SRC TR 87-179

Extended Database Logic: Complex Objects and Deduction

by

T. K. Sellis and J. Grant

Extended Database Logic: Complex Objects and Deduction

John Grant
Department of Computer
and Information Sciences
Towson State University
Towson, MD 21204

Timos K. Sellis
Department of Computer Science
and Systems Research Center
University of Maryland
College Park, MD 20742

Abstract

Database Logic was proposed in the late 1970's as a generalization of first–order logic in order to deal in a uniform manner with relational, hierarchic, and network databases. At about the same time, the study of deductive (relational) databases has become important, primarily as a vehicle for the development of expert database systems. Also, Prolog, the main logic programming language, has become prominent for many applications in artificial intelligence, and its connections with deductive databases have been investigated. Although the relational model provides a suitable framework for traditional, essentially data processing applications, several researchers have found the need for complex objects in newer applications, such as engineering databases. In this paper we show how database logic can be extended in two directions: 1) to include complex objects, and 2) to provide deductive capabilities for hierarchic and network databases.

This research was sponsored partially by the National Science Foundation under Grants DCR–84–12662 and CDR–85–00108.

1. Introduction

Database logic was proposed in the late 1970's as a generalization of first–order logic in order to deal in a uniform manner with relational, hierarchic, and network databases. Just as first–order logic provides a precise syntax and semantics for data representation and manipulation in relational databases, database logic provides the same capabilities for hierarchic and network (as well as relational, as a special case) databases. This is achieved by allowing relations to be nested within relations. By the mid 1980's several important issues have been investigated using database logic [J]. However, the increasing commercial success of the relational model and the movement away from the hierarchic and network models by database systems vendors decreased the interest in database logic.

Although the relational model provides a suitable framework for traditional data processing applications, several researchers have found a need for explicit hierarchies in modeling some newer applications, such as engineering databases [BK1,BK2,D,VKC]. The terminology used for these hierarchic structures is *complex objects*; however, complex objects allow for the direct representation of more complicated structures than standard hierarchic databases. Deductive databases have gained prominence in the past decade [GMN], primarily as a vehicle for the development of expert database systems. A *deductive database* consists of facts and rules, where the rules allow for the derivation of additional facts. One advantage of this is that the storage requirements are decreased. But also, the incorporation of rules yields important capabilities for providing expert knowledge.

Since deductive databases can be directly implemented in a logic programming language, such as Prolog, logic programming is closely related to deductive databases. The connection between logic programming and complex objects was investigated in [Z], where it was shown that functions in Prolog can be applied to the representation of complex objects. In [GS] we presented some ideas about tying together concepts involving deductive databases, complex objects, logic programming, and database logic. The purpose of this paper is to present these connections in more detail and to provide formal definitions for extended database logic, which is a suitable framework for representing hierarchic, network, and deductive databases, including complex objects.

The organization of this paper is as follows. In Section 2 we review the connections between deductive databases and logic programming by showing how such deductive databases are represented in Prolog. In Section 3 we explain the notion of complex object and show how complex objects may be represented in Prolog. We introduce database logic for heterogeneous databases in Section 4. Section 5 contains several examples of extended database logic, illustrating its application to complex objects and its deductive capabilities. Section 6 provides the formal definitions for extended database logic including syntax and semantics. We discuss implementation considerations for extended database logic in Section 7. Section 8 is the summary.

2. Deductive Databases

A deductive relational database, *DB*, consists of two parts, a theory T and a set of integrity constraints *IC*. Such a theory has three groups of axioms. The first group consists of standard axioms: the *domain closure* axiom, the *unique name* axioms and the *equality* axioms [GMN]. The second group consists of atomic formulas which stand for facts: tuples in relations. This is called the *extensional database*. The third group represents the deductive rules; this comprises the *intensional database*. Additionally, a meta–rule is added to deal with negative information; one choice for this purpose is the *Closed World Assumption*. Under this assumption, failure to prove an atom P permits us to infer that $\neg P$ is true. The set of integrity constraints must be satisfied by the database; hence, $T \bigcup IC$ must be consistent. Additionally, it is assumed that the axioms do not contain function symbols.

As an example, suppose that the database consists of two extensional relations

```
EMPLOYEE(ENAME,DNAME,AGE,SALARY)
DEPARTMENT(DNAME,MANAGER,PHONE)
```

with the obvious meanings, and one intentional relation

```
SUPERVISOR(DNAME,MANAGER)
```

that indicates employees that supervise other employees. We use a typed first–order logic with attributes for variables of that type, as will be seen presently. The following is an instance for this database schema:

Extensional Database

EMPLOYEE			
ENAME	DNAME	AGE	SALARY
Jones	Sales	33	28500
Smith	Sales	41	34100
Baker	Accounting	25	22400
Adams	Service	50	27000
Rogers	Service	47	26000
Wilson	Accounting	42	35000
Osmond	Sales	21	17500
Moore	Sales	61	42300

DEPARTMENT		
DNAME	MANAGER	PHONE
Sales	Samuels	555-1113
Accounting	Lever	434-2219
Service	Simmons	913-2425

Intensional Database

```
∀ ENAME ∀ DNAME ∀ AGE ∀ SALARY ∀ MANAGER ∀ PHONE
(EMPLOYEE(ENAME,DNAME,AGE,SALARY) & DEPARTMENT(DNAME,MANAGER,PHONE)
 → SUPERVISOR(ENAME,MANAGER))
```

In logic, queries are written in the relational calculus. We give two queries:

1) Find the name, age, and manager of every employee whose salary is greater than 25000

```
{<ENAME,AGE,MANAGER> | ∃ DNAME ∃ SALARY ∃ PHONE
 ( EMPLOYEE (ENAME,DNAME,AGE,SALARY)
 & DEPARTMENT (DNAME,MANAGER,PHONE)
 & SALARY > 25000)}
```

2) Find the names of all employees supervised by Lever

```
{<ENAME> | SUPERVISOR(ENAME,"Lever")}
```

We discuss now the connections between deductive databases and logic programming. Logic programming involves the use of logical formulas, basically clauses, as the building blocks of a computer program. The most prominent logic programming language is Prolog, which provides a restricted set of clauses, the Horn clauses, for writing programs. A fundamental concept in logic programming is the dual declarative and procedural interpretation of clauses. Thus, a clause $A \leftarrow B_1,B_2,...,B_k$ (terms omitted) may be understood declaratively as indicating that if B_1, $B_2,...,B_k$ are all true then A is true, and procedurally by implying that to solve for A requires solving for B_1, $B_2,...$, B_k. In the database context non–Horn clauses express indefinite informa-

tion. We will restrict our consideration to Horn clauses and thus will deal only with definite information.

A deductive relational database is represented in Prolog by writing the second and third groups of axioms of T as Prolog clauses. The Prolog interpreter takes care of aspects of the first group in T as well as the meta-rule for negative information. The formulas in IC are handled separately. For example, the above database can be represented by the following Prolog program:

```
employee(jones,sales,33,28500).
employee(smith,sales,41,34100).
employee(baker,accounting,25,22400).
employee(adams,service,50,27000).
employee(rogers,service,47,26000).
employee(wilson,accounting,42,35000).
employee(osmond,sales,21,17500).
employee(moore,sales,61,42300).
department(sales,samuels,"555-1313").
department(accounting,lever,"434-2219").
department(service,simmons,"913-2425").
supervisor(Ename,Manager):- employee(Ename,Dname,Age,Salary),
                            department(Dname,Manager,Phone).
```

We now write the above queries in Prolog. For the first query, a new predicate is defined to obtain only the attributes required for the answer.

```
1) answer1(Ename,Age,Manager):- employee(Ename,Dname,Age,Salary),
                                department(Dname,Manager,Phone), Salary>25000.
  ?- answer1(Ename,Age,Manager).

2) ?- supervisor(Ename,lever).
```

Finally, we show how standard relational database languages, such as SQL, can be used to implement the above deduction model. The intensional database can be formulated using relational views, with the exception of recursive definitions. For example, SUPERVISOR can be defined as follows

```
CREATE VIEW SUPERVISOR(ENAME,MANAGER)
       AS   SELECT  ENAME,MANAGER
            FROM    EMPLOYEE,DEPARTMENT
            WHERE   EMPLOYEE.DNAME = DEPARTMENT.DNAME
```

We conclude this section by showing how the above queries can be written in SQL.

1) Find the name, age, and manager of every employee whose salary is greater than 25000

```
SELECT   ENAME,AGE,MANAGER
FROM     EMPLOYEE,DEPARTMENT
WHERE    EMPLOYEE.DNAME = DEPARTMENT.DNAME
AND      SALARY > 25000
```

2) Find the names of all employees supervised by Lever

```
SELECT   ENAME
FROM     SUPERVISOR
WHERE    MANAGER = "Lever"
```

3. Complex Objects

In modeling some newer applications, such as engineering databases, the notion of complex objects has been found useful [BK1,BK2]. In fact, complex objects are useful in many applications, including standard data processing databases. In this section we illustrate the notion of a complex object and show how complex objects can be represented in a natural way within the framework of logic.

A complex object may involve hierarchies in several different ways. In a traditional hierarchic database the purpose of the hierarchy is to represent a one–to–many relationship. In a complex object a hierarchy may also be used in other ways. We deal with two typical cases. One application is for breaking up an object into several parts (or grouping several objects together). For example, it may be convenient to divide a name into first name and last name, while allowing for the possibility of treating a name as a single object. Another application allows for different alternatives, as in our example later in this section, where different data is stored about an employee's education depending on the employee's degree.

In representing complex objects, Zaniolo has suggested in [Z] the use of Prolog. Complex objects may be represented in Prolog by using functions in predicates; however, functions are not needed for deductive (relational) databases. We start by showing how a traditional hierarchy, involving the departments and employees of the previous section, can be represented in Prolog:

```
department(sales,samuels,"555-1313",employee(jones,33,28500)).
department(sales,samuels,"555-1313",employee(smith,41,34100)).
department(sales,samuels,"555-1313",employee(osmond,21,17500)).
department(sales,samuels,"555-1313",employee(moore,61,42300)).
department(accounting,lever,"434-2219",employee(baker,25,22400)).
department(accounting,lever,"434-2219",employee(wilson,42,35000)).
department(service,simmons,"913-2425",employee(adams,50,27000)).
department(service,simmons,"913-2425",employee(rogers,47,26000)).
```

Next we show how to write a query from the previous section as well as two additional queries in Prolog using the hierarchic representation.

1) Find the name, age, and manager of every employee whose salary is greater than 25000

```
answer1(Ename,Age,Manager) :- department(Dname,Manager,Phone,
                              employee(Ename,Age,Salary)), Salary > 25000.
?- answer1(Ename,Age,Manager).
```

3) Find all employee information for each department

```
?- department(Dname,_,_,Employee).
```

4) Find the name and salary of every employee who works for a manager younger than him/her

```
answer4(Ename,Salary):- department(Dname,Manager,Phone,employee(Ename,Age1,
                Salary)), managers(Manager,Age2,children(Child)), Age1 < Age2.
?- answer4(Ename,Salary).
```

Question 4 assumes the existence of a **managers** predicate in the Prolog program and involves a join of hierarchies.

Our next example illustrates all three uses of hierarchies, that we discussed earlier, in Prolog. This example is based on, but is not identical to, an example in [Z]. Essentially we use the above data about employees, but break up the ENAME attribute, change DNAME to JOBCLASS, and add an attribute for DEGREE. The Prolog program is as follows:

```
employee(name(jones,mary,j),sales,33,28500,degree(hs,1972)).
employee(name(wilson,david),accounting,42,35000,degree(bs,1970,accounting,nyu)).
employee(name(adams,john,b),service,50,27000).
employee(name(moore,pat),sales,61,42300,degree(hs,1944)).
employee(name(harris,susan),management,31,32500,degree(bs,1980,business,gwu)).
employee(name(harris,susan),management,31,32500,degree(ms,1982,business,ub)).
employee(name(cramer,bill,j),dp,35,30000,degree(bs,1974,cs,umcp)).
```

Note that in this example all college degrees (if any) of an individual are listed, with year and school information; otherwise the high school degree is listed with the year value only. **name** is a grouping of first, last names and, optionally, middle name initial. **degree** by itself illustrates the application of alternatives, as a high–school degree is treated differently than a university degree.

We now write a few queries for this database.

5) Find the name and salary of all employees who are only high school graduates

```
?- employee(Name,_,_,Salary,degree(hs,_)).
```

6) Find the name, job classification, and major of all employees who obtained a B.S. after 1970

```
answer6(Name,Jobclass,Major) :- employee(Name,Jobclass,Age,Salary,
                                degree(bs,Year,Major,School))), Year > 1970.
?- answer6(Name,Jobclass,Major).
```

Prolog also allows for the definition of intensional predicates for complex objects. In fact, the definition for answer6 can be thought of as the definition of an intensional predicate. Another example is the following, where collegegrad is a predicate containing information about college graduates

```
collegegrad(Name,Salary,School) :- employee(Name,Jobclass,Age,Salary,
                                   degree(Type,Year,Major,School)).
```

We end this section by showing how recursive complex objects can be defined. The most typical representative here is the parts explosion example. We assume a predicate part(pid,pname,part), where the third argument is itself an element of part and indicates which parts are sub–parts of other parts. For example, the following can be an instance of part

```
part(p1,compartment1,part(p2,tank2,part(p3,tank3,EMPTY))).
part(p4,water-tank,EMPTY).
part(p5,tank5,part(p6,tank6,EMPTY)).
part(p7,compartment2,part(p2,tank2,part(p3,tank3,EMPTY))).
part(p8,fuel-tank,EMPTY).
part(p9,tank9,part(p6,tank6,EMPTY)).
part(p10,tank10,part(p11,tank11,EMPTY)).
part(p12,tank12,part(p13,tank13,part(p14,tank14,EMPTY))).
```

where the keyword EMPTY is used to indicate that no further decomposition of the part is possible. The following are possible queries on such a database

7) Find the sub–parts of compartment1

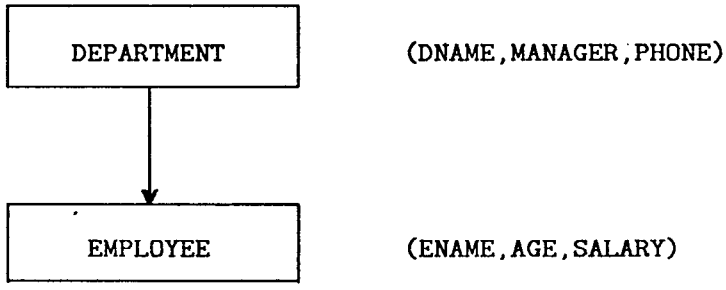
```
?- part(_,compartment1,Sub-part).
```

8) Find the names of top–level parts that have p2 as direct sub–part

```
?- part(_,Pname,part(p2,_,_)).
```

4. Database Logic

The fundamental idea in database logic is to allow relations within relations: a column of a table may itself be a table. This way database logic can be used to generalize non–procedural languages, such as the relational calculus and SQL, to hierarchic and network databases. We illustrate database logic on a hierarchic database obtained from the relational database portion of the deductive database presented in Section 2.



The schema for such a database written in database logic and the extensional database, look as follows

```
SCHEMA          DEPTEMP
TABLE           ATTRIBUTES
  DEPARTMENT      DNAME,MANAGER,PHONE,EMPLOYEE
  EMPLOYEE        ENAME,AGE,SALARY
```

DEPARTMENT					
DNAME	MANAGER	PHONE	EMPLOYEE		
			ENAME	AGE	SALARY
Sales	Samuels	555-1113	Jones	33	28500
			Smith	41	34100
			Osmond	21	17500
			Moore	61	42300
Accounting	Lever	434-2219	Baker	25	22400
			Wilson	42	35000
Service	Simmons	913-2425	Adams	50	27000
			Rogers	47	26000

This DEPARTMENT table has three rows and four columns. The fourth column is itself a table: it contains data about the employees of a particular department. In database logic a predicate may have a predicate nested within it; it is also convenient to use so–called *cluster* predicates which are predicate combinations, as we will soon demonstrate. Now we rewrite the first query from Section 2 in database logic. We give two versions, the first one uses database logic predicates, the second one uses a cluster predicate.

1) Find the name, age, and manager of every employee whose salary is greater than 25000

```
(a) {<ENAME,AGE,MANAGER> | ∃ DNAME ∃ PHONE ∃ EMPLOYEE ∃ SALARY
     ( DEPARTMENT (DNAME,MANAGER,PHONE,EMPLOYEE)
     & EMPLOYEE (ENAME,AGE,SALARY)
     & SALARY > 25000)}

(b) {<ENAME,AGE,MANAGER> | ∃ DNAME ∃ PHONE ∃ EMPLOYEE ∃ SALARY
     ( DEPARTMENT-EMPLOYEE (DNAME,MANAGER,PHONE,EMPLOYEE(ENAME,AGE,SALARY))
     & SALARY > 25000)}
```

The next two queries are from Section 3. Note that question (3) yields a hierarchic subtable of DEPARTMENT for its answer and question (4) involves the join of two hierarchic tables.

3) Find all employee information for each department

```
{<DNAME,EMPLOYEE> | ∃ MANAGER ∃ PHONE
 DEPARTMENT (DNAME,MANAGER,PHONE,EMPLOYEE)}
```

4) Find the name and salary of every employee who works for a manager younger than him/her

```
{<ENAME,SALARY> | ∃ DNAME ∃ MANAGER ∃ PHONE ∃ EMPLOYEE ∃ AGE1 ∃ AGE2 ∃ CHILDREN
 ( DEPARTMENT-EMPLOYEE (DNAME,MANAGER,PHONE,EMPLOYEE(ENAME,AGE1,SALARY))
 & MANAGERS (MANAGER,AGE2,CHILDREN)
 & AGE1 < AGE2)}
```

While the relational calculus is the database language directly based on logic, another language based on the relational calculus, SQL, is the one widely used in applications. Just as the relational calculus can be generalized to the database logic language, SQL has a generalization, GSQL, in the database logic framework [J]. We illustrate GSQL by rewriting the previous database logic queries in it.

```
1) SELECT ENAME,AGE,MANAGER
   FROM   DEPARTMENT-EMPLOYEE
   WHERE  SALARY > 25000
```

```
3) SELECT DNAME,EMPLOYEE
   FROM   DEPARTMENT

4) SELECT ENAME,SALARY
   FROM   DEPARTMENT-EMPLOYEE,MANAGERS
   WHERE  DEPARTMENT-EMPLOYEE.MANAGER = MANAGERS.MANAGER
   AND    DEPARTMENT-EMPLOYEE.AGE < MANAGERS.AGE
```

Next we illustrate database logic by reconsidering the example of the recursive complex object from the previous section. In database logic such an object is represented as a network database. The schema for such a database is written in database logic as

```
SCHEMA        PARTS
TABLE         ATTRIBUTES
  PART          PID,PNAME,PART
```

In database logic the instance corresponding to the extensional database of the previous section would look as follows:

PART					
PID	PNAME	PART			
		PID	PNAME	PART	
				PID	PNAME
p1	compartment1	p2	tank2	p3	tank3
p4	water-tank				
p5	tank5	p6	tank6		
p7	compartment2	p2	tank2	p3	tank3
p8	fuel-tank				
p9	tank9	p6	tank6		
p10	tank10	p11	tank11		
p12	tank12	p13	tank13	p14	tank14

Next we write the queries for this database from the previous section both in database logic and in GSQL. Database logic allows us to generalize query (8) by eliminating the restriction to top–level parts.

7) Find the sub–parts of compartment1

```
{<PART2> | ∃ PART1 ∃ PID1
 PART1 (PID1,"compartment1",PART2)}
```

```
SELECT PART
FROM   PART
WHERE  PNAME = "compartment"
```

8) Find the names of parts that have p2 as direct sub–part

```
{<PNAME1> | ∃ PART1 ∃ PID1 ∃ PART2 ∃ PNAME2 ∃ PART3
 PART1-PART (PID1,PNAME1,PART2("p2",PNAME2,PART3))}
```

```
SELECT FIRST.PNAME
FROM   FIRST PART - SECOND PART
WHERE  SECOND.PID = "p2"
```

We end this section by showing how to write integrity constraints in database logic. The first three constraints are for the hierarchic DEPARTMENT database, while the last constraint is for the recursive PARTS database.

A) DEPARTMENT: DNAME → MANAGER,PHONE,EMPLOYEE

This is the functional dependency that makes DNAME the key for DEPARTMENT.

```
∀ DNAME1 ∀ MANAGER1 ∀ MANAGER2 ∀ PHONE1 ∀ PHONE2 ∀ EMPLOYEE1 ∀ EMPLOYEE2
( DEPARTMENT (DNAME1,MANAGER1,PHONE1,EMPLOYEE1)
& DEPARTMENT (DNAME1,MANAGER2,PHONE2,EMPLOYEE2)
→ MANAGER1=MANAGER2 & PHONE1=PHONE2 & EMPLOYEE1=EMPLOYEE2)
```

B) DEPARTMENT-EMPLOYEE: DNAME,ENAME → AGE,SALARY

This functional dependency says that for a specific department, ENAME is the key for EMPLOYEE.

```
∀ DNAME1 ∀ MANAGER1 ∀ MANAGER2 ∀ PHONE1 ∀ PHONE2 ∀ EMPLOYEE1 ∀ EMPLOYEE2
∀ ENAME1 ∀ AGE1 ∀ AGE2 ∀ SALARY1 ∀ SALARY2
( DEPARTMENT-EMPLOYEE (DNAME1,MANAGER1,PHONE1,EMPLOYEE1(ENAME1,AGE1,SALARY1))
& DEPARTMENT-EMPLOYEE (DNAME1,MANAGER2,PHONE2,EMPLOYEE2(ENAME1,AGE2,SALARY2))
→ AGE1=AGE2 & SALARY1=SALARY2)
```

C) EMPLOYEE: 16 ≤ AGE ≤ 75

This is a domain constraint for the AGE attribute in the EMPLOYEE relation.

```
∀ EMPLOYEE ∀ ENAME ∀ AGE ∀ SALARY
( EMPLOYEE (ENAME,AGE,SALARY) → 16 ≤ AGE & AGE ≤ 75)
```

D) PART: PID → PNAME,PART

This functional dependency says that for a specific part, PID is the key for PART.

```
∀ PART1 ∀ PNAME1 ∀ PID1 ∀ PART2 ∀ PART3 ∀ PNAME2 ∀ PART4
( PART1 (PID1,PNAME1,PART2) & PART3 (PID1,PNAME2,PART4)
→ PNAME1 = PNAME2 & PART3 = PART4
```

5. Extended Database Logic Examples

As we demonstrated in the previous section, database logic generalizes first–order logic to hierarchic and network databases. However, as indicated in Section 3, complex objects may involve hierarchies in several different ways, not just for one–to–many relationships. In this section we give examples to show how a new version of database logic, called *Extended Database Logic*, can be used to represent complex objects. We also show that extended database logic has built–in deductive capabilities. This way, extended database logic provides a common framework for the representation of deductive relational, hierarchic, and network databases that may include complex objects.

We use the second example of Section 3 to illustrate extended database logic. We start by describing the schema for this example. As pointed out in Section 4, database logic was developed as an extension to first–order logic to provide the standard hierarchic and network models with the capabilities that first–order logic provides for the relational model: a non–procedural query language with precise semantics and the expression of queries and constraints in one language.

We illustrate what needs to be done for complex objects by showing how database logic can be extended to deal with the following extended database logic schema

```
SCHEMA        EMPED
TABLE         ATTRIBUTES
  EMPLOYEE      NAME,JOBCLASS,AGE,SALARY,DEGREE
  NAME          <LAST,FIRST (,MI)>
  DEGREE        TYPE,YEAR [TYPE ≠ hs →,MAJOR,SCHOOL]
```

The angle brackets denote the grouping of objects, as opposed to one–to–many relationships. Thus an employee has only one name, which is composed of several parts, but an employee may have several degrees. Parentheses are used to enclose optional items: for example, a name may have, but is not required to contain a middle initial. Finally, brackets represent a situation where additional attributes depend on some condition. In this case only non–high school degrees require major and school information.

The instance of this schema written in Section 3 in Prolog would be represented as the following database logic table.

EMPLOYEE									
NAME			JOBCLASS	AGE	SALARY	DEGREE			
LAST	FIRST	MI				TYPE	YEAR	MAJOR	SCHOOL
Jones	Mary	J	Sales	33	28500	HS	1972		
Wilson	David		Accounting	42	35000	BS	1970	Accounting	NYU
Adams	John	B	Service	50	27000				
Moore	Pat		Sales	61	42300	HS	1944		
Harris	Susan		Management	31	32500	BS	1980	Business	GW
						MS	1982	Business	UB
Cramer	Bill	J	Dp	35	30000	BS	1974	Cs	UMCP

Now we write in extended database logic the queries for this database that we wrote in Prolog in Section 3.

5) Find the name and salary of all employees who are only high school graduates

```
{<NAME,SALARY> | ∃ JOBCLASS ∃ AGE ∃ DEGREE ∃ YEAR
 (EMPLOYEE-DEGREE(NAME,JOBCLASS,AGE,SALARY,DEGREE("HS",YEAR))}
```

6) Find the name, job classification, and major of all employees who obtained a B.S. after 1970

```
 {<NAME,JOBCLASS,MAJOR> | ∃ AGE ∃ SALARY ∃ DEGREE ∃ YEAR ∃ SCHOOL
(EMPLOYEE-DEGREE(NAME,JOBCLASS,AGE,SALARY,DEGREE("BS",YEAR,MAJOR,SCHOOL))
 & YEAR > 1970)}
```

We continue by rewriting these queries in GSQL.

```
5)  SELECT NAME,SALARY
    FROM   EMPLOYEE-DEGREE
    WHERE  TYPE = "HS"

6)  SELECT NAME,JOBCLASS,MAJOR
    FROM   EMPLOYEE-DEGREE
    WHERE  TYPE = "BS"
    AND    YEAR > 1970
```

Next we give examples of integrity constraints for this database in database logic.

E) EMPLOYEE: NAME → JOBCLASS,AGE,SALARY,DEGREE

This functional dependency establishes NAME as the key for EMPLOYEE.

```
∀ NAME1 ∀ JOBCLASS1 ∀ JOBCLASS2 ∀ AGE1 ∀ AGE2 ∀ SALARY1 ∀ SALARY2
∀ DEGREE1 ∀ DEGREE2
( EMPLOYEE (NAME1,JOBCLASS1,AGE1,SALARY1,DEGREE1)
& EMPLOYEE (NAME1,JOBCLASS2,AGE2,SALARY2,DEGREE2)
→ JOBCLASS1=JOBCLASS2 & AGE1=AGE2 & SALARY1=SALARY2 & DEGREE1=DEGREE2)
```

F) EMPLOYEE-DEGREE: NAME,TYPE → YEAR,MAJOR,SCHOOL

This functional dependency says that for a specific NAME, TYPE is the key for DEGREE.

```
∀ NAME1 ∀ JOBCLASS1 ∀ JOBCLASS2 ∀ AGE1 ∀ AGE2 ∀ SALARY1 ∀ SALARY2 ∀ DEGREE1
∀ DEGREE2 ∀ TYPE1 ∀ YEAR1 ∀ YEAR2 ∀ MAJOR1 ∀ MAJOR2 ∀ SCHOOL1 ∀ SCHOOL2
( EMPLOYEE-DEGREE (NAME1,JOBCLASS1,AGE1,SALARY1,DEGREE1(TYPE1,YEAR1,
                   MAJOR1,SCHOOL1))
& EMPLOYEE-DEGREE (NAME1,JOBCLASS2,AGE2,SALARY2,DEGREE2(TYPE1,YEAR2,
                   MAJOR2,SCHOOL2))
→ YEAR1=YEAR2 & MAJOR1=MAJOR2 & SCHOOL1=SCHOOL2)
```

G) EMPLOYEE: 16 ≤ AGE ≤ 75

This is a domain constraint for the AGE attribute in EMPLOYEE.

```
∀ NAME ∀ JOBCLASS ∀ AGE ∀ SALARY ∀ DEGREE
( EMPLOYEE (NAME,JOBCLASS,AGE,SALARY,DEGREE) → 16 ≤ AGE & AGE ≤ 75)
```

Finally, we show how deductive capabilities may be added to database logic by writing the definition of the intensional predicate given near the end of Section 3 in database logic.

```
∀ NAME ∀ JOBCLASS ∀ AGE ∀ SALARY ∀ DEGREE ∀ TYPE ∀ YEAR ∀ MAJOR ∀ SCHOOL
( EMPLOYEE-DEGREE-SCHOOL (NAME,JOBCLASS,AGE,SALARY,DEGREE(TYPE,YEAR,MAJOR,
          SCHOOL)) → COLLEGEGRAD(NAME,SALARY,SCHOOL))
```

In Section 2 we briefly described how to express a deductive relational database in first-order logic. In this section we showed that both a theory and integrity constraints for more complex structures can be expressed in extended database logic. The Closed World Assumption can also be applied to take care of negative information. Thus the framework of extended database logic can be used to define deductive databases that may involve complex objects.

6. Formal Definitions

This section provides a sketch for the formal definitions of the syntax and semantics of extended database logic. We omit some of the details and illustrate the definitions on the example of the previous section. More complete definitions (for database logic) can be found in [J].

A database schema S consists of two parts: 1) a set of table names with each table name comprised of a sequence of attributes, and 2) a typing for the table names and attributes. We start by giving rules for part 1). No attribute may be repeated within one sequence and no table name may be defined more than once. There are three optional constructs involving angle brackets, parentheses, and brackets. First, the whole sequence of attributes may be enclosed within angle brackets. Second, a rightmost subsequence may be enclosed within parentheses. The third option is for the rightmost portion of the attribute to consist of a bracketed expression. A bracketed expression must contain an arrow; to the left of the arrow is a condition involving one or more attributes that must also appear to the left of the bracketed expression; to the right of the arrow are one or more additional attributes. Part 2), the typing, assigns a data type to each atomic attribute. An atomic attribute is one that is not also a table name. We identify each data type with a set of values. In the previous sections we omitted the typing.

Now we write a complete schema for the example given in the previous section.

```
SCHEMA        EMPED
TABLE         ATTRIBUTES
  EMPLOYEE      NAME,JOBCLASS,AGE,SALARY,DEGREE
  NAME          <LAST,FIRST (,MI)>
  DEGREE        TYPE,YEAR [TYPE ≠ "HS" → ,MAJOR,SCHOOL]
TYPE          ATTRIBUTES
  INTEGER       AGE,YEAR
  REAL          SALARY
  CHAR(1)       MI
  STRING        LAST,FIRST,JOBCLASS,TYPE,MAJOR,SCHOOL
```

Next we define the database language L associated with the schema S. L contains *predicate symbols, constant symbols, variable symbols, connectives, quantifiers, punctuation, and built-in predicate symbols.* There is a predicate symbol for each table. The type of a predicate symbol is the sequence of typings for its attributes. The constant symbols of L contain all the values for the data types represented by the atomic attributes. Additional constant symbols exist for table names that are also attributes, as illustrated below for the EMPLOYEE example. A single constant symbol is associated with each table name that is not an attribute. Also, for all table names that are also attributes and all atomic attributes, L contains an infinite set of variables indicated by writing the name optionally followed immediately by a digit.

The database language L for the schema EMPED contains the following symbols:

Predicate symbols:

```
EMPLOYEE   -   NAME,JOBCLASS,AGE,SALARY,DEGREE
NAME       -   <LAST,FIRST (,MI)>
DEGREE     -   TYPE,YEAR [TYPE ≠ HS → ,MAJOR,SCHOOL]
```

Constant symbols:

all integers, reals, char(1) values, and strings,
    c-EMPLOYEE (the constant symbol for EMPLOYEE)
constant symbols for NAME of the form
    <string,string> or <string,string,char(1)>
constant symbols for DEGREE of the form
    DEGREE(NAME-constant-symbol)

Variable symbols:

```
NAME, NAME1, NAME2, ...
JOBCLASS, JOBCLASS1, ...
   .
   .
SCHOOL, SCHOOL1, SCHOOL2, ...
```

Connectives: ¬ & ∨ →

Quantifiers: ∃ ∀

Punctuation: ( ) [ ] < > ,

Built–in predicate symbols: $=$ $\leq$ $<$ $\neq$ $>$ $\geq$

This example does not contain any implicitly defined predicates. In the case of a deductive database, a predicate symbol and a constant symbol are required for each implicitly defined predicate.

The constant symbols for NAME and DEGREE require explanation. Since NAME consists of two or three parts, the constant symbols for NAME must also contain those parts. In the case of DEGREE, there is a one–to–many relationship between EMPLOYEE and DEGREE. Assuming that a NAME uniquely determines a DEGREE table, it is sufficient to indicate the NAME constant symbol. Note that each predicate symbol is followed by its typing.

The formulas of L are obtained in the usual manner. A term is either a constant or a variable. This example contains no function symbols; however, function symbols can be added as needed. An atomic formula is a predicate symbol followed by terms in conformance with the typing, separated by commas, and enclosed within parentheses. Infix notation may be used for the built–in predicate symbols. Formulas are obtained from atomic formulas by applying the connectives and quantifiers in the standard way. In the previous section we gave several examples of formulas of L.

The semantics for extended database logic is expressed by means of *interpretations*. Essentially, an interpretation I is a database instance that conforms to a database schema. The interpretation of an attribute is the set of all elements defined by its type. The interpretation of each atomic constant symbol is itself and the interpretation of each built–in predicate symbol is the standard interpretation of that symbol. A specific interpretation I is determined by the interpretation of each non–atomic constant symbol.

To illustrate interpretations, we provide the interpretation for the EMPED schema that corresponds to the instance given in the previous section.

```
I(c-EMPLOYEE) =
   {(<Jones,Mary,J>,Sales,33,28500,DEGREE(<Jones,Mary,J>)),
    (<Wilson,David>,Accounting,42,35000,DEGREE(<Wilson,David>)),
    (<Adams,John,B>,Service,50,27000,DEGREE(<Adams,John,B>)),
    (<Moore,Pat>,Sales,61,42300,DEGREE(<Moore,Pat>)),
    (<Harris,Susan>,Management,31,32500,DEGREE(<Harris,Susan>)),
    (<Cramer,Bill,J>,Dp,35,30000,DEGREE(<Cramer,Bill,J>))}

I(DEGREE(<Jones,Mary,J>)) =  {(HS,1972)}
I(DEGREE(<Wilson,David>)) =  {(BS,1970,Accounting,NYU)}
I(DEGREE(<Adams,John,B>)) =  {}
I(DEGREE(<Moore,Pat>)) =     {(HS,1944)}
I(DEGREE(<Harris,Susan>)) =  {(BS,1980,Business,GWU),(MS,1982,Business,UB)}
I(DEGREE(<Cramer,Bill,J>)) = {(BS,1974,Cs,UMCP)}
```

Note how the interpretation of a NAME constant is different from the interpretation of a DEGREE constant. For a NAME constant it is simply made up of its component parts. But for a DEGREE constant it is a table. Each element of this table must be a 2–tuple if the first value is BS, otherwise it is a 4–tuple to conform to the schema. If the database contains implicitly defined predicates, then the interpretation of the constant symbol for the implicitly defined predicate is obtained by evaluating the formula on the appropriate constant symbols for the explicitly defined predicates according to the definition.

Once the interpretation of the above symbols is given, the interpretation for all formulas of L can be defined in the standard logical manner. We actually use additional predicates, called *cluster predicates*, also in some of our formulas. Such a predicate is essentially an abbreviation for writing out all the predicates in the cluster. All sentences (formulas with no free variables) of L evaluate to true or false. It is not difficult to verify that all the integrity constraints given in the previous section are true for this interpretation. Queries are also formulas of L, but they contain free variables (except for yes–no queries). The answers for the queries are the sequences

of constants that make the formula true when substituted for the variables. Consider, for example, query (5). There are two answers:

<<Jones,Mary,J>,28500> and <<Moore,Pat>,42300>.

The first tuple is an answer because the sentence

```
∃ JOBCLASS ∃ AGE ∃ DEGREE ∃ YEAR
( EMPLOYEE-DEGREE(<Jones,Mary,J>,JOBCLASS,AGE,28500,DEGREE("HS",YEAR)) )
```

is true for the given database. The second tuple is also an answer for a similar reason and there are no more answers.

7. Implementation Aspects

Although complex objects have received a lot of attention recently, not many ideas on implementation have been suggested. In this section we present some previous proposals and suggest improvements. We will illustrate the various approaches on a slightly modified example for an employee–degree database, shown in the following table

EMPLOYEE									
NAME		JOBCLASS	AGE	SALARY	DEGREE				
LAST	FIRST				TYPE	YEAR	MAJOR	SCHOOL	
								SNAME	STATE
Jones	Mary	Sales	33	28500	HS	1972			
Wilson	David	Accounting	42	35000	BS	1970	Accounting	NYU	NY
Adams	John	Service	50	27000					
Moore	Pat	Sales	61	42300	HS	1944			
Harris	Susan	Management	31	32500	BS	1980	Business	GWU	DC
					MS	1982	Business	UB	MD
Cramer	Bill	Dp	35	30000	BS	1974	Cs	UMCP	MD

Notice that this is similar to the example of the previous section, yet we have added one more

level of nesting by decomposing the SCHOOL field to the name of the school and the state in which it is found.

Clearly, one way to store complex objects is by simply storing the object hierarchy in a "flattened" way. For example, the most natural way would be to store the hierarchy representing an object in *pre–order*. That is, store the top–level component, then each one of its children followed by their children, etc. This approach provides high clustering which enables the fast retrieval of all the components of an object. However, it is very inefficient if some partial retrieval is requested. For example, in the case of the EMPLOYEE object shown above, if only the information on SCHOOL is requested, the whole hierarchy needs be traversed. Moreover, this approach suffers in the case where the instance of an object does not fit in one physical page. In this case, multiple page accesses are required, even in the case where the requested part of the object is really small in size.

As a solution to this problem, Lorie et al. [L] suggested that the hierarchy be stored in a simple way using logical pointers to connect the various components. Figure 1 shows how a portion of the EMPLOYEE database would be stored using this proposal. The reason that physical addresses are not used for pointers is because reorganization of the database would result in major reorganization of the data structures themselves. The solution suggested in [L] is the *Indirection Table* shown in Figure 1 which simply serves as a translation mechanism for logical pointers. Of course, the use of the indirection table has the effect of more costly retrievals since the various components of an object may be scattered to random places on the physical device. However, updates are less costly compared to the first proposal because there is no need to move data around. Kim at. al [KCB] also use a similar structure in their implementation of complex objects.

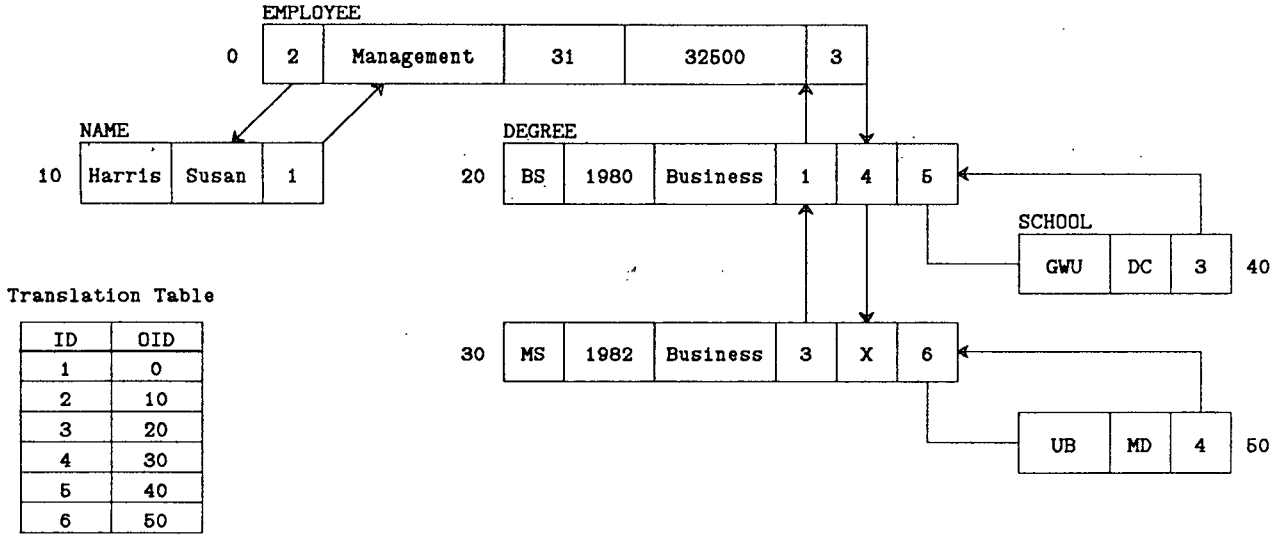


Figure 1: The Lorie et al. [L] approach

Although the Lorie et al. approach is very straightforward and easy to implement within relational database systems, it is useful only for full object retrievals. If the retrieval of just a sub–object is requested, still the hierarchy needs be traversed. For example, in the above example, if one needs to find out the high school graduation date for a given employee, all DEGREE records need be accessed in the worst case. This is due to the fact that DEGREE records are linked together through pointers, making the cost of retrieving the last record in the chain much higher than the cost of retrieving the first record.

To overcome this undesirable effect, Valduriez et al. [VKC] suggested a different approach. Each instance of an object is assigned a unique object identifier (*surrogate*). All instances of an object along with their associated surrogates are stored in a relation. Surrogates are used to link the various objects with their components, in the same sense that pointers were used in the Lorie et al. proposal. However, no ordering is assumed in the way the various instances of an object are stored. To give an example, the following shows how the EMPLOYEE database is stored using

the proposal of [VKC].

EMPLOYEE					
E-sur	N-sur	JOBCLASS	AGE	SALARY	D-sur
E001	N001	Sales	33	28500	D001
E002	N002	Accounting	42	35000	D002
E003	N003	Service	50	27000	NULL
E004	N004	Sales	61	42300	D003
E005	N005	Management	31	32500	D004
E006	N005	Management	31	32500	D005
E007	N006	Dp	35	30000	D006

NAME		
N-sur	LAST	FIRST
N001	Jones	Mary
N002	Wilson	David
N003	Adams	John
N004	Moore	Pat
N005	Harris	Susan
N006	Cramer	Bill

DEGREE				
D-sur	TYPE	YEAR	MAJOR	S-sur
D001	HS	1972		
D002	BS	1970	Accounting	S001
D003	HS	1944		
D004	BS	1980	Business	S002
D005	MS	1982	Business	S003
D006	BS	1974	Cs	S004

SCHOOL		
S-sur	SNAME	STATE
S001	NYU	NY
S002	GWU	DC
S003	UB	MD
S004	UMCP	MD

Again, in this proposal physical independence is achieved. In addition, full retrieval of an object or partial retrieval of sub–components of an object can be done efficiently using the join operator. For example, retrieving all the degrees of the employee Susan Harris can be performed as follows

```
{<TYPE,YEAR,MAJOR> | ∃ E-sur ∃ N-sur ∃ JOBCLASS ∃ AGE ∃ SALARY ∃ D-sur ∃ S-sur
 ( EMPLOYEE(E-sur,N-sur,JOBCLASS,AGE,SALARY,D-sur)
 & NAME(N-sur,"Harris","Susan")
 & DEGREE(D-sur,TYPE,YEAR,MAJOR,S-sur))}
```

Notice that joins are performed on surrogates only. Since the join is a very costly operation,

special support is required to accelerate its execution. In [VKC] the use of *join indices* is suggested. Join indices were originally introduced by Roussopoulos [R] and later rediscovered by Valduriez [V]. A join index for two relations R and S and for the join on columns A and B of R and S respectively, is defined as a set of pairs of surrogates

$$\{(r,s) \mid R.A = S.B\}$$

where r and s are the surrogates associated with the corresponding tuples of R and S. Hence, in order to retrieve sub–objects of a given object, a simple consultation of the join index between the relation that stores instances of the object and the corresponding relation that stores instances of the sub–object is needed. For example, again using the above example, the EMPLOYEE, NAME, DEGREE and SCHOOL records are stored according to the following schema

```
EMPLOYEE(E-sur,JOBCLASS,AGE,SALARY)
NAME(N-sur,LAST,FIRST)
DEGREE(D-sur,TYPE,YEAR,MAJOR)
SCHOOL(S-sur,SNAME,STATE)
```

and the join indices used are

```
EMPLOYEE-NAME(E-sur,N-sur)
EMPLOYEE-DEGREE(E-sur,D-sur)
DEGREE-SCHOOL(D-sur,S-sur)
```

Using join indices, the above query that requests all the degrees of Susan Harris is implemented as follows

```
{<TYPE,YEAR,MAJOR> | ∃ E-sur ∃ N-sur ∃ D-sur ∃ S-sur
 ( EMPLOYEE-NAME(E-sur,N-sur)
 & EMPLOYEE-DEGREE (E-sur,D-sur)
 & NAME(N-sur,"Harris","Susan")
 & DEGREE(D-sur,TYPE,YEAR,MAJOR))}
```

Although we show the join indices EMPLOYEE-NAME and EMPLOYEE-DEGREE in the above query, they are only *implicitly* used, i.e. they are only consulted to retrieve pairs of surrogates from the various relations involved.

The approach of [VKC] is very interesting because it allows direct manipulation of object sub–components and can comply with recursive complex objects (such as the parts example of section 3). In addition, it can be easily used as an implementation of either Zaniolo's proposal or extended database logic. Clearly, it is compatible with the relational model and can be implemented on top of any existing relational database system. One possible improvement to this scheme is the addition of a *cache*. This is simply a piece of secondary storage where previously retrieved sub–components can be stored. The reason is that even with the use of join indices, recovering all the pieces of a complex object may require a lot of time. Of course, the decision to cache an object or not depends on the frequency with which such an object is requested as well as on the probability that such an object or its components have to be updated. Sellis in [S] discusses various caching techniques. In general the support for complex objects can range from

1) none, in which case joins must be used to retrieve the various components, to

2) caching surrogate pairs (join indices), in which case the surrogates provide direct access to sub–objects, to

3) caching full object components, in which case no retrieval from the database is needed.

Clearly the cost of maintaining the above schemes increases in going from strategy (1) to (2) to (3). A compromise of the above is probably the best solution. Some objects can be stored to their whole extent, others using surrogates to identify sub–objects, etc. The specific storage requirements, retrieval and update frequencies should be used for the most efficient implementation.

8. Summary

In this paper we showed how database logic can be extended to deal with complex objects and to have deductive capabilities. Extended database logic is a powerful framework that ties together concepts involving database logic, deductive databases, and complex objects. Thus extended database logic can be used to model traditional relations, hierarchies, and networks, as well as inference mechanisms, including expert systems rules, and complex objects. Deductive heterogeneous databases may be represented in this manner.

Future research in the area should focus on several aspects. Deduction should be investigated. A formal theory should be developed to cover modeling of complex objects and the most

common operators on such objects. Also, implementation techniques need to be studied in more detail. Although preliminary work has been done in the past, there are a lot of issues that need further consideration. Such issues include a model for representing objects using relations, support for speeding–up the execution of joins, and caching techniques. Finally, a friendly user interface based on some non–procedural language should also be developed to facilitate the interaction with such a database.

9. References

[BK1] Batory, D.S. and Kim, W., "*Modeling Concepts for VLSI CAD Objects*", ACM TODS **10** (1985), pp. 322–346.

[BK2] Bancilhon, F. and Khoshafian, S., "*A Calculus for Complex Objects*", Proceedings of the Fifth PODS Symposium, 1986, pp. 53–59.

[D] Dadam, P. at al, "*A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables and Hierarchies*", Proceedings of 1986 ACM SIGMOD, pp. 356–367.

[GMN] Gallaire, H, Minker, J. and Nicolas, J.M., "*Logic and Databases: A Deductive Approach*", Computing Surveys **16** (1984), pp. 153–185.

[GS] Grant, J., and Sellis, T., "*Deductive Heterogeneous Databases*", Proceedings of the Second International Symposium on Methodologies for Intelligent Systems (1987). (to appear)

[J] Jacobs, B.E., **Applied Database Logic Vol.I**, Prentice Hall, Inc., 1985.

[KCB] Kim, W., Chou, H–T., and Banerjee, J., "*Operations and Implementation of Complex Objects*", Proceedings of the 1987 Data Engineering Conference, pp. 626–633.

[L] Lorie, R. et al.,"*Supporting Complex Objects in a Relational System for Engineering Databases*", in **Query Processing in Database Systems**, Eds. W. Kim, D.S. Reiner and D.S. Batory, Springer–Verlag, 1984.

[R] Roussopoulos, N., "*View Indexing in Relational Databases*", ACM TODS **7** (1982), pp. 258–290.

[S] Sellis, T., "*Efficiently Supporting Procedures in Relational Database Systems*", Proceedings of 1987 ACM SIGMOD, pp. 278–291.

[V] Valduriez, P., "*Join Indices*", ACM TODS **12** (1987), pp. 218–246.

[VKC] Valduriez, P., Khoshafian, S. and Copeland, G., "*Implementation Techniques for Complex Objects*", Proceedings of the 12th VLDB Conference, 1986, pp. 101–109.

[Z] Zaniolo, C., "*The Representation and Deductive Retrieval of Complex Objects*", Proceedings of the 11th VLDB Conference, 1985, pp. 458–469.