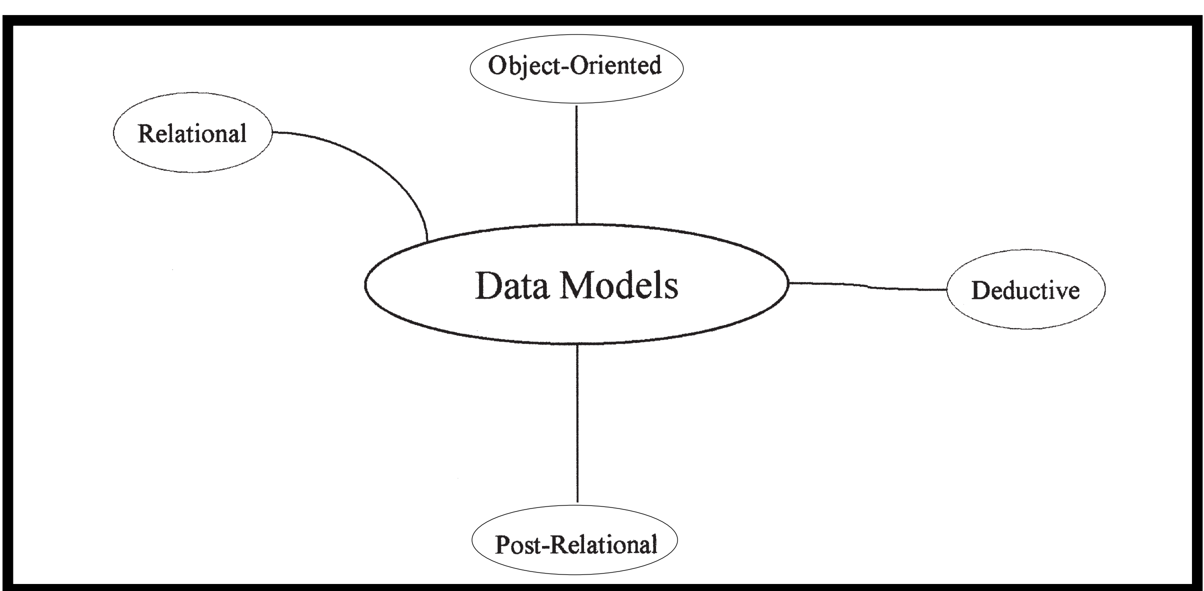
DATA MODELS

No man is an island, entire of itself,

Every man is a piece of the continent, a part of the main.

John Donne



Part 2 is devoted to the issue of data management architectures, or what we have referred to in Part 1 as a data model. An understanding of key data models is important for the following reasons:

- A data model allows us to treat a database as an abstract machine. In other words, we can concentrate on the principles of design divorced from an immediate concern with implementation. We can get the data organised suitably before building the technology
- Data models constitute formal languages for defining data structures declaring integrity and for manipulating data. A data model is a mechanism for specifying the schema of some database
- Data models establish the principles underlying DBMS

In this part we consider a number of alternative data models – relational, object-oriented, deductive and post-relational. Each data model has its own strengths and weaknesses in terms of commercial applications.

By far the largest chapter is devoted to the relational data model. This is for two reasons. First, the relational data model is unusual in having a uniform, theoretical foundation. Second, database systems based upon this data model are currently the most popular in commerce and industry.

We also consider two advanced data models: the object-oriented data model and the deductive data model. The object-oriented data model is particularly important because of its ability to capture complex processing within the remit of a database system. The deductive data model is important in that it permits the incorporation of a number of 'intelligent' features into database systems.

We conclude Part 2 with a brief examination of a number of features popular in contemporary DBMS that can be seen as extensions to the relational data model in that they incorporate certain features that may be interpreted as having an object-oriented foundation. We consider this collection of features very loosely as a data model and refer to it as the post-relational data model.

A loose chronology for the development of the data models discussed in this part of the book is illustrated in Figure P2.1. Semantic data models are addressed in the chapter on the post-relational data model.

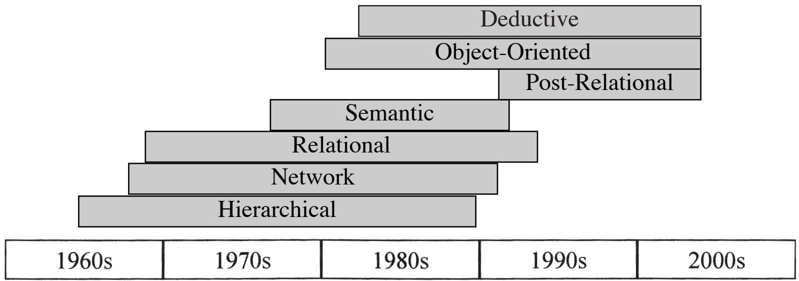


Figure P2.1 A chronology of data models.

It is tempting to describe the development of data models as an evolutionary one. For instance, the relational data model can be seen to build upon some of the foundations established by two data models which held sway in the commercial world for a number of years during the 1970s and 1980s – the hierarchical and network models. In turn, the object-oriented data model can be seen to have developed some of its features upon foundations established by the relational data model. However, in practice, of course, the picture is not so clear-cut. As an example of this, the relational model was in fact specified before any true specification emerged for the network data model.

One should also note that the chronology of data models does not correspond precisely with the chronology of DBMS founded in these data models. An example here is the way in which, although the relational data model was specifed in the early 1970s, it took some fifteen years or so for the model to start to experience a commercial impact in terms of actual DBMS.

RELATIONAL DATA MODEL

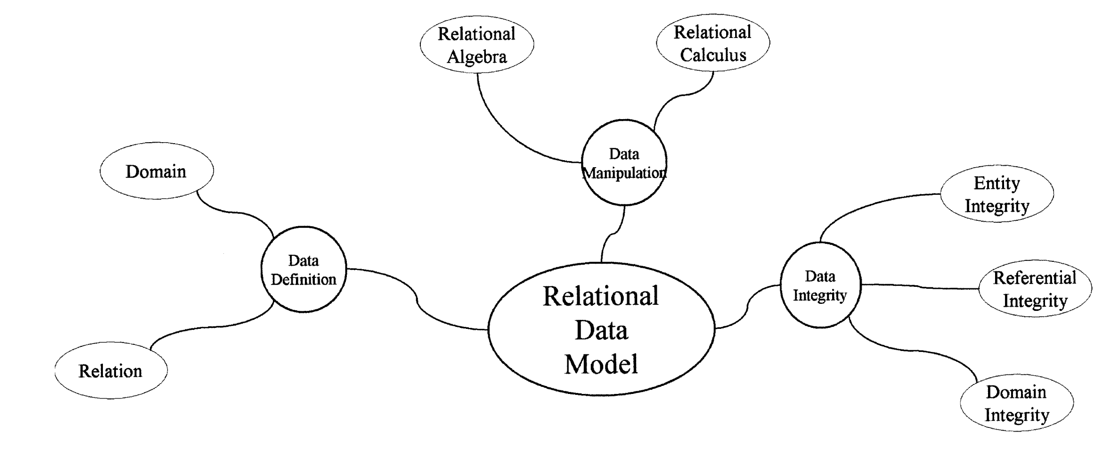
No hatred is so bitter as that of near relations.

Cornelius Tacitus (AD 55–117)

LEARNING OUTCOMES

At the end of this chapter the reader will be able to:

- Describe the history of the relational data model
- Describe the key features of data definition in the relational data model
- Describe the key features of data manipulation in the relational data model
- Describe the key features of data integrity in the relational data model



7.1 INTRODUCTION

The relational data model is unusual in being largely due to the efforts of one man, E.F. Codd. In 1970 E.F. Codd published a seminal paper which laid the foundation for probably the most popular of the contemporary data models. From 1968 to 1988 Codd published more than 40 technical papers on the relational data model. He refers to the total content of the pre-1979 papers as Version 1 of the Relational Data Model (Codd, 1990).

Early in 1979, Codd presented a paper to the Australian Computer Society at Hobart, Tasmania entitled 'Extending the relational database model to capture more meaning'. The paper was later to be published in the ACM Database Transactions (Codd, 1979). The extended version of the relational data model discussed in this paper Codd named RM/T (T for Tasmania).

Early in 1990 Codd published a book entitled *The Relational Model for Database Management: Version 2* (Codd, 1990). One of the main reasons Codd cited for publishing this work is that he believed that many vendors of DBMS products with the relational stamp had failed to understand the implications not only of RM/T but also of many aspects of Version 1 of the model. We consider the issue again in Chapter 11.

In this chapter we concentrate on describing the major elements of Version 1 of the data model. A discussion of elements of RM/T and Version 2 are included in appropriate sections.

In Chapter 10 we discuss a number of extensions that have been added to the relational data model to increase its functionality. Most of the key players in the DBMS market are built on the foundations of the relational data model but include a number of what we have called post-relational extensions.

7.1.1 OBJECTIVES

Codd has always maintained that there were three problems that he wanted to address with his theoretical work. First, he maintained that data models prior to his (see Chapter 6) treated data in an undisciplined fashion. His model was proposed as a disciplined way of handling data using the rigour of mathematics, particularly set theory. As a result of this increased rigour Codd expected two consequences to follow. Codd believed that his ideas would enhance the concept of program – data independence. He also expected to improve programmer productivity. As evidence of this, in 1982, when Codd received the ACM Turing award, the title of his acceptance paper was 'Relational database: a practical foundation for productivity' (Codd, 1982).

7.2 DATA DEFINITION

A database is effectively a set of data structures for organising and storing data. In any data model, and consequently in any DBMS, we must have a set of principles for exploiting such data structures for information systems applications

within organisations. Data definition is the process of exploiting the inherent data structures of a data model for a particular organisational application.

7.2.1 RELATIONS

There is only one data structure in the relational data model – the relation. Because the idea of a relation is modelled on a mathematical construct, a relation is a table which obeys a certain restricted set of rules:

- Every relation in a database must have a distinct name
- Every column in a relation must have a distinct name within the relation
- All entries in a column must be of the same kind. They are said to be defined on the same domain
- The ordering of columns in a relation is not significant
- Each row in a relation must be distinct. In other words, duplicate rows are not allowed in a relation
- The ordering of rows is not significant
- Each cell or column/row intersection in a relation should contain only a so-called atomic value. In other words, multiple-values are not allowed in the cells of a relation

Example The tables below conform to all of these rules and hence constitute relations:

Modules

moduleName	*level*	*courseCode*	*staffNo*
Relational Database Systems	1	CSD	244
Relational Database Design	1	CSD	244
Deductive Databases	3	CSD	445
Object-Oriented Databases	3	CSD	445
Distributed Databases	2	CSD	247

Lecturers

staffNo	*staffName*	*status*
244	Davies T	L
247	Patel S	SL
445	Evans R	PL

In contrast, the table below is not a relation because the *moduleName* column contains multiple values:

Courses	
courseCode	***moduleName***
CSD	Relational Database Systems
	Relational Database Design
	Deductive Databases
	Object-Oriented Databases
	Distributed Database Systems
BSD	Intro to Business
	Basic Accountancy

Codd originally borrowed from the terminology of mathematics to denote elements of his data model. Columns of tables are known as attributes. Rows of tables are known as tuples. The number of columns in a table is referred to as the table's degree. The number of rows in a table is referred to as the table's cardinality. Hence the cardinality of the modules relation above is 5 and the degree of the relation is 4.

Primarily because most commercial DBMS that refer to themselves as relational use the more colloquial terms table, row and column, we shall tend to conform to this practice both in this chapter and throughout the text.

7.2.2 PRIMARY KEYS

Each relation must have a primary key. This is to enforce the property that duplicate rows are forbidden in a relation. A primary key is one or more columns of a table whose values are used to uniquely identify each of the rows in a table.

In any relation there may be a number of *candidate keys*; that is, a column or group of columns which can act in the capacity of a unique identifier. The primary key is chosen from one of the candidate keys.

Example Consider the relation named Lecturers above. The attributes staffNo, staffName and status are currently candidate keys since the values in these columns are currently unique. We know in practice, of course, that as the size of the lecturers table grows, we may store information about more than one lecturer named Evans R, and more than one lecturer is likely to be a senior lecturer – an SL. This leaves staffNo – a unique code for each lecturer of a university – as the only practicable primary key.

Any candidate key, and consequently any primary key, must have two properties. It must be unique, and it must not be null – a special character being used to indicate a missing or incomplete datum. First, by definition any candidate key must be a unique identifier. Hence, there can be no duplicate values in a candidate or primary key column. Second, we must have a primary key value for each row in a table. In other words, we cannot have a null (non-existent) value within a primary key column or columns.

7.2.3 DOMAINS

The primary unit of data in the relational data model is the data-item. Such data-items are said to be non-decomposable or atomic. A set of such data-items of the same type is said to be a domain. Domains are therefore pools of values from which actual values appearing in the columns of a table are drawn.

Examples ➠ Examples of data-items include a staff number such as 244, a lecturer name such as S Patel or a student's date of birth such as 1/7/1986.

Suppose there are four members of staff in our institution with the staff numbers 244, 386, 534 and 222. The domain of staff numbers is the set of all possible staff numbers, in our case {244, 386, 534, 222}.

7.2.4 FOREIGN KEYS

Foreign keys are the means of interconnecting the data stored in a series of disparate tables. A foreign key is a column or group of columns of some table which draws its values from the same domain as the primary key of some related table in the database.

Example ➠ In our university example staffNo is a foreign key in the modules table. This column draws its values from the same domain as the staffNo column – the primary key of the lecturers table. This means that when we know the staffNo of some lecturer we can cross-refer to the lecturers table to see, for instance, the status of that lecturer.

7.2.5 NULL VALUES

A special character is used in relational systems to indicate incomplete or unknown data – the character null. This character, which is distinct from zero or space, is particularly useful in the context of primary–foreign key links.

However, the concept of null values is not without controversy. Codd maintains that the introduction of null values into relational systems turns conventional two-valued logic (true, false) into a three-valued logic (true, false, unknown). In normal two-valued logic if question 1 is true and question 2 is true the combined question is also true. In three-valued logic if question 1 is

true and question 2 is unknown then the truth-value of the combined question is unknown. This introduces additional complications into query-processing in relational systems which some commentators (e.g. Chris Date) believe is unnecessary.

7.2.6 SYNTAX FOR DATA DEFINITION

There is no agreed syntax for expressing the structure of relations. Here, we express a relational schema as a series of templates made up of relation names, attributes and primary and foreign keys declarations

Example The two templates below define the current schema for the academic database:

Domains
moduleNames: CHARACTER(15)
levels: INTEGER{1,2,3}
courseCodes: CHARACTER(3)
staffNos: INTEGER
statuses: CHARACTER(6): {L,SL,PL,Reader,Prof,HOD}
staffNames: CHARACTER(20)

Relation Modules
Attributes
moduleName: moduleNames
level: levels
courseCode: courseCodes
staffNo: staffNos
Primary Key moduleName
Foreign Key staffNo references Lecturers

Relation Lecturers
Domains
Attributes
staffNo: staffNos
staffName: staffNames
status: statuses
Primary Key staffNo

Note how we have separated domain definitions from relation definitions. We have assumed some predefined domains: character and integer – and also specified two domains using sets – lists contained in curly brackets. For instance, in the case of the column *level* we are allowing only the values 1, 2 or 3 as valid values.

7.2.7 THE BRACKETING NOTATION

It is frequently convenient to use a shorthand notation for a full relational schema definition This is known as the bracketing notation. We list a suitable

mnemonic name for the table first. This is followed by a list of data-items or column names delimited by commas. It is conventional to list the primary key for the table first and underline this data-item. If the primary key is made up of two or more attributes, we underline all the component data-items.

Example The two relations for our university example would look as follows:

Modules(moduleName, level, courseCode, staffNo)

Lecturers(staffNo, staffName, status)

7.3 DATA MANIPULATION

In this section we discuss the second of the three parts of the relational data model, that part dealing with data manipulation.

Data manipulation has four aspects:

- How we input data into a relation
- How we remove data from a relation
- How we amend data in a relation
- How we retrieve data from a relation

When Codd first proposed the relational data model, by far the most attention was devoted to the final aspect of data manipulation – data retrieval. Retrieval in the relational data model is conducted using a series of operators known collectively as the relational algebra.

7.3.1 RELATIONAL ALGEBRA

The relational algebra is a set of eight operators. Each operator takes one or more relations as input and produces one relation as output. The three main operators of the algebra are restrict, project and join. Using these three operators most of the manipulation required of relational systems can be accomplished.

The additional operators – product, union, intersection, difference and division – are modelled on the traditional operators of set theory. Figure 7.1 illustrates the operators of the relational algebra.

There is no standard syntax for the operators of the relational algebra. We therefore use here a notation designed more for the purposes of explanation than for rigour.

7.3.2 RESTRICT

Restrict is an operator which takes one relation as input and produces a single relation as output. Restrict can be considered a ‘horizontal slicer’ in

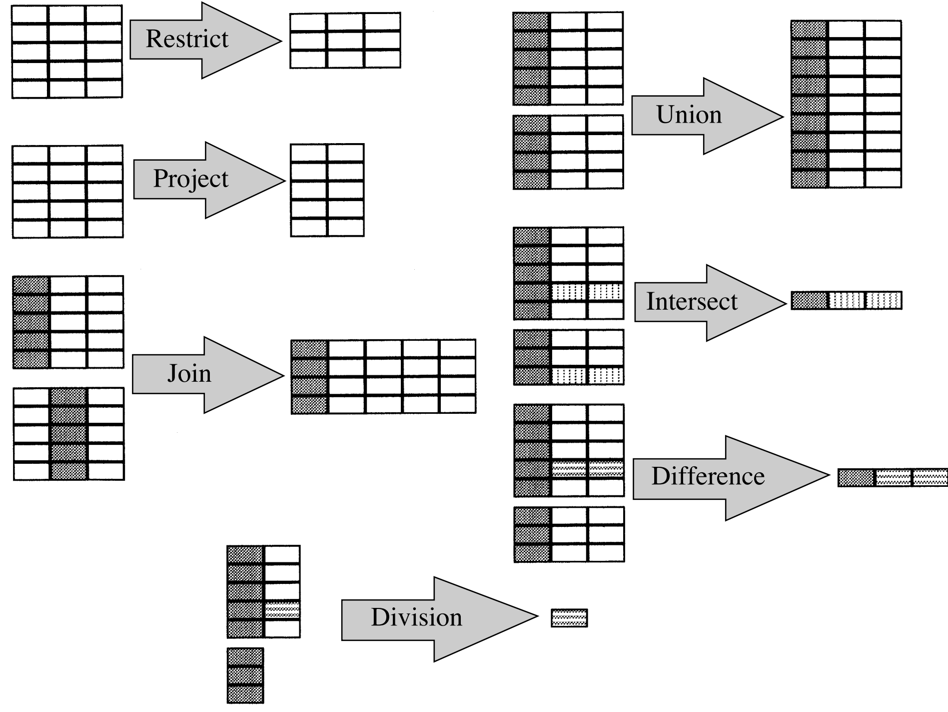


Figure 7.1 The relational algebra.

that it extracts rows from the input relation matching a given condition and passes them to the output relation. Our syntax for the restrict operator is as follows:

RESTRICT <table name> [WHERE <condition>] → <result table>

Example RESTRICT Modules WHERE level = 1 → R1

This statement would produce the following relation as output:

R1

moduleName	***level***	***courseCode***	***staffNo***
Relational Database Systems	1	CSD	244
Relational Database Design	1	CSD	244

Note that we need to provide a name, in this case R1, for the output relation. R1 must adhere to all the rules described in section 7.2.1.

7.3.3 PROJECT

The project operator takes a single relation as input and produces a single relation as output. Project is a 'vertical slicer' in that it produces in the output relation a subset of the columns in the input relation. The syntax for the project operator is as follows:

PROJECT <table name> [<column list>] → <result table>

Example PROJECT Modules(moduleName) → R1

R1

moduleName
Relational Database Systems
Relational Database Design
Deductive Databases
Object-Oriented Databases
Distributed Databases

7.3.4 PRODUCT

Joins are based on the relational operator product, a direct analogue of an operator in set theory known as the Cartesian product. Product takes two relations as input and produces as output one relation composed of all the possible combinations of input rows. Product is a little-used operator in practice because of its potential for generating an 'information explosion'. Hence a product of lecturers with modules will list all possible lecturer–module combinations. However, product is the base from which the join operator is derived. The syntax of the product operator is given below:

PRODUCT <table 1> WITH <table 2> → <result table>

Example PRODUCT Lecturers WITH Modules → R1

7.3.5 EQUI-JOIN

The join operator takes two relations as input and produces one relation as output. A number of distinct types of join have been identified. Probably the most commonly used is the natural join, a development of the equi-join.

The equi-join operator is a product with an associated restrict. In other words, we combine two tables together but only for records where the values match in the join columns of two tables. We shall assume that the primary key of one table and the foreign key of the other table form the default join

columns. (Actually, there is nothing preventing us from performing an equi-join across any two common columns between two relations.) Hence an equi-join of Lecturers with Modules will produce the table R1 below. The syntax is:

EQUIJOIN <table 1> WITH <table 2> → <result table>

Example EQUIJOIN Lecturers WITH Modules → R1

R1

moduleName	***level***	***course Code***	***staff No***	***staff No***	***staff Name***	***status***
Relational Database Systems	1	CSD	244	244	Davies T	L
Relational Database Design	1	CSD	244	244	Davies T	L
Deductive Databases	3	CSD	445	445	Evans R	PL
Object-Oriented Databases	3	CSD	445	445	Evans R	PL
Distributed Databases	2	CSD	247	247	Patel S	SL

Here we have joined Lecturers and Modules, but only produced a row in R1 where a lecturer's staffNo value matches a module's staffNo value.

7.3.6 NATURAL JOIN

Note that an equi-join does not remove the duplicate join column in the resulting table above – staffNo appears twice in R1. A natural join removes one of these join columns. The natural join operator is a product with an associated restrict followed by a project of one of the join columns. Hence a natural join of Lecturers with Modules will produce the table R1 below:

JOIN <table 1> WITH <table 2> → <result table>

Example JOIN Lecturers WITH Modules → R1

R1

moduleName	***level***	***course Code***	***staff No***	***staff Name***	***status***
Relational Database Systems	1	CSD	244	Davies T	L
Relational Database Design	1	CSD	244	Davies T	L
Deductive Databases	3	CSD	445	Evans R	PL
Object-Oriented Databases	3	CSD	445	Evans R	PL
Distributed Databases	2	CSD	247	Patel S	SL

7.3.7 OUTER JOINS

Natural joins are certainly the most common type of join used in practice. Another type of join is however important for commercial database processing: the outer join.

Example Suppose we extend our academic database as below. Note that some modules now temporarily have no lecturer (indicated by null), and some lecturers do not currently have any modules:

Modules

moduleName	*level*	*courseCode*	*staffNo*
Relational Database Systems	1	CSD	244
Relational Database Design	1	CSD	244
Deductive Databases	3	CSD	445
Object-Oriented Databases	3	CSD	445
Distributed Databases	2	CSD	247
Database Development	2	CSD	null
Data Administration	2	CSD	null

Lecturers

staffNo	*staffName*	*status*
244	Davies T	L
247	Patel S	SL
124	Smith J	L
145	Thomas P	SL
445	Evans R	PL

A natural join is designed to produce a result from two input relations R and S of only those rows in R that have matching rows in S, and vice versa. In the case above, a natural join will list all modules with lecturers.

A related set of operators known as outer joins has been proposed for use when we want to keep all rows in R or S or both in the result, whether or not they have matching rows in the other relation. There are three types of outer join: left outer joins, right outer joins and two-way outer joins (sometimes called full outer joins).

A left outer join preserves unmatched rows in the first table expressed in a join.

Examples In our case, a left outer join will list all modules with or without lecturers as below:

RLeft

moduleName	*level*	*course Code*	*staff No*	*staff Name*	*status*
Relational Database Systems	1	CSD	244	Davies T	L
Relational Database Design	1	CSD	244	Davies T	L
Deductive Databases	3	CSD	445	Evans R	PL
Object-Oriented Databases	3	CSD	445	Evans R	PL
Distributed Databases	2	CSD	247	Patel S	SL
Database Development	2	CSD	null	null	null
Data Administration	2	CSD	null	null	null

Note that those modules with no lecturers have had the staffNo, staffName and status attributes padded with nulls.

A right outer join will list all lecturers with or without modules as below.

Example

RRight

moduleName	*level*	*course Code*	*staff No*	*staff Name*	*status*
Relational Database Systems	1	CSD	244	Davies T	L
Relational Database Design	1	CSD	244	Davies T	L
Design Deductive Databases	3	CSD	445	Evans R	PL
Object-Oriented Databases	3	CSD	445	Evans R	PL
Distributed Databases	2	CSD	247	Patel S	SL
null	null	null	124	Smith J	L
null	null	null	145	Thomas P	SL

A two-way outer join will list all modules with or without lecturers and all lecturers with or without modules, as below.

Example

R2way

moduleName	***level***	***course Code***	***staff No***	***staff Name***	***status***
Relational Database Systems	1	CSD	244	Davies T	L
Relational Database Design	1	CSD	244	Davies T	L
Deductive Databases	3	CSD	445	Evans R	PL
Object-Oriented Databases	3	CSD	445	Evans R	PL
Distributed Databases	2	CSD	247	Patel S	SL
null	null	null	124	Smith J	L
null	null	null	145	Thomas P	SL
Database Development	2	CSD	null	null	null
Data Administration	2	CSD	null	null	null

7.3.8 UNION

Union is an operator which takes two compatible relations as input and produces one relation as output. By compatible is meant that the tables have the same structure – the same columns defined on the same domains.

Example Suppose we wished to union a table of lecturers information with a table of administrators data. We might do this as follows:

Administrators

staffNo	***staffName***	***status***
1001	Davies P	Clerk
1024	Jones S	Senior Clerk
445	Evans R	PL

<table 1> UNION <table 2> → <result table>

Lecturers UNION Administrators → R1

R1

staffNo	*staffName*	*status*
1001	Davies P	Clerk
1024	Jones S	Senior Clerk
244	Davies T	L
247	Patel S	SL
445	Evans R	PL

Note that although Evans R appears in both the Lecturers and Administrators table, he only appears once in R1. This is because, since R1 is a relation, it cannot have duplicate rows.

7.3.9 INTERSECTION

Intersection is fundamentally the opposite of union. Whereas union produces the combination of two sets or tables, intersection produces a result table which contains rows common to both input tables.

<table 1> INTERSECTION <table 2> → <result table>

Example

Lecturers INTERSECTION Administrators → R1

Applying intersection to the relations used in the previous section would give us a relation containing staff who were both lecturers and administrators.

R1

staffNo	*staffName*	*status*
445	Evans R	PL

7.3.10 DIFFERENCE

In most operators of the relational algebra, the order of specifying input relations is insignificant. A union of table 1 with table 2, for instance, is exactly the same as a union of table 2 with table 1. Using difference, in contrast, the order of specifying the input tables does matter.

Example

Hence

Lecturers DIFFERENCE Administrators → R1

will produce all lecturers who are not administrators

R1		
staffNo	***staffName***	***status***
244	Davies T	L
247	Patel S	SL

while

Administrators DIFFERENCE Lecturers → R1

will produce all administrators who are not lecturers

R1		
staffNo	***staffName***	***status***
1001	Davies P	Clerk
1024	Jones S	Senior Clerk

7.3.11 DIVISION

Division or divide takes two tables as input and produces one table as output. One of the input tables must be a binary table (i.e. a table with two columns). The other input table must be a unary table (i.e. a table with one column). The unary table must also be defined on the same domain as one of the columns in the binary table.

The fundamental idea of division is that we take the values of the unary table and check them off against the associated column in the binary table. Whenever all the values in the unary table match with the same value in the binary table then we output a value to the output table.

Example Suppose that we maintain a table with a list of dates on which particular modules are taught:

ModuleDays	
Module Name	***Day***
Relational Database Systems	19/12/2003
Relational Database Design	12/12/2003
Relational Database Design	19/12/2003
Object-Oriented Database Systems	20/12/2003
Object-Oriented Database Systems	20/12/2003

We might want to find a common date on which both relational database systems and relational database design is taught. Hence, our unary table would be:

PairedModules
Module Name
Relational Database Systems
Relational Database Design

If we divide the relation ModuleDays by the relation PairedModules we would get the following output relation:

R1
Day
19/12/2003

7.3.12 A PROCEDURAL QUERY LANGUAGE

The relational algebra as described above is a procedural query language. This means that to extract information from a database using the algebra we must specify a sequence of operators in which the result from each step in the sequence constitutes all or part of the input to the next step.

Example Consider the following query: *List all module names taken by T Davies*. There are a number of ways we can implement this query using the algebra. Two possible solutions are given below:

```
RESTRICT Lecturers WHERE staffName = 'Davies T' → R1
JOIN R1 WITH Modules ON staffNo → R2
PROJECT R2(moduleName) → R4

JOIN Lecturers WITH Modules ON staffNo → R1
RESTRICT R1 WHERE staffName = 'Davies T' → R2
PROJECT R2(moduleName) → R4
```

The relational algebra is a procedural query language because it demonstrates the property of closure. In other words, the output from the application of one relational operator can be passed as input to another relational operator, and so on. As we shall see some query languages, of which SQL (Part 3) is an example, do not demonstrate this property of closure.

The property of closure means that we can rewrite any sequence of algebraic statements as one nested expression, in that the result of one statement can be passed as a parameter to the next statement, and so on. This will be particularly useful when we come to specify additional constraints.

Example Below we provide a nested version of the query above.

```
PROJECT(JOIN Modules WITH
  (RESTRICT Lecturers WHERE staffName = 'Davies T')
  ON staffNo) (moduleName) → R1
```

7.3.13 RELATIONAL CALCULUS

The relational calculus is an alternative to the relational algebra as far as the manipulative part of the relational data model is concerned. The main difference between the algebra and calculus is that whereas the algebra may be described as procedural or prescriptive, the calculus is non-procedural or declarative. By this we mean that in the calculus we write an expression which specifies what is to be retrieved rather than how to retrieve it.

The algebra and calclulus are however identical in the sense that it can be shown that any retrieval specified in the algebra can also be specified in the calculus, and vice versa. The relational calculus is a language based on a branch of formal logic known as the predicate calculus. There are two variants of the relational calculus, one known as the tuple relational calculus, the other the domain relational calculus. The tuple relational calculus has formed the basis for SQL (Chapter 11), and the domain relational calculus has formed the basis for QBE (Query By Example) interfaces (Chapter 24). We shall concentrate on a brief description of the tuple relational calculus in this section.

An expression in the tuple relational calculus has the following basic form:

```
RANGE OF <tuple variable> IS <table name>
GET <tuple variable>.<attribute name>
INTO <result table>
WHERE <conditional statement>
```

A tuple variable is some named placeholder which is said to range over a particular database relation.

Example For instance:

```
RANGE OF l IS Lecturers
GET l.status
INTO R1
WHERE l.status = SL
```

This query declares a tuple variable l to range over the table Lecturers. It places all tuples matching the condition in the result table R1.

A statement in the calculus can include logical connectives (and, or, not) and the quantifiers (the existential quantifier Forsome, and the universal

quantifier Forall). Forsome, sometimes read as *there exists*, has the meaning that in a given set of tuples there is at least one tuple which satisfies a given condition.

Example The statement:

```
RANGE OF l IS Lecturers
GET l.status
INTO R1
WHERE Forsome l (l.status = SL)
```

has exactly the same result as the first statement.

Forall has the meaning that in a given set of tuples exactly all tuples satisfy a given condition.

Examples The statement:

```
RANGE OF m IS Modules
GET m.level
INTO R1
WHERE Forall m (m.level)
```

will retrieve all the levels associated with modules. This enables quite complex queries to be built as one statement, such as:

```
RANGE OF l IS Lecturers
RANGE OF m IS Modules
GET l.staffName
INTO R1
WHERE Forall m Forsome l (l.staffNo = m.staffNo)
```

which retrieves the names of all those lecturers who take all modules. In our case, of course, this would produce the empty relation as a result.

7.3.14 MAINTENANCE OPERATIONS ON RELATIONS

There are three basic maintenance operations needed to support retrieval activities specified in the algebra or calculus: insert, delete and update. We illustrate a suggested syntax for these operations below:

```
INSERT (<value>,<value>, . . .) INTO <table name>
DELETE <table name> WITH <condition>
UPDATE <table name> WHERE <condition> SET <column name> = <value>
```

Example INSERT ('Distributed Database Systems', 2,'CSD',247,'Jones S','SL') INTO Modules
DELETE Modules WITH level = 1
UPDATE Modules WHERE level = 1 SET level = 2

Whenever we apply any update operations we must ensure that none of the integrity constraints specified in a relational schema are violated. It is to this issue of integrity that we now turn.

7.4 DATA INTEGRITY

When we say a person has integrity we normally mean we can trust what that person says. We assume, for instance, a close correspondence between what that person says he or she did and what he or she actually did.

When we say a database has integrity we mean much the same thing. We have some trust in what the database tells us. There is a close correspondence between the facts stored in the database and the real world it models. Hence, in terms of our university database we believe that the fact – Jones is a lecturer – is an accurate reflection of the workings of our organisation.

Integrity is enforced via integrity rules or constraints. In the relational data model there are two inherent integrity rules: entity integrity and referential integrity. They are inherent because every relational database should demonstrate these two aspects of integrity.

7.4.1 ENTITY INTEGRITY

Entity integrity concerns primary keys. Entity integrity is an integrity rule which states that every table must have a primary key and that the column or columns chosen to be the primary key should be unique and not null.

A direct consequence of the entity integrity rule is that duplicate rows are forbidden in a table. If each value of a primary key must be distinct, no duplicate rows can logically appear in a table. Entity integrity is enforced by adding a primary key clause to a schema definition as, for instance, in the Lecturers relation in section 7.2.1.

7.4.2 REFERENTIAL INTEGRITY

Referential integrity concerns foreign keys. The referential integrity rule states that any foreign key value can only be in one of two states. The usual state of affairs is that the foreign key value refers to a primary key value of some table in the database. Occasionally, and this will depend on the rules of the business, a foreign key value can be null. In this case we are explicitly saying that either there is no relationship between the objects represented in the database or that this relationship is unknown.

Example Take the case of our university database. We observe only one foreign key – staffNo in Modules. By examining the Modules table we can verify that each Module has a staffNo value, and that each staffNo value corresponds to a value existing in the Lecturers table. In this case we can say that the relationship between module and lecturer is mandatory. Every module is assigned to a lecturer. No module record has a null staffNo value. We enforce this by adding a not null clause to the foreign key definition as below:

Relation Modules
Attributes
moduleName: moduleNames
level: levels
courseCode: courseCodes
staffNo: staffNos
Primary Key moduleName
Foreign Key staffNo **References** Lecturers **Not Null**

Domains
moduleNames: character
levels: {1,2,3}
courseCodes: character
staffNos: integer

However, suppose the rules of this organisation change. The university must now allow for the fact that new modules may not have an assigned lecturer for a certain period of time. For such modules it is decided that their staffNo value will be null until they are assigned to a lecturer (as in section 7.2.1). In this case, we assume that null is the default state for foreign keys and hence we do not add a not null clause to our foreign key declaration.

Maintaining referential integrity in a relational database is not simply a case of defining when a foreign key should be null or when it should not. It also entails defining propagation constraints. A propagation constraint details what should happen to a related table when we update a row or rows of a target table.

For every relationship between tables in our database we should define how we are to handle deletions of target and related tuples. Three possibilities exist:

- *Restricted delete.* This is the cautious approach. It means we forbid the deletion of the target row until all module rows for that key have been deleted from the related table
- *Cascades delete.* This is the confident approach. If we delete a target row all associated rows in the related table are deleted as a matter of course
- *Nullifies delete.* This is the median approach. If we delete a target row we set the foreign key values of relevant rows in the related table to null

Example In our university database if we delete a row from the Lecturers table (our target table) we must decide what should happen to related rows in the Modules table. Three possibilities exist:

- *Restricted delete*. This means we forbid the deletion of the lecturer row until all module rows for that lecturer have been deleted
- *Cascades delete*. If we delete a lecturer row all associated module rows are deleted as a matter of course
- *Nullifies delete*. If we delete a lecturer row we set the staff numbers of relevant modules to null

Relation Modules
Attributes
moduleName: moduleNames
level: levels
courseCode: courseCodes
staffNo: staffNos
Primary Key moduleName
Foreign Key staffNo **References** Lecturers **Not Null**
Delete Restricted
Update Cascades

In the definition above we have forbidden lecturer rows from being deleted until all associated module rows have been deleted. We have also enforced the constraint that any changes made to the staffNo within the Lecturers table should be reflected in the Modules table.

7.4.3 ADDITIONAL INTEGRITY

Not all aspects of integrity appropriate to a particular application can be represented using inherent integrity. This means that additional integrity must be implemented via other mechanisms.

We can add aspects of additional integrity to a relational schema by including a series of constraint definitions. Such definitions can be written as relational algebra or relational calculus expressions. We choose to write them as nested expressions of the relational algebra.

Examples For instance, although we can ensure that every module has a requisite lecturer by placing a not null clause against the foreign key staffNo, we cannot inherently represent the fact that every lecturer should have responsibility for at least one module. To do this, we must write a constraint as follows:

Constraint (Project Lecturers(staffNo)) Difference (Project Modules(staffNo)) is empty

In effect we are saying that any maintenance activity should trigger the running of this query. This constraint may then be added to the relation definition for Lecturers.

Another constraint might be to ensure that every level 3 module should be taught by a principal lecturer (PL). We write the following constraint to enforce this:

Constraint (Restrict Lecturers Where status <> 'PL') Intersection (Restrict Modules Where level = 3) is empty

7.5 FORMAL NOTATION FOR THE RELATIONAL ALGEBRA

In this chapter we have deliberately used a rather informal notation for the syntax of the relational algebra. A more formal notation for expressing operations in this formal language is presented in the table below.

Operator	Syntax	Definition
Restrict	The restrict operator transforms a single relation R into resulting tuples matching the specified condition (predicate)	$\sigma_{predicate}$ (R)
Project	The project operator transforms a single relation R into a subset consisting of specified attributes $a_1, \ldots, a_n$	$\Pi\ a_1, \ldots, a_n$ (R)
Union	The union operator produces output from two relations R and S containing all the tuples of R or S, or both R and S. Duplicate tuples are removed from the output	$R \cup S$
Difference	The difference operator produces output from two relations R and S in which output tuples exist in R but not S	R – S
Intersection	The intersection operator produces output from two relations R and S in which output tuples exist both in R and S	$R \cap S$
Natural join	A natural join is an equi-join of two relations R and S. One instance of each of the common attributes in the output relation is projected out	R |><| S

7.6 CASE STUDY: A RELATIONAL SCHEMA FOR A UNIVERSITY DATABASE

In this chapter we introduced two relations from a database of interest to a university. This was clearly heavily simplified from an actual case. Below we present a more complete version of this schema expressed in the bracketing

notation. Primary keys are underlined. Foreign keys are tagged with an asterisk – *.

Courses(courseNo, courseName, deptName, courseLeader)
CourseEnrolment(courseNo*, studentNo*, enrolmentDate, feePaid)
Modules(moduleNo, moduleName, level, courseCode)
ModuleEnrolment(moduleNo*, studentNo*, enrolmentDate)
ModuleAssessment(moduleNo*, studentNo*, assessmentNo, grade)
Lecturers(staffNo, staffName, status, salary)
Students(studentNo, studentName, gender, studentTermAddress, studentTermTelNo, studentHomeAddress, studentHomeTelNo)
Teaches(staffNo*, moduleNo*)
Prerequisites(moduleNo*, moduleNo*)

Note that we have introduced a different primary key for the relation Modules. We have also introduced a link relation (Chapter 16) to connect modules with lecturers. The primary key of this relation – ModuleEnrolment – contains two foreign keys from the linked relation. A number of other link relations exist in this schema – CourseEnrolment, ModuleEnrolment, ModuleAssessment, Teaches and Prerequisites. This last relation is interesting in that it builds a hierarchy of prerequisite conditions between modules. In other words, you cannot take module A without first having completed module B successfully.

The reason why moduleEnrolment and moduleAssessment are two separate relations will be explained in the chapter on normalisation (Chapter 18).

7.7 SUMMARY

- The relational data model has one data structure, the relation. The relation is a restricted table
- Relations are defined in terms of attributes, tuples, primary keys, foreign keys and domains
- A special character exists in relational systems for representing incomplete or missing information – null
- The manipulative part of the relational data model has some eight operators bundled together as the relational algebra
- The relational calculus is an alternative but equivalent formalism for the manipulative part of the relational data model
- The model also has two inherent integrity rules: entity and referental integrity

7.8 ACTIVITIES

1. Create a relational schema to hold information about insurance policies and policy holders. Insurance policies have properties policyNo, startDate, premium,

renewalDate, policyType. Policy holders are characterised by holderNo, holderName, holderAddress and holderTelno.

2. What would be suitable primary keys in this database?
3. How do we relate insurance policy information with policy holder information?
4. The insurance company want to enforce the business rule – every policy must have a holder. How do we enforce this rule in our database?
5. Declare a suitable domain for policy type.
6. Write a relational algebra statement to insert new holder information into the database.
7. Write a statement to update existing holder information in the database.
8. Write a statement to delete existing holder information from the database.
9. Write a relational algebra statement to list all standard life policies.
10. Write a statement to list the holder numbers and names of all persons holding standard life policies.
11. Write a constraint to enforce the fact that every policy holder should hold at least one policy in the database.
12. Given the implications of activity 4 and activity 11, are inner and outer joins relevant to this insurance database?

7.9 REFERENCES

Codd, E.F. (1979). Extending the relational database model to capture more meaning. *ACM Trans. on Database Systems* **4**(4): 397–434.

Codd, E.F. (1982). Relational database: a practical foundation for productivity. *Comm. of ACM* **25**(2).

Codd, E.F. (1990). *The Relational Model for Database Management: Version 2*. Reading, MA, Addison-Wesley.

OBJECT-ORIENTED DATA MODEL

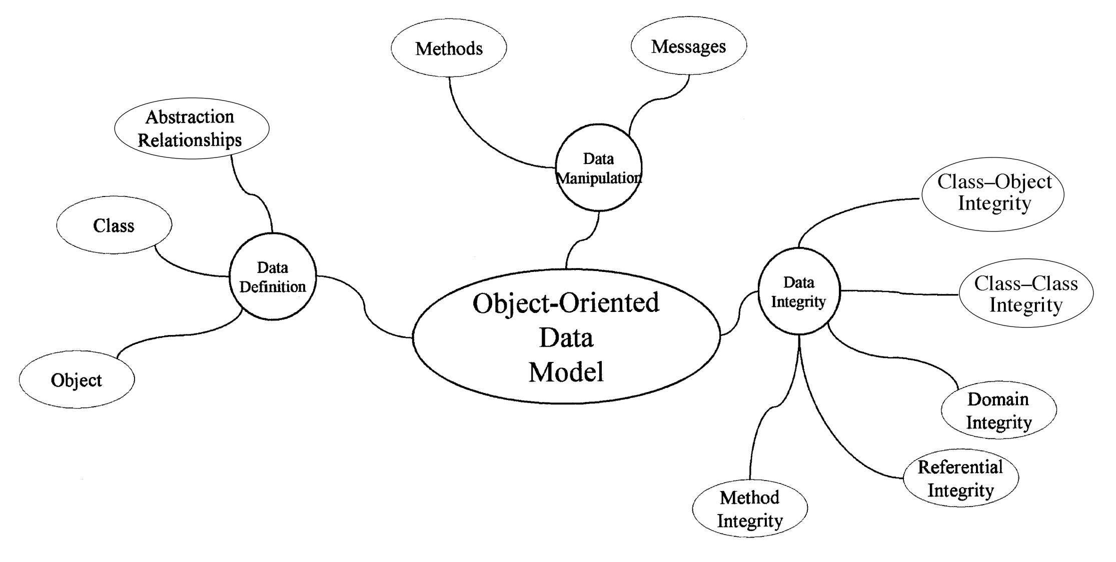
I never saw an ugly thing in my life: for let the form of an object be what it may – light, shade, and perspective will always make it beautiful.

John Constable (1776–1837)

LEARNING OUTCOMES

At the end of this chapter the reader will be able to:

- Explain the evolution of the object-oriented (OO) data model from semantic data models
- Introduce an approach to object-orientation founded in extensions to the relational data model
- Describe the key features of the OO data model in terms of data definition, data manipulation and data integrity



8.1 INTRODUCTION

An OVUM report published in 1988 predicted that database systems adhering to an object-oriented (OO) data model as opposed to a relational data model would overtake relational database systems by the mid-1990s (Ovum, 1988). Although this clearly has not happened, there is no doubt that object-oriented concepts are influencing the development of information systems in general and database systems in particular. As evidence of this many relational DBMS vendors are beginning to offer OO features, and SQL3 (Chapter 31) includes some features of object-orientation in what was originally a relational database sublanguage. However, there is some confusion about what it actually means to be object-oriented. The present chapter attempts a brief discussion of this issue in terms of database systems.

The term object-oriented is used in a number of different senses within contemporary computing. The term was first applied to a group of programming languages descended from a Scandinavian invention known as SIMULA. SIMULA was the first language to introduce the concept of an abstract data type (Chapter 10), an integrated package of data structures and procedures. However, probably the best known object-oriented programming language is SMALLTALK – a programming environment developed at the Xerox Park Research Institute.

It is only comparatively recently that the term object-oriented has been applied to database systems. The main difference between object-oriented programming languages and databases is that object-oriented database systems require the existence of persistent objects (Chapter 1). In object-oriented programming languages, objects exist only for the span of program execution. In object-oriented database systems, objects have a life in secondary storage over and above the execution of programs.

There is still some debate about whether object-oriented database systems offer a better alternative to database management than relational database systems. There is no doubt that an object-oriented approach seems particularly well-suited to certain non-standard applications such as geographical information systems or computer-aided design. However, a question remains as to the use of object-orientation in standard commercial applications for databases.

Having said this, many vendors of relational DBMS have included facilities such as user-defined data types, triggers and stored procedures and described these as object-oriented features. We address this issue of so-called object-relational systems in Chapter 10.

8.1.1 SEMANTIC DATA MODELS

In Chapter 3 we made a distinction between the classic data models, such as the network and relational data models, and the semantic data models. The classic data models represent schemas using the concepts of record structures and links between record structures. The semantic data models are so named because they attempt to provide a richer set of constructs with which to build

schemas. Such constructs, it is maintained, allow the database developer to incorporate within a database system more of the meaning or semantics of some application domain.

The semantic data models were originally developed to address some of the inherent weaknesses of the relational data model, such as:

- The relational data model is held to provide poor representation of 'real-world' entities. This is because of the fragmentation of an entity or classes into numerous relations through the process of normalisation (Chapter 18)
- The relational model is said to suffer from semantic overloading. This is because the data model uses one construct for both entities and relationships
- The relational data model suffers from the difficulty of managing complex objects. The relational constraint of atomic values means that hierarchical/nested structures cannot be accommodated easily

During the 1970s and 1980s a great number of alternative models to the relational model were proposed. Many of these approaches are surveyed in a paper by Hull and King (1987). One of the most popular groups of semantic data models is based around the entity–relationship approach of Chen (1976). This approach is the topic of Chapter 16. Most of the semantic data models extend Chen's basic approach with a number of abstraction mechanisms, the most popular being generalisation and aggregation. An example from this group of data models is SDM developed by Hammer and McCleod (1981). Generalisation and aggregation, as we shall see, form the basis for many object-oriented approaches.

There is little agreement also concerning the difference between an object-oriented data model and many of the semantic data models such as SDM. Perhaps a possible distinction can be made using a structural/behavioural framework. Semantic data models are usually seen as mechanisms for producing structural abstraction. In contrast, object-oriented data models are geared more towards providing behavioural abstraction. In other words, semantic data models are geared more towards the representation of data while object-oriented data models are geared more towards the manipulation of data (King, 1988). In this sense, some people have questioned whether an OO data model is not something of a misnomer, since it contains a process as well as data component.

8.2 EXTENDED RELATIONAL APPROACHES

Two alternative approaches to object-orientation in the context of databases have been proposed. The first is to replace the relational data model with a semantically richer data model more clearly suited to the needs of object-orientation. This is the topic of the current chapter. The second alternative is to exploit the capabilities of the relational model and to extend them with object-oriented facilities. This is the topic of Chapter 10.

8.3 COMPONENTS OF AN OO DATA MODEL

In this chapter we consider a more radical solution to the problem of introducing object-orientation into the database arena. However, unlike the relational data model, no one person has defined the critical elements underlying OO database systems. Nevertheless, there is a developing concensus emerging as to the key elements of an OO data model (Atkinson *et al.*, 1990). A number of groups have attempted to develop standards for object data management. For instance, the Object Database Management Group (ODMG) was formed in 1991 by members of a limited number of companies, the objective being to develop a set of standards for object data management that would encourage portability of applications (Eaglestone and Ridley, 1998).

Because there is no readily agreed syntax for an OO data model, we use here a simplified syntax designed to display the general features of a possible model. It is very loosely based on features of the ODMG model (Chapter 32) and the facilities of the OODBMS described in Chapter 35.

8.4 DATA DEFINITION

An OO database is made up of objects and object classes linked via a number of abstraction mechanisms.

8.4.1 OBJECTS

An object is a package of data and procedures. Data is contained in attributes of an object. Procedures are defined by an object's methods. Methods are activated by messages passed between objects.

An OO data model must provide support for object identity. This is the ability to distinguish between two objects with the same characteristics. The relational data model is a value-oriented data model. It does not inherently support the notion of a distinct identifier for each object in the database. Hence, in the relational data model two identical tuples or rows must fundamentally refer to the same object. In an object-oriented database two identical records can refer to two distinct objects via the notion of a unique system-generated identifier.

All objects must demonstrate the property of encapsulation. This is the process of packaging together of both data and process within a defined interface and controlled access across that interface. Encapsulation is implicit in the definition of an object since all manipulation of objects must be done via defined procedures attached to objects. The metaphor of an object as a gene or egg has been used to emphasise its character as containing all the information needed for it to behave appropriately in a given environment.

8.4.2 OBJECT CLASSES

An object class is a grouping of similar objects. It is used to define the attributes, methods and relationships common to a group of objects. Objects are therefore instances of some class. They have the same attributes and methods. In other words, object classes define the intension of an OO database – the central topic of database design. Objects define the extension (sometimes referred to as the extent) of an OO database – the central topic of database implementation.

8.4.3 ABSTRACTION MECHANISMS

An object data model supports two mechanisms which allow the database builder to construct hierarchies or lattices of object classes. Two such abstraction mechanisms are normally discussed: generalisation and aggregation.

Implicit in the construction of object classes is the support for the abstraction mechanism of generalisation. This allows us to declare certain object classes as subclasses of other object classes. For instance, *Manager*, *Secretary* and *Technician* might all be declared subclasses of an *Employee* class. Likewise, *SalesManager*, *ProductionManager* etc. would all be declared subtypes of the *Manager* class. Aggregation is the process by which a higher-level object is used to group together a number of lower-level objects. For instance, a *Car* entity might be built up of an assembly of wheels, chassis, engine etc.

An OO data model must also supply some means of linking objects to object classes. In this case an object class is said to classify an object, or in contrast an object is said to instantiate an object class.

8.4.4 INHERITANCE

Associated with the idea of a generalisation hierarchy is the process of inheritance. There are two major types of inheritance: structural inheritance and behavioural inheritance.

In structural inheritance a subclass is said to inherit the attributes of its superclass. Hence, suppose an employee class has attributes name, address, salary, then a subclass such as manager will also be defined by such attributes. In behavioural inheritance a subclass is said to inherit the methods of its superclass. Hence, if the employee class has a method calculatePay, then a subclass such as manager will also be characterised with this method.

We may also distinguish between single and multiple inheritance. If an OO database displays single inheritance then a class may be the subclass of only one superclass. Hence, a manager could not both be an employee and a customer. If an OO database displays multiple inheritance then a class may inherit the attributes or methods associated with more than one superclass. In this case, managers could be both employees and customers.

8.4.5 DEFINING A SCHEMA

Defining a schema for an OO database means defining object classes.

Example We define three simple object classes for a university database below:

```
CREATE CLASS Student
SUPERCLASS: object
ATTRIBUTE studentName: CHARACTER,
ATTRIBUTE termAddress: CHARACTER,
ATTRIBUTE dateOfBirth: DATE,
ATTRIBUTE dateOfEnrolment: DATE

CREATE CLASS Undergraduate
SUPERCLASS: Student
RELATIONSHIP Course memberOfCourse INVERSE courseEnrolment
RELATIONSHIP SET(Module) memberOfModule INVERSE Module moduleEnrolment

CREATE CLASS Postgraduate
SUPERCLASS: student
RELATIONSHIP Department assignedTo INVERSE Department registers

CREATE CLASS Assessment
ATTRIBUTE studentDetails: Student,
ATTRIBUTE moduleDetails: Module
```

The schema definition is made up of the following elements:

- Each class is defined by a unique name in the schema: Student, Undergraduate, Postgraduate, Assessment
- Three classes are defined on superclasses. The Student class is said to form a subclass of a system class (meta-class) called object
- The attributes of the class Student have been defined on pre-established data types integer, character and date. These are said to be simple attributes since they take a literal value and are defined on a standard data type such as integer
- The subclasses Undergraduate and Postgraduate contain no attribute declarations, but contain relationship declarations for memberOf and assignedTo. For instance, the Undergraduate class is declared as being related to the Module class. It is implicitly defined as a one-to-many link from Undergraduate to Module since the relationship is implemented as a set of Module identifiers. Both relationships have inverses defined. The inverse indicates a name for the traversal of the relationship from the linked class. Hence, the inverse relationship from Module to Undergraduate is defined as moduleEnrolment. This traversal will be defined in the Module class and is also likely to be one-to-many

- The class Assessment might be considered one in which the concept of aggregation was important. We might define an assessment as being made up of student and module details. The two attributes in this class are complex attributes since they contain collections and/or references – values that refer to other objects. Here the attributes are defined as references to specific instances of other classes

8.5 DATA MANIPULATION

Data manipulation in the OO data model is accomplished by passing messages to methods defined on objects.

8.5.1 METHODS

Methods come in four major forms:

- *Constructor methods*. These operations create new instances of a class
- *Destructor methods*. These operations allow the user to remove unwanted objects
- *Accessor methods*. These operations yield properties of objects
- *Transformer methods*. These operations yield new objects from existing objects

Examples In the case of a banking application the following methods might be appropriate:

- *Constructor methods*. We would need a method for creating a new instance of a bank account given values for attributes such as accountNumber, customer details etc.
- *Destructor methods*. This type of method would allow users to remove existing bank accounts
- *Accessor methods*. A typical accessor method might be the calculation of service charges for a given bank account
- *Transformer methods*. Examples of such methods might be debit and credit operations on a bank account

We may also distinguish between public methods, which are visible to the user and to other objects, and private methods, which are used mainly to perform internal operations for a class.

Methods would normally be assigned to classes at the definition stage.

Example

```
CREATE CLASS Module
    ATTRIBUTE moduleName: STRING,
    ATTRIBUTE level: INTEGER,
    ATTRIBUTE roll: INTEGER,
    RELATIONSHIP Course givenOn,
    RELATIONSHIP List(Lecturer) taughtBy,
    METHOD increaseRoll(amount: INTEGER)
```

The transformer method declaration illustrated here constitutes only the method's 'signature'. In other words, it merely indicates the name of the method, and the name and types of the arguments. The body of a method would be defined using a standard programming language.

8.5.2 MESSAGES

Messages need to contain the name of a method, the object identifier (or some other way of accessing objects) and the appropriate parameters for the method.

Example A message to initiate the increaseRoll method declared above might look like:

```
Module where moduleName = relationalDatabaseSystems increaseRoll(30)
```

8.5.3 CREATING AND MANIPULATING A DATABASE

In this section we give two examples of the use of methods for data manipulation.

Examples Suppose we wish to create a student object. We might write a message:

```
Student create(studentName: 'Johns S', termAddress: '14, BelleVue Ave, Cardiff', dateOfBirth: 07/3/1977, date Of Enrolment: 1/10/1998)
```

We might delete objects using the folowing statement:

```
Student where dateOfEnrolment > 01/01/1990 delete)
```

Here we are passing a condition as a parameter to the delete method.

8.6 DATA INTEGRITY

Inherent integrity in the OO model arises from the relationship between objects and classes. Since all access to objects must take place via methods,

additional integrity can be implemented via constraint definitions embedded in methods.

8.6.1 INHERENT INTEGRITY

An object-oriented data model can display:

- *Class–class integrity*. A superclass should not be deleted until all associated subclasses have been deleted
- *Class–object integrity*. A class should not be deleted until all associated objects have been deleted
- *Domain integrity*. Attributes are defined on pre-established classes such as integer, user classes such as Lecturer or sets of object identifiers
- *Referential integrity*. Since classes can be related to other classes via relationships, referential integrity issues similar to those discussed for relations exist in the OO data model

Examples

- *Class–class integrity*. Hence a student class should not be deleted until undergraduate and postgraduate are deleted
- *Class–object integrity*. Hence, the class undergraduate should only be deleted if all objects like 'Steve Austin' and 'Steve Jones' have been deleted
- *Domain integrity*. For example, a course attribute of a class module might be defined on the set of course IDs
- *Referential integrity*. Hence a lecturer should not be deleted who has outstanding modules in the database

8.6.2 ADDITIONAL INTEGRITY

Additional integrity can be added to an OO data model by adding constraints to the body of a method.

Example

Suppose we attach the following method to the Student class:

```
enrolStudent(courseCode: Course)
```

The body of this method might be expressed in pseudo-code as:

```
body enrolStudent(courseCode: Course)
conditions
  student exists
  courseCode exists
  roll of course not exceeded
```

actions

```
create new instance of Enrolment class
update Student dateOf Enrolment
```

The conditions of this method enforce a number of constraints before performing any action.

8.7 CASE STUDY: CLASS SCHEMA FOR ACADEMIC DATABASE

To illustrate how we can use two different data models for specifying the same universe of discourse we present a simplified class schema for the academic database presented in the case in Chapter 7. We have only included the class name and any generalisation or association relationships in the schema.

```
CREATE CLASS Department
SUPERCLASS Object
RELATIONSHIP SET(PostgraduateStudent) registers INVERSE PostgraduateStudent
assignedTo

CREATE CLASS Course
SUPERCLASS Object
RELATIONSHIP SET(UndergraduateStudent) courseEnrolment INVERSE
UndergraduateStudent memberOfCourse
RELATIONSHIP Lecturer courseLeader INVERSE Lecturer Leads

CREATE CLASS Module
SUPERCLASS Object
RELATIONSHIP SET(UndergraduateStudent) moduleEnrolment INVERSE
UndergraduateStudent memberOfModule
RELATIONSHIP LIST(Lecturer) taughtBy INVERSE Lecturer teaches
RELATIONSHIP SET(Module) prerequisite

CREATE CLASS Student
SUPERCLASS Object

CREATE CLASS UndergraduateStudent
SUPERCLASS Student
RELATIONSHIP Course memberOfCourse INVERSE courseEnrolment
RELATIONSHIP SET(Module) memberOfModule INVERSE Module moduleEnrolment

CREATE CLASS PostgraduateStudent
SUPERCLASS Student
RELATIONSHIP Department assignedTo INVERSE Department registers

CREATE CLASS ModuleAssessment
ATTRIBUTE studentDetails:Student
ATTRIBUTE moduleDetails:Student
```

```
CREATE CLASS Lecturer
RELATIONSHIP Set(Module) Teaches INVERSE Module taughtBy
RELATIONSHIP Course Leads INVERSE Course courseLeader
```

8.8 SUMMARY

- An OO database is made up of objects and object classes linked via a number of abstraction mechanisms
- An object is a package of data and procedures. Data is contained in attributes of an object. Procedures are defined by an object's methods. Methods are activated by messages passed between objects
- An object class is a grouping of similar objects. It is used to define the attributes, methods and relationships common to a group of objects. Objects are therefore instances of some class
- An object data model supports two abstraction mechanisms: generalisation and aggregation
- There are four types of methods: constructor, destructor, accessor and transformer
- There are four types of inherent integrity: class–class, class–object, domain and referential integrity
- Additional integrity can be defined in constraints associated with methods

8.9 ACTIVITIES

1. Declare the classes discussed in section 8.4.3 (employee, manager etc.) as an OO schema.
2. Declare a range of appropriate methods for the schema.
3. Declare some integrity constraints for the schema.
4. Describe the main differences between the OO data model and the relational data model.
5. Investigate in what ways the OO data model is similar to aspects of the hierarchical and network data models.
6. Declare the schema in section 7.6 as an OO schema.
7. What methods might be appropriate for this schema?

8.10 REFERENCES

Atkinson, M., D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich and S. Zdonik (1990). The object-oriented database system manifesto. In W. Kim, J.-M. Nicolas and S. Nishio (Eds), *Deductive and Object-Oriented Databases*. North-Holland, Elsevier Science.

Chen, P.P.S. (1976). The entity–relationship model: towards a unified view of data. *ACM Trans. on Database Systems* **1**(1): 9–36.

Eaglestone, B. and M. Ridley (1998). *Object Databases: An Introduction*. Maidenhead, Berkshire, UK. McGraw-Hill.

Hammer, M. and D. McCleod (1981). Database description with SDM: a semantic database model. *ACM Trans. on Database Systems* **6**(3): 351–86.

Hull, R. and R. King (1987). Semantic database models: survey, applications and research issues. *ACM Computing Surveys* **19**(3): 201–60.

King, R. (1988). My cat is object-oriented. In W. Kim and F. Lochovsky (Eds), *Object-Oriented Languages, Applications and Databases*. Reading, MA, Addison-Wesley.

Ovum (1988). *The Future of Databases*. London, Ovum Press.

DEDUCTIVE DATA MODEL

The real problem is not whether machines think but whether men do.

B.F. Skinner (1904–90)

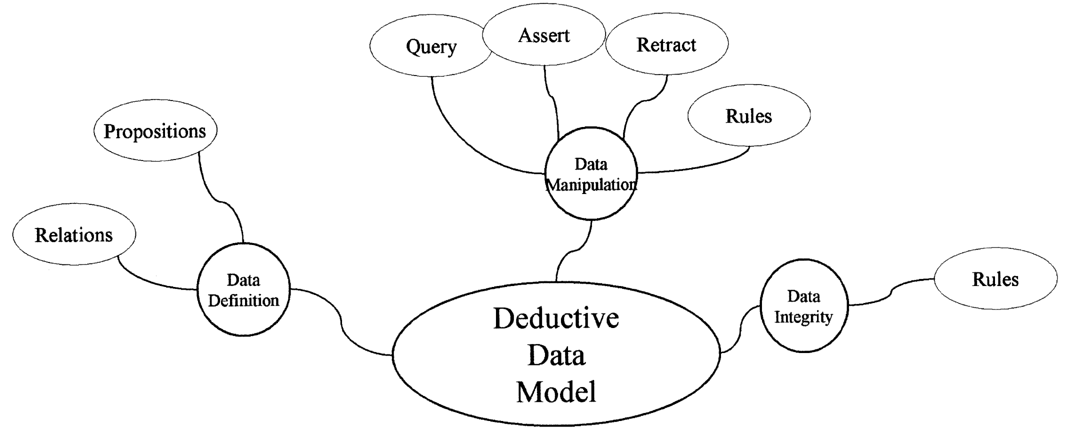
The grand aim of all science is to cover the greatest number of empirical facts by logical deduction from the smallest number of hypotheses or axioms.

Albert Einstein (1879–1955)

LEARNING OUTCOMES

At the end of this chapter the reader will be able to:

- Discuss the genesis of the deductive data model in formal logic
- Describe the major elements of the deductive data model
- Explain why deductive databases are sometimes described as 'intelligent' databases



9.1 INTRODUCTION

The deductive data model fundamentally involves the application of formal logic to the problems of data definition, data manipulation and data integrity. Logic can act as a formal basis for expressing in one uniform manner these three aspects of a data model. Logic also provides the capacity to extend the idea of a conventional database with deductive facilities.

Research on the relationship between databases and formal logic dates back at least to the late 1970s. Codd's early work on the relational data model was heavily motivated by formal logic, particularly the development of the relational algebra and the relational calculus. The principal stimulus for the interest in this relationship seems to have been the publication of a landmark paper by Reiter (1984). In this paper Reiter characterised the traditional perception of databases as being model-theoretic. He contrasted this with a proof-theoretic view of databases. It is to this proof-theoretic view of databases that the term deductive databases is normally applied. This paper built upon the earlier work published by Reiter and others in a collected volume edited by Gallaire and Minker (1978).

Most of the work on deductive databases has been done using various variants of standard Prolog known collectively as Datalog. Prolog is an Artificial Intelligence language designed to implement aspects of formal logic as a programming environment. Datalog is fundamentally a variant of Prolog specifically designed for database work. This chapter constitutes a tutorial description of Datalog. For a more formal discussion of Datalog see Gardarin and Valduriez (1989).

9.2 DATA DEFINITION IN DATALOG

In this section we describe the basic data structures of the deductive data model. We assume that a database sublanguage based on formal logic can be built on the basis of the data model.

9.2.1 PROPOSITIONS AND RELATIONS

The fundamental data elements in the deductive data model are propositions. Propositions may combine to form data structures called relations.

Example Let us suppose that we want to represent the following assertions relevant to the university database first introduced in Chapter 1:

Relational Database Systems is a module
John Davies is a student
John Davies is taking Relational Database Systems

In Datalog we represent such assertions as:

```
module(relationalDatabaseSystems).
student(johnDavies).
takes(johnDavies, relationalDatabaseSystems).
```

Within logic the terms 'module', 'takes' and 'student' are called predicates, while the terms 'relationalDatabaseSystems' and 'johnDavies' are called arguments. Predicates are written first followed by arguments. Arguments are enclosed in brackets, delimited by commas. The whole assertion is concluded with a period.

Predicates and arguments form a collection known as a proposition, and each proposition may take only one of two possible values, namely, true or false.

Example Hence, each of the following propositions has a truth-value.

```
module(relationalDatabaseSystems).
module(relationalDatabaseDesign).
```

In the area of databases we assume that if a proposition is written with concrete arguments then it is assumed to be true. A collection of propositions with the same predicate is known as a relation.

9.2.2 CONSTANTS AND VARIABLES

Each argument in a proposition can be either a constant or a variable. A constant indicates a particular individual or class of individuals. A variable is a placeholder in the sense that it indicates that such an individual or class of individuals remains unspecified.

Example Hence, one interpretation of the following proposition would be *there is some entity X that is a student*:

```
student(X).
```

In Datalog, it is conventional to write constants beginning with a lower-case letter. Variables begin with an upper-case letter.

Example Hence, the following are constants:

johnDavies
hywelEvans

while the following are variables:

Student
Lecturer

When a variable becomes filled with or bound to a constant, that variable is said to be instantiated.

Example We might therefore instantiate the variable X in the proposition student(X) with the constant johnDavies.

A proposition or assertion with all constants for arguments is said to be a ground proposition or clause.

Example Hence, the proposition below is a ground proposition:

student(johnDavies).

The extension of a deductive database consists primarily of ground propositions or clauses.

9.2.3 CONNECTIVES

Individual propositions are often referred to as being 'atomic'. By this is meant that the internal structure and meaning of such propositions are not determined by the logic. They are in a sense the primitive material from which we build. Such atomic propositions may however be combined by the use of logical connectives with special symbols. These include:

$\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT) and $\Rightarrow$ IMPLIES

Example Thus we might have the following compound proposition:

lecturer (hywelEvans) $\wedge$ teaches(hywelEvans, relationalDatabaseSystems).

which reads, *Hywel Evans is a lecturer and Hywel Evans teaches Relational Database Systems*.

Because the early Prolog systems could not handle symbols such as $\wedge$, $\vee$, $\Rightarrow$, these connectives are usually written in the following way:

$\wedge$ as , (comma)
$\vee$ as ; (semicolon)
$\Rightarrow$ as :- (colon followed by a hyphen)

Example Hence, the statement above would be rewritten:

lecturer (hywelEvans), teaches(hywelEvans, relationalDatabaseSystems).

9.3 DATA MANIPULATION IN DATALOG

The uniformity of the deductive database approach can be demonstrated by the way in which there exists a seamless join between data definition and data manipulation.

9.3.1 ASSERT AND RETRACT

We add assertions to our database using the in-built function assert. We deny assertions by using the in-built function retract.

Example Hence, to initialise the university database we might issue the following assert statements:

```
assert(module(relationalDatabaseSystems))
assert(student(johnDavies))
assert(takes(johnDavies, relationalDatabaseSystems))
```

To remove reference to johnDavies from our database we issue the following statement:

```
retract(student(johnDavies))
```

Note that we assume that assert and retract modify propositions stored in a persistent store.

9.3.2 BASIC QUERY FUNCTION

To issue a basic query we merely write an atomic proposition with constants for arguments. The system then sees if it has an assertion which matches in the database. If it does, it replies *true* or *yes*. If it does not, it replies *false* or *no*.

Example Suppose we wished to see if hywelEvans is a lecturer. We would issue the following statement as a query:

```
lecturer(hywelEvans)
```

If an assertion with this exact form is found in the database then the system would respond *true*.

9.3.3 QUERIES WITH VARIABLES

Let us now assume that we wish to list all the lecturers in our database. To do this we need to write a proposition with a variable as an argument.

Example The simplest form of query of this type would be:

```
lecturer(X)
```

where X is a variable.

In a Prolog system, the inference engine would look in its program and match on the first assertion of this form – i.e. with lecturer as predicate and a single argument. It then binds the constant it finds to X and displays the result.

Example

```
lecturer(hywelDavies)
```

The Prolog engine would then respond 'Next Solution'. If the user replies 'yes' it then attempts to make the next possible binding. This process continues until the engine runs out of bindings, or the user responds 'no'.

In Datalog we assume that the engine will not return one solution, but it will return a set of solutions by default. Hence, to perform practicable queries in Datalog we need to assume the presence of some function such as *forall*. Bundling a query within this function will return the entire set of base assertions matching the query.

Example Hence, the statement:

```
forall(lecturer(X))
```

will produce all the possible lecturer names in our database.

9.4 DATA INTEGRITY IN DATALOG

Enforcing data integrity in the deductive data model means writing rules. Rules can also be used to produce virtual data, and handle recursive information.

9.4.1 THE IMPLIES CONNECTIVE

The IMPLIES connective (section 9.2.3) is particularly important in the sense that it allows us to construct rules of the form:

takes(Student, Module) $\wedge$ teaches(Lecturer, Module)
$\Rightarrow$ assesses (Lecturer, Student).

That is, *if a student takes a module and a lecturer teaches that module, then that lecturer assesses that student.* This could be rewritten using Datalog symbols as:

```
assesses (Lecturer, Student) :–
    takes(Student, Module),
    teaches(Lecturer, Module).
```

Note that the conclusion of the rule, the rule head, is written first followed by an implies connective (:–), followed in turn by the body of the rule, a series of conditions delimited with commas.

9.4.2 UPDATE FUNCTIONS

In Chapter 1 we identified the following update function as being of relevance to the academic database:

Offer Module – assert that the module is offered.

Update functions are written as rules in Datalog. Hence offer module might be written in its simplest form as:

```
offerModule(X):–assert(module(X))
```

Hence, if we issued a statement such as:

```
offerModule(relationalDatabaseDesign)
```

to the Datalog engine the system would look for a matching predicate. X would then be instantiated to relationalDatabaseDesign. Finding the rule above it would proceed to execute the rule body. This would add the assertion:

```
module(relationalDatabaseDesign).
```

to the database.

9.4.3 INTEGRITY CONSTRAINTS

We have already mentioned that integrity constraints are rules. In fact, integrity constraints are usually associated with update functions.

Example Suppose we wish to implement the following update function and its associated constraints:

> Enrol Student in Module – if the module is offered and the person is a student, then assert that the student takes the module.

This might be implemented as follows:

```
enrolStudentModule(S, M):–
   student(S),
   module(M),
   assert(takes(S,M)).
```

For this rule to work then all the conditions of the rule have to be true.

Example Hence, if we issued the statement:

```
enrolStudentModule(terryConnolly, relationalDatabaseDesign)
```

the system would not add an assertion to the database. It would not do this because the first condition, namely that S be a student, is currently false.

9.4.4 QUERIES ON A DATABASE WITH RULES

Introducing rules into our database causes a small complication in the way in which queries are handled.

Example Suppose our database (deductive) looks as follows:

```
module(relationalDatabaseSystems).
student(johnDavies).
lecturer(hywelEvans).
teaches(hywelEvans, relationalDatabaseSystems).
takes(johnDavies, relationalDatabaseSystems).

assesses (Lecturer, Student) :–
takes(Student, Module),
teaches(Lecturer, Module).
```

Suppose also that we run the following query against our database:

```
forall(assesses(L, S))
```

In other words, we wish to find all combinations of assessment relationships. The Datalog engine would look in the database for the predicate assesses. It finds no assertions for this but finds the predicate attached to a rule. It proceeds to execute

the conditions of the rule body in sequence from first condition to last condition. The first thing it does is try to instantiate takes(Student, Module). This it can do with the values johnDavies for Student and relationalDatabaseSystems for Module. It then tries to instantiate the variable Lecturer in teaches(Lecturer, Module). This it can do with the value hywelEvans. Since both conditions have been set to true, the rule head is true and the values johnDavies for S and hywelEvans for L can be displayed for the user. The engine then unravels the instantiations and starts the process again.

In a sense, querying on a database with rules produces virtual or derived data from the database.

9.5 DATALOG AND THE RELATIONAL DATA MODEL

A database which contains a collection of positive assertions is equivalent to a conventional database – a factbase. A database which contains both facts and rules is said to be a deductive database. It is deductive because we can use the rules to deduce virtual data – data not actually stored in the factbase.

Datalog without rules is essentially similar to that of the relational model (Chapter 7). Predicates in Datalog denote relations. However, in Datalog attributes are not named. Reference to a column is only through its position among the arguments of a given predicate.

Introducing rules into a database means that there are now two ways in which a relation can be defined. A predicate whose relation is declared explicitly by assertions is said to be part of the extensional database. A predicate defined by rules is said to be part of the intensional database.

In general the deductive data model is more powerful than the relational data model. The power of a deductive database over a conventional relational database can be demonstrated by considering the issue of recursive query processing.

9.5.1 RECURSION

Recursion occurs naturally in many domains having hierarchical structures – inventories, schedules, part hierarchies, routes.

However, conventional database sublanguages such as SQL (Part 3) are too weak to express recursion effectively. In such cases, data needs first to be retrieved from the database into an application program in which recursion is performed.

One advantage of applying formal logic to databases is that it allows recursive information to be explicitly specified in databases.

Example Suppose we wish to generate an organisational hierarchy based on a manages relation. The database is given below:

```
manages(jSmith, fJones).
manages(fJones, tDavies).
manages(jSmith, pEvans).
```

The first argument of the relation specifies the employee. The second argument specifies the employees' manager. To produce an organisational hierarchy we would include the following rules in our database:

```
manager(E, M) :– manages(E, M).
manager(E, M) :– manager(E, N), manages(N,M).
```

The first rule allows us to find someone's immediate superior. The second rule defines the fact that a manager of a manager is also a manager, and so on. Running a query such as Forall (manager(X,Y)) will recursively generate all the managerial relationships.

9.6 DIFFERENCES BETWEEN DATALOG AND PROLOG

The syntax of Datalog as described in previous sections looks very much like standard Prolog. So what are the differences between the programming language and the abstract or ideal database sublanguage? Some of the differences are presented below:

- *Set processing*. In Prolog results are returned one at a time. This is implicitly as we have described it in section 9.3.3. In Datalog results should be returned in sets. In other words, we assume the existence of built-in predicates such as forall
- *Order insensitivity*. Processing in Prolog is affected by the order of the rules or facts in a program and by the order of predicates in the body of a rule. A Prolog programmer frequently uses order sensitivity to improve the performance of a program. Datalog is a database sublanguage and as such should ideally be non-procedural. In other words, the execution of database queries should be insensitive to the order of retrieval of predicates
- *Special predicates*. Prolog programmers control the execution of programs through special predicates (e.g. the cut). This reduces the declarative nature of the language. Datalog does not include any comparable predicates
- *Function symbols*. Prolog has function symbols, which are typically used for building recursive functions and complex data structures. Base Datalog does not contain any function symbols, although a number of extensions to Datalog have been proposed, particularly for handling complex objects

Hence, syntactically Datalog is very similar to pure Prolog. Functionally they are different. Datalog is non-procedural, set-oriented, with no order sensitivity, no special predicates and no function symbols.

9.7 EXTENSIONS TO PURE DATALOG

The Datalog syntax described above corresponds to a restricted subset of first order logic and is often referred to as pure Datalog. Several extensions to pure Datalog have been proposed. Some of the most important extensions are built-in predicates, negation and complex objects.

9.7.1 BUILT-IN PREDICATES

Built-in predicates are expressed by special predicate symbols such as >, <, >= and <=, < >, each of which has a predefined meaning. These symbols can appear in the conditions of a rule and are usually written in infix notation.

Example For instance:

```
sibling(X,Y):– parent(Z,X), parent(Z,Y), X <> Y.
```

Here the symbol < > is taken to mean *not equal to*.

9.7.2 NEGATION

In pure Datalog the negation operator ¬ is not allowed. Normally the database is expected to adhere to the *closed world assumption*. This means that negative facts can be inferred from a set of positive Datalog clauses.

In general, a given fact about some universe of discourse (UoD) implies a large number of other facts. For example, knowing that John Smith teaches module relational database systems only (positive information) leads one to infer that John Smith does not teach modules relational database design, object-oriented databases etc. (negative information). This allows one to answer not only the query – Does John Smith teach module relational database systems? – but also queries of the form – Does John Smith teach module relational database design?

In the relational data model the usual semantic assumption made is that if a tuple does not exist in a relation then it exists in the complement of that relation. Thus the relation teaches(Lecturer, Module) represents all module offerings and NOT(teaches(Lecturer, Module)) lists all modules that are not taught by the corresponding lecturers. The complement of a relation is usually not made explicit for reasons of efficiency.

Two analogous assumptions are made in deductive database systems: *the closed world assumption* and *negation as failure*. The closed world assumption merely

asserts that the only things held to be true about a UoD are the positive assertions about this UoD – all else is held to be false. The negation as failure assumption asserts that if a statement Q cannot be proved to be true, not(Q) should be taken to be true. Hence, if one cannot prove that John Smith teaches module relational database systems then it should be concluded that John Smith does not teach relational database systems – failure to prove something equates to negation.

The closed world assumption however does not allow the use of negative facts to allow us to deduce further facts. In real life it is frequently important to express rules whose premises contain negative information.

Example For instance:

If X is a student and X is not a postgraduate student then X is an undergraduate student.

In pure Datalog such a rule would not be allowed. In extended Datalog such a rule might be written:

```
undergraduate(X):–student(X), ¬postgraduate(X).
```

9.7.3 COMPLEX OBJECTS AND FUNCTIONS

The 'objects' handled by pure Datalog programs correspond to the tuples of relations in that all attribute values are atomic. Datalog can be extended to handle complex objects, mainly by the inclusion of function symbols and set constructors.

Example We might construct a series of facts representing the complex object person as follows:

```
person(name(joe,bloggs),dob(5,march,1958),children({john,paul,george})).
person(name(joe,smith),dob(5,march,1958),children({john,susan})).
person(name(jill,jones),dob(5,april,1960),children({bill,sarah})).
```

Here name, dob and children are function symbols, the symbols {} are set constructors. In this scheme, variables may represent atomic objects or compound objects. This is similar to the idea of nested relations (Chapter 10). Hence the following rule produces the last names of all persons having the same first name and date of birth:

```
similar(X,Y):–person(name(Z,X),B,C),person(name(Z,Y),B,D).
```

By applying this rule we can infer the fact, similar(bloggs,smith).

9.8 INTELLIGENCE AND DATABASE SYSTEMS

The question of what is intelligence has of course been an important one for philosophers and psychologists for hundreds of years. No concensus definition

has yet been reached. Many people have portrayed the concept of intelligence as a continually moving target. This target has been moving faster of late with the rise of a particularly mechanistic discipline devoted to the study of intelligence: artificial intelligence (AI).

One major hypothesis lies at the heart of all work in AI: the physical symbol system (PSS) hypothesis (Newell and Simon). Put simply, this hypothesis states that a physical symbol system has the necessary and sufficient means for generating intelligent action. So what is a physical symbol system?

A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure). Thus a symbol structure is composed of a number of instances of symbols which are related in some physical way. At any instant of time the system will contain a collection of these symbol structures. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction. A physical symbol system is a machine that produces through time an evolving collection of symbol structures. Such a system exists in a world of objects wider than just these symbol structures themselves (Newell and Simon, 1976).

The important thing about this definition is that this hypothesis cannot be proven or disproved on logical grounds. We can only test the truth of the hypothesis by empirical validation; that is, by building physical symbol systems that generate intelligent action.

Of course we are still left with the question of what is intelligent action. It is extremely difficult to delineate the exact characteristics of intelligent action. The idea of the Turing test frequently discussed in this context only takes us so far. Many maintain that the idea of a computer system generating intelligence is nonsensical. For instance, Winograd and Flores (1986) maintain that the idea of intelligent action cannot be divorced from context and human interpretation. In this sense, they place key emphasis on the last sentence in the above quote: *that a PSS exists in a complex, ever-changing world wider than itself.*

9.8.1 DATABASE SYSTEMS AND PHYSICAL SYMBOL SYSTEMS

In Chapter 1 we discussed the essential features of a conventional database system. Such a system clearly meets many of the features of a physical symbol system. However, a conventional database system would not normally be described as being intelligent. So what distinguishes an intelligent from a non-intelligent database? This question is best answered in terms of the idea of a representation formalism.

Every database system must use some representation formalism. A representation formalism is 'a set of syntactic and semantic conventions that make it possible to describe things' (Winston, 1984). In database terms the idea of a representation formalism corresponds with the concept of a data model (Tsitchizris and Lochovsky, 1982). A data model is an architecture for data. A

data model is a formalism for implementing some or all of the abstract features of a database system described in Chapter 1.

Each database formalism has its strengths and weaknesses. Consider probably the most popular data model of the present day – the relational data model. The relational model offers an extremely flexible way of representing facts and querying such facts but does not offer a convenient way of representing the intricacies of integrity constraints and update functions. Integrity constraints and update functions are normally implemented in application programs which sit on top of a relational database.

In contrast, deductive databases, the topic of this chapter, offer a powerful notation for representing facts, integrity constraints, queries and update functions. However, they currently suffer from a lack of efficient implementations for large factbases.

Both relational and deductive databases are forms of PSS. A pure relational database would not normally be described as being intelligent. A deductive database would normally be seen as an example of an intelligent database. One part of the definition of an intelligent database is therefore involved with considering how certain formalisms offer a more powerful and uniform set of facilities for implementing the component parts of a database system.

9.9 INTELLIGENT DATABASES AND BUSINESS APPLICATIONS

In the sections above we have implicitly defined one of the important characteristics of an intelligent database as being its ability to store rules and use such rules to provide an active dimension. An intelligent database no longer sits there receiving data from its environment; it can react to its environment.

We would probably make the claim that an intelligent database system is an important tool for improving the productivity of informations systems development and maintenance (Beynon-Davies, 2002). Take, for instance, the example of integrity management. Some people have estimated that as much as 50–80% of a standard application is made up of integrity management. Traditionally, each program that accesses a database has to replicate this integrity management.

An intelligent database offers the possibility of storing integrity constraints centrally within the database itself. This is likely to mean:

- Replication of coding is reduced
- Information systems are likely to be smaller
- Information systems are likely to be more maintainable
- Better performance is likely to result in a cooperative processing environment

However, deductive databases are not the only approach to building intelligent database systems. It is useful to distinguish between two approaches to intelligent databases: evolutionary and revolutionary approaches.

Evolutionary approaches involve melding together existing technology or extending existing technology. For instance, one approach to building deductive databases is to meld a Prolog system to a relational DBMS.

Many of the in-built procedure and trigger mechanisms being offered by relational vendors can be cast in this light (Chapter 10), as can the current attempts to extend relational technology to handle complex objects. The current development of SQL3 (Chapter 31) is important in this context.

Revolutionary approaches propose new architectures. Datalog, for instance, is a proposal which unifies features of logic programming with databases. People are even attempting to extend Datalog to produce deductive object-oriented systems.

9.10 CASE STUDY: A DEDUCTIVE SCHEMA FOR A UNIVERSITY DATABASE

Below we present a more complete example of the deductive schema alluded to throughout this chapter for a university application. Comments are provided between the symbols /* and */.

Each entity in the schema is presented as a series of ground clauses – a proposition in which all the arguments are constants. The set of ground clauses with the same predicate constitutes a relation. Hence, there are three clauses in the *CourseEnrolment* relation below. The meaning of each attribute is defined by its position in the list of arguments for each predicate. Hence, the second argument in the *CourseEnrolment* relation always stands for a *studentNo*.

```
/*Courses – courseCode, courseName, deptName, courseLeader*/
courses('ISD', 'Information Systems Development', 'Computer Studies', '345').
/*Course enrolments – courseNo, studentNo, enrolmentDate, feePaid*/
courseEnrolment('ISD', '200001', '2002, 'Y').
courseEnrolment('ISD', '200002', '2002', 'Y').
courseEnrolment('ISD', '200003', '2000', 'Y').
/*Modules – moduleNo, moduleName, level, courseCode*/
modules('CS101', 'Relational Database Systems', 'l', 'ISD').
modules('CS102', 'Relational Database Design', 'l', 'ISD').
modules('CS301', 'Deductive Databases', '3', 'ISD').
/*Module enrolments – moduleNo, studentNo, year, semester enrolmentDate*/
moduleEnrolment('CS101', '200001', '2002', '1').
moduleEnrolment('CS101', '200002', '2002', '1').
moduleEnrolment('CS102', '200001', '2002', '2').
moduleEnrolment('CS102', '200002', '2002', '2').
/*Module assessments – moduleNo, studentNo, assessmentNo, grade*/
moduleAssessment('CS101', '200001', '1', '80%').
moduleAssessment('CS101', '200001', '2', '60%').
moduleAssessment('CS101', '200002', '1', '65%').
/*Lecturers – staffNo, staffName, status, salary*/
```

```
lecturers('234', 'Davies T', 'L', '20000').
lecturers('237', 'Patel S', 'SL', '27500').
lecturers('345', 'Evans R', 'PL', '35500').
/*Students – studentNo, studentName, gender, studentTermAddress, studentTermTelNo,
studentHomeAddress, studentHomeTelNo*/
students('200001', 'Morgan P', 'M', '11 Bronwyn St.', '562854', '12 Candice St., Liverpool',
'678923').
students('200002', 'Mirani F', '11 Mair Rd.', '564354', '12 Temple St., Reading', '228943').
students('200003', 'Smith T', 'M', '11 Bronwyn St.', '562854', '12 Canterbury Ave., Leeds',
'442223').
/*Teaches – lecturerNo, moduleNo*/
teaches('234', 'CS101').
teaches('234', 'CS102').
teaches('345', 'CS301').
/*Prerequisites – moduleNo, moduleNo*/
prerequisites('CS301', 'CS101').
prerequisites('CS301', 'CS102').
```

A number of basic update functions might also be offered. We provide a fuller version of one of the update functions defined in the text.

```
enrolStudentModule(Student, Module, Year, Semester):–
    students(Student,A,B,C,D,E,F),
    modules(Module,X,Y,Z),
    NOT(moduleEnrolment(Student,Module,Year,Semester)),
    assert(moduleEnrolment(Student,Module,Year,Semester)).
```

9.11 SUMMARY

- A database system is a physical symbol system. Different data models have different capacities for fulfilling intelligent action. By intelligence in this context most people refer to the active nature of such databases
- Logic as a data model consists of data structures, operators and integrity rules all represented in the same uniform way, namely as axioms in a logic language. In this chapter we have discussed Datalog as such a logic language and have illustrated the key features of this approach
- The Datalog discussed in this chapter bears a close resemblance to the logic programming language Prolog. The main difference is that we assume that Datalog demonstrates persistence
- The notion of intelligence is a slippery concept. Newell and Simon define intelligence in terms of a physical symbol system (PSS). A PSS has the necessary and sufficient means for generating intelligent action
- Both relational and deductive databases can be considered as physical symbol systems; but only deductive databases have been referred to as being 'intelligent'

- Because deductive databases contain an inferential component they are frequently discussed in the context of intelligent database systems. Intelligent databases have the possibility of significantly improving the productivity of information systems development and maintenance
- The idea of building intelligence into database systems will also prove important in contributing to the management of a number of other emergent properties of database systems: distribution, complex data and multiple media

9.12 ACTIVITIES

1. Write Datalog statements for the assertions: a share is an investment; a stock is an investment.
2. Write a Datalog statement to add deductive databases as a module to our database.
3. Given our academic database, how would the Datalog engine respond to the following query?

   ```
   takes(johnDavies, relationalDatabaseSystems)
   ```
4. Given our current academic database, what would happen to the following query?

   ```
   module(X)
   ```
5. Write a rule to implement the fact that a lecturer of a module is also the author of that module.
6. Given the database described in the chapter, run through the process of solving the query:

   ```
   teaches(L, relationalDatabaseSystems).
   ```
7. Implement the update functions described in section 1.4 for the university database in Datalog.
8. Given the rule described in section 9.5.1, list all the results of running the query:

   ```
   manager(X, Y).
   ```

9.13 REFERENCES

Beynon-Davies, P. (2002). *Information Systems: An Introduction to Informatics in Organisations*. Basingstoke, Palgrave (formerly Macmillan).

Gallaire, H. and G. Minker (1978). *Logic and Databases*, New York, Plenum Press.

Gardarin, G. and P. Valduriez (1989). *Relational Databases and Knowledge Bases*. Reading, MA, Addison-Wesley.

Newell, A. and H.A. Simon (1976). Computer science as empirical inquiry: symbols and search. *Comm. of ACM* **19**(3).

Reiter, R. (1984). Towards a logical reconstruction of relational database theory. In M.L. Brodie, J. Mylopoulos and J.W. Schmidt (Eds), *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*. New York, Springer-Verlag.

Tsitchizris, D.C. and F.H. Lochovsky (1982). *Data Models*. Englewood Cliffs, NJ, Prentice-Hall.

Winograd, T. and F. Flores (1986). *Understanding Computers and Cognition: A New Foundation for Design*. Norwood, NJ, Ablex Publishing.

Winston, P.H. (1984). *Artificial Intelligence*. Reading, MA, Addison-Wesley.

POST-RELATIONAL DATA MODEL

It's always useful to know where a friend-and-relation is, whether you want him or whether you don't.

Rabbit, *Pooh's Little Instruction Book*, inspired by A.A. Milne

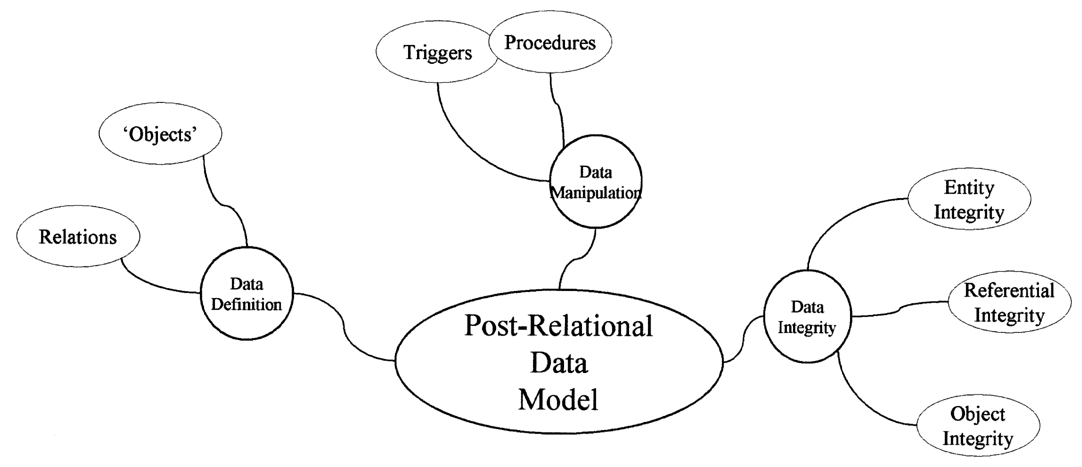
Hell, there are no rules here – we're trying to accomplish something.

Thomas A. Edison (1847–1931)

LEARNING OUTCOMES

At the end of this chapter the reader will be able to:

- Distinguish between the relational and the post-relational data model
- Describe some of the key features of the post-relational data model: triggers, procedures and abstract data types



10.1 INTRODUCTION

A number of extensions to the relational data model have been proposed in the three decades or so since its invention. Many of these extensions have been implemented in commercial DBMS. What is termed the post-relational data model here is not strictly a data model in that no coherent theory has been developed. Nevertheless it is useful to discuss it here in terms of a set of mechanisms found in many contemporary DBMS. Such a data model is also referred to by the terms extended-relational and object-relational data model. In Chapter 18 we discuss how the proposed SQL3 standard addresses many of these features. In Chapter 34 we also consider how the ORACLE DBMS supports some of these features.

In the first half of the chapter we consider two extensions to the data definition part of the relational data model: abstract data types and nested relations. In the second half of the chapter we consider two constructs – triggers and stored procedures – that have been used both for data manipulation and data integrity purposes. The incorporation of these features into a relational DBMS provides it with the ability to handle complex objects and behaviour. Hence many of the DBMS with these features have termed themselves object-relational systems.

10.2 THE RELEVANCE OF OBJECT-RELATIONAL SYSTEMS

Michael Stonebraker (1996) has proposed a simplistic but useful characterisation of the relevance of different types of data management approach (see Figure 10.1):

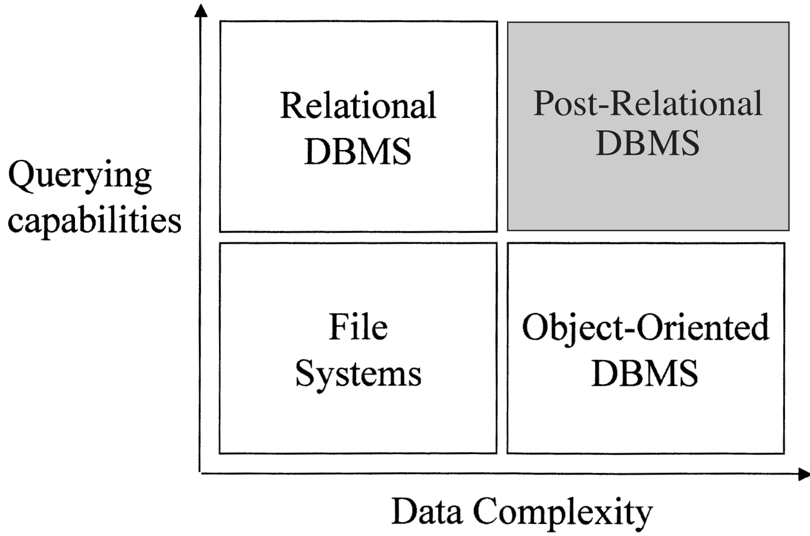


Figure 10.1 The place of the post-relational data model.

- In file systems the data is relatively simple in terms of structure and there is no real requirement for querying the data
- In relational systems the data is structurally simple but there is a heavy demand for querying
- In object-oriented database applications the data is usually inherently complex but there is usually no significant demand for querying the data
- In object-relational or post-relational systems the data may be complex but there is also an associated need for complex querying

This categorisation clearly portrays post-relational systems as an attempt to build on the strengths of both the relational and object-oriented (OO) approaches to managing data. It particularly seeks to extend the proven ability of relational DBMS to handle the data relevant to standard commercial information systems with the ability to manage complex data characteristic of OO DBMS.

10.3 DATA DEFINITION

In this section we consider how the concepts of an abstract data type and a nested relation can be added to the relational model to improve its object orientation.

10.3.1 ABSTRACT DATA TYPES

Most existing relational database systems provide only a limited number of data types for the user. For example, SQL-based systems (Part 3) provide integer, character, date and real data types, among others. These are usually sufficient for most standard data processing applications. However, for other applications, such as office automation, geographical information systems (Chapter 39) or computer aided design, these data types are not sufficiently rich. An office automation system, for instance, might want a document data type and a geographical information system might want a polygon data type. One solution to this problem is to extend the range of primitive data types offered to the user by the DBMS. However, the vast number of data types needed to support the infiltration of computing into more and more areas is probably too large to make this a practicable solution. A better approach is therefore to make the DBMS infinitely extendable by the user through user-defined and implemented abstract data types (ADT), sometimes called user-defined types (UDTs) (Figure 10.2).

An ADT or UDT is a type of object that defines a domain of values and a set of operations designed to work with these values. The objective of an ADT is to provide a characteristic known as information hiding. Implementation details of the ADT are hidden from higher levels of an application system. If the implementation of an ADT is changed, the upper layers of a system are

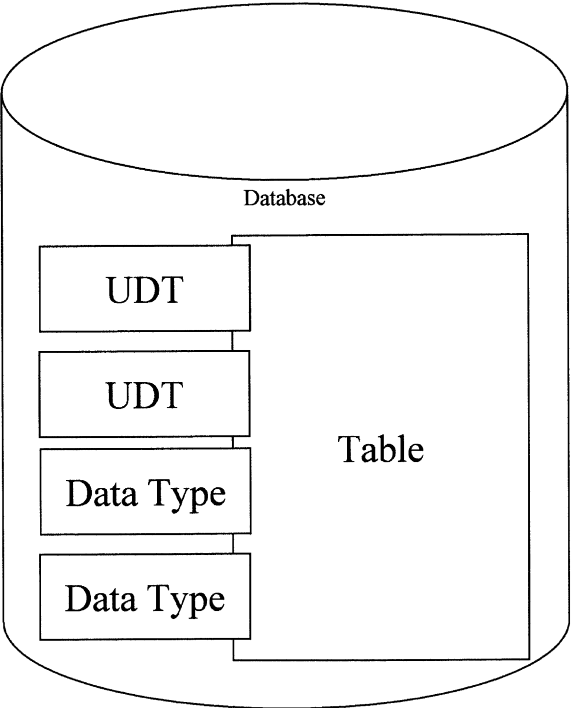


Figure 10.2 User-defined types.

unaffected since they communicate with the ADT only through a set of abstract operations composing its interface.

Example Suppose we have a clinical information system designed to store the written materials of a consultant psychiatrist. In such a system, a patient note might be a new ADT defined by the system designer. Associated with this new data type might be the operation count_note. This would count the number of words in a consultation note. Assuming that the number of words per consultation would give the psychiatrist some idea of the importance of the record, a query which extracted the health service numbers of patients seen after 30 June 2002 and ordered the results in descending order of the size of their consultation record might prove useful.

10.3.2 NESTED RELATIONS

Nested relations, sometimes known as NF 2 (non-first normal form) relations, have been proposed as a means of handling complex objects. A complex object can be defined as an object having a hierarchical structure whose attributes can be atomic or indeed complex objects themselves. In other words, an object can have any internal complexity we wish to define.

Example To continue with the psychiatry example, suppose that PatientHistory was a relation in our database. Such a nested relation might be represented as follows:

PatientHistory

patientNo	***patientName***	***DOB***	***consultation***			
			date	***time***	***treatment***	
					treatmentCode	***duration***
2312	Poulson J	7/4/70	1/2/2003	09:00	I/XS	30
			7/2/2003	09:30	A/XS	30
4377	Gearing P	4/3/73	1/2/2003	10:00	L/TP	70
			2/2/2003	10:00	L/TP	30
			3/2/2003	10:00	L/TP	47

Like normal relations, nested relations have a list of column names. However, a column name in a nested relation may refer to a group of attributes, each of which may also refer to a group, and so on. Hence, consultation in the table above refers to the group of attributes date, time and treatment. The attribute treatment in turn refers to the group treatmentCode and duration.

To operate on such relations we need two in-built operations: nest and unnest. Nest takes a relation and an attribute name (or composite attribute) and returns a corresponding nested relation.

Example Hence, if we supplied the relation Modules and the attribute courseCode we would obtain a nested relation with two attributes: courseCode and a composite group (moduleName, level, staffNo).

R1

courseCode			
	moduleName	***level***	***staffNo***
CSD	Relational Database Systems	1	234
	Relational Database Design	1	234
	Deductive Databases	3	347
	Object-Oriented Databases	3	347
	Distributed Database Systems	2	237
BSD	Intro to Business	1	123
	Basic Accountancy	1	147

Unnest is the opposite of nest. Applying unnest to the table above would produce the original relation.

10.4 PASSIVE AND ACTIVE DATABASE SYSTEMS

Many of the extensions to the relational data model have been proposed to enable more functionality to be placed within the database itself. This functionality was described in abstract terms in Chapter 1 as a set of update functions associated with the database. To refresh, update functions can be used to encapsulate both constraints and events that cause changes to the state of some database.

A database with embedded update functions may be referred to as an active database. It is active in the sense that it may perform operations traditionally performed in application systems external to the database. This idea of an active database may be contrasted with the traditional idea of a database as a passive repository for data. This distinction is illustrated in Figure 10.3.

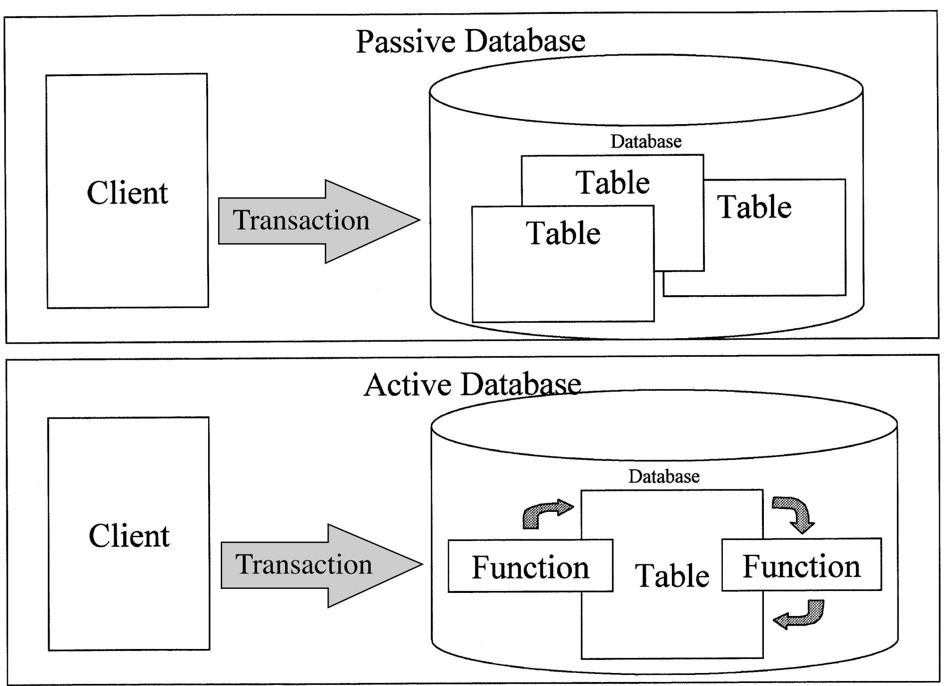


Figure 10.3 Passive and active databases.

Triggers and stored procedures are two mechanisms available for implementing integrity constraints and update functions and embedding these directly within the database.

10.4.1 TRIGGERS

Triggers are units of logic embedded in a database. Triggers are event-driven (see Figure 10.4). The logic is triggered by transactions which seek to update the database. A trigger is inherently associated with a single table within the database and is activated when insertions, updates or deletions are commanded against rows in that table. Such events are said to 'fire' a trigger.

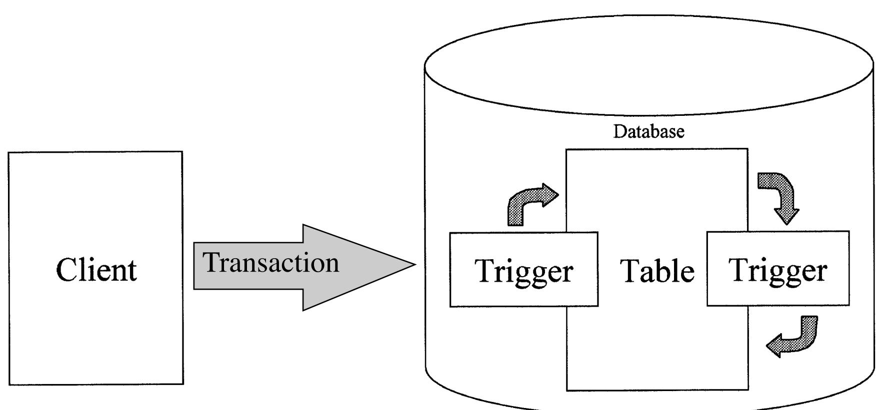


Figure 10.4 Triggers.

Triggers are used to perform a number of different database functions:

- Validating input data and maintaining complex integrity constraints: certain integrity constraints may not be declared using the inherent integrity mechanisms of a given data model, and triggers may be used to program such constraints
- Use as alerters – in other words, indicating actions that have been performed on the database to particular users
- Maintaining audit information for database administrators
- Supporting distribution actions such as the automatic replication of data across a network

Examples

- *Validating input data*. For instance, in terms of the academic database we might use a trigger to check that a lecturer teaches no more than eight modules in an academic session
- *Use as alerters*. Hence, a trigger might be used to flag to school administrators when a particular module is full

- *Maintaining audit information for database administrators*. Triggers might be used by a DBA to record information about updates made to a particular table in the database
- *Supporting distribution actions*. A trigger may be used to automatically update a desktop system with information off the central student information system at regular intervals

10.4.2 STORED PROCEDURES

Stored procedures are also units of logic embedded in the database. However, unlike triggers, stored procedures are relatively autonomous units of logic (see Figure 10.5). A stored procedure may be called from an application program, from another stored procedure or from a trigger.

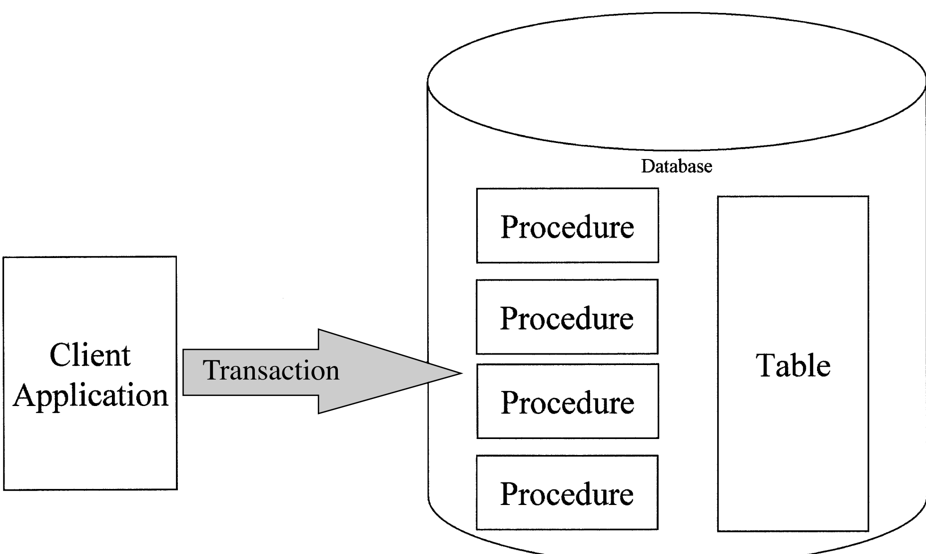


Figure 10.5 Stored procedures.

Example In terms of our academic database, stored procedures might then be used to: offer modules, cancel modules, enrol students on courses, enrol students in modules or transfer students between modules. In other words, all of the update functions defined for our academic database.

10.4.3 ADVANTAGES AND DISADVANTAGES OF TRIGGERS AND PROCEDURES

There is some debate about the appropriate use of triggers and stored procedures.

Advantages

Triggers and procedures have been developed to fulfil an active database need in contemporary applications. There are hence clear advantages to the use of these constructs such as:

- *Centralisation, standardisation and reuse of application logic.* The advantages of centralising replicated data (Chapter 3) also apply to replicated processes or code
- *Siting processing appropriately on the database server.* In client–server architectures (Chapter 36), siting applications on powerful servers can reduce the traffic along communication lines
- *Improving the productivity of application building.* The reuse of centralised database logic can enhance the productivity of the development of information systems

Disadvantages

Triggers and procedures are rather recent inventions in database history. The database development industry is still learning about the appropriate use of these constructs. Some possible disadvantages are associated with the modern use of triggers and procedures:

- There is some evidence that a system built substantially out of triggers and stored procedures may be difficult to manage. It may prove difficult to design, implement and perform administration tasks with active databases
- There are certain problems of transparency. Moving functionality to the database has the tendency of hiding functionality from the user. The database, not the user, controls much of the activity in an active database
- There is currently no standard syntax for writing stored procedures or triggers. Hence, there is the inherent concern of being tied-in to a particular DBMS vendor if an organisation chooses to write most of its application logic as triggers and procedures
- Using triggers and stored procedures normally has a performance implication. As the number of triggers increases then the processing overhead increases. For this reason, many systems have mechanisms for enabling and disabling triggers when performance proves an issue

10.5 TABLES AS OBJECTS

It is possible to treat post-relational extensions to the relational data model in a uniform way as having elements of an object-based approach. Note we call it an object-based, not an object-oriented approach. We can treat a table with extensions as an object but we usually cannot treat it in the same way as objects defined in Chapter 8.

A table with its associated attributes, constraints, triggers and procedures can be mapped on to the object-oriented concepts discussed in Chapter 8. Attributes of a table can map to the attributes of an object. Nested relations also allow all the attributes of an object to be kept together, not fragmented through normalisation. Stored procedures and triggers can, with a little imagination, be envisaged as the methods of the associated object. Tables can also be included in generalisation hierarchies with subtables inheriting the attributes, constraints, triggers and procedures of its supertable.

10.6 CASE STUDY: STORED PROCEDURES IN AN ACADEMIC DATABASE

Stored procedures can be used to implement some or all of the update functions against a database. In terms of a database system used at a university (see Case Study of Chapter 7), the following update functions might be appropriate. We have expressed these functions in a sort of pseudo-code of conditions and actions, similar in nature to the rules of the deductive data model (Chapter 9).

OfferModule(ModuleCode M, ModuleName N, . . .)
Conditions: NOT(Modules(M));
Actions: INSERT INTO Modules(M,N, . . .)
CancelModule(ModuleCode M)
Conditions: Modules(M);
Actions: DELETE Modules(M)
RegisterStudentDetails(StudentNo S, StudentName N, . . .)
Conditions: NOT(Students(S));
Actions: INSERT INTO Students(S,N, . . .)
EnrolStudentOnCourse(StudentNo S, CourseCode C)
Conditions: Students(S); Courses(C); NOT (CourseEnrolment(S,C));
Actions: INSERT INTO CourseEnrolment(S,C)
EnrolStudentOnModule(StudentNo S, ModuleNo M, CourseCode C)
Conditions: Students(S); Modules(M); Courses(C);
NOT (ModuleEnrolment(S,M)); CourseEnrolment(S,C);
Actions: INSERT INTO ModuleEnrolment(S,M)
TransferStudentBetweenCourses(StudentNo S, CourseCode C1, C2)
Conditions: Students(S); Courses(C1); Courses(C2);
CourseEnrolment(S,C1); NOT (CourseEnrolment(S,C2));
Actions: DELETE CourseEnrolment(S,C1); INSERT INTO CourseEnrolment(S,C2)

TransferStudentBetweenModules(StudentNo S, ModuleNo M1, M2)
Conditions: Students(S); Modules(M1); Modules(M2);
ModuleEnrolment(S,M1); NOT (ModuleEnrolment(S,M2));
Actions: DELETE ModuleEnrolment(S,M1); INSERT INTO ModuleEnrolment(S,M2)

10.7 SUMMARY

- The post-relational or extended relational data model incorporates the following extensions to the relational data model: user-defined types, nested relations, triggers and stored procedures
- An abstract data type or user-defined data type is a type of object that defines a domain of values and a set of operations designed to work with these values
- Nested relations have been proposed as a means of handling objects having a hierarchical structure whose attributes can be atomic or indeed complex objects themselves
- Triggers are event-driven units of logic embedded in a database inherently associated with a table
- Stored procedures are autonomous units of logic embedded in the database

10.8 ACTIVITIES

1. Define what is meant by the term post-relational database system.
2. What features of a database can be said to make it active?
3. Distinguish between a stored procedure and a trigger.
4. In terms of an application known to you, investigate how much of it could be implemented using triggers and procedures.

10.9 REFERENCE

Stonebraker, M. (1996). *Object-Relational DBMSs: The Next Great Wave*. San Francisco, CA, Morgan Kauffman.