Chapter 1

Logic Programming and Databases: An Overview

This book deals with the *integration of logic programming and databases to generate new types of systems,* which extend the frontiers of computer science in an important direction and fulfil the needs of new applications. Several names are used to describe these systems:

a) The term *deductive database* highlights the ability to use a logic programming style for expressing deductions concerning the content of a database.
b) The term *knowledge base management system (KBMS)* highlights the ability to manage (complex) knowledge instead of (simple) data.
c) The term *expert database system* highlights the ability to use expertise in a particular application domain to solve classes of problems, but having access over a large database.

The confluence between logic programming and databases is part of a general trend in computer science, where different fields are explored in order to discover and profit from their common concepts.

Logic programming and databases have evolved in parallel throughout the seventies. *Prolog*, the most popular language for PROgramming in LOGic, was born as a simplification of more general theorem proving techniques to provide efficiency and programmability. Similarly, the relational data model was born as a simplification of complex hierarchical and network models, to enable set-oriented, nonprocedural data manipulation. Throughout the seventies and early eighties, the use of both *Prolog* and relational databases has become widespread, not only in academic or scientific environments, but also in the commercial world.

Important studies on the relationships between logic programming and relational databases have been conducted since the end of the seventies, mostly from a theoretical viewpoint. The success of this confluence has been facilitated by the fact that *Prolog* has been chosen as the programming language paradigm within the Japanese *Fifth Generation Project.* This project aims at the development of the so-called "computers of the next generation", which will be specialized in the execution of Artificial Intelligence applications, hence capable of performing an extremely high number of *deductions per time unit.* The project also includes the use of the relational data model for storing large collections of data.

The reaction to the Japanese *Fifth Generation Project* was an incentive to research in the interface area between logic programming and relational databases. This choice indicated that this area is not just the ground for theoretical investigations, but also has great potential for future applications.

S. Ceri et al., *Logic Programming and Databases*
© Springer-Verlag Berlin Heidelberg 1990

By looking closely at logic programming and at database management, we discover several features in common:

a) *DATABASES.* Logic programming systems manage small, single-user, main-memory databases, which consist of deduction rules and factual information. Database systems deal instead with large, shared, mass-memory data collections, and provide the technology to support efficient retrieval and reliable update of persistent data.
b) *QUERIES.* A query denotes the process through which relevant information is extracted from the database. In logic programming, a query (or *goal*) is answered by building chains of deductions, which combine rules and factual information, in order to *prove* or *refute* the validity of an initial statement. In database systems, a query (expressed through a special-purpose data manipulation language) is processed by determining the most efficient access path in mass memory to large data collections, in order to extract relevant information.
c) *CONSTRAINTS.* Constraints specify correctness conditions for databases. Constraint validation is the process through which the correctness of the database is preserved, by preventing incorrect data being stored in the database. In logic programming, constraints are expressed through general-purpose rules, which are activated whenever the database is modified. In database systems, only a few constraints are typically expressed using the data definition language.

Logic programming offers a greater power for expressing queries and constraints as compared to that offered by data definition and manipulation languages of database systems. Furthermore, query and constraint representation is possible in a homogeneous formalism and their evaluation requires the same inferencing mechanisms, hence enabling more sophisticated reasoning about the database content. On the other hand, logic programming systems do not provide the technology for managing large, shared, persistent, and reliable data collections.

The natural extension of logic programming and of database management consists in building new classes of systems, placed at the intersection between the two fields, based on the use of *logic programming as a query language.* These systems combine a logic programming style for formulating queries and constraints with database technology for efficiency and reliability of mass-memory data storage.

1.1 Logic Programming as Query Language

We give an informal presentation of how logic programming can be used as a query language. We consider a relational database with two relations:

$$PARENT(PARENT, CHILD), \quad \text{and} \quad PERSON(NAME, AGE, SEX).$$

The tuples of the $PARENT$ relation contain pairs of individuals in parent-child relationships; the tuples of the $PERSON$ relation contain triples whose first,

PARENT	
PARENT	*CHILD*
john	*jeff*
jeff	*margaret*
margaret	*annie*
john	*anthony*
anthony	*bill*
anthony	*janet*
mary	*jeff*
claire	*bill*
janet	*paul*

PERSON		
NAME	*AGE*	*SEX*
paul	*7*	*male*
john	*78*	*male*
jeff	*55*	*male*
margaret	*32*	*female*
annie	*4*	*female*
anthony	*58*	*male*
bill	*24*	*male*
janet	*27*	*female*
mary	*75*	*female*
claire	*45*	*female*

Fig. 1.1. Example of relational database

second, and third elements are the person's name, age, and sex, respectively. We assume that each individual in our database has a different name. The content of the database is shown in Fig. 1.1.

We express simple queries to the database using a logic programming language. We use *Prolog* for the time being; we assume the reader has some familiarity with *Prolog*. We use two special *database predicates, parent* and *person* with the understanding that the ground clauses (facts) for these predicates are stored in the database. We use standard *Prolog* conventions on upper and lower case letters to denote variables and constants. For instance, the tuple *<john, jeff>* of the database relation PARENT corresponds to the ground clause:

$$parent(john,jeff).$$

The query: *Who are the children of John?* is expressed by the following *Prolog* goal:

$$? - parent(john, X).$$

The answer expected from applying this query to the database is:

$$X = \{jeff,\ anthony\}.$$

Let us consider now which answer would be given by a *Prolog* interpreter, operating on facts for the two predicates *parent* and *person* corresponding to the database tuples; we assume facts to be asserted in main memory in the order shown above.

The answer is as follows: After executing the goal, the variable X is first set equal to *jeff*; if the user asks for more answers, then the variable X is set equal to *anthony*; if the user asks again for more answers, then the search fails, and the interpreter prompts *no*. Note that *Prolog* returns the result one tuple at a time, instead of returning the set of all result tuples.

The query: *Who are the parents of Jeff?* is expressed as follows:

$$? - parent(X,jeff).$$

The set of all answers is:

$$X = \{john, mary\}.$$

Once again, let us consider the *Prolog* answer: After executing this goal, the variable X is set equal to *john*; if the user asks for more answers, then the variable X is set equal to *mary*; if the user asks again for more answers, then the search fails.

We can also express queries where all arguments of the query predicate are constants. For instance:

$$? - parent(john,jeff).$$

In this case, we expect a positive answer if the tuple *<john, jeff>* belongs to the database, and a negative answer otherwise. In the above case, a *Prolog* system would produce the answer *yes*.

Rules can be used to build an *Intensional Database (IDB)* from the *Extensional Database (EDB)*. The EDB is simply a relational database; in our example it includes the relations PARENT and PERSON. The IDB is built from the EDB by applying rules which define its content, rather than by explicitly storing its tuples. In the following, we build an IDB which includes the relations FATHER, MOTHER, GRANDPARENT, SIBLING, UNCLE, AUNT, ANCESTOR, and COUSIN. Intuitively, all these relationships among persons can be built from the two EDB relations PARENT and PERSON.

We start by defining the relations FATHER and MOTHER, by indicating simply that a father is a male parent and a mother is a female parent:

$$father(X,Y) \ :- \ person(X,_,male),\ parent(X,Y).$$
$$mother(X,Y) \ :- \ person(X,_,female),\ parent(X,Y).$$

As a result of this definition, we can deduce from our sample EDB the IDB shown in Fig. 1.2.

Note that here we are presenting the tuples of the IDB relations as if they actually existed; in fact, tuples of the IDB are not stored. One can regard the two rules *father* and *mother* above as *view definitions*, i.e., programs stored in the database which enable us to build the tuples of *father* starting from the tuples of *parent* and *person*.

The IDB can be queried as well; we can, for instance, formulate the query: *Who is the mother of Jeff?*, as follows:

$? - mother(X,jeff).$

FATHER	
FATHER	*CHILD*
john	*jeff*
jeff	*margaret*
john	*anthony*
anthony	*bill*
anthony	*janet*

MOTHER	
MOTHER	*CHILD*
margaret	*annie*
mary	*jeff*
claire	*bill*
janet	*paul*

Fig. 1.2. The IDB relations FATHER and MOTHER

With a *Prolog* interpreter, after the execution of this query X is set equal to *mary*. Notice that the interpreter does not evaluate the entire IDB relation MOTHER in order to answer the query, but rather it finds just the tuple which contributes to the answer.

We can proceed with the definition of the IDB relations GRANDPARENT, SIBLING, UNCLE, and AUNT, with obvious meanings:

$$grandparent(X, Z) \ :- \ parent(X, Y), \ parent(Y, Z).$$
$$sibling(X, Y) \ :- \ parent(Z, X), \ parent(Z, Y), \ not(X = Y).$$
$$uncle(X, Y) \ :- \ person(X, _, male), \ sibling(X, Z), \ parent(Z, Y).$$
$$aunt(X, Y) \ :- \ person(X, _, female), \ sibling(X, Z), \ parent(Z, Y).$$

Complex queries to the EDB and IDB can be formulated by building new rules which combine EDB and IDB predicates, and then presenting goals for those rules; for instance, *Who is the uncle of a male nephew?* can be formulated as follows:

$$query(X) \ :- \ uncle(X, Y), \ person(Y, _, male).$$
$$?-\ query(X).$$

More complex IDB relations are built from *recursive rules*, i.e., rules whose head predicate occurs in the rule body (we will define recursive rules more precisely below). Well-known examples of recursive rules are the ANCESTOR relation and the COUSIN relation.

The ANCESTOR relation includes as tuples all ancestor-descendent pairs, starting from parents.

$$ancestor(X, Y) \ :- \ parent(X, Y).$$
$$ancestor(X, Y) \ :- \ parent(X, Z), ancestor(Z, Y).$$

The COUSIN relation includes as tuples either two children of two siblings, or, recursively, two children of two previously determined cousins.

$cousin(X,Y)$:- $parent(X1,X)$, $parent(Y1,Y)$, $sibling(X1,Y1)$.
$cousin(X,Y)$:- $parent(X1,X)$, $parent(Y1,Y)$, $cousin(X1,Y1)$.

The IDB resulting from the two definitions above is shown in Fig. 1.3.

ANCESTOR	
ANCESTOR	*DESCENDENT*
john	*jeff*
jeff	*margaret*
margaret	*annie*
john	*anthony*
anthony	*bill*
anthony	*janet*
mary	*jeff*
claire	*bill*
janet	*paul*
john	*margaret*
mary	*margaret*
jeff	*annie*
john	*bill*
john	*janet*
anthony	*paul*
john	*annie*
mary	*annie*
john	*paul*

COUSIN	
PERSON1	*PERSON2*
margaret	*bill*
margaret	*janet*
annie	*paul*

Fig. 1.3. The IDB relations ANCESTOR and COUSIN

This example shows that recursive rules can generate rather large IDB relations. Furthermore, the process of generating IDB tuples is quite complex; for instance, a *Prolog* interpreter operating on the query *ancestor(X, Y)* would generate some of the IDB tuples more than once. Therefore, the efficient computation of recursive rules is quite critical. On the other hand, recursive rules are very

important because they enable us to derive useful IDB relations that cannot be expressed otherwise.

Rules can also express integrity constraints. Let us consider the EDB relation PARENT; we would like to express the following constraint:

a) *SelfParent constraint:* a person cannot be his (her) own parent.

The formulation of this constraint in *Prolog* is as follows:

a) $incorrectdb : -parent(X, X).$

This constraint formulation enables us to inquire about the correctness of the database. For instance, consider the *Prolog* goal:

$? - incorrectdb.$

If no individual X exists satisfying the body of the rule then the answer to this query is *no.* In this case, the database is correct. If, instead, such an individual does exist, then the answer is *yes.*

Let us consider a few more examples of constraints. For instance:

b) $OneMother:$ *Each person has just one mother.*
c) $PersonParent:$ *Each parent is a person.*
d) $PersonChild:$ *Each child is a person.*
e) $AcyclicAncestor:$ *A person cannot be his(her) own ancestor.*

These constraints are formulated as follows:

b) $incorrectdb \; : - \; mother(X, Z), \; mother(Y, Z), \; not(X = Y).$
c) $incorrectdb \; : - \; parent(X, _), \; not(person(X, _, _)).$
d) $incorrectdb \; : - \; parent(_, Y), \; not(person(Y, _, _)).$
e) $incorrectdb \; : - \; ancestor(X, X).$

Note that constraints b) and e) also use in their formulation some IDB relations, while contraints a), c), and d) refer just to EDB relations. The two cases, however, are not structurally different. Moreover, note that constraint b) is a classic functional dependency, while constraints c) and d) express inclusion dependencies (also called referential integrity).

Let us consider a collection of constraints of this nature. Constraint evaluation can be used either to preserve the integrity of an initially correct database, or to determine (and then eliminate) all sources of inconsistency.

Let us consider the former application of constraints, namely, how to preserve the integrity of a correct database. We recall that the content of a database is changed by the effects of the execution of *transactions.* A transaction is an atomic unit of execution, containing several operations which insert new tuples, delete existing tuples, or change the content of some tuples. Atomicity of transactions means that their execution can terminate either with an *abort* or with a *commit.* An abort leaves the initial database state unchanged; a commit leaves the database in a final state in which all operations of the transaction are successfully performed. Thus, to preserve consistency, we should accept the

commit of a transaction iff it produces a final database state that does not violate any constraint. Efficient methods have been designed for testing the correctness of the final state of a transaction. These methods assume the database to be initially correct, and test integrity constraints on the part of the database that has been modified by the transaction.

Let us consider, then, the application of constraints to restore a valid database state. The above constraint formulation enables a *yes/no* answer, which is not very helpful for such purposes. However, we might, for instance, restate the constraints as follows:

a) $incorrectdb(selfparent, [X]) \ :- \ parent(X, X).$
b) $incorrectdb(onemother, [X, Y, Z]) \ :-$
$\qquad mother(X, Z),\ mother(Y, Z),\ not(X = Y).$
c) $incorrectdb(personparent, [X]) \ :- \ parent(X, _),\ not(person(X, _, _)).$
d) $incorrectdb(personchild, [Y]) \ :- \ parent(_, Y),\ not(person(Y, _, _)).$
e) $incorrectdb(acyclicancestor, [X]) \ :- \ ancestor(X, X).$

In this formulation, the head of the rule has two arguments; the first argument contains the constraint name, and the second argument contains the list of variables which violate the constraint. This constraint formulation enables us to inquire about the causes of incorrectness of the database. For instance, consider the *Prolog* goal:

$$?-incorrectdb(X, Y).$$

If there exists no constraint X which is invalidated, then the answer to this query is *no*. In this case, the database is correct. If instead one such constraint exists, then variables X and Y are set equal to the constraint name and the list of values of variables which cause constraint invalidity. For instance, the answer:

$$X = personparent, Y = [Karen]$$

reveals that Karen belongs to the relation PARENT but not to the relation PERSON; this should be fixed by adding a tuple for Karen to the relation PERSON.

However, the answer to the above query might not be sufficient to understand the action required in order to restore the correctness of the database. This happens with rules b) and e), which express a constraint on IDB relations. We have already observed that IDB relations are generally not stored explicitly; they are defined by rules, and their value depends on the EDB relations which appear in these rules. Thus, violations to constraints b) and e) should be compensated by actions applied to the underlying EDB relations.

We conclude this example by showing, in Table 1.1, the correspondence between similar concepts in logic programming and in databases that we have seen so far:

DATABASE CONCEPTS	LOGIC PROGRAMMING CONCEPTS
Relation	Predicate
Attribute	Predicate argument
Tuple	Ground clause (fact)
View	Rule
Query	Goal
Constraint	Goal (returning an expected truth value)

Table 1.1. Correspondence between similar concepts in logic programming and in databases

1.2 Prolog and Datalog

In the previous section, we have shown how a *Prolog* interpreter operates on a database of facts, and we have demonstrated that *Prolog* can act as a powerful database language. The choice of *Prolog* to illustrate the usage of logic programming as a database language is almost mandatory, since *Prolog* is the most popular logic programming language. On the other hand, the use of *Prolog* in this context also has some drawbacks, which have been partially revealed by our example:

1) *Tuple-at-a-time processing.* While we expect that the result of queries over a database be a *set* of tuples, *Prolog* returns individual tuples, one at a time.
2) *Order sensitivity and procedurality.* Processing in *Prolog* is affected by the order of rules or facts in the database and by the order of predicates within the body of the rule. In fact, the *Prolog* programmer uses order sensitivity to build efficient programs, thereby trading the so-called *declarative nature* of logic programming for procedurality. Instead, database languages (such as SQL or relational algebra) are *nonprocedural*: the execution of database queries is insensitive to order of retrieval predicates or of database tuples.
3) *Special predicates.* *Prolog* programmers control the execution of programs through special predicates (used, for instance, for input/output, for debugging, and for affecting backtracking). This is another important loss of the declarative nature of the language, which has no counterpart in database languages.
4) *Function symbols.* *Prolog* has function symbols, which are typically used for building recursive functions and complex data structures; neither of these applications are useful for operating over a *flat* relational database, although they might be useful for operating over *complex database objects.* We will not address this issue in this book.

These reasons motivate the search for an alternative to *Prolog* as a database and logic programming language; such an alternative is the new language *Datalog.*

Datalog is a logic programming language designed for use as a database language. It is nonprocedural, set-oriented, with no order sensitivity, no special predicates, and no function symbols. Thus, *Datalog* achieves the objective of eliminating all four drawbacks of *Prolog* defined above.

Syntactically, *Datalog* is very similar to pure *Prolog*. All *Prolog* rules listed in the previous section for expressing queries and constraints are also valid *Datalog* rules. Their execution produces the *set* of all tuples in the result; for instance, after executing the goal:

$$? - parent(john, X).$$

We obtain:

$$X = \{ \textit{jeff}, \textit{anthony} \}.$$

As an example of the difference between *Prolog* and *Datalog* in order sensitivity, consider the following two programs:

Program Ancestor1:

$ancestor(X, Y) \; :- \; parent(X, Y).$
$ancestor(X, Y) \; :- \; parent(X, Z), \; ancestor(Z, Y).$

Program Ancestor2:

$ancestor(X, Y) \; :- \; ancestor(Z, Y), \; parent(X, Z).$
$ancestor(X, Y) \; :- \; parent(X, Y).$

Input goal:

$$? - ancestor(X, Y).$$

Both Ancestor1 and Ancestor2 are syntactically correct programs either in *Prolog* or in *Datalog*. *Datalog* is neither sensitive to the order of rules, nor to the order of predicates within rules; hence it produces the correct expected answer (namely, the set of all ancestor-descendent pairs) in either version. A *Prolog* interpreter, instead, produces the expected behavior in version Ancestor1 (namely, the first ancestor-descendent pair); but it loops forever in version Ancestor2. In fact, the *Prolog* programmer must avoid writing looping programs, while the *Datalog* programmer need not worry about this possibility.

The process that has led to the definition of *Datalog* is described in Fig. 1.4. The picture was shown by Jeff Ullman at Sigmod 1984; it indicates that the evolution from *Prolog* to *Datalog* consists in going from a procedural, record-oriented language to a nonprocedural, set-oriented language; that process was also characteristic of the evolution of database languages, from hierarchic and network databases to relational databases. Even though *Datalog* is a declarative language and its definition is independent of any particular search strategy, *Datalog* goals are usually computed with the *breadth-first search strategy*, which produces the *set of all answers*, rather than with the *depth-first search strategy* of *Prolog*, which produces answers with a tuple-at-a-time approach. This is consistent with the set-oriented approach of relational query languages. Furthermore,

in *Datalog* the programmer does not need to specify the *procedure* for accessing data, which is left as system responsibility; this is again consistent with relational query languages, which are nonprocedural.

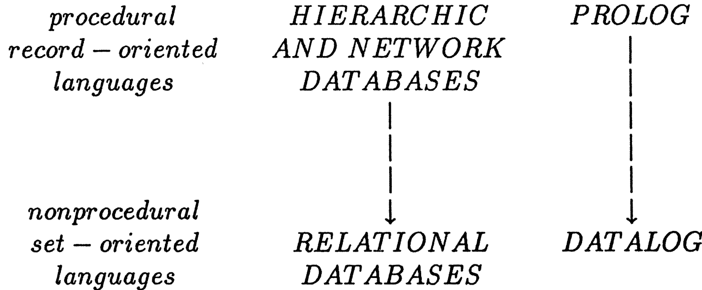

Fig. 1.4. Datalog as an evolution of Prolog

On the other hand, these features limit the power of *Datalog* as a general-purpose programming language. In fact, *Datalog* is obtained by subtracting some features from *Prolog*, but not, for the time being, by adding to it features which belong to classical database languages. Therefore, *Datalog* is mostly considered a good abstraction for illustrating the use of logic programming as a database language, rather than a full-purpose language. We expect, however, that *Datalog* will evolve to incorporate a few other features and will turn into a full-purpose database language in the near future.

In Table 1.2, we summarize the features that characterize *Datalog* in contrast to *Prolog*.

PROLOG	DATALOG
Depth-first search	(usually) Breadth-first search
Tuple-at-a-time	Set-oriented
Order sensitive	No order sensitivity
Special predicates	No special predicates
Function symbols	No function symbols

Table 1.2. Comparison of Prolog and Datalog

1.3 Alternative Architectures

Turning *Prolog* and *Datalog* into database languages requires the development of new systems, which integrate the functionalities of logic programming and

database systems. Several alternative architectures have been proposed for this purpose; in this section, we present a first classification of the various approaches.

The first, broader distinction concerns the relationship between logic programming and relational systems.

a) We describe the development of an interface between two separate subsystems, a logic programming system and a database system, as *coupling.* With this approach, we start from two currently available systems, and we couple them so as to provide a single-system image. Both subsystems preserve their individuality; an interface between them provides the procedures required for bringing data from the persistent database system into the main-memory logic programming execution environment in order to evaluate queries or to validate constraints.
b) We describe the development of a single system which provides logic programming on top of a mass-memory database system as *integration.* This approach corresponds to the development of an entirely new class of data structures and algorithms, specifically designed to use logic programming as a database language.

Given the above alternatives, it is reasonable to expect that *Prolog*-based systems will mostly use the coupling approach, and *Datalog*-based systems will mostly use the integration approach. This is due to the present availability of many efficient *Prolog* systems that can be coupled with existing database systems with various degrees of sophistication. In fact, several research prototypes and even a few commercial products that belong to this class are already available. On the other hand, *Datalog* is an evolution of *Prolog* specifically designed to act as a database language; hence it seems convenient to use this new language in the development of radically new integrated systems. This mapping of *Prolog* to coupling and of *Datalog* to integration should not be considered mandatory. Indeed, we should recall that the *Fifth Generation Project* will produce an integrated system based on a parallel version of *Prolog.*

The coupling approach is easier to achieve but also potentially much less efficient than the integration approach. In fact, we cannot expect the same efficiency from the interface required by the coupling approach as from a specifically designed system. Furthermore, the degree of complexity of the interfaces can be very different. At one extreme, the simplest interface between a *Prolog* system and a relational system consists in generating a distinct SQL-like query in correspondence to every attempt at unification of each database predicate. This approach is very simple, but also potentially highly inefficient.

Hence, we expect that coupling will be sufficient for dealing with some applications, while other applications will require integration; further, coupling may be made increasingly efficient by superimposing ad-hoc techniques to the standard interfaces, thus achieving the ability of dealing with several special applications.

Within coupling, we further distinguish two alternative approaches:

a) *Loose coupling.* With this approach, the interaction between the logic programming and database systems takes place independently of the actual inference

process. Typically, coupling is performed at compile time (or at program load time, with interpreters), by extracting from the database all the facts that might be required by the program; sometimes, coupling is performed on a rule-by-rule basis, prior to the activation of that rule. Loose coupling is also called *static coupling* because coupling actions are performed independently of the actual pattern of execution of each rule.

b) *Tight coupling.* With this approach, the interaction between the logic programming and database systems is driven by the inference process, by extracting the specific facts required to answer the current goal or subgoal. In this way, coupling is performed whenever the logic programming system needs more data from the database system in order to proceed with its inference. Tight coupling is also called *dynamic coupling* because coupling actions are performed in the frame of the execution of each rule.

It follows from this presentation that loose and tight coupling are very different in complexity, selectivity, memory required, and performance. With loose coupling, we execute fewer queries of the database, because each predicate or rule is separately considered once and for all; while with tight coupling each rule or predicate can be considered several times. However, queries in loosely coupled systems are less selective than queries in tightly coupled systems, because variables are not instantiated (bound to constants) when queries are executed. If coupling is performed at compile or load time, queries are presented a priori, disregarding the actual pattern of execution of the logic program. In fact, it is even possible to load data at compile or load time concerning a rule or predicate that will not be used during the work session.

From these considerations, we deduce that the amount of main memory required for storing data which is to be retrieved by a loosely coupled system is higher than that required by a tightly coupled system. On the other hand, this consideration does not allow us to conclude that the performance of tightly coupled systems is always better; in general, tight coupling requires more frequent interactions with the database, and this means major overhead for the interface, with frequent context switching between the two systems. Thus, a comparison between the two approaches is difficult, and includes a trade-off analysis between memory used and execution time.

In Table 1.3, we summarize the comparison between loose and tight coupling.

LOOSE COUPLING	TIGHT COUPLING
fewer queries	more queries
less instantiated queries	more instantiated queries
all queries applied	relevant queries applied
more memory required	less memory required

Table 1.3. Comparison between loose and tight coupling

1.4 Applications

There are a number of new applications needing integrated systems at the confluence between databases and logic programming; the following is a list of the features of these applications.

a) *Database need.* The application must need to access data stored in a database. This means access to persistent, shared data that are resilent to failures and that can be accessed and updated concurrently by other applications. If we consider most expert systems or knowledge bases presently available, we observe that these systems have access to persistent data files, but that these are locally owned and controlled, with no sharing, concurrency, or reliability requirement with other applications.
b) *Selective access.* We cannot postulate that the entire database will be examined by the application during a work session, or else we should deal with data retrieval loads which exceed those of traditional database applications. Instead, we can postulate that during the work session the application will retrieve only a limited portion of the database, due to its selective access to data.
c) *Limited working set.* As a result of the previous assumption, the *working set* of data, namely the data required in main memory at a given time, is limited. This requirement is particularly important for loosely coupled systems, as loose coupling does not profit from the access selectivity, which is not expressed at compile time.
d) *Demanding database activity.* It is quite important to understand that systems which perform *millions of deductions per second*, as stated in the requirements of the *Fifth Generation Project*, are likely to put quite a heavy demand on the database. For instance, the computation of recursive rules requires a high number of interactions with the database. This feature contrasts with the typical database transactions, which serve *thousands of transactions per second*, each one responsible for a small amount of input/output operations.

Dealing with the above features extends the current spectrum of applicability of expert systems and other artificial intelligence applications; it also solves classical database problems, such as the *bill-of-materials* or the *anti-trust control* problems. These problems will be described in Chap. 3.

1.5 Bibliographic Notes

The relationship between logic programming and databases has been investigated since November 1977, when the conference on *Logic and Databases* took place in Toulouse; this event was followed by other two conferences on *Advances in Database Theory*, held in 1979 and 1982, again centered on this subject. The proceedings of the conferences, edited by Gallaire and Minker [Gall 78] and by Gallaire, Minker, and Nicolas [Gall 81, Gall 84a], contain fundamental papers

for the systematization of this field. Perhaps the best synthesis of the results in Logic and Databases achieved before 1984 is contained in a paper, again by Gallaire, Minker, and Nicolas, which appeared in *ACM Computing Surveys* [Gall 84b].

Two events characterize the growth of interest in this field: the selection, by the Japanese *Fifth Generation Project*, of an architecture based on *Prolog* as main programming language and of the relational model for data representation [Itoh 86]; and the growth of interest in the database theory community in logic queries and recursive query processing, marked by the seminal paper of Ullman [Ullm 85a]. Ullman has also presented, at the ACM-SIGMOD conference in 1984, the picture shown in Sect. 1.2 indicating the relationship between *Prolog* and *Datalog*. All recent database conferences (ACM-SIGMOD, ACM-PODS, VLDB, Data Engineering, ICDT, EDBT) have presented one or more sessions on logic programming and databases.

Parallel interest in *Expert Database Systems*, characterized by a more pragmatic, application-oriented approach, has in turn been presented through the new series of Expert Database Systems (EDS) conferences, organized by L. Kerschberg [Kers 84], [Kers 86], and [Kers 88]. The reading of the conference proceedings makes it possible to follow the growth, systematization, and spread of this area.

Specialized workshops on knowledge base management systems and on deductive databases were held in Islamorada [Brod 86], Washington [Mink 88], and Venice [Epsi 86]. Good overview papers describing methods for recursive query processing have been presented by Bancilhon and Ramakrishnan [Banc 86b], by Gardarin and Simon [Gard 87], and by Roelants [Roel 87]. A comparison of ongoing research projects for integrating databases and logic is provided by a special issue of IEEE-Database Engineering, edited by Zaniolo [Zani 87].