Logic and Databases

Johann Eder

Universität Klagenfurt, Institut für Informatik
Universitätsstr. 65-67, A-9022 Klagenfurt, Austria
eder@ifi.uni-klu.ac.at

Abstract. Logic and databases have gone a long way together since the advent of relational databases. Already the first basic query languages for relational databases beside relational algebra - tuple calculus and domain calculus - are actually a subset of first order predicate logic. Furthermore logic proved to be very adequate for establishing a sound theory for relational databases.
When attempts were made to integrate AI and database technology in form of expert database systems or knowledge base management systems logic provided a unifying framework although several differences in the use of logic in the both fields have been discovered. The confluence of logic programming and databases triggered deductive databases as new area of research.
In this overview paper we will discuss shortly the relationship between relational databases and logic and present the possibilities for coupling Prolog with databases. The main part of this paper concentrates on deductive databases, in particular on the database language Datalog. We conclude by mentioning some facets of recent research on logic and databases. The goal of this paper is to provide an overview of the most relevant developments in the field and providing pointers to the literature.

1 Introduction

Database systems in general are characterized by the properties enumerated below. These properties define a generic database system and distinguish databases from all other systems.

1. persistence
2. management of secondary memory
3. concurrent manipulation of data
4. reliabilty, security, safety
5. ad hoc queries

The conceptual structure of a database is described in a conceptual schema using a database model. Logic became relevant for the relational data model [Cod70], where data is organized in form of relations, which are defined as follows: Let $D_1, \ldots, D_n$ be sets, called domains, let D be the Cartesian product $D_1, \times \ldots \times D_n$. A *relation* is any subset of D.

The logical schema of a relational database consists of a set of relations, their attributes and domains and a set of integrity constraints which define restrictions on the possible instances of the relations.

Data can be retrieved from relations by applying queries against the database. These queries can be expressed in different languages derived from relational algebra and relational calculus. Relational algebra provides the usual set operations (union, difference, intersection) and relational operations (projection, selection, Cartesian product, join). Relational calculus is based on logic.

In a logical interpretation of relational databases, the relations correspond to predicates of first order predicate logic [GMN81, GM78, GMN84b, GMN84a]. For an example consider the relation R with the attributes A and B. We interpret R as being a predicate. The predicate is true for all instances which are actually stored in R and false for all other (closed world assumption, [Rei78]).

This interpretation is used for relational calculus, a pureley declarative query language [Cod72]. Actually, we have two different languages [Pir78], tuple calculus, where variables range over tuples and domain calculus, where variables range over the elements of the domains. In relational calculus queries are expressed by logical formulas. The result of the query consists of all tuples satisfying this formula.

The following example shows a query on a relation *hasPart* with the attributes $part\#, subpart\#, quantity$, and a relation *parts* with attributes $part\#, name$ retrieving the name and quantity of all subparts of part 123:

$$\{< pn, quan > | hasPart(p\#, sp\#) \wedge parts(sp\#, pn) \wedge p\# = 123\}$$

Most query languages of relational database systems like SQL, QUEL, or QBE have their foundation in relational calculus. The declarative nature of query languages is essential for relational databases, as the actual physical storage structure is hidden from the user. The actual procedural evaluation of a query is performed by the database management system (DBMS) using a query optimizer.

Query languages also allow the definition of derived relations. Since the result of a query is again a relation, a query can be used to define relations whose values are not stored in the database but are retrieved from the actually stored relations. Such relations are called views and form the so called *Intensional Database - IDB*, while the relations actually physically stored are called base relations forming the *Extensional Database - EDB* [GMN84b].

Beside being the foundation of query languages, logic is needed to represent integrity constraints. These are logical formulas which have to be satisfied by the database extensions, i.e. they define all valid database extensions.

Developments in the field of logic programming, expert systems, and knowledge based systems which use logic to represent knowledge in form of facts and rules again triggered research in logic and databases. While those systems are tailored towards representation and processing of knowledge they usually lack the fundamental properties of database systems stated above, although they have to represent, store, and manipulate data like classical database systems.

The already available logical interpretation of databases seemed a good starting point for integrating these technologies. However, relational calculus had severe limitations in expressiveness restricting its use for knowledge processing. While non-recursive rules can be represented in form of view definitions, it is not possible to represent recursive queries or recursive rules in relational calculus [AJ79]. For example it is not possible to explode the bill-of-materials table in the example above.

In a first attempt logic programming systems have been coupled with database systems, such that the database takes care of the data and the logic programming system is responsible for knowledge representation. It turned out, that coupling is very easy from a conceptual point of view, but naive coupling showed to pay big performance penalties due to the mismatch of these systems. Considerable effort has been directed towards improvement of the interfaces. The problems and achievments in coupling Prolog and relational databases are outlined in the following section.

2 Coupling Prolog and Relational Databases

Prolog has a lot in common with relational domain calculus and can, therefore, easily be used as query language for relational databases [Zan86]. As a Turing complete programming language the expressive power of Prolog is strictly greater than that of relational languages. Therefore, it overcomes their restrictions sketched above as the rules of Prolog powerfully enrich the possibility for defining and evaluating derived relations. In particular, predicates and rules in Prolog can be recursively defined. When using Prolog as query language, the tuples in relations of the database are considered as facts in Prolog.

Nevertheless, coupling Prolog and relational databases show some dissonances. Facts and rules in Prolog are organized in a total order and the semantics of a Prolog program depends on this order. In contrast, relations in a database are considered as unordered sets of tuples and the result of a query is independent from any physical order. The processing of Prolog programs is tuple oriented while relational databases are set oriented. Prolog offers procedural features like the cut predicate to allow the programmer to control the inference process. The order of evaluation of a Prolog program is pre-determined, whereas expressions in relational calculus are purely declarative and the actual evaluation is left to a query processor which may rearrange the query for optimization purposes. Optimization of queries was crucial for the success of relational databases. The procedural nature of the Prolog engine leaves the burden of optimization with the programmer.

To give an example, consider the evaluation of a join operation in Prolog:

$$? - p(X, Y), q(Y, a)$$

The inference process of Prolog is equivalent to the nested loop algorithm for evaluating joins, while other algorithms (sort-merge, hash join) are much more efficient. Prolog makes little use of the binding of an attribute to a constant (a in our example) and cannot rearrange the query while rewriting the query as $? - q(Y, a), p(X, Y)$ might reduce evaluation cost by orders of magnitude. The Prolog engine has no information about the physical storage structure and cannot make use of indices. Furthermore, the tuple-at-a-time processing of Prolog leads to poor buffer performance.

Different approaches of coupling Prolog and databases have been presented (e.g. [CW86, Boc86, CD88, Deno86, CGW89, Qui87, WW88]. They differ in the Prolog engine (especially how the main memory database is managed), the Prolog interface (how database predicates are discovered, ranging from full transparency- i.e. database predicates are recognized without user support - to no transparency, where the programmer has to write explicit queries in a database language), the database interface (how complex the queries sent to the database may be, ranging from simple single-predicate queries to unrestricted complex -even recursive - queries, and the way query results are returned, either single tuples, or the whole result sets), and the database engine (the level of the query language offered from the database). Beside conceptual considerations of the interaction, engineering techniques like cacheing of data and queries are used to improve the performance of the coupled system.

In *loosely coupled systems* the interaction between the Prolog engine and the database takes place at load time, i.e. the Prolog engine recognizes uninstantiated database predicates, issues according queries to the database and asserts the results as facts to the program. During the execution of the Prolog program the database is not consulted.

In *tightly coupled systems* Prolog and the database interact during the inference process. In less sophisticated systems each time the Prolog engine tries to satisfy a database predicate, a query is sent to the database. In more sophisticated systems the Prolog engine discovers succeeding database and comparison predicates (base conjunctions) and transforms them together to a single query which is then transfered to the database. This approach overcomes some of the problems detailed above.

Coupling approaches achieved interesting results by improving the interface between relational databases and Prolog. Nevertheless, substantial problems remain, Most researchers agree that deep integration of database management and rule processing is required.

3 Datalog

Datalog is a language for deductive databases which are deductive systems in the sense of AI and logic with the characteristics of database systems. It is a simplified logic programming language which is integrated with database management. From the point of view of logical programming a Datalog program consist of function free Horn clauses, from a database point of view it is a relational domain calculus language extended with recursion.

3.1 Pure Datalog

The syntax of Datalog is similar to that of Prolog. Datalog distinguishes two sets of predicates: *extensional predicates* which are the relations stored in the database and *intensional predicates* defined by rules in the Datalog program. Facts are the tuples of extensional predicates. Rules are represented in the general pattern:

$$L_0 : -L_1, \ldots, L_n$$

Each L_i is a literal of the form $P_i(t_1, \ldots, t_k i)$ such that p_i is a predicate symbol and the t_j are terms. A term may only be a constant or a variable, but not a functor term as in other logic programming languages. Like in Prolog the name of a variable has to start with a capital letter while constants and predicate names start with a lower case letter. We call L_o the head of the rule and $L_1, \ldots, L_n$ the body of the rule. A literal, fact, rule, or clause without any variable is called ground.

The extensional database (EDB) of a Datalog program consists of all ground facts physically stored in the database. The intensional database (IDB) is Datalog program (P) consisting of rules. EDB-predicates may appear in P only in the body of rules, IDB-predicates in the head and in the body.

To assure that all results of Datalog programs are finite any Datlog program has to obey the following safety conditions:

- Each fact of P is ground.
- Each variable occuring in the head of a rule must also occur in its body

The following example gives a solution to the problem of exploding the relation $hasPart$ of the example in section 2 to a bill-of-material:

$bill(P, SubP) : -hasPart(P, SubP);$
$bill(P, SubP) : -bill(P, I), hasPart(I, SubP)$

A goal is represented by a list of literals of the following pattern:

$$L_1, \ldots, L_n$$

where the L_i are defined as above. (Some authors require goals to consist of only one literal. This is no restriction, since we can write a goal-rule with a special goal predicate as it's head and the multi-literal goal as its body. And then we use the goal-predicate as literal for the goal.)

In database terms, a Datalog program defines a set of views. A query is represented as goal. Materialization of these views for query processing is the task of a Datalog system.

Datalog programs are interpreted in First-Order Logic (FOL) as follows:

- Each fact F corresponds to an atomic formula in FOL.
- Each rule $R : L_0 : -L_1, \ldots, L_n$ corresponds to a FOL formula $\forall X_1 \ldots \forall X_m (L_1 \wedge \ldots \wedge L_n \Rightarrow L_0)$, where $X_1, \ldots, X_m$ are all variables occuring in R.
- A set of Datalog clauses corresponds to the conjunction of all corresponding formulas.

The *Herbrand Base HB* of a Datalog Programm is the set of all ground facts which can be expressed. We divide HB in its extensional part *EHB* which consists of all ground facts with EDB predicates, and accordingly, in its intensional part *IHB*.

The semantics of a Datalog program P is defined as a mapping $\mathcal{M}_P$ from EHB to IHB which maps each possible EDB to the set of intensional result facts, i.e the set of all facts of IHB which are logical consequences of the corresponding formulas of P and EDB. In the presence of a goal this set is restricted to all facts subsumed by the goal.

In a model theoretic interpretation, $\mathcal{M}_P$ is defined by the least Herbrand model of $P \cup EDB$.

The least Herbrand model is the disjunction of all Herbrand models. A Herbrand model is a Herbrand interpretation which satisfies $P \cup EDB$. A Herbrand interpretation is a subset of the Herbrand Base, i.e. all ground facts which are considered as being true. A Herbrand interpretation $\mathcal{I}$ satisfies EDB, if all facts of EDB are also in $\mathcal{I}$. It satisfies a rule $L_0 : -L_1, \ldots, L_n$ of P, iff for each substitution θ of the rule which replaces variables with constants, whenever all literals of the body of the substituted rule are in $\mathcal{I}$ also its head literal is in $\mathcal{I}$.

In a proof theoretic interpretation the semantics of a Datalog program is defined as the set of all ground facts which can be derived by successive application of the elementary production principle (EPP) on P and EDB. EPP is a meta rule defining which facts can be derived from a rule and a set of facts in one step. Consider a Datalog rule $R : L_0 : -L_1, \ldots, L_n$ and a set of ground facts $F = \{F_1, \ldots, F_n\}$. For a substitution θ of R we can infer the fact represented by the substituted head literal in a one step, if all substituted literals in the body are in the set of known facts. If we apply EPP to all rules and all known facts we derive all facts which may be infered from P and F in one step and we can take them into the set of known facts. Applying this procedure in turn delivers all facts which are logical consequences of F and P.

The proof theoretic semantics leads directly to an evaluation procedure for Datalog programs by fixpoint iteration. Starting with F = EDB and applying this procedure until no new facts can be infered from F and P with EPP delivers the materialization of all IDB predicates. For all Datalog programs this evaluation procedure will terminate with a finite result.

3.2 Optimization

Although the proof theoretic interpretation of Datalog programs leads directly to an evaluation procedure, this method of processing queries is very inefficient. In the last years various techniques for efficient evaluation have been developed and research on optimization methods still go on. Here, we only can give a rough overview of these techniques.

According to [CGT90] the techniques can be classified according to the following criteria:

- *formalism*: Some methods use logical formalism, while others transform a Datalog Program to a set of algebraic equations and evaluate these equations.
- *search strategy*: Bottom-up methods start from the facts of EDB and infer new facts, while Top-down methods start from the goal and search for facts which satisfy the premises of a rule yielding the goal as conclusion. Within the top-down approach we distinguish between depth first and breadth first search.
- *objectives*: In pure evaluation methods the optimization is done during the evaluation. Rewriting methods map a Datalog program P to another program P' which is more efficient to evaluate with a basic evaluation procedure.

- *type of considered information*: Syntactic optimization methods consider only the syntactic structure of the program while semantic optimization methods take additional semantic knowledge about the database (e.g. integrity constraints) into account.

[BR86] gives an overview and comparison of different evaluation and optimization techniques. Important evaluation techniques are the seminaive evaluation (or differential fixpoint evaluation) [GKB87, HQC88] the method of Henschen and Naqvi [HN84] and the Query-Subquery algorithm [Vie86, Vie88]. The most renowned optimization algorithms include the Magic Set method [BMSU86, BR87, Ram88, Sag91], the Counting method [SZ88], and Static Filtering [KL86].

3.3 Extensions of Pure Datalog

An important extension of pure Datalog is the use of negation in rule bodies. With this extensions Datalog no longer requires all clauses to be Horn clauses. A drawback of the use of negation is that a unique minimal Herbrand model is not guaranteed. Stratified Datalog allows only a restricted use of negation. If it is possible to arrange the predicates of a Datalog program in a series of numbered sets called strata such that no predicate of a lower stratum appears in the body of any rule with a predicate from a higher stratum as head predicate we call the program stratified.

To determine whether a program is stratified an extended dependency graph is constructed. The nodes in this graph are the intensional predicates of the program. A directed arc is drawn from p to q, if q appears in the body of a rule with p as head predicate. If q is negated in one of the rule, the arc is marked with $\neg$. If ther is no cylcle in the graph containing a marked arc, the program is stratified.

Stratified programs are evaluated bottom up in the sequence of their strata. This procedure uniquely defines a Herbrand model as result of the program. Although there may exist several stratifications for a Datalog program with negation they all have the same result. Note, however, that not all programs with negated literals in the body are stratified.

Further extensions of Datalog include the definition of built in predicates like $=, >, <$, or for arithmetic operations $(+, *, -, /)$, the use of functors for dealing with complex objects, or the the use of sets as arguments. Such extensions can be studied in LDL, a deductive database systems implementing extended Datalog [NT89].

4 Conclusions

Logic turned out to provide a solid and fruitful basis for the integration of database technology with knowledge processing capabilities. Research in logic and databases brought theoretically sound foundations for the building of deductive database systems. Several prototype implementations are in development or have been already presented, among them EDUCE and EKS-V1 from ECRC, LDL from MCC, and NAIL! from Stanford University. Ongoing research in the field includes the development of yet more sophisticated optimization techniques, higher order deduction (eg.

[ERMS91], and the application of logic to object oriented databases (e.g. [Bee90]. Deductive databases will have a major impact on future knowledge based systems.

We will conclude with some references for further reading. The topic of logic and databases is subject of a specialized textbook [CGT90], and is also extensively covered in [Ull89]. [CGT89] provides an introduction to Datalog, [NT89] presents the logic database language LDL, an already implemented extension of Datalog. An Overview of logic and databases can be found in [GM92], [UZ90] discuss achievements and furture directions of research in the field.

References

[AJ79] A. Aho and J.Ullmann. Universality of data retrieval languages. In *Proc. ACM Symp. on Principles of Programming Languages*, 1979.

[Bee90] C. Beeri. Formal methods for object oriented databases. *Data & Knowledge Engineering*, 5, 1990.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullmann. Magic sets and other strange ways to implement logic programs. In *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986.

[Boc86] J. Bocca. On the evaluation strategy of EDUCE. In *Proc. ACM SIGMOD Conf.*, 1986.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing. In *Proc. ACM SIGMOD Conference*, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the power of Magic. In *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1987.

[BR88] F. Bancilhon and R. Ramakrishnan. Performance evaluation of data intensive logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[CD88] F. Cuppens and R. Demolombe. A Prolog-relational DBMS interface using delayed evaluation. In C. Beeri, J.W. Schmidt, and U. Dayal, editors, *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, Jerusalem, 1988.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer Verlag, New York, 1990.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared ask). *IEEE Trans. on Knowledge and Data Eng.*, 1(1), 1989.

[CGW89] S. Ceri, G. Gottlob, and G. Wiederhold. Efficient database access through Prolog. *IEEE Trans. on Software Engineering*, 1989.

[Cod70] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[Cod72] E. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[CW86] C.L. Chan and A. Walker. PROSQL: A Prolog programming interface with SQL/DS. In L. Kerschberg, editor, *Expert Database Systems*. Bejamin-Cummings, 1986.

[Deno86] D. Denoel, et al. Query translation for coupling Prolog with a relational DBMS. In *Workshop on Integration of Logic Programming ans Databases*, Venice, 1986.

[ERMS91] J. Eder, A. Rudloff, F. Matthes, and J.W. Schmidt. Data construction with recursive set expressions. In J. W. Schmidt and A.A. Stogny, editors, *Next Generation Information System Technology*, LNCS 504. Springer Verlag, 1991.

[GMN81] H. Gallaire, J. Minker, and J-M. Nicolas, editors. *Advances to Database Theory*, volume I. Plenum Press, 1981.

[GKB87] U. Güntzer, W. Kiessling, and R. Bayer. On the evaluation of recursion in (deductive) database systems by efficient differential fixpoint iteration. In *Proc. 3rd Intern. Conf on Data Engineering.* IEEE-CS Press, 1987.

[GM78] H. Gallaire and J. Minker, editors. *Logic and Databases.* Plenum Press, 1978.

[GM92] J. Grant and J. Minker. The impact of logic programming on databases. *Comm. of the ACM*, 35(3), 1992.

[GMN84a] H. Gallaire, J. Minker, and J-M. Nicolas, editors. *Advances in Database Theory, Vol II.* Plenum Press, 1984.

[GMN84b] H Gallaire, J. Minker, and J-M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 2, 1984.

[HN84] L. J. Henschen and S.A. Naqvi. On compiling queries in recursive first order databases. *Journal of the ACM*, 31(1), 1984.

[HQC88] J. Han, G. Qadahand, and C. Chaou. The processing and evaluation of transitive closure. In J.W. Schmidt et al., editor, *Advances in Database Technology*, LNCS 303. Springer Verlag, 1988.

[KL86] M. Kifer and E. L. Lozinski. Filtering data flow in deductive database systems. In *Proc. 1st Int. Conf. on Database Theory*, Rome, 1986.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, New York, 1989.

[Pir78] A. Pirotte. High level database query languages. In Gallaire and Minker [GM78].

[Qui87] Quintus Computer Systems Inc., Mountain View, California. *Quintus Prolog Data Base Interface Manual*, 1987.

[Ram88] R. Ramakrishnan. Magic Templates, a spellbinding aproach to logic evaluation. In *Proc. of the Logic Programming Conf.*, 1988.

[Rei78] R. Reiter. On closed world databases. In Gallaire and Minker [GM78].

[Sag91] Y. Sagiv. On testing effective computability of Magic programs. In C. Delobel, M Kiferd Y. Masunaga, editors, *Proc. 2nd Int. Conf. Deductive and Object-Oriented Databases*, volume 566 of *LNCS.* Springer Verlag, 1991.

[SZ88] D. Sacca and C. Zaniolo. Magic counting methods. In *Proc. ACM SIGMOD Conf.*, 1988.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I + II.* Computer Science Press, 1988/89.

[UZ90] J.D. Ullmann and C. Zaniolo. Deductive databases: Achievements and future directions. *ACM SIGMOD Record*, 19(4), 1990.

[Vie86] L. Vieille. Recursive axioms in deductive databases: The Query-Subquery approach. In L. Kerschberg, editor, *Proc. Int. Conf. Expert Database Systems*, Charlston, 1986.

[Vie88] L. Vieille. From QSQ to QoSaQ: Global optimization of recursive queries. In *Proc. 2nd Int. Conf Expert Database Systems*, Tyson Corner, 1988.

[WW88] K.F. Wong and M.H. Williams. Design considerations for a Prolog database engine. In C. Beeri, J.W. Schmidt, and U. Dayal, editors, *Proc. 3rd Intern. Conf. on Data and Knowledge Bases*, Jerusalem, 1988.

[Zan86] C. Zaniolo. Prolog: A database language for all seasons. In L. Kerschberg, editor, *Expert Database Systems.* Benjamin-Cummings, 1986.

This article was processed using the LaTeX macro package with LMAMULT style