prohibitive in terms of computation overhead, but it may not be so depending on the context of application.

Finally, one should note that DDB's are not yet fully understood ; however they already permit various generalizations of conventional DB's among which generalized notions of views, integrity constraints, query languages, data dependencies studies, etc (29, 30, 31, 32,...). Obviously, not all of these notions have an acceptable treatment : among them one can mention update of views, recursive DR's, checking IC's, etc.

It should be clear from the above discussion how close are some of the problems which are dealt with both from the DB viewpoint and from the logical one ; what to emphasize and how to solve problems, is where these two fields separate.

CONCLUSION

In this overview paper, two main trends for enhancing data bases on one side, logic on the other, have been examined. Both aim at bridging the gap between DB and logic. One puts the emphasis on efficiency, the other on functionalities. As a result there is no single logic & DB system : a taxonomy of systems including DB's, knowledge-based systems, logic interpreters handling large sets of assertions, etc can be developed ; corresponding to this taxonomy which is rather intuitive and well-known, another one has been proposed here according to the emphasis on logic or on DB's : PROLOG+, PROLOGDB, logic DB, deductive DB. Yet, another taxonomy is still to be developed : it has to do with the types of axioms that could be sufficient for the purpose of each type of system corresponding to the above taxonomies. As an example, consider recursive axioms : what is the complexity of such axioms when one adopts the deductive DB perspective ? Isn't it sufficient to have the power of transitive closure ? Then, isn't it possible to take advantage of such a simplification in the deductive system to be built. Such questions are important and the task of finding such a taxonomy is now to be undertaken. It may be presently undertaken in the framework of the Japanese 5G Project which aims at the same objective : bring together logic and database system. One should note that we have not covered the use of logic as an implementation language for interfacing DB's, e.g. for a natural language interface (19) or for menus and other tools (37). Finally recall that an important topic has not been discussed here at all : the knowledge representation problem and the contribution of logic databases to it.

ACKNOWLEDGMENTS

Views expressed here result from many years of studies in common with J.M. NICOLAS, and also discussions with J. MINKER ; our collaboration started with the logic and databases publication and is still continuing. The influence of R. KOWALSKI and of R. REITER's work should be obvious throughout.

REFERENCES

(1) **GREEN C.**, "Theorem proving by resolution as a basis for Question-Answering systems", Machine Intelligence 4 (MELTZER, B., and MICHIE, D., eds), American Elsevier Pub. Co., NEW-YORK (1969), pp. 137-147

(2) **GALLAIRE H.**, **MINKER J.**, eds, "Logic and Data Bases Plenum Press, NEW-YORK (1978).

(3) **NICOLAS J.M.** and **SYRE J.C.**, "Natural question-answering and automatic deduction in the system SYNTEX", Proc. IFIP 74, North-Holland, AMSTERDAM (1974), pp. 595-599

(4) **MINKER J.**, "An Experimental relational database system based on logic" in (2), pp. 107-147

(5) **REITER R.**, "Towards a logical reconstruction of relational database theory", unpublished manuscript

(6) **KOWALSKI R.A.**, "Logic as a database language", Proc. advanced seminar on TIDB, Cetraro (Sept. 1981)

(7) **BRUYNOOGHE M.**, "PROLOG-C implementation", University of LOUVAIN, 1981

(8) **MIYAZAKI N.**, "A data sublanguage approach to inferfacing predicate logic and relational databases", ICOT report, 1982

(9) **CLARK K.L.** and **Mc CABE F.**, "The Control facilities of IC-PROLOG", In "Expert Systems in the Micro Electronic Age" (Ed. MICHIE), EDINBURGH University Press, 1979

(10) **GALLAIRE H.**, **MINKER J.** and **NICOLAS J.M.**, "An overview and introduction to logic and databases", in (2).

(11) **CHAKRAVARTY U.S.**, **MINKER J.** and **TRAN D.**, "Interfacing predicate logic languages and relational databases", Proc. 1st Int. Conf. on logic programming, MARSEILLE (Sept. 1982), pp. 91-98

(12) **KUNIFUJI S.** and **YOKOTA H.**, "PROLOG and relational databases for fifth generation computer systems", ICOT Report presented at CERT 82 workshop "Logical Bases for Databases".

(13) **CHANG C.L.**, "DEDUCE 2 : further investigations of deduction in relational databases", in (2), pp. 201-236

(14) **MINKER J.** and **NICOLAS J.M.**, "On recursive axioms in deductive databases", Information Systems 7,4 (1982)

(15) **HENSCHEN L.** and **NAQVI S.**, "Compiling recursive databases", submitted to JACM (1982)

(16) **SICKEL S.**, "A search technique for clause interconnectivity graphs", IEEE Transactions on computers, Vol. C-25, n° 8, 1976

(17) **NAQVI S., FISHMAN D.** and **HENSCHEN L.J.**, "An Improved compiling technique for first-order databases", Presented at CERT 82 Workshop "Logical Bases for Databases", Bell laboratories and Northwestern University

(18) **KOWALSKI R.**, "Logic Programming", Invited paper IFIP PARIS Sept. 19-23

(19) **WARREN D.H.D.**, "Efficient processing of interactive relational database queries expressed in logic", Proc. 7th VLDB Conf., CANNES (Sept.1981), pp. 272-281

(20) **BOWEN K.A.** and **KOWALSKI R.A.**, "Amalgamating language and meta-language in logic programming", in "Logic programming" (K.I CLARK and S.A. TARNLUND eds), Academic Press, LONDON (1982), pp. 153-172

(21) **NEVES J.C., ANDERSON S.O.** and **WILLIAM H.**, "A PROLOG implementation of Query-by-Example", Proceedings 7th Int. Computing Symposium, March 22-24, 1983, NURNBERG

(22) **LI D.Y.** and **HEATH F.G.**, "ILEX : an intelligent relational database system", HERIOT-WATT University, Dept. of Electrical and Electronic Engineering, EDINBURGH 1982

(23) **GRUMBACH A.**, "Knowledge Acquisition in PROLOG", 1st Int. Logic Programming Conf., Sept. 14-17th, MARSEILLE

(24) **KELLOGG C.** and **TRAVIS L.**, "Reasoning with data in a deductively augmented data management system", in Advances in Database Theory, Vol.1 (Plenum 1981), pp.261-295 (H.GALLAIRE, J. MINKER, J.M. NICOLAS editors)

(25) **NICOLAS J.M.** and **GALLAIRE H.**, "Database : theory vs. interpretation", in (2), pp. 33-54

(26) **BOSSU G.** and **SIEGEL P.**, "La saturation au secours de la non monotonicité", Thèse de 3è cycle, Université de MARSEILLE-LUMINY, MARSEILLE (Jun.-1981), to appear in A.I.

(27) **MINKER J.**, "On indefinite databases and the closed world assumption", Proc. 6th Conf. on Automated Deduction, in Lecture Notes in Computer Science, Vol. 138, Springer-Verlag, NEW-YORK (1982)

(28) **NICOLAS J.M.** and **YAZDANIAN K.**, "An outline of BDGEN : a deductive DBMS", Techn. Rep., ONERA-CERT, TOULOUSE (Oct. 1982)

(29) **NICOLAS J.M.** and **YAZDANIAN K.**, "Integrity checking in deductive databases", in (2), pp. 325-344

(30) **BLAUSTEIN B.T.**, "Enforcing database assertions : techniques and applications", Ph.D. Thesis, HARVARD Univ., CAMBRIDGE (Aug. 1981)

(31) **PIROTTE A.**, "High level database query languages", in (2), pp. 409-436

(32) **FAGIN R.**, "HORN Clauses and databases dependencies", J.ACM 29,4 (Oct. 1982), pp. 952-985

(33) **CLARK K.L.**, "Negation as failure", in (2), pp. 293-322

(34) **GALLAIRE H.**, **MINKER J.** and **NICOLAS J.M.**, "Logic and Databases - An overview and survey", Joint report CERT-CGE-Univ. of MARYLAND

(35) **DEMOLOMBE R.**, "Utilisation du calcul des prédicats comme langage d'interrogation des bases de données", Thèse de doctorat d'Etat, ONERA-CERT, TOULOUSE, Feb. 1982

(36) **REITER R.**, "On closed world databases", in (2), pp. 55-76

(37) **PEREIRA L.**, **FIGUEIRO M** : Relational Databases à la carte, Centro de Informatica, Universidade Nova de Lisboa, PORTUGAL