Databases Essentials

Antonio Albano

University of Pisa
Department of Informatics
Largo B. Pontecorvo 3, 56127 Pisa – Italy
albano@di.unipi.it

September 23, 2011

CONTENTS

Chapter 1

INTRODUCTION

The importance of information in today's society is widely recognized. In this report, the attention will be focused particularly on the need for information management in *organizations*, considered as an organic collection of resources (people and materials), tools and procedures, which are finalized to create and offer a product or a service. For example, a manufacturer converts raw materials into a finished product, a bank provides financial services, and a hospital supplies medical services.

Nowadays, information is considered to be a critical resource of any organization, as fundamental as capital or machinery, and, in fact, the majority of the labor force in the industrialized countries works in some way with information.

Since organizations operate in a competitive environment, their information must not only be accurate but must also be provided rapidly, in time to support the decision processes. For this reason, all organizations have a structure which is dedicated to the management of information, the *information system*: an organized collection of resources, people, and procedures finalized to collect, store, process and communicate the information needed to support the ongoing activities.

Information can be represented as data, images, text, and voice. Clearly, different kinds of organizations will have differing needs with respect to the types of information they use. However, the attention here will be on information represented as *structured data*, shared by a variety of users within an organization, and managed using computers. Reductions in the costs of computer technology, improvements in performances, and new facilities to support the development of applications have created an increasing demand for data processing systems. We use the term *computerized information system* to refer to the hardware and software which is used for storing, retrieving, and processing the information which supports the functions of an organization[1].

When a computerized information system is implemented using database technology, it will consist of an *operational database* and a collection of *application programs* (*transactions*) which are used to access and update the data quickly and efficiently (Figure 1.1). The main goal of such a transaction processing system is to maintain the correspondence between the database and

1. Frequently in the literature, "information system" is used as synonym of "computerized information system". Here, we prefer to make a distinction between the two terms to evidence the fact that a "computerized information system" will never completly substitute the global "information system" of an organization.

the real-world situation it is modeling as events occur in the real world.

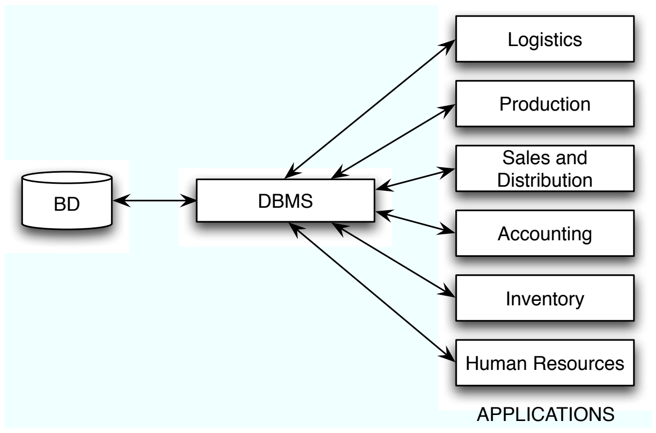


Figure 1.1 Transaction processing system

The data are under the control of a *Data Base Management System* (DBMS), a centralized or distributed software system, which provides the tools to define the database, to select the data structures needed to store and retrieve the data easily, and to access the data, interactively or by means of a programming language.

Another application domain in which databases play a key role is *Decision Support*. The main goal of such applications is to turn the data into *information* useful to support management decisions. Three categories of decision support are *reporting data*, *analyzing data*, and *knowledge discovery* with *data mining* techniques.

Decision support applications, sometimes called *online analytic processing* (OLAP), involve quite complex queries which cannot be efficiently executed against operational databases, optimized for *online transaction processing* (OLTP). For this reason, organizations maintain a separate database, called *data warehouse*, specifically organized for such complex OLAP queries.

This report presents and discusses the principal topics in the database area. The emphasis is on the concepts underlying database languages, systems and design. The discussion is organized into three main sections: database from the designer perspective, DBMS from a user perspective, and DBMS from a system perspective.

Section 4 presents introductory and fundamental concepts regarding information modeling. The problem of building a symbolic model of the knowledge on some aspect of the world is addressed and an object formalism is introduced to define this model. The formalism is used to explain the basic concepts used in the rest of the paper. The basic features of the *relational* model is also presented. The emphasis will be on the relational model since it has gained wide acceptance among database researchers and practitioners and has a solid theoretical basis. Moreover, an overview is given of the fundamental results of normalization theory to design relational databases.

Section 3, DBMS from a user perspective, presents the functionality of a DBMS: the separation of database description and application programs; database languages; data control; facilities for the database administrator. A large part of the presentation is devoted to the most important feature characterizing a DBMS: the data model it supports, i.e. the abstraction mechanisms used to model the databases. The basic features of the relational language SQL are also presented to define and use databases. The inclusion of SQL statements in a program written in a conventional programming language is also discussed.

Chapter 2

DATABASE: THE CONCEPTUAL DESIGNER PERSPECTIVE

The notion of model is fundamental to all methods of analysis and design. A model reproduces the essential characteristics of a real world situation, ignoring those details which would only represent an unnecessary complication with respect to the specific scope of the study being undertaken. A model is used for explicative or descriptive purposes (*descriptive models*), to predict actions and events (*predictive models*), or to provide recommended courses of actions (*normative models*). Models are distinguished by their structure: an *iconic model* retains some of the physical characteristics of the entities represented (e.g., a scale model car in a wind tunnel), whereas a *symbolic model* uses symbols to describe the real world. Symbolic models are used in the analysis and design of information systems and we define them as follows:

Definition 2.0.1 A *symbolic model* is a subjective formal representation of ideas and knowledge about a phenomenon.

Three fundamental aspects of this definition must be underlined: a) a model is a representation of only some aspects of a phenomenon; b) the representation is given by a formal language; c) the model is the result of an interpretation process which depends on the knowledge already possessed by the interpreter.

Computer science offers different formalisms which can be used to build symbolic models and, in particular, to represent information effectively, at varying degrees of detail. Different features vary in significance at different levels of abstraction. This is well illustrated, for instance, by a book of road maps. In planning a route, a traveller will look at the front of the book which may well list the major cities and the numbers of the page where more detailed information can be found. Looking at the relevant page, he finds the major roads with towns represented as shaded areas. At the back of the book, detailed street maps which pinpoint the destination more closely are often found. Clearly, in this case, there is a need for each type of representation, according to the particular problem to be solved.

We will thus first consider the kind of knowledge for which we wish to build a computerized information system, we will then show a graphic formalism which can be used to build models to analyze information systems, i.e. to build a static representation of the information content of a system, and finally we will present a formal language which can be used to implement the model. The emphasis will be on the construction of a *conceptual model*, i.e. on a model built using a formalism which is suitable for natural and direct modeling. The examples in the following sections mainly refer to a simplified library or a university administration system.

2.1 What to Model

When constructing a computerized information system, the reality to be modeled is generally considered with respect to: *concrete knowledge, abstract knowledge, procedural knowledge, dynamics*, and *communications.*

Concrete knowledge concerns specific facts known of the system to be represented. Adopting a simplified point of view, we will assume that the reality consists of *entities*, with certain *characteristics* or *properties*, and of *relationships between* the entities, which evolve over time.

Definition 2.1.1 An *entity* is anything for which certain facts should be recorded, independently of the existence of other entities.

In a library administration system, for instance, examples of entities can be a bibliographic description, a book, a loan record or a user.

Definition 2.1.2 A *property* is a fact about an entity which is not meaningful in itself, but only because it describes an entity of interest.

Examples of properties in a library are the user's name and address. The difference between a property and an entity results from a different interpretation of the represented fact in the model: properties are facts which are of interest only because they describe other facts which are considered as entities.

Entities with the same properties are said to have the same type and they are classified into the same *collection* (called also *entity set*). For instance, John, Mary, and Ann may be classified into the collection Persons based on the fact that they have the same properties and represent humans.

Collection of entities with the same type are certainly an important aspect of the knowledge about the reality to be modeled, but much more information is carried by facts which establish associations among entities.

Definition 2.1.3 A *relationship* is a fact which correlates independent entities. As with entities, a collection of similar relationships is called *relationship set.*

In the library, examples of relationships between entities can be the fact that a bibliographic description refers to a book, or more than one book if more than one copy of the same book exists, or the fact that the user Smith has borrowed a copy of a particular book.

A relationship is usually *binary*, that is involves two entities, but in general it may be n-ary. Moreover, several relationship sets might involve the same entity sets.

If we take a picture of a given time slice of the reality, the entities of interest, the values of their properties and the relationships in which they participate constitute a *state* of this reality. In general, the reality undergoes changes because entities are subjects of processes. These may be continuous processes or discrete event processes such as a change in the address of a user, the loan of a book, the acquisition of a new book, etc.

Abstract knowledge concerns general facts which impose restrictions on the admissible values of the concrete knowledge and on the way in which the values of the concrete knowledge can evolve in time, or expresses rules to derive new information (*integrity constraints*).

In the library, examples of abstract knowledge are (a) the user properties *Name*, and *Address*, which must have values of type *string*, whereas the *BirthYear* property will have values of type *integer*, (b) a book can be borrowed for two weeks, and two one-week extensions are allowed, if the extension is performed before the due data, (c) any person may have on loan at most five books at any time; the title of a book cannot be changed (d) the age of a person is computed as the difference between the current year and the year of birth.

Relationships usually have certain constraints that limit the possible correlated entities. The most important ones are the so called *structural constraints* or *properties*:

– *Cardinality*, *one* or *many*, to specify how many entities of one collection may be associated with entities of another collection.
– *Partecipation*, *total* or *partial*, to specify whether an entity of one collection can have entities of another collection associated to it.

For example, a book is borrowed by at most one person, but a person can borrow several books (the relationship is said *one-to-many* or 1:N). In contrast, the relationship *AppearsIn* between authors and bibliographic descriptions, in which an author has written several books and a book has been written by several authors, is said to be *many-to-many* or N:M. A book must be related to a bibliographic description (total), but a bibliographic description may not be related to a library book (partial).

Procedural knowledge concerns the elementary actions (or operations) in the application environment which are applied to concrete knowledge to cause changes. It must be understood that concrete knowledge is about the *structure* of the entities and procedural knowledge is about their *behavior*. Moreover, while abstract knowledge imposes restrictions on possible values of concrete knowledge, procedural knowledge imposes restrictions on the possible ways in which concrete knowledge can be used or modified.

Examples of elementary actions for a university student are: enroll, graduate, change address, and change telephone number.

Dynamics concerns how concrete and procedural knowledge can be used to model complex activities in the application environment.

Dynamics regards changes in the reality triggered by events and accomplished by standard procedures. An example of such a procedure in a university situation is: When a professor moves to another university, then stop salary; exclude the professor from mailing lists (usually more than one); for each course held by the professor, start procedure to assign new professor; for each commission of which the professor was a member, start procedure for new nominations; etc.

Finally, *communications* concern how information is entered in the information system and is exchanged among members of the organization.

For the sake of simplicity we will not consider in the following, procedural knowledge, dynamics and communications.

2.2 How to Model

To construct a conceptual model of an information system we define the *schema*, a collection of time-invariant definitions which model respectively (a) the structure of admissible data, as well as integrity constraints, (b) the procedural knowledge (*intensional aspects*). The creative part of conceptual modeling is deciding what collection of entities, relationships, and constraints to include in the schema to model the observed reality. So, this modeling activity requires a good deal of creativity, technical expertise, and understanding of the application domain. After the conceptual schema has been defined, there are straightforward ways of converting the design into an implementation, as it will be shown later.

Different formalisms, each supporting a specific data model, can be used to define the conceptual schema.

Definition 2.2.1 A *data model* is a set of abstraction mechanisms, with associated operators and implicit integrity constraints, used to define a database schema.

As a first example of a data model, let us examine the features of a so-called *object data model* (ODM), with abstraction mechanisms to model the user's conceptualization of the application domain, naturally and directly. This kind of data model was originally proposed as a formalism for the analysis and design of information systems, but nowadays such model is also supported by a new generation of DBMS, as it will be shown in Section 5.1.

2.3 ODM: An Object Data Model

The basic abstraction mechanisms of an object data model (ODM) are: *object, type, class, relationship, inheritance, type hierarchies*, and *class hierarchies.* For simplicity, in this section we will only describe how structural aspects of the reality can be modeled using a graphic formalism; in the Section 5.1, languages supporting the abstraction mechanisms of an object data model will be shown, and examples of how to model procedural knowledge will also be given.

2.3.1 Object

An object is the computer representation of certain facts about an entity of the observe world. An object is a software entity which has an internal state (*instance variables*) and it is equipped with a set of local operations (*methods*) to manipulate that state. The request to an object to execute an operation is called a *message*, to which the object can reply. The structure of an object state is modeled by a set of variables (or *attributes*) which can have values of arbitrary complexity, including other objects which become components of the object. When the state of an object can only be accessed and modified through operations associated to that object, we say that the object is a *data abstraction* or that it *encapsulate* its state.

Finally, each object is distinct from all other objects and has an identity that persists over time, independently of changes to the value of its state, e.g., if X and Y are identifiers bound to objects of type T, X will be equal to Y if they are bound to the *same* object. For instance, the object representing the person John is different from any other object representing another person, but will remain the same even if his address or some other attribute changes.

2.3.2 Type

An object is an instance of a type defined with a *generative type constuctor*, i.e. each object type definition produces a new type, which is different from any other previously defined types. An object type describes the state fields and the implementation of methods of its possible instances. An object type definition introduces a constructor of its instances, and so an object can be constructed only after its object type definition has been given.

In the object programming context this approach to objects is called *class-based* since the description of objects is called a *class*; we prefer the term "type" since we will use "class" with a different meaning according to the database tradition.

The signature $\Downarrow\mathcal{T}$of an object type $\mathcal{T}$ is the set of label-type pairs of the messages which can be sent to its instances.

Each object is a value of a certain type and objects of the same type have the same properties, i.e. they have the same structure and the same operations, specified by the type definition. The operations (the methods) to manipulate the state are specified by giving a specific implementation (*concrete behavior*)

The type mechanism makes it possible to create many objects of the same type using an appropriate constructor.

Example 2.1 Figure 2.1 shows a graphic representation of types.[1] Attributes are represented by the pair (Name : Type). Attributes can be *multivalued* (have a type seq T; they can be optional, meaning that the value can be left unspecified. Methods are represented by (Name (Parameters) := Body).

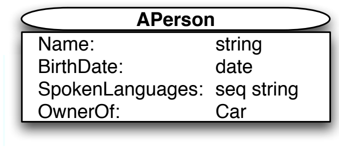


Figure 2.1 A graphic representation of object types

2.3.3 Class

An object data model supports a mechanism to define a collection of homogeneous values to model multivalued attributes or collections of objects to model databases. Usually two different mechanisms are provided:

1. To model multivalued attributes, type constructors are available for bags, lists (or sequences), and sets. For the sake of simplicity we will only consider sequences.

1. Currently there is no standard notation for an ODM model. Most books use the ER notation. We instead use a notation based on UML (Unified Modelling Language).

2. To model databases a mechanism called *class* is provided. A class is a modifiable sequence of objects with the same type. A class definition has two different effects:

   – It introduces the definition of the type $\mathcal{T}$ of its elements and a constructor for values of this type (*intensional aspect*).
   – It supplies a name to denote the modifiable sequence of the elements of type $\mathcal{T}$ currently in the database (*extensional aspect*).

We assume that when an object with the type of the elements of a class is constructed, then the object will itself become an element of that class.

2.3.4 Relationship

Classes of objects model sets of entities of the observed world, while relationships between such entities of are represented with a separate mechanism as shown with the following axamples.

Example 2.2 Figure 2.2 shows a graphic representation of classes with different level of details: (a) class name only, (b) class name and the attributes of its elements, and (c) class name, the attributes of its elements together with their values type.

A binary relationship between classes is represented by an oriented arc (Figure 2.3a). The arc is labeled with the relationships name. A binary relationship with attributes is represented by a relationship class attached to the arc using a dashed line (Figure 2.3b). The arcs may be labeled to clarify the role that entities play in the relationship: In this case the labels are used to name direct and inverse relationships; the labels are mandatory in the case of recursive relationships (Figure 2.3c).

The graphic notation represents also the structural properties of relationships: *cardinality* and *partecipation*, to model respectively how many elements of one class can be associated with elements of another class and whether an element of one class can have elements of another class associated to it. Multivalued relationships are represented graphically with a double arrow; optional relationships with a crossed line.

For example, a student might have passed zero or more exams, but an exam result must be associated to a student. Figure 2.4 shows a schema for a library information system.

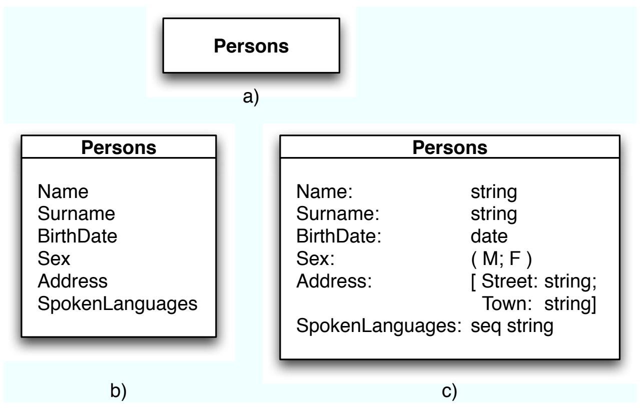


Figure 2.2 Graphic representation of classes

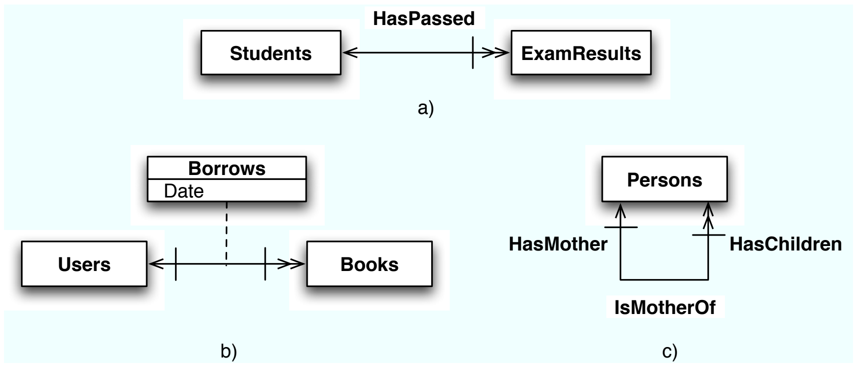


Figure 2.3 Graphic representation of relationships

2.3.5 Inheritance and Type Hierarchies

Inheritance is a mechanism which allows something to be defined, typically an object type, by only describing how it differs from a previously defined one. Inheritance should not be confused with subtyping: subtyping is a relation between types such that when $\mathcal{T} \leq \mathcal{S}$, then any operation which can be applied to any value of type $\mathcal{S}$ can also be applied to any value of type $\mathcal{T}$. The subtype relation (*IsA*) is asymmetric, reflexive and transitive.

The two notions are sometimes confused because, in object languages, inheritance is generally only used to define object subtypes, and object subtypes can only be defined by inheritance. However, we will keep the two terms distinct and will use each of them with its proper meaning.

Inheritance can be *strict*, when properties of the supertype can only be redefined in a controlled fashion, or *non-strict*, when they can be redefined freely. When inheritance is strict, we assume that properties can be redefined only by *specializing* their type and thus a value of the subtype T_1 can be used in all contexts in which an element of the supertype T_2 is expected (*context inheritance*).

In a subtype definition, a property of the supertype can be redefined (*overriding*), and its meaning in an object is then that given in the most specialized type to which the object belongs (*late binding*).

A subtype can be defined from a single supertype (*simple inheritance*) or from several supertypes (*multiple inheritance*).

2.3.6 Class Hierarchies

This is an asymmetric, reflexive and transitive relation in the set of classes, such that if $(C_1 \mathit{SubsetOf}\, C_2)$, then C_1 is said to be a *subclass* of C_2 and the following properties hold:

a) The type of the elements of C_1 is a subtype of the type of the elements of C_2 (*intensional constraint*).
b) The elements of C_1 are a subset of the elements of C_2 (*extensional constraint*).

Example 2.3 If we are interested both in Persons and Students, we have to model two different and essential facts: the *type* of Students elements is a *subtype* of the

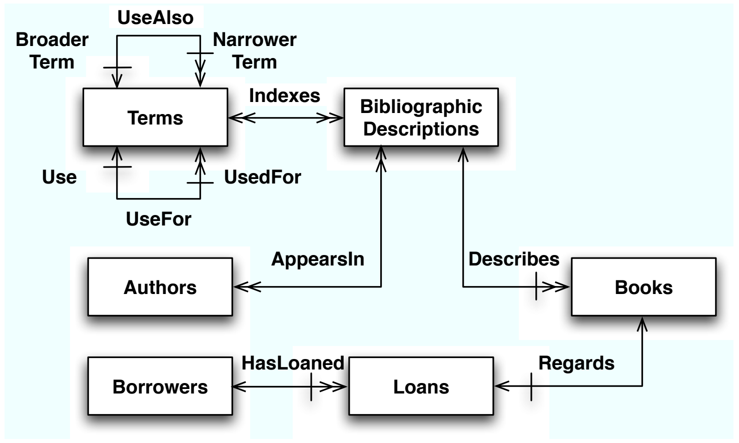


Figure 2.4 A schema for a library

type Persons elements, because all the *possible* students are a subset of all the *possible* persons; the set of all *actual* Students, is a subset of all *actual* Persons (i.e. the *class* Studentsis a *subclass* of the *class* Persons) (Figure 2.5).

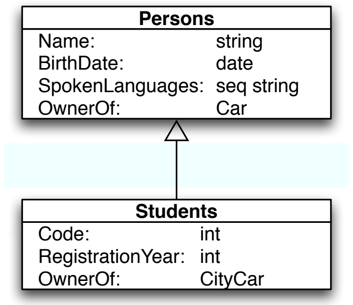


Figure 2.5 Subclasses

A subclass can be defined from a single superclass (*simple inheritance*) or from several superclasses (*multiple inheritance*). Moreover on subclasses of the same superclass can be defined two kinds of constraints: *overlap* and *covering.*

A no overlap constraints specify whether two subclasses are not allowed to contain the same element. We denote this by drawing a small black circle (Figure 2.7b). In the absence of this constraint, we assume by default that the subclasses are allowed to contain same elements (Figure 2.7a).

A covering constraints specify whether the objects in the subclasses collectively include all the elements in the superclass. We denote this by drawing the hierarchy with a double line (Figure 2.7c). In the absence of this constraint, we assume by default that there is no covering constraint. When the union of the sets of the elements of the subclasses are disjoint and equal to the set of the elements of the superclass, we call the hierarchy a *generalization* (Figure 2.7d).

A refined library schema using subclasses is shown in Figure 2.8 with the element attributes.

Usually there are two ways to populate subclasses:

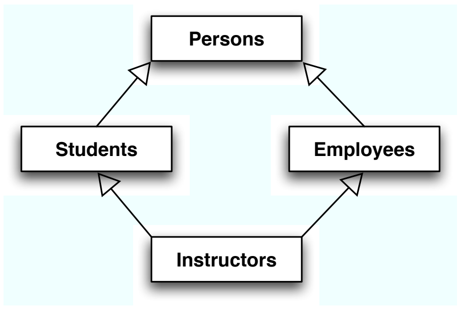


Figure 2.6 Subclasses with multiple inheritance

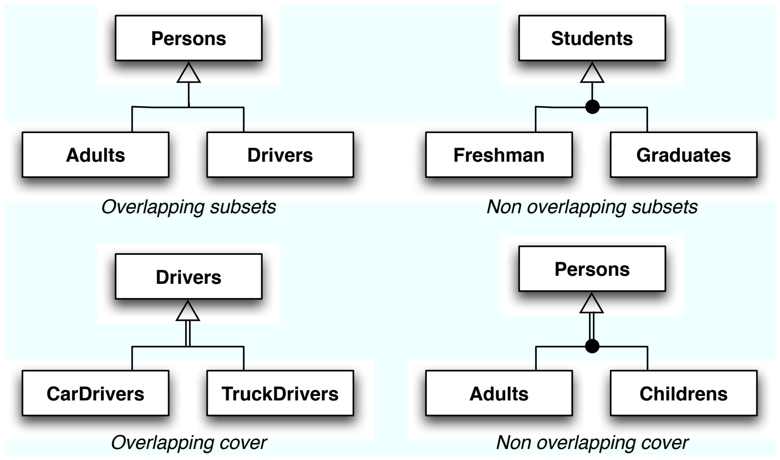


Figure 2.7 Kinds of subclasses

- A subclass can be populated simply by creating elements with an appropriate constructor, and these elements will also appear as elements of its superclasses, because of the extensional constraint of the subclass relation.
- A subclass can be populated also by moving objects from a superclass into the subclass. Thus, objects can change the most specific class to which they belong during their life-time. For example, a person can belong to the subclass of students, then employees, and finally be just a person again.

Because of the semantics of the extensional constraint of the subclass relation, when an object is removed from a class, it is also removed from its subclasses; but when it is removed from a subclass, it will remain in the superclasses.

Subtype, inheritance, and subset are three different kinds of relations between types and values of an object language. *Subtype* is a relation between types which implies value substitutability; *inheritance* is a relation between definitions, which means that the inheriting definition is specified "by difference" with respect to the super-definition; *subset* is a subset relation between collections of objects, which also implies a subtype relation between the types of their elements. Languages exist that support only subtypes, or subtypes and inheritance, or subtypes, inheritance and subsets.

Several alternative graphical notations have proposed for modeling databases. The most popular is the entity-relationship (ER) diagram, introduced by Chen

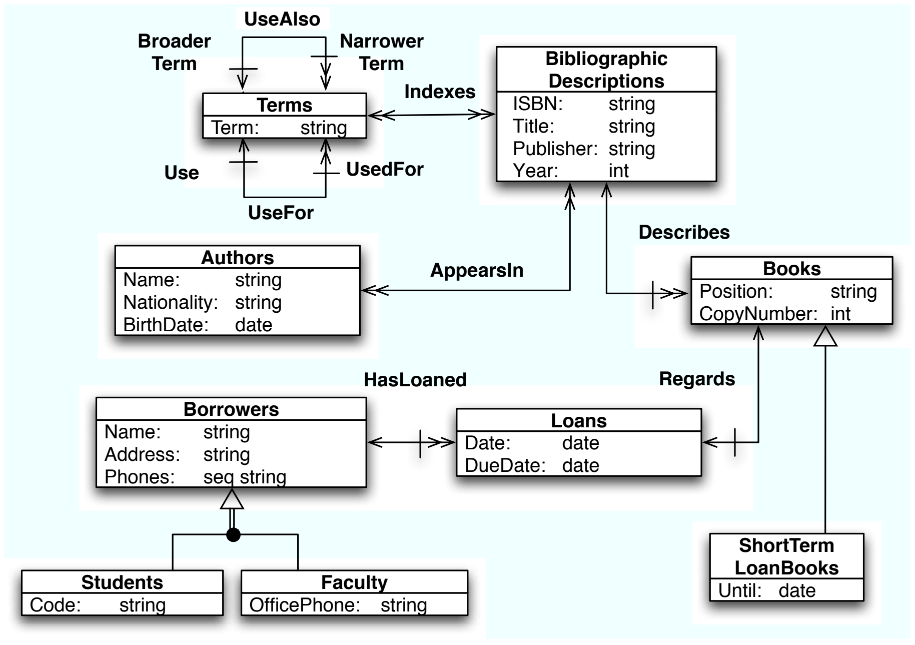


Figure 2.8 A refined schema for a library with element attributes

in the 1976 and later extended with hierarchies. More recently the Unified Modeling Language (UML) is becoming the standard notation for object modeling. Several tools also exists to specify diagrams, examples are: *ERwin* from Computer Associates, *ER/Studio* from Embarcadero Technologies, and *Relational Rose* from Rational Software for UML. In addition, DBMS vendors provide their own design tools, such as *Oracle Designer* and *Power Designer* from Sybase. Just to give an idea of the alternative graphical notations, the library schema in Figure 2.4 is shown in Figure 2.9 using the ER notation.

2.4 Exercises

1. A university database contains information about professors (identified by social security number, or SSN) and courses (identified by courseid). Professors teach courses; each of the situations described below concerns the Teaches relationship set. For each of the following situations, draw a diagrams that capture this (assuming that no further constraints hold):
   (a) Professors can teach the same course in several semesters, and each offering must be recorded.
   (b) Professors can teach the same course in several semesters, and only the most recent such offering needs to be recorded. (Assume that this is the case in all subsequent questions.)
   (c) Every professor must teach some course.
   (d) Every professor teaches exactly one course (no more, no less).
   (e) Every professor teaches exactly one course (no more, no less), and every course must be taught by some professor.
   (f) Now suppose that certain courses can be taught by a team of professors jointly, but it is possible that no one professor in a team can teach the course. Model this, introducing additional entity sets and relationship sets if necessary.

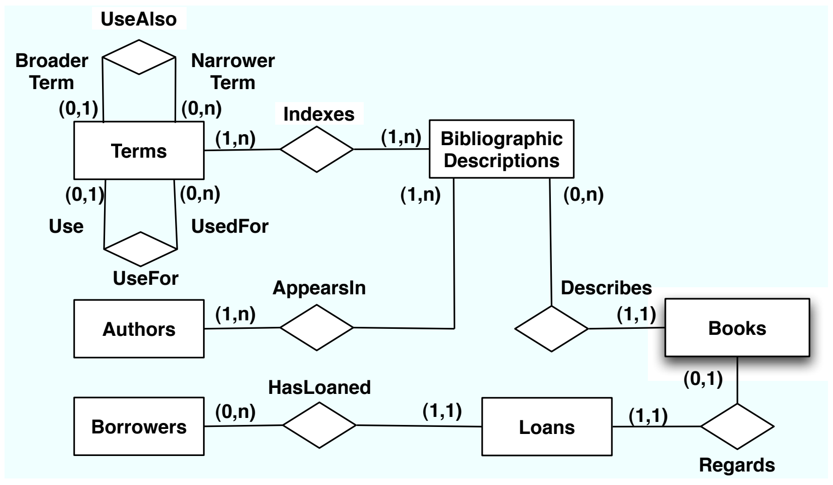


Figure 2.9 The library schema in ER

2. Let us assume that a company uses the following worksheet to store data about its computers.

InvNo	*Model*	*Description*	*SerialNo*	*Cost*	*UCode*	*UName*	*UPhone*
...	...	...	...	...	...	...	...
111	SUN3	WS Sun	ajk078	25000	1	John	576
112	PBG4	Notebook Mac	a908m	6000	2	Ron	587
113	SUN3	WS Sun	ajp890	27000	2	Ron	587
114	ThinkPad	IBM PC	ajp890	7000	3	Bob	588
...	...	...	...	...	...	...	...

The inventory number identifies a computer. A computer has a cost, a model, a description and a serial number. Computers with the same model can have a different cost, but have the same description. The serial number is different for the computers with the same model. Each computer has a user, who can have several computers, but only one phone number. A user has a code and a name. Design a conceptual schema for the database.

3. A bank database keeps track of loans, clients and montly payments to produce reports of the kind shown in the Figure 2.10. A client may apply for several loans and can have more than one approved loan. Design a conceptual schema for the database.

LOAN REPORT

LOAN NUMBER: 250	**DATA**: 7/2/02
LOAN DATA OF EXPIRY: 1/1/10	
LOAN AMOUNT: 70.000,00	
ANNUAL INTEREST RATE: 5%	**CLIENT CODE**: 2000
No. OF PAYMENT MADE: 4	**CLIENT NAME**: Mario Rossi
CURRENT BALANCE: 14.000	**CLIENT ADDRESS**: Via Roma, 13 -Pisa

PAYMENT No	PAYMENT DATA OF EXPIRY	AMOUNT	PAYMENT DATA
1	1/7/00	3.500	29/06/00
2	1/1/01	3.500	30/12/00
3	1/7/01	3.500	30/06/01
4	1/1/02	3.500	30/12/01

Figure 2.10 Loan report

4. Design a conceptual schema for a database to keep track of actors and directors of films. Each actor o director has a unique name, a birth year, and a nationality. An

actor may be also a director. Each film has a title, the production year, the actors, a director, and a producer. Films produced the same year have different titles.

5. We would like to design a database to maintain the following facts. Trains are either local trains or express trains, but never both. A train has a unique number and an engineer. Stations are either express stops or local stops, but never both. A station has a name (assumed unique) and an address. All local trains stop at all stations. Express trains stop only at express stations. For each train and each station the train stops at, there is a time. Design a conceptual schema for the database.
6. Consider the following information about a manufacturing company's parts and suppliers database. The database contains information about the way certain parts are manufactured out of other parts: the subparts that are involved in the manufacture of a part, the number of subparts used, the cost of manufacturing a part from its subparts, the mass of the part as result of the subparts assemblage. The manufactured parts may themselves be subparts in a further manufacturing process. In addition, certain information must be held on the parts themselves: their code, name and, if they are imported (i.e., manufactured externally), the supplier and the purchase cost. Suppliers have a code, a name, several phones and an address. Design a conceptual schema for the database.
7. Design a conceptual schema for a Company database to keep track of a company's employees, departments, and projects. The company is organized into departments. Each department has a unique name, a unique number, a location, and a manager who is one of its employees. We keep track of the start date when the employee began managing the department. A department controls a number of projects, each of which has a unique name, a unique number. An employee has a name, a social security number, address, salary, sex (m or f), and birthdate. An employee is assigned to one department but may work on several projects, which are not necessarily controlled by the same department. We keep track of the percent-time that an employee works on each project. We also keep track of the direct supervisor of each employee, who belong to the same department, and the start date when the employee began acting as supervisor. We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's name, sex, birthdate, and relationship (spouse or child or other) to the employee (assume that only one parent works for the company). We are not interested in information about dependents once the parent leaves the company.

Chapter 3

DATABASE: THE RELATIONAL DESIGNER PERSPECTIVE

3.1 The Relational Data Model

The relational data model, defined by Codd in 1970, has been supported by DBMSs from the mid-1970s on. Such systems soon became popular, mainly because of the simplicity of the data model and the facilities they provide to allow easy access to the data for non-expert users. Several implementations exist and are available on many types of personal computers, and workstations (e.g., ORACLE, DB2, SQL Server, Sybase).

The relational data model supports a very simple, tabular view of the data, with a direct correspondence to the mathematical concept of a relation. Following the proposal of the relational data model, an important theory has been developed to assist in the design of relational databases; this theory will be presented in the next section.

The relational data model describe databases in terms of sets of *tuples* (records) and associations among data in terms of values of attributes, and not using a specific abstraction mechanism. This way of describing associations looks similar to the solution adopted in object data models, but there are important differences in the modeling capabilities of these two data models:

- In the object data model the structure of the objects can be complex, whereas in the relational data model the structure of a tuple is simple, i.e. the values of the components of a tuple are elementary.
- In the object data model the associations model set of object tuples, whereas in the relational data model associations are described by attributes which can only have the value of the key of the associated elements of some other relation as their values.
- In the object data model the structure of an object is defined together with the representation of the procedural knowledge, whereas in the relational data model only a mechanism to describe the structure of the tuples is provided.

A number of studies which aim at overcoming some of the limitations of the relational data model are now in course, and references to them are given in the bibliographic notes.

Definition 3.1.1 A relational database is described by a set of relation schemas $R : \{T\}$ defined as follows:

- *integers, floats, booleans*, and *strings* are *primitive* types.
- If $T_1, \ldots, T_n$ are primitive types, and $A_1, \ldots, A_n$ are distinct attribute names, then $(A_1 : T_1, \ldots, A_n : T_n)$ is a *tuple type of degree* n. Two tuple types are equal if they have the same set of pairs $(A_i : T_i)$.
- If T is a tuple type, then $\{T\}$ is a *relation type*;
- A relation schema $R : \{T\}$ is a variable R with a relation type.

Definition 3.1.2 A tuple $(A_1 := V_1, \ldots, A_n := V_n)$ of type $T = (A_1 : T_1, \ldots, A_n : T_n)$ is a set of pairs (A_i, V_i) with V_i of type T_i. Two tuples with the same type are equal if they have the same set of pairs (A_i, V_i). The extension of an relation schema $R : \{T\}$ is a finite set of tuples of type $\{T\}$, called a *relation*. The *cardinality* of a relation is the number of its tuples. The extension of a relational database is a collection of the extensions of its relation schemas.

Definition 3.1.3 A *key* for a relation is a *minimal* subset of attributes whose values identify a tuple. Out of all the possible keys the database designer identify a *primary* key.

An example of a relational database schema is:

Students:{*Name*: string, *StudentCode*: string, *City*: string, *BirthYear*: int}
ExamResults:{*Subject*: string, *Candidate*: string, *Date*: string, *Grade*: int}

where the attribute StudentCode is the *primary key* for the relation Students, and the attribute Candidate in ExamResults, whose values match those of the primary key of the Students relation, is called an *external key*. An external key is used to model associations.

For simplicity in the following, instead of the notation $R : \{A_1 : T_1, \ldots, A_n : T_n\}$, we will use as standard notation $R(A_1 : T_1, \ldots, A_n : T_n)$ to denote a relation with name R and type $\{A_1 : T_1, \ldots, A_n : T_n\}$, which will be further abbreviated to $R(A_1, \ldots, A_n)$ when the type of the attributes is not important.

Example 3.1 A relational database schema for the example in Figure 3.1 is the following (primary keys are underlined):

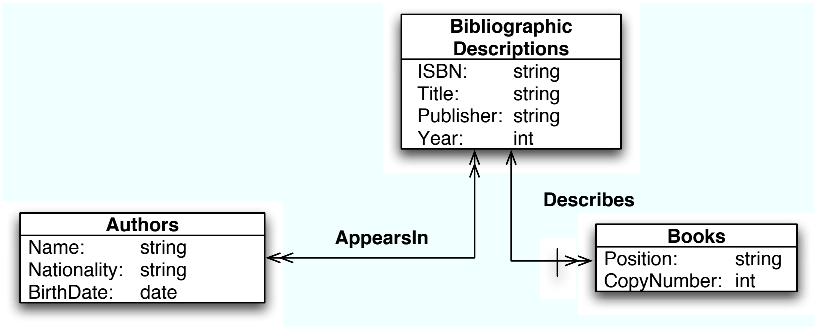


Figure 3.1 A database schema using the ODM

Authors(<u>Name</u>, BirthDate, Nationality)
BibliographicDescriptions(<u>ISBN</u>, Title, Publisher, Year)
Books(ISBN, <u>Position</u>, CopyNumber)
Authors-BibliographicDescriptions(<u>Name</u>, <u>ISBN</u>)

Figure 3.2 shows a graphical notation for representing relational schemas: A rectangle represent a relation schema and directed arrows from R to S represent an association between them with a foreign key defined in R for S.

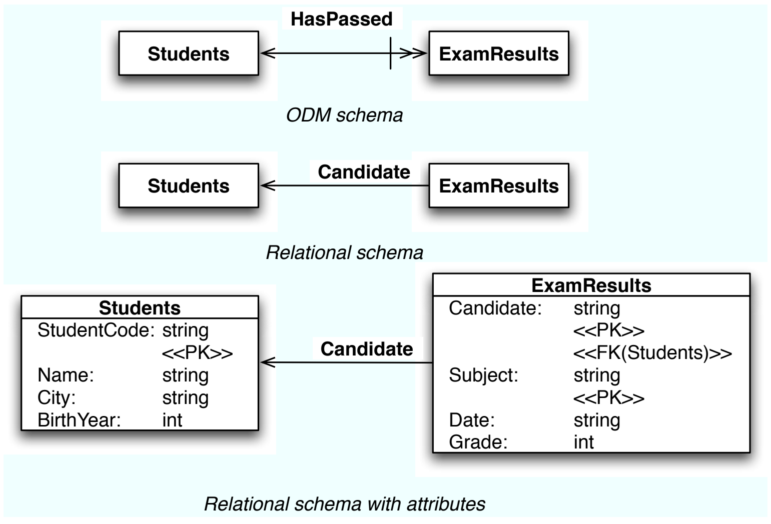


Figure 3.2 A graphical notation for a relational schema

3.2 Relational Algebra

The relational data model supports operations on relations whose results are themselves relations. These operations can be combined using an algebraic notation called *relational algebra*. Let R, S, and E be a relational expression defined using relations in the database or constant relations.[1] There are six fundamental operations in relational algebra: *rename*, *project*, *select*, *set union*, *set difference*, and *product*; we shall also mention some additional operations which serve as useful shorthand.

3.2.1 Fundamental Operations

Rename: $\rho_{A_1 \to B_1, A_2 \to B_2, \ldots, A_m \to B_m}(R)$

$A_1, A_2, \ldots, A_m$ are attributes of R, and $B_1, B_2, \ldots, B_m$ are not attributes of R. The result is the relation R with the attributes A_i renamed to B_i.

Project: $\pi_{A_1, A_2, \ldots, A_m}(R)$

$A_1, A_2, \ldots, A_m$ are attributes of R. The result is a relation with type $\{A_1 : T_1, A_2 : T_2, \ldots, A_m : T_m\}$ whose tuples are those of R with only the attributes $A_1, A_2, \ldots, A_m$. Since the result is a set, any duplicate tuples are eliminated.

Select: $\sigma_{Condition}(R)$

1. A constant relation is written by listing its tuples within { }, for example $\{(A_1 := 2, A_2 := 125); (A_1 := 3, A_2 := 250)\}$.

The result is a relation with the same type as R, whose tuples are those of R which satisfy the condition.

Set union: $R \cup S$

R and S are relations of the same type, and the result is a relation with tuples which are in R or S or both.

Set difference: $R - S$

R and S are relations of the same type, and the result is a relation with tuples which are in R but not in S.

Product: $R \times S$

$R\{A_1 : T_1, \dots, A_n : T_n\}$ and $S\{A_{n+1} : T_{n+1}, \dots, A_{n+m} : T_{n+m}\}$ are relations with disjoint set of attributes. The result is a relation of type $\{A_1 : T_1, \dots, A_n : T_n, A_{n+1} : T_{n+1}, \dots, A_{n+m} : T_{n+m}\}$ whose tuples are all possible tuples whose first n components form a tuple in R and whose last m components form a tuple in S.

Let us show how these operators can be used to write queries using the following database:

Students

Name	StudentCode	City	BirthYear
Isaia	071523	Pisa	1962
Rossi	067459	Lucca	1960
Bianchi	079856	Livorno	1961
Bonini	075649	Pisa	1962

ExamResults

Subject	Candidate	Date	Grade
DA	071523	12/01/85	28
DA	067459	15/09/84	30
MTI	079856	25/10/84	30
DA	075649	27/06/84	25
LFC	071523	10/10/83	18

Example 3.2 First, we find the name, and the student code of all the students of Pisa.

$\pi_{Name,StudentCode}(\sigma_{\text{City} = \text{'Pisa'}}(Students))$

Name	StudentCode
Isaia	071523
Bonini	075649

Next, suppose we want to find the names of all those students, who have passed the exam "DA" with grade 30, plus the examination date. Let us compute the result in more than one step, using the following strategy: since we need information from both the Students relation and the ExamResults relations, let us first compute the product of the two relations, producing the following temporary relation T:

$T :=$ Students $\times$ ExamResults

which can be very large: if there are n tuples in Students and m tuples in ExamResults, then there are $n \times m$ tuples in T.

Name	StudentCode	City	BirthYear	Subject	Candidate	Date	Grade
Isaia	071523	Pisa	1962	DA	071523	12/01/85	28
Isaia	071523	Pisa	1962	DA	067459	15/09/84	30
Isaia	071523	Pisa	1962	MTI	079856	25/10/84	30
Isaia	071523	Pisa	1962	DA	075649	27/06/84	25
Isaia	071523	Pisa	1962	LFC	071523	10/10/83	18
Rossi	067459	Lucca	1960	DA	071523	12/01/85	28
Rossi	067459	Lucca	1960	DA	067459	15/09/84	30
Rossi	067459	Lucca	1960	MTI	079856	25/10/84	30
Rossi	067459	Lucca	1960	DA	075649	27/06/84	25
Rossi	067459	Lucca	1960	LFC	071523	10/10/83	18
Bianchi	079856	Livorno	1961	DA	071523	12/01/85	28
Bianchi	079856	Livorno	1961	DA	067459	15/09/84	30
Bianchi	079856	Livorno	1961	MTI	079856	25/10/84	30
Bianchi	079856	Livorno	1961	DA	075649	27/06/84	25
Bianchi	079856	Livorno	1961	LFC	071523	10/10/83	18
Bonini	075649	Pisa	1962	DA	071523	12/01/85	28
Bonini	075649	Pisa	1962	DA	067459	15/09/84	30
Bonini	075649	Pisa	1962	MTI	079856	25/10/84	30
Bonini	075649	Pisa	1962	DA	075649	27/06/84	25
Bonini	075649	Pisa	1962	LFC	071523	10/10/83	18

However the only meaningful tuples in T are those with equal values for the attributes StudentCode and Candidate.

$R := \sigma_{\text{StudentCode} = \text{Candidate}}(T)$

Name	StudentCode	City	BirthYear	Subject	Candidate	Date	Grade
Isaia	071523	Pisa	1962	DA	071523	12/01/85	28
Isaia	071523	Pisa	1962	LFC	071523	10/10/83	18
Rossi	067459	Lucca	1960	DA	067459	15/09/84	30
Bianchi	079856	Livorno	1961	MTI	079856	25/10/84	30
Bonini	075649	Pisa	1962	DA	075649	27/06/84	25

The final answer to our query is the result of the expression:

$\pi_{Name,Date}(\sigma_{\text{Subject} = \text{'DA'} \wedge \text{Grade} = 30}(R))$

The same result might have been obtained with the expression

$\pi_{Name,Date}(\sigma_{\text{Subject} = \text{'DA'} \wedge \text{Grade} = 30 \wedge \text{StudentCode} = \text{Candidate}}(Students \times ExamResults))$

As matter of fact, the result of the above expression can be computed in a more efficient way than that shown above. This is a property of a relational manipulation language: a complex expression is a way of specifying the result declaratively, without forcing the system to follow certain steps, as happens in the first case shown above. The system chooses the best strategy by estimating the cost of obtaining the query answer according to different alternatives. We will discuss this aspect in more detail later on.

3.2.2 Additional Operations

Examples of additional and very useful operators that can be expressed in terms of the five basic operators above are *intersect*, *join*, and *natural join.*

Set intersection: $R \cap S$

R and S are relations of the same type, and the result is a relation with tuples which are both in R and in S.

Join: $R \underset{R.A_i=S.A_j}{\bowtie} S$

$R\{A_1 : T_1, \ldots, A_n : T_n\}$ and $S\{A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m}\}$ are relations with a disjoint set of attributes, A_i an attribute of R and A_j an attribute of S. The join $R \underset{R.A_i=S.A_j}{\bowtie} S$ is equivalent to $\sigma_{A_i=A_j}(R \times S)$.

Natural Join: $(R \bowtie S)$

The natural join is only applicable when both R and S have attributes with the same name. Let us assume that R and S have the common attribute A_i. The result is computed by selecting those tuples of $R \times S$ that have the same value for the common attribute A_i, and excluding one of the common attributes from the result.

The following operators are examples of useful extended relational algebra operations.

Generalized projection: $\pi_{e_1 \text{ AS } \mathsf{ide}_1, e_2 \text{ AS } \mathsf{ide}_2, \ldots, e_n \text{ AS } \mathsf{ide}_n}(E)$

where E is any relational algebra expression, each of $e_1, \ldots, e_n$ is an arithmetic expression involving constants and attributes in the schema of E, and $\mathsf{ide}_1, \ldots, \mathsf{ide}_n$ is a set of different attributes.

For example, if $A_1, A_2, \ldots, A_m$ are integer attributes in R, we can write the following expression:

$$\pi_{A_1,\, 2 \text{ AS } \mathsf{Two},\, A_1+A_3 \text{ AS } \mathsf{A1PlusA3}}(R)$$

The expressions in the generalized projection can be defined using *aggregate functions*, which take a collection of values and return a single value as result. Common aggregate functions include *min*, *max*, *count*, *sum*, and *avg*. If a generalized projection contains an aggregate function, then all the expressions must be aggregate functions, and the result is a relation with a single tuple with attributes values the results of the aggregation functions.

Bag-project: $\pi^b_{A_1,A_2,\ldots,A_m}(R)$

the result is the projection of R onto the attributes $A_1, A_2, \ldots, A_m$, without duplicates elimination, as it happens with the project operator.

Grouping operator: ${}_{A_1,\ldots,A_n}\gamma_{f_1(B_1),\ldots,f_m(B_m)}(E)$,

where E is any relational algebra expression; $A_1, \ldots, A_n$ is a list of attributes of E on which to group; each f_i is an aggregate function applied to attributes of E. The meaning of the operation is as follows (Figure 3.3):

1. The tuples of E are partitioned in groups in such a way that all the tuples in a group have the same values for $A_1, \ldots, A_n$.
2. For each group with attributes values $a_1, \ldots, a_n$, the result has a tuple

$$(a_1, \ldots, a_n, v_1, \ldots, v_m)$$

where for each i, v_i is the result of applying the aggregation function f_i on the multiset of B_i values in the group.

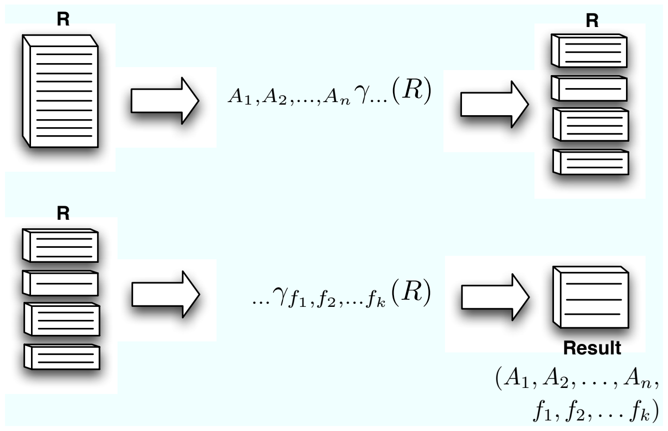

R
$_{A_1,A_2,\ldots,A_n}\gamma_{\ldots}(R)$
R
R
$_{\ldots}\gamma_{f_1,f_2,\ldots f_k}(R)$
Result
$(A_1, A_2, \ldots, A_n,$
$f_1, f_2, \ldots f_k)$

Figure 3.3 Grouping evaluation

For example, to find for each value of A_1 the maximum value of A_2, and the sum of the A_3 values, we write the expression:

$$_{A_1}\gamma_{\mathsf{max(A_2),sum(A_3)}}(E)$$

As in the generalized projection, attributes of a grouping operation can be renamed as follows:

$$_{A_1}\gamma_{\mathsf{max(A_2)}\,AS\,M,\mathsf{sum(A_3)}\,AS\,S}(E)$$

Sort: $\tau_{A_1,A_2,\ldots,A_m}(R)$

where $A_1, A_2, \ldots, A_m$ are attributes of R. The operator returns the tuples of R sorted in ascending order on the attributes $A_1, A_2, \ldots, A_m$. To sort in descending order the attributes becomes pairs $A_i\,d$, where d stands for "descending".

3.2.3 Equivalence rules

Two relational algebra expressions are said to be *equivalent* if, on every legal database instance, the two expressions generate the same set of tuples. Note that the order of the tuples is irrelevant.

An *equivalence rule* says that expressions of two forms are equivalent. A query optimizer uses equivalence rules to transform expressions into other logically equivalent expressions. We now present some of them.

1. *Cascade of select*

   $\sigma_{\phi_X}(\sigma_{\phi_Y}(E)) = \sigma_{\phi_X \wedge \phi_Y}(E),$
2. *Select and project are commutative*

   $\pi_Y(\sigma_{\phi_X}(E)) = \sigma_{\phi_X}(\pi_Y(E)),\ if\ X \subseteq Y.$

   If $X \not\subseteq Y$, then:

   $\pi_Y(\sigma_{\phi_X}(E)) = \pi_Y(\sigma_{\phi_X}(\pi_{XY}(E))).$

3. *Select distributes over product and join*

   $\sigma_{\phi_X}(E_1 \times E_2) = \sigma_{\phi_X}(E_1) \times E_2,$
   $\sigma_{\phi_X}(E_1 \bowtie E_2) = \sigma_{\phi_X}(E_1) \bowtie E_2,$

   if X are attributes from E_1.

   $\sigma_{\phi_X \wedge \phi_Y}(E_1 \times E_2) = \sigma_{\phi_X}(E_1) \times \sigma_{\phi_Y}(E_2),$
   $\sigma_{\phi_X \wedge \phi_Y}(E_1 \bowtie E_2) = \sigma_{\phi_X}(E_1) \bowtie \sigma_{\phi_Y}(E_2),$

   if X are attributes from E_1 and Y are attributes from E_2.

   $\sigma_{\phi_X \wedge \phi_Y \wedge \phi_J}(E_1 \times E_2) = \sigma_{\phi_X}(E_1) \underset{\phi_J}{\bowtie} \sigma_{\phi_Y}(E_2),$

   if X are attributes of E_1, Y are attributes of E_2 and ϕ_J is a join condition.
4. *Cascade of project*

   $\pi_Z(\pi_Y(E)) = \pi_Z(E),$

   where Z and Y are attributes from E, and $Z \subseteq Y$.
5. *Project distributes over product and join*

   $\pi_{XY}(E_1 \times E_2) = \pi_X(E_1) \times \pi_Y(E_2),$
   $\pi_{XY}(E_1 \underset{\phi_J}{\bowtie} E_2) = \pi_X(E_1) \underset{\phi_J}{\bowtie} \pi_Y(E_2),$

   where X are attributes from E_1, Y are attributes from E_2, and ϕ_J is the join condition with attributes $J \subseteq XY$.

   $\pi_{XY}(E_1 \underset{\phi_J}{\bowtie} E_2) = \pi_{XY}((\pi_{XX_{E_1}}(E_1)) \underset{\phi_J}{\bowtie} (\pi_{YX_{E_2}}(E_2))),$

   where X are attributes from E_1, Y are attributes from E_2, X_{E_1} are the join attributes from E_1 which are not in XY, and X_{E_2} are the join attributes from E_2 which are not in XY.
6. *Product and join are commutative*

   $E_1 \times E_2 = E_2 \times E_1$
   $E_1 \underset{\phi_J}{\bowtie} E_2 = E_2 \underset{\phi_J}{\bowtie} E_1$
7. *Product and join are associative*

   $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3),$
   $(E_1 \underset{\phi_{J_1}}{\bowtie} E_2) \underset{\phi_{J_2} \wedge \phi_{J_3}}{\bowtie} E_3 = E_1 \underset{\phi_{J_1} \wedge \phi_{J_3}}{\bowtie} (E_2 \underset{\phi_{J_2}}{\bowtie} E_3),$

   where ϕ_{J_2} contains only attributes of E_2 and E_3. If the join condition is empty, it follows that $\times$ is associative.
8. *Set union and intersection are commutative*

   $E_1 \cup E_2 = E_2 \cup E_1$
   $E_1 \cap E_2 = E_2 \cap E_1$
   $E_1 - E_2 \neq E_2 - E_1$
9. *Set union and intersection are associative*

   $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
   $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
10. *Select distributes over set operations*

    $\sigma_{\phi_X}(E_1 - E_2) = \sigma_{\phi_X}(E_1) - \sigma_{\phi_X}(E_2)$

    the equivalence holds with $-$ replaced by $\cup$ or $\cap$, while

    $\sigma_{\phi_X}(E_1 - E_2) = \sigma_{\phi_X}(E_1) - E_2$

    holds with replaced $-$ by $\cup$, but does not hold if $-$ is replaced by $\cap$.

11. *Project distributes over set operations*

    $\pi_X(E_1 \cup E_2) = (\pi_X(E_1)) \cup (\pi_X(E_2))$
12. *Select distributes over grouping*

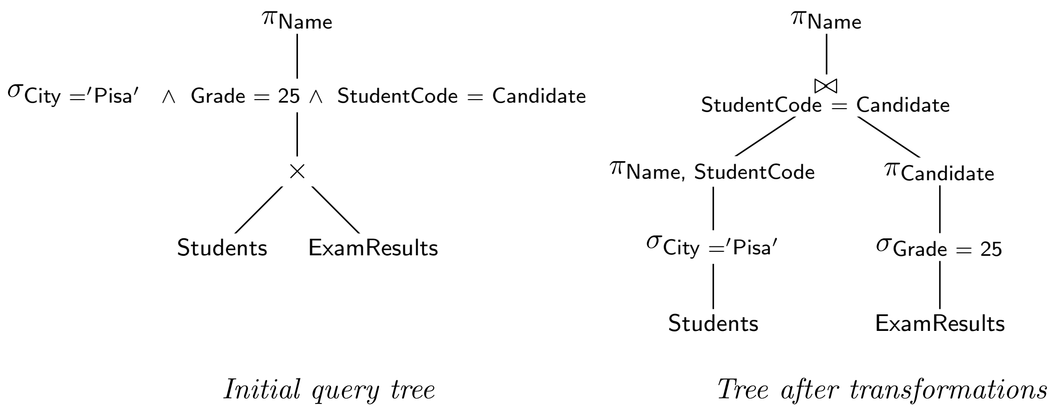
    $\sigma_X({}_A\gamma_F(E)) = {}_A\,\gamma_F(\sigma_X(E))$

    if X uses only attributes from A.

We now illustrate the use of the equivalence rules. The expression

$\pi_{\mathsf{Name}}(\sigma_\phi(\mathsf{Students} \times \mathsf{ExamResults}))$,

where $\phi = (\mathsf{StudentCode} = \mathsf{Candidate} \wedge \mathsf{City} = \mathsf{'Pisa'} \wedge \mathsf{Grade} = 30)$, can be represented as the initial expression tree (*query tree*) in Figure 3.4, and then it can can be represented also with the transformed query tree, taking into account the previous equivalence rules.



π_{Name}
$\sigma_{\mathsf{City} = \mathsf{'Pisa'} \wedge \mathsf{Grade} = 25 \wedge \mathsf{StudentCode} = \mathsf{Candidate}}$
$\times$
Students
ExamResults
Initial query tree
π_{Name}
$\bowtie_{\mathsf{StudentCode} = \mathsf{Candidate}}$
$\pi_{\mathsf{Name,\ StudentCode}}$
$\pi_{\mathsf{Candidate}}$
$\sigma_{\mathsf{City} = \mathsf{'Pisa'}}$
$\sigma_{\mathsf{Grade} = 25}$
Students
ExamResults
Tree after transformations

Figure 3.4 Expression trees

3.3 Relational Database Design: ODM-to-Relational Mapping

The growing use of DBMSs, the complexity of the new applications, and the need to implement database applications that can be readily adapted to changes in user requirements, have all led to an increasing demand for environments with integrated sets of automated tools to support both the design and the maintenance of database applications. The problem is similar to that of software engineering and the following strategies have been adopted: a) the definition of a design methodology composed of a set of structured steps in which design decisions are considered one at a time to achieve a satisfactory result; b) the definition of techniques to be used during the design steps; c) the definition of tools for an automated development support system.

The aim of a design methodology is to transform a user-oriented linguistic representation of the information needs of an organization into a DBMS-oriented description. There is a general consensus among researchers and practitioners on the static and dynamic aspects that should be modeled during the design process. Static aspects regard the data structures and integrity constraints, while dynamic aspects concern the transactions modifying the database from one consistent state to another. We shall consider here only the static aspects.

Different phases of design have been suggested to cope with the complexity of the database design process. *User requirements analysis and specification* consists of collecting user needs and normalizing them according to established standards. *Conceptual design* is the phase in which requirements are formalized and integrated into a global conceptual schema, using a DBMS-independent conceptual language. In the next phase, *logical design*, the conceptual schema is mapped into a logical schema using the data model supported by the DBMS chosen for the implementation. Finally, *physical design* concerns the selection of the data structures used to store and retrieve the data.

When a relational DBMS is used, in the logical design phase the designer can benefit from a well-developed theory, called *normalization theory*, which provides algorithms to produce a set of relation schemas with certain desirable properties, and in particular to avoid certain "bad" design decisions, both with respect to semantics and to performance.

In this section the steps of an algorithm are described to design a relational schema from a conceptual design. For simplicity only structural aspects are considered:

STEP 1: Representation of 1:N and 1:1 associations with the rules in Figure 3.5.

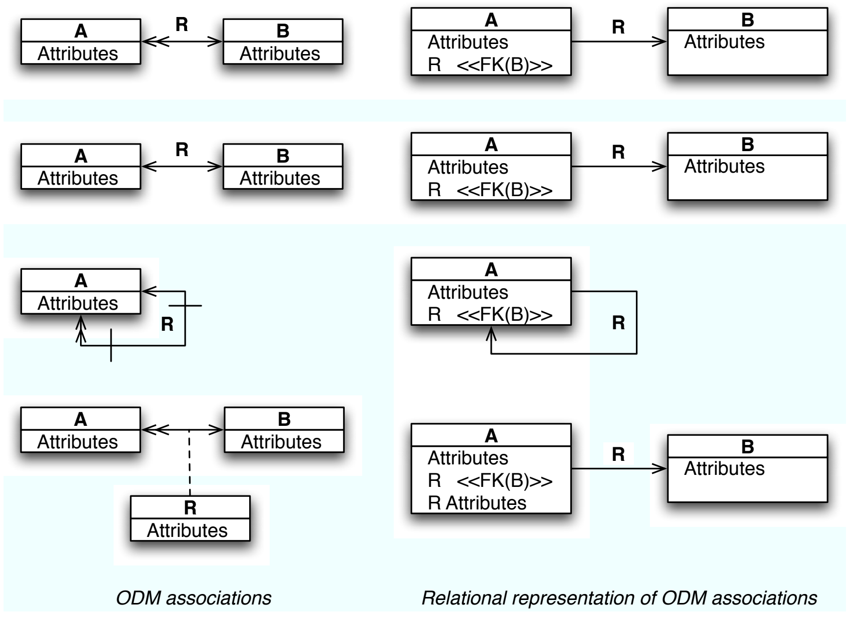


Figure 3.5 Rules for STEP 1

STEP 2: Representation of N:M associations with the rules in Figure 3.6.

STEP 3: Representation of class hierarchies with the rules in Figure 3.7. For simplicity we assume that the subclasses are disjoint and that the class attributes are not redefined in subclasses. Three main options are possible:

1. A single relation with the attributes of the class and subclasses, and a

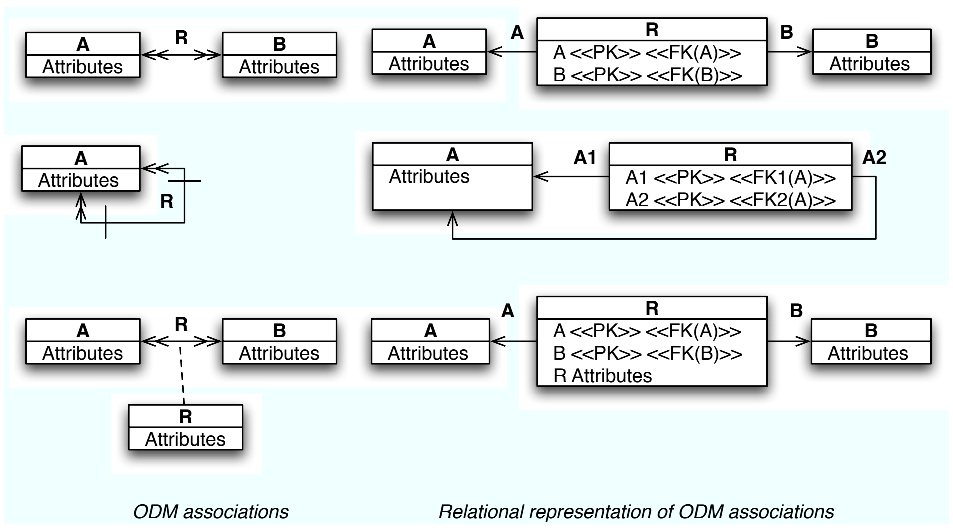


Figure 3.6 Rules for STEP 2

special attribute D to discriminate to which subclass a tuple belong, if any.

2. A relation for the class and a relation for each subclass (*vertical partitioning*). The relation for the class contains the class elements and the elements of the subclasses.
3. A relation for the class and a relation for each subclass which include the class attributes too (*horizontal partitioning*). The relation for the class contains the class elements which do not belong to the subclasses.

STEP 4: Define the primary key for each relation representing a class of the conceptual schema. For each relation representing a subclass, the primary key is that selected for the superclass.

STEP 5: Representation of multi-valued attributes: if a class C has a multi-valued attribute A, define a new relation with attributes corresponding to A, plus the primary key for C as a foreign key.

STEP 6: Representation of composite attributes: if an attribute A is a record with fields A_i, A is replaced by the A_i.

The applications of the above steps to the schema in Figure 2.8 produces the relational schema in Figure 3.8.

3.3.1 Exercises

1. Convert the following conceptual schemas to a relational database schema.
   (a) Your solution to Exercise 2.4(3).
   (b) Your solution to Exercise 2.4(5).
   (c) Your solution to Exercise 2.4(6).

3.4 Relational Database Design: Normalization Theory

In this section, we shall show how normalization theory can be applied directly following an approach to relational database design which is based on analyzing

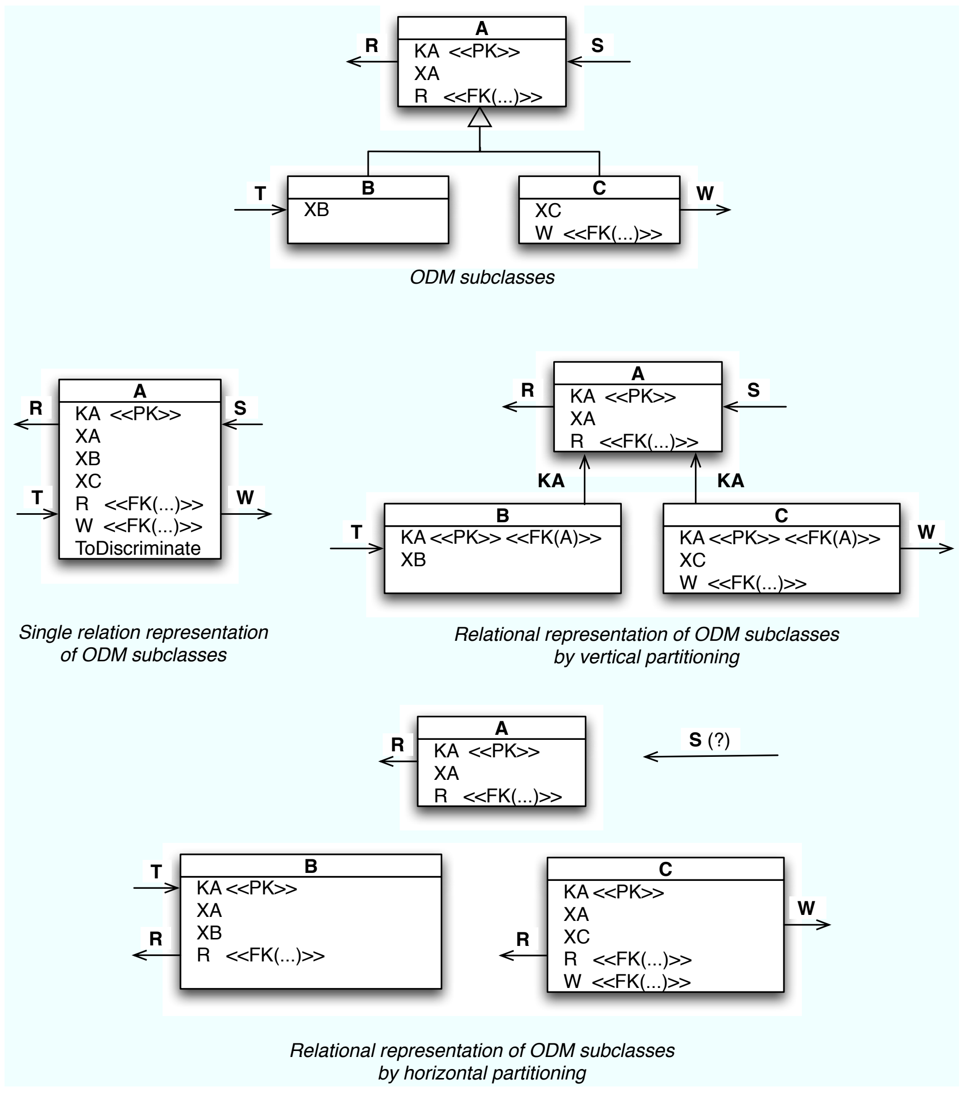


Figure 3.7 Rules for STEP 3

an application in terms of its elementary facts and the functional relationships among them, and then synthesizing a "good" set of relation schemas.

To show in which sense a relation schema can be considered "bad", let us assume that we are interested in representing certain information in a simplified library administration system, and we have decided to represent it in one relation with the following schema:

Library (UserName, Address, Tel, CallNumber, Author, Title, Date)

The library has a set of books (not more than one copy per book), each identified by a unique book number. Books may be loaned to borrowers, each identified by a unique name, and having an address and telephone number; a library user can have more than one book on loan at the same time; the lending date is also recorded. The key of the relation is {*UserName, CallNumber*}. An example of an instance of the relation is:

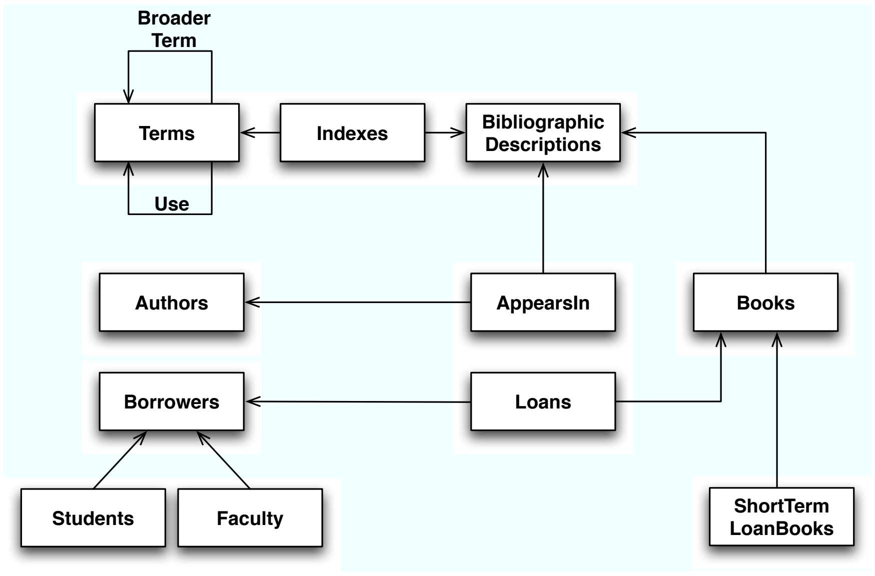


Figure 3.8 A relational schema for a library

Name	StudentCode
Isaia	071523
Bonini	075649

UserName	Address	Tel	CallNumber	Author	Title	Date
Rossi Carlo	Carrara	75444	XY188A	Boccaccio	Decameron	07-07
Paolicchi Luca	Avenza	59729	XY256B	Verga	Novelle	07-08
Pastine Maurizio	Dogana	661338	XY090C	Petrarca	Canzoniere	01-08
Paolicchi Laura	Avenza	59729	XY101A	Dante	Vita Nova	05-08
Paolicchi Luca	Avenza	59729	XY701B	Manzoni	Adelchi	14-01
Paolicchi Luca	Avenza	59729	XY008C	Moravia	La noia	17-08

The above schema is "bad" because it presents the following main undesirable properties:

- *Repetition of information.* Every time a user borrows another book, the information about his address and telephone will be repeated; this wastes space and complicates database updating when a user changes address.
- *Inability to represent certain information.* Information about users can be stored only when they borrow a book.

An alternative design is to replace the schema with two relation schemas, but a careless decomposition may lead to another kind of "bad" design. Consider the following rather absurd decomposition where the association between loans and borrowers is modeled by the telephone numbers:

Users(UserName, Address, Tel)
Loans(CallNumber, Author, Title, Date, Tel)

The instances of the two relations are obtained by projections of the *Library* relation as follows:

$Users = \pi_{\text{UserName, Address, Tel}}(Library) =$

UserName	Address	Tel
Rossi Carlo	Carrara	75444
Paolicchi Luca	Avenza	59729
Pastine Maurizio	Dogana	661338
Paolicchi Laura	Avenza	59729

$Loans = \pi_{\text{CallNumber, Author, Title, Date, Tel}}(Library) =$

CallNumber	Author	Title	Date	Tel
XY188A	Boccaccio	Decameron	07-07	75444
XY256B	Verga	Novelle	07-07	59729
XY090C	Petrarca	Canzoniere	01-08	661338
XY101A	Dante	Vita Nova	05-08	59729
XY701B	Manzoni	Adelchi	14-01	59729
XY008C	Moravia	La Noia	17-08	59729

This decomposition eliminates data duplications, but presents another anomaly when we need to reconstruct the *Library* relation. For example, suppose that we wish to send a letter to solicit users to return books borrowed in January. To obtain the required information, the following query can be formulated:

$\pi_{\text{UserName, Address}}(Users \bowtie \sigma_{\text{Data}\in(01-01,31-01)}(Loans))$

The result is

UserName	Address
Paolicchi Luca	Avenza
Paolicchi Laura	Avenza

which is wrong since Laura Paolicchi has not borrowed a book in January. Thus, when we join *Users* and *Loans* we have more tuples in the result than those we expect. This anomaly is called a *loss of information* and the decomposition is called a *lossy decomposition.* The reason for this anomaly is that we have selected a wrong external key to describe the association of users and loans. A correct design would have been

Users(UserName, Address, Tel)
Loans(CallNumber, Author, Title, Date, UserName)

The main goal of relational design theory is to give formal criteria to design databases without anomalies of the types represented by the above examples.

In the following, we will assume that attributes have a *global meaning*, i.e. attributes mean the same wherever they occur in a database schema, and we adopt the following conventions:

- Capital letters near the beginning of the alphabet stand for single attributes (A, B, A_1, A_2, etc.).
- Capital letters near the end of the alphabet stand for sets of attributes (X, Y, U, Z, etc.).
- XY is used as a shorthand for $X \cup Y$, AB as a shorthand for $\{A, B\}$, and AX as a shorthand for $\{A\} \cup X$.
- $A_1A_2 \dots A_n$ is a shorthand for $\{A_1, A_2, \dots, A_n\}$.
- Names beginning with a capital letter denote relation schemas, and $R(T)$ a relation with a set of attributes T.
- Let t be a tuple, $R(T)$ a relation schema, and $X \subseteq T$, then $t[X]$ denotes the X-value of t.

3.4.1 Functional Dependencies

In order to formalize the notion of schema without anomalies, we need a formal description of the semantics of the facts stored in a relation. Codd [?] proposed a particular kind of formalism based on the notion of functional dependency:

Definition 3.4.1 Given a relation schema $R(T)$ and $X, Y \subseteq T$, a functional dependency (FD) is a constraint on R of the form $X \rightarrow Y$, i.e. X *functionally determines* Y or Y is *determined* by X, if for any legal instance r of R a value of X uniquely determines a value of Y

$$\forall t_1, t_2 \in r \text{ such that } t_1[X] = t_2[X] \text{ it is also the case that } t_1[Y] = t_2[Y]. \tag{3.1}$$

We say that an instance r of R *satisfies* the FD $X \rightarrow Y$ if condition (3.1) holds, and that an instance r of R satisfies a set F of FD if, for each $X \rightarrow Y \in F$, condition (3.1) holds.

Condition (3.1) formally expresses the following constraint: in any legal instance r of R, if two tuples have the same X value, then they will also have the same Y value. These kinds of constraints depend on the semantics of the represented facts and consequently must be true for any legal instance r of R; we cannot look at a particular instance of R and deduce what functional dependencies hold for R. Functional dependencies might be enforced by a DBMS if this is specified by the database designer, but relational systems usually enforce only those functional dependencies that follow from the fact that a key determines the other attributes of a relation. Since functional dependencies are an important aspect in database design, in the following we will use the convention that $R < T, F >$ denotes a schema with a set T of attributes and a set F of functional dependencies over T.

Let us consider a legal instance r of $R < T, F >$, with $F = \{X \rightarrow Y, X \rightarrow Z\}$, $X, Y, Z \subseteq T$, and $W \subseteq X$. Many other functional dependencies are satisfied by r including, for example, $X \rightarrow W$ and $X \rightarrow YZ$. In fact, in the first case, if two tuples have the same value on X, they will certainly have the same value on W which is a subset of X (trivial FD); in the second case if $t_1[X] = t_2[X]$, since t_1, t_2 satisfy the FDs in F, it is also the case that $t_1[Y] = t_2[Y]$ and $t_1[Z] = t_2[Z]$, and consequently $t_1[YZ] = t_2[YZ]$.

Thus, given a set F of FDs, other FDs will generally be 'implied' by this set in the following sense:

Definition 3.4.2 Given a set F of FDs on a schema R, we say that $F \models X \rightarrow Y$, i.e. F *logically implies* $X \rightarrow Y$, if every instance r of R that satisfies F also satisfies $X \rightarrow Y$.

From this definition, the previous example has shown that

$$\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$$
and
$$W \subseteq X \ \{\} \models X \rightarrow W$$

An interesting question is whether there is a way of computing all the possible FDs logically implied by a set F, using a set of inference rules with the property of being *sound* and *complete* so that we can derive mechanically all the FDs implied by F, and only those.

3.4.2 Inference Rules

A set of inference rule to derive new FDs mechanically from a given set F are the Armstrong axioms[2]:

F1 (*reflexivity*) If $Y \subseteq X$, then $X \rightarrow Y$
F2 (*augmentation*) If $X \rightarrow Y$, $Z \subseteq T$, then $XZ \rightarrow YZ$
F3 (*transitivity*) If $X \rightarrow Y, Y \rightarrow Z$, then $X \rightarrow Z$

Definition 3.4.3 $F \vdash X \rightarrow Y$ iff $X \rightarrow Y$ can be inferred from F using Armstrong's axioms as inference rules.

Using these rules, the following rules can also be proved correct

$$\begin{array}{l} \{X \rightarrow Y, X \rightarrow Z\} \vdash X \rightarrow YZ \,(\textit{union rule}) \\ Z \subseteq Y\,\{X \rightarrow Y\} \vdash X \rightarrow Z(\textit{decomposition rule}) \\ \{\} \vdash X \rightarrow X \\ \{X \rightarrow Y\} \vdash XZ \rightarrow Y \\ W \subseteq Z, V \subseteq Y\,\{X \rightarrow Y\} \vdash XZ \rightarrow VW \end{array}$$

So far, we have discussed derived dependencies in two ways: we have talked about logically implied dependencies ($\models$) and about dependencies which are inferred using Armstrong's axioms as deduction rules ($\vdash$). In fact, these two ways of defining derived dependencies are the same: if a functional dependency f can be inferred from a set F using Armstrong's axioms, then f is logically implied by F (*soundness*), and, vice versa, if f is logically implied by F, then f can also be inferred using Armstrong's axioms (*completeness*).

Theorem 3.4.1 Armstrong's axioms are sound and complete.

A consequence of this theorem is that we can substitute $\models$ with $\vdash$ and vice versa in all the previous results.

3.4.3 Closure of a Set of FDs

Definition 3.4.4 Given a set F of FDs, the closure of F, denoted by F^+, is: $F^+ = \{X \rightarrow Y | F \vdash X \rightarrow Y\}$

Therefore, to test whether an FD $V \rightarrow W$ is in F^+ (the *implication problem*) we can generate all the FDs in F^+, which is a finite set, by applying Armstrong's axioms repeatedly. This way of solving the implication problem is generally found time-consuming, simply because the set of dependencies in F^+ can be large even when F itself is small. Consider the set $F = \{A \rightarrow B_1, \ldots, A \rightarrow B_n\}$, then F^+ will includes all the dependencies $A \rightarrow Y$, where Y is a subset of $\{B_1, \ldots, B_n\}$ and there are 2^n of sets Y.

A simpler way of solving the implication problem follows from the following notion of *closure* of a set of attributes and theorem.

Definition 3.4.5 Given a schema $R < T, F >$, and $X \subseteq T$, the *closure* of X, denoted by X^+, is $X^+ = \{A \in T | F \vdash X \rightarrow A\}$.

Theorem 3.4.2 $F \vdash X \rightarrow Y$ iff $Y \subseteq X^+$.

2. There are several equivalent sets of rules and we present just one of them here.

Instead of computing F^+, compute X^+ and then test whether $Y \subseteq X^+$. Therefore an algorithm to compute X^+ is

```
$X^+ = X$
**while** (changes to $X^+$) **do**
  **for each** $W \to V$ **in** $F$ **with** $W \subseteq X^+$ **and** $V \not\subseteq X^+$
    **do** $X^+ = X^+ \cup V$
```

It turns out that in the worst case this algorithm has time complexity $O(ap \min\{a, p\})$, where a is the number of attributes and p the number of FDs. A faster algorithm, with time complexity $O(ap)$, has been given by [**?**].

Using the notions of functional dependency and closure of sets of dependencies, we can formally define the concept of *key* of a relation.

Definition 3.4.6 Given the schema $R < T, F >$, we say that $W \subseteq T$ is a *key* (or a *candidate key*) of R if

$$\begin{array}{l} 1.\ W \to T \in F^+ \\ 2.\ \forall V \subset W, V \to T \notin F^+ \end{array}$$

In general, there are many candidate keys for a relation, and we designate one of them as the *primary* key to be used in representing associations. We also use the term *superkey* for any superset of a key and the term *prime attribute* for an attribute which belongs to a candidate key. The following results have been proved for keys:

1. The problem of finding all the keys of a relation requires an algorithm with an exponential time complexity.
2. The problem of testing whether an attribute is prime is $\mathcal{NP}$-complete.

3.4.4 Covers of Sets of Dependencies

Let F and G be sets of dependencies on the same attributes. Using the notion of closure we can determine when two sets of dependencies are equivalent and thus when two schemas on the same attributes represent the same information.

Definition 3.4.7 Two sets of FDs, F and G, over schema R are *equivalent*, written $F \equiv G$, iff $F^+ = G^+$. If $F \equiv G$, then F is a *cover* for G (and G a *cover* for F).

It is easy to test whether F and G are equivalent: test if every dependency in F is in G^+, and every dependency in G is in F^+.

It is useful to have a cover for a given set of FDs which is easy to deal with and which has simple and important properties. An example is given in the following definition.

Definition 3.4.8 Let F be a set of FDs

1. Given $X \to Y \in F$, we say that X contains an *extraneous attribute* A_i iff $X - \{A_i\} \to Y \in F^+$;
2. $X \to Y$ is a *redundant dependency* iff $X \to Y \in (F - \{X \to Y\})^+$;
3. F is called a *canonical cover* iff
   - every right side of a dependency in F is a single attribute;
   - no attribute on any left side is extraneous;

- no dependency in F is redundant.

Theorem 3.4.3 Every set of dependencies F is equivalent to a set F' that is a canonical cover.

The following example shows that in general a set F of FDs can have more than one canonical cover.

Example 3.3 For the set $F = \{AB \to C, A \to B, B \to A\}$ both $\{A \to C, A \to B, B \to A\}$ and $\{B \to C, A \to B, B \to A\}$ are canonical covers.

An algorithm to compute a canonical cover based on definition 3.4.8 has time complexity $O(a^2p^2)$.

3.4.5 Schema Decomposition

It has been shown that in order to eliminate anomalies from a bad schema, the schema must be decomposed into smaller schemas. Let us define formally the notion of a decomposition and its desirable properties.

Definition 3.4.9 A decomposition of a schema $R(T)$ is the substitution of $R(T)$ with a set $\rho = \{R_1, \dots, R_k\}$ of schemas $R_i(T_i)$ such that $\cup T_i = T$.

There are two desirable properties of a decomposition, *data preserving* (*lossless join*) and *dependency preserving.*

Data Preserving Decomposition

Definition 3.4.10 Given a schema $R < T, F >$, the decomposition $\rho = \{R_1, \dots, R_k\}$ is data preserving if for every legal instance r of R:

$$r = (\pi_{T_1} r) \bowtie (\pi_{T_2} r) \bowtie \cdots \bowtie (\pi_{T_k} r)$$

That is, every legal instance r is the natural join of its projections onto the R_i's. From the definition of the natural join operator, the following result can be proved.

Theorem 3.4.4 Let $R < T, F >$ be a relation schema, $\rho = \{R_1, \dots, R_k\}$ be any decomposition of R, and r any legal instance of R. Then:

$$r \subseteq (\pi_{T_1} r) \bowtie (\pi_{T_2} r) \bowtie \cdots \bowtie (\pi_{T_k} r)$$

This theorem clarifies the notion of loss of information: in general a relation is not recoverable from its decomposition, as it is shown by the following example.

Example 3.4 Let us consider the following instance of the relation $R(A, B, C)$:

A	B	C
a_1	b	c_1
a_2	b	c_2

The following decomposition is not data preserving because $r \subseteq (\pi_{A,B}r) \bowtie (\pi_{B,C}r)$.

$\pi_{A,B}(r) =$

A	B
a_1	b
a_2	b

$\pi_{B,C}(r) =$

B	C
b	c_1
b	c_2

Since it is desirable for a decomposition to be data preserving, the following theorem gives a condition which can be used to establish when this property holds.

Theorem 3.4.5 Let $R < T, F >$ be a relation schema, the decomposition $\rho = \{R_1, R_2\}$ is data preserving iff $T_1 \cap T_2 \rightarrow T_1 \in F^+$ or $T_1 \cap T_2 \rightarrow T_2 \in F^+$.

This result has been extended by providing an algorithm to test whether a decomposition in more than two smaller relations is data preserving.

3.4.6 Dependency Preserving Decomposition

Definition 3.4.11 Given the schema $R < T, F >$, and $T_i \subseteq T$, the projection of F onto T_i is

$$\pi_{T_i}(F) = \{X \rightarrow Y \in F^+ | X, Y \subseteq T_i\}$$

Proposition 3.4.1 Given a schema $R < T, F >$, and $X \subseteq T$, the problem of finding a canonical cover of the projection of F on X is $\mathcal{NP}$-complete.

A simple algorithm for computing $\pi_{T_i}(F)$ is

```
Algorithm Projection of $F$ onto $T_i$
    input     $R\langle T, F\rangle$ and $T_i \subseteq T$
    output    A cover of the projection of $F$ onto $T_i$
begin
    for each $Y \subseteq T_i$ do
        begin
            $Z := Y_F^+$
            return $Y \rightarrow (Z \cap T_i)$
        end
end
```

Definition 3.4.12 Given a schema $R < T, F >$, the decomposition $\rho = \{R_1, \ldots, R_n\}$ is dependency preserving iff $\cup\pi_{T_i}(F) \equiv F$.

A trivial algorithm for testing whether a decomposition $\rho = \{R_1, \ldots, R_n\}$ preserves a set of dependencies F is to compute the projections of F onto the attributes T_i, take the union $\cup T_i$, and test whether this set is equivalent to F. This algorithm will have an exponential time complexity. However a faster algorithm exist which does not require the computation of the projections of F onto the attributes T_i, and takes time that is polynomial in the size of F [?].

The reason why it is desirable for a decomposition to preserve a set of dependencies F is that the dependencies in F are integrity constraints for the relation R. If the projected dependencies did not imply F, then every update to one of the R_i's would require a join to check that the constraints were not violated.

The data preserving and dependency preserving properties of a decomposition are independent, i.e. there exist lossless decompositions which do not preserve dependencies and vice versa. The following result relates the two properties and gives a sufficient but not necessary condition to establish if a dependency preserving decomposition is data preserving.

Definition 3.4.13 Given a schema $R < T, F >$ and a dependency preserving decomposition $\rho = \{R_i < T_i, F_i >\}$ such that a T_j is a superkey for $R < T, F >$, then ρ is data preserving.

3.4.7 Normalization Using Functional Dependencies

We now examine how functional dependencies can be used to define several normal forms which represent "good" database design. The most important are the *third normal form* (3NF) and the *Boyce-Codd normal form* (BCNF).

Definition 3.4.14 $R < T, F >$ is in 3NF if, when $X \to A \in F^+$, and $A \notin X$, then X includes a key or A is prime.

Example 3.5 A schema which is not in 3NF is

Employees(#Employee, NameOfEmployee, NameOfDept, InformationOnDept)
#Employee → NameOfEmployee NameOfDept InformationOnDept
NameOfDept → InformationOnDept

The relation *Employees* is not in a desirable form since there is a repetition of information: if there are several employees working in the same department, then we are forced to repeat the information on the department for each employee.

If F is a canonical cover, then the following result holds

Proposition 3.4.2 $R < T, F >$ is in 3NF if, when $X \to A \in F$, X is a key or A is prime.

Since, for both the definitions, we need to know if an attribute is prime in order to test whether a relation schema is in 3NF, we will have the following result.

Proposition 3.4.3 The problem of deciding whether a relation schema $R < T, F >$ is in 3NF is $\mathcal{NP}$-complete.

Example 3.6 Let us consider the following schema *ZipCodes(City, Street, Zip)*, with FDs

City Street → Zip
Zip → City

That is, the address (city and street) determines the zip code, and the zip code determines the city, although not the street address. Since the candidate keys are {*City, Street*}, {*Street, Zip*}, all attributes are primes, and thus the schema is in 3NF, but it suffers from the repetition of information problem. Consequently, 3NF does not solve the problem of detecting "bad" schemas completely and another normal form is required.

Definition 3.4.15 $R < T, F >$ is in BCNF if, when $X \to A \in F^+$, and $A \notin X$, then X is a superkey.

The schema *ZipCodes(City, Street, Zip)* from Example 3.6 is a well known example showing that a relation schema can be in 3NF without being in BCNF. If F is a canonical cover, then the following result holds

Proposition 3.4.4 $R < T, F >$ is in BCNF if, when $X \to A \in F$, X is a key.

From this definition, it follows that an algorithm to test whether a single relation schema is in BCNF has a complexity $O(ap^2)$.

Proposition 3.4.5 Given a schema $R < T, F >$, $X \subseteq T$, and F' the projection of F onto X, the problem of deciding if $R' < X, F' >$ is in BCNF is $\mathcal{NP}$-complete.

3.4.8 Polynomial Algorithms to Normalize in 3NF and BCNF

A Synthesis Algorithm for 3NF

The best known synthesis algorithm was proposed by Bernstein [**?**]. The basic steps are the followings: (1) first a canonical cover G of the FDs is computed; (2) G is then partitioned into groups G_i such that all the FDs in each G_i will have the same left-hand side and no two groups will have the same left-hand side; (3) each G_i then produces a 3NF relation schema composed of all the attributes in G_i. The algorithm will obviously provide a dependency preserving decomposition.

However, in order to avoid the synthesis of superfluous schemas, this basic algorithm must be extended, as shown in the following example.

Example 3.7 Let $F = \{A \to B, B \to A, C \to D, D \to C\}$; F is a canonical cover and the basic algorithm generates the following schemas: $R_1(A, B)$, $R_2(A, B)$, $R_3(C, D)$, and $R_4(C, D)$, while two relations are sufficient, $R_1(A, B)$ and $R_3(C, D)$.

The extension of the basic algorithm is reported in Bernstein [**?**] where it is also shown that its complexity is $O(a^2p^2)$. In [**?**], another step has been added to the algorithm to produce a set of relation schemas in 3NF that has both the data and dependency preservation properties. It requires that the final set of relation schemas includes a relation whose key is also the key of the relation which contains all the attributes in the initial FDs, which are the inputs to the synthesis algorithm.

As a consequence of this result, we have that it is faster to produce a set of relation schemas in 3NF, than to test whether a single relation schema is already in 3NF.

As there is no synthesis algorithm which can be used to produce a relation schema in BCNF, another approach must be used.

A Decomposition Algorithm for BCNF

The goal of a decomposition algorithm is to convert a relation schema which is not in BCNF into a set of relations: If $R(X, Y, Z)$ is not in BCNF because of $X \to Y$, R is decomposed into: $R_1(X, Y)$ and $R_2(X, Z)$. The process continues as long as the R_i are not in BCNF. Therefore a decomposition algorithm is:

```
$\rho = \{R < T, F >\}$
**while** exists in $\rho$ a $R_i < T_i, F_i >$ not in BCNF because of the FD $X \to A$ **do**
  $T_1 = X\,A$
  $F_1 = \pi_{T_1}(F_i)$
  $T_2 = T_i - A$
  $F_2 = \pi_{T_2}(F_i)$
  $\rho = \rho - R_i + \{R_1 < T_1, F_1 >, R_2 < T_2, F_2 >\}$
**end**
```

The decomposition is data preserving but, in general, not dependency preserving, as shown by the following example: $R < \{J, K, L\}, \{JK \to L, L \to K\} >$ is not in BCNF, however every decomposition will fail to preserve $JK \to L$. Thus, obtaining a data and dependency preserving decomposition is an impossible goal.

[**?**] gave an algorithm with a polynomial time complexity $O(a^5 p)$ to compute a data preserving decomposition in BCNF, although it will sometimes decompose a relation that is already in BCNF. However, the problem of deciding whether a relation schema has a dependency preserving decomposition in BCNF is $\mathcal{NP}$-hard.

3.4.9 Multivalued Dependencies and Fourth Normal Form

We have introduced the concepts of functional dependency, 3NF, and BCNF normal forms to avoid schemas with anomalies. Unfortunately, 3NF and BCNF are insufficient to solve the problem. For example, the relation *Employees(EmplName, ChildName, Salary, Year)*, used to store information about the children and salary histories of employees, is in BCNF (there are no FDs), however there is a lot of data redundancy.

Employees

EmplName	ChildName	Salary	Year
Bragazzi	Maurizio	1000000	1980
Bragazzi	Maurizio	1200000	1984
Bragazzi	Maurizio	1400000	1988
Bragazzi	Marcello	1000000	1980
Bragazzi	Marcello	1200000	1984
Bragazzi	Marcello	1400000	1988
Fantini	Maria	1000000	1980
Fantini	Maria	800000	1984
Fantini	Maria	600000	1988

Informally, data redundancy occurs whenever a multivalued property is represented in a relation schema together with another simple or multivalued independent property. An example is when we attempt to represent the children and salary histories properties for employees. If we had represented only one of these properties in a relation, we would not have had this a problem:

EmployeeSalaries (EmplName, Salary, Year).
EmployeeChildren (EmplName, ChildName)

To deal with this redundancy, the concept of *multivalued dependencies* (MVDs) is needed. MVDs unlike FDs, are not only a property of the information represented by relations, but are dependent on the way attributes are grouped into relations.

Definition 3.4.16 Given a schema $R(T), X, Y$ subsets of T, $Z = T - XY$, there is in R a multivalued dependency (MVD) $X \to\to Y$, read "*X multidetermines*

Y", iff in any instance r of R, for any two tuples $t_1, t_2 \in r$ with $t_1[X] = t_2[X]$, there exists a tuple $t_3 \in r$ such that $t_3[X] = t_1[X] = t_2[X], t_3[Y] = t_1[Y]$, and $t_3[Z] = t_2[Z]$.

Exchanging the roles of t_1 and t_2 in the definition, we have that in addition to t_3, a tuple t_4 must exist such that $t_4[X] = t_1[X] = t_2[X], t_4[Y] = t_2[Y]$, and $t_4[Z] = t_1[Z]$.

For example, in the relation *Employees*, the MVD *EmplName* $\rightarrow\rightarrow$ *ChildName* holds; let

t_1 = (Bragazzi, Maurizio, 1000000, 1980)
t_2 = (Bragazzi, Marcello, 1200000, 1984)

then the following tuples will also exist in the relation

t_3 = (Bragazzi, Maurizio, 1200000, 1984)
t_4 = (Bragazzi, Marcello, 1000000, 1980)

Let D denote the set of functional and multivalued dependencies. The closure, D^+, of D is the set of all functional and multivalued dependencies logically implied by D. As was the case for functional dependencies, we can reason about D using a set of inference rules.

Theorem 3.4.6 [?] The following axioms are sound and complete for functional and multivalued dependencies:

F1 (*reflexivity*) If $Y \subseteq X$, then $EX \rightarrow Y$
F2 (*augmentation*) If $X \rightarrow Y, Z \subseteq T$, then $XZ \rightarrow YZ$
F3 (*transitivity*) If $X \rightarrow Y, Y \rightarrow Z$, then $X \rightarrow Z$
M1 (*complemention*) If $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow T - XY$
M2 (*multivalued augmentation*) If $V \subseteq W, W \subseteq T, X \rightarrow\rightarrow Y$, then $XW \rightarrow\rightarrow YV$
M3 (*multivalued transitivity*) If $X \rightarrow\rightarrow Y, Y \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow Z - Y$
M4 (*replication*) If $X \rightarrow Y$, then $X \rightarrow\rightarrow Y$
M5 If $Z' \subseteq Z, Y \cap Z = \emptyset, X \rightarrow\rightarrow Y, Y \rightarrow Z'$, then $X \rightarrow Z'$

The following theorem shows how MVDs are related to lossless decomposition.

Theorem 3.4.7 Let $R < T, D >$ be a relation schema and $\rho = \{R_1, R_2\}$ a decomposition of R. Then ρ is data preserving iff $T_1 \cap T_2 \rightarrow\rightarrow T_1 \in D^+$ (or equivalently $T_1 \cap T_2 \rightarrow\rightarrow T_2 \in D^+$).

If we consider a particular instance r, we will have the following result, which is a more general statement of the similar result for FDs.

Theorem 3.4.8 Let $R < T, D >$ be a relation schema, and $\rho = \{R_1, R_2\}$ a decomposition; an instance r of R which satisfies D decomposes on ρ without loss of information iff r satisfies $T_1 \cap T_2 \rightarrow\rightarrow T_1$ (or equivalently $T_1 \cap T_2 \rightarrow\rightarrow T_2$)

This theorem gives us a method which can be used to test if an instance r of a relation $R < T, D >$ satisfies the MVD $X \rightarrow\rightarrow Y$. We project r onto XY and $X(T - XY)$, join the two projections, and test if the result is r.

Finally, there is a generalization of a Boyce-Codd normal form, called *fourth normal form* (4NF), that applies to relation schemas with functional and multivalued dependencies.

Definition 3.4.17 An MVD over $R < T, D >$ is trivial if $X \rightarrow\rightarrow \emptyset$, where $\emptyset$ is an empty set of attributes, or $X \rightarrow\rightarrow T - X$.

Definition 3.4.18 A relation schema $R < T, D >$ is in 4NF if for every non-trivial MVD $X \rightarrow\rightarrow Y$ in R, X is a superkey of R.

The following result shows that 4NF is a generalization for BCNF.

Definition 3.4.19 If $R < T, D >$ is in 4NF then it is in BCNF.

A relation that is not in 4NF can be decomposed in much the same way as we constructed BCNF database schemas. The resulting decomposition is data preserving. However, in general, it is not possible to design a database schema that meets the three criteria: 4NF, dependency preservation, and data preservation. Moreover, it is not known how (or if) a synthesis algorithm can handle MVDs.

Other kinds of dependencies have been defined to avoid other forms of data redundancy in a relation schema. The interested reader may consult [**?**] for a fuller discussion of dependency theory, including other topics which have not been addressed here.

3.4.10 Exercises

1. Prove that for a schema $R < T, F >$, with F a canonical cover, if an attribute A_i does not appear in the right side of any FD, then A_i belongs to every key of R.
2. Prove that if a schema $R < T, F >$ has two attributes only, then it is in BCNF.
3. Prove that if a schema $R < T, F >$ is in 3NF, and all keys are made of one attributes, then it is in BCNF. *Hint*: prove that for each $X \rightarrow A \in F$, X is a superkey.
4. For each of the following relational schemas and set of functional dependencies:
   (a) $R(A, B, C, D)$ with functional dependencies $AB \rightarrow C$, $C \rightarrow D$, and $D \rightarrow A$.
   (b) $R(A, B, C, D)$ with functional dependencies $A \rightarrow B$, and $A \rightarrow C$.
   (c) $R(A, B, C, D)$ with functional dependencies $A \rightarrow B$, and $B \rightarrow C$.

   do the following:

   (a) Find all the keys of R,
   (b) Indicate all the BCNF violations.
   (c) Decompose the relations, as necessary, into collections of relations that are in BCNF. Say if the decomposition is dependency preserving.
   (d) Indicate all the 3NF violations.
   (e) Decompose the relations, as necessary, into collections of relations that are in 3NF and are data preserving.
5. Consider the following poorly designed relational schema:

   UnivInfo(studID, studName, course, profID, profOffice)

   Each tuple in relation UnivInfo encodes the fact that the student with the given ID and name took the given course from the professor with the given ID and office. Assume that students have unique ID's but not necessarily unique names, and professors have unique ID's but not necessarily unique office. Each student has one name; each professor has one office.

   (a) Specify a set of completely nontrivial functional dependencies for relation UnivInfo that encodes the assumptions described above but no additional assumptions.

(b) Decompose relation UnivInfo into BCNF according to your functional dependencies in part (1).

(c) Now add the following two assumptions: (1) No student takes two different courses from the same professor; (2) No course is taught by more than one professor. Modify your set of functional dependencies from part (a) to take these new assumptions into account.

Chapter 4

DBMS: THE USER PERSPECTIVE

4.1 Objectives of a DBMS

The most common class of computer applications is used to store, maintain, and retrieve large quantities of persistent data, i.e. data that are required to last longer than the duration of the execution of the programs using them. All computerized information systems, whether in a public or private environment, fall into this class.

During the 1950s and most of the 1960s, these kinds of applications were developed using programming languages with *files*, collections of homogeneous records with the property of persistency. The responsibility for organizing and maintaining data rested entirely on the application programmers. The logical and physical structure of the data was described in the programs and the code to manipulate the data was dependent on these structures. In addition, this coupling of programs and data tended to make files specific to individual applications, precluding the sharing of common data among related applications. Consequently, it was common to have multiple copies of the same data which comported problems of consistencies between different versions and inefficient use of storage. Finally, the need for familiarity with programming languages in order to use data, often prevented the end-users, i.e. non computer professionals, from getting direct access to the data without going through a programmer intermediary.

In the late 1960s and early 1970s, a series of software systems were developed to simplify the task of maintaining and accessing persistent data. These systems began evolving to database management systems by centralizing the control of data and providing a uniform interface to it: the system rather than a user's application program has the responsibility for maintaining and manipulating data by providing the application programs with a logical view of the data, hiding the details of the structures employed to store and access them. In addition, to simplify the programming task of each user, the database management system promotes the sharing of data among users.

The term database is sometimes used for any computerized collection of data. Here, we use a more narrow definition which restricts the use of the term to what is sometimes called *formatted* data.

Definition 4.1.1 A database is a collection of persistent data, partitioned into two:

a) the schema, a collection of time-invariant definitions which describe the structure of admissible data, as well as constraints on legal data values, i.e. integrity constraints, (the *intensional database*);
b) the data, a time-variant representation of specific facts (the *extensional database*), with the following characteristics:
   - they are organized in sets, and associations are defined between these sets using the abstraction mechanism of a *data model*;
   - they occur in large quantities and do not fit in a conventional main memory;
   - they are persistent, i.e. once created, the data continue to exist until being explicitly deleted;
   - they are accessed by an *atomic* work unit (called *transaction*) which, when executed, commits either all or none of the changes effected to the extensional database;
   - they are protected both from unauthorized users and from hardware and software failures;
   - they are shared concurrently by several users[1].

All above features are guaranteed by a *Data Base Management System* (DBMS), defined as follows:

Definition 4.1.2 A DBMS is a centralized or distributed software system, which provides the tools to define the database schema, to select the data structures needed to store and retrieve data easily, and to access the data, interactively using a query language or by means of a programming language.

A more detailed presentation of the operational facilities provided by a DBMSs follows.

4.2 Functions of a DBMS

A DBMS will provide a number of different services and utilities. However, some DBMSs, especially those designed for personal computers, provide only a subset of the capabilities that will be discussed below. For example, in order to keep the system price low, many small systems do not provide facilities for concurrency control and data recovery. In general, however, such facilities are considered essential in the computerized information systems implemented in medium-sized or large organizations.

4.2.1 Separation of Data Description and Data Manipulation

In programming languages, the data declarations and the executable statements usually constitute a single program module. With DBMSs, instead, there is a separation of the database description, the *schema*, from application programs that use data. Several levels of data description are supported: *physical level, logical level, logical view level.*

1. The term "user" is adopted throughout this paper to mean either an end-user or an application program which is performing data manipulation operations.

The *physical level* is the lowest level of abstraction at which the database is described. This level contains the description of the data structures used to store and access the data. The principal data structures used will be discussed in sections ??–??.

The *logical level*, often called the *conceptual level*, is the next level of abstraction and describes the logical structure of the data and the relationships established among them, i.e. the schema, using a language which supports the abstraction mechanisms of a particular data model. The language used for the classical data models — the hierarchical, network, and relational data models, discussed below — is called the *Data Description Language* (DDL), since only data are described in the database schema and not procedural aspects.
The *logical view level* is the level at which that part of the entire database which is accessible to a certain class of users is described (*external schema*). There may be many views of the same database, and all of them are defined in terms of the schema given at the logical level. For example, only some classes may be accessible and only a subset of the attributes of an element are visible for a particular user category. An external schema is not necessarily a subset of a schema, it can also contain new classes, defined in terms of those actually present in the database.

The description of the database at these different levels is given by the person responsible for creating the database, usually known as the *database administrator* (DBA), and the information in the schema is usually stored in a system *catalog*, described in the following, which constitutes an additional database that can be queried by users .

Example 4.1 The difference between the levels of data description can be understood using an example of a relational database for university employees. At the logical level, the database structure is described in terms of the following table:

```
CREATE TABLE Persons (Name        CHAR(30),
                      FiscalCode  CHAR(15),
                      Salary      INTEGER,
                      Status      CHAR(6),
                      Address     CHAR(8))
```

At the logical view level, to the administration office and to the library is not allowed to access all the information in the table Persons, but only a subset of them:

```
CREATE  VIEW PersonsForAdministration  AS
        SELECT Name, FiascalCode, Salary, Status
        FROM   Persons

CREATE VIEW  PersonsForLibrary AS
        SELECT  Name, Address
        FROM    Persons
```

A *view* is a table computed from others as we will see later.

Finally, at the physical level, the database designer selects a data organization for each database table from a set of possible options, e.g., sequential, hash or tree structured organizations. However, the user of a class will be unaware of the physical organization selected for this class:

```
MODIFY  Persons  TO HASH ON  Name
```

These three levels of data description were proposed in 1978 by the ANSI/X3/SPARC study group on DBMSs, with the aim of guaranteeing two important properties: *physical* and *logical data independence.*

Physical data independence means that modifications to the physical database organization will not imply modifications to applications programs.

Logical data independence means that the mechanism used to define external schemas should ensure that certain modifications to the logical schema, such as adding new definitions for example, will not comport changes to the application programs, but simply a redefinition of the associated external schemas in terms of the new logical schema. The only kind of change in the logical schema that cannot be reflected in a redefinition of an external schema is the deletion of information in the logical schema which corresponds to information present in the external schema. Logical data independence is highly desirable because of the costs involved in software maintenance.

Although these three levels of data description are not supported in most DBMSs, some systems, for example the relational ones, have physical and logical data independence.

4.2.2 Database Languages

The operators associated to the data model used to access or modify the database constitute the so-called *Data Manipulation Language* (DML) of a DBMS. Typically, the DML may be used either in a stand-alone mode as a query or update language, or it may be used in a host language mode, i.e. embedded in a programming language.

There are two kinds of data manipulation operators:

- *procedural*, which are "record oriented", in the sense that they deliver one record at a time and require that a user, wishing to retrieve a particular set of records, writes a procedure which implements an appropriate search strategy to "navigate" through the database structure;
- *nonprocedural*, or *declarative*, which are "set oriented", in the sense that they deliver a set of records satisfying a condition and require a user to characterize the data he wants, with the system assuming the responsibility for devising an appropriate search strategy.

In addition to query language and programming language interfaces for the application programmer, a DBMS will offer a language for *report generation*, i.e. a language in which the user can specify a query together with requirements on the visual form of output, and a language for *data entry*, i.e. a language in which non-computer professionals can specify database entry and update on-line.

4.2.3 Data Control

A DBMS provides a number of facilities to control the physical and logical integrity of data. These facilities are:

- *access control* which limits the kind of access to the database allowed to a particular user. In fact, although the purpose of a DBMS is to facilitate database sharing by users, this sharing must be selective. The owner of data should be able to specify the nature of the access privileges allowed to those

users who will access the data (i.e. read only or read/write), to allow certain users to see only certain fields or certain records, or even to allow only a view of aggregate values (such as averages);
- *integrity control* which prevents data which violate the constraints declared in the database schema from being entered into the database;
- *concurrency control* which ensures that users simultaneously accessing a database do not interfere with one another. In fact, when more than one user accesses the same data, unpredictable results can occur.

Example 4.2 Let us assume that John and Jane have a joint savings account and both go to different tellers. The current balance is \$350. Jane wishes to add \$400 to the account. John wishes to withdraw \$50. Let us assume the following events happen in the order in which they are shown:
Jane's teller reads \$350,
John's teller reads \$350,
Jane's teller writes \$750,
John's teller writes \$300,

The account now reads \$300, and this certainly is not a correct way to allow more than one person to use the same account.

- *data recovery* which entails restoring the database to a consistent state after the occurrence and detection of a failure. A database may become inconsistent because of a *transaction failure*, a *system failure*, or a *media (disk) failure.*

Definition 4.2.1 A *transaction* is a sequential program with embedded database operations and the following properties, often called the ACID properties:

- *atomicity*: only committed transactions change the database, if a transaction aborts, the state of the database should remain unchanged as if no operations of the aborted transaction had occurred;
- *persistency*: the effects of a committed transaction are permanent and must survive system and media failures, i.e. commitment is an irrevocable act;
- *serializability*: when a transaction is executed concurrently with others, the final effect must be the same as a *serial* execution of transactions that execute to completion; i.e. the DBMS behaves as if it executes the transactions one at a time.

Definition 4.2.2 A *transaction failure* is an interruption of a transaction which does not damage the content of both the temporary memory (*buffers*) and the permanent memory.

A transaction can be interrupted because (a) the program has been coded in such a way that if certain conditions are detected then an abort must be issued, (b) because the DBMS detects a violation by the transaction of some integrity constraint or access right, or (c) because it was decided to terminate the transaction since it was involved in a deadlock detected by the DBMS. When a transaction aborts, its actions are undone automatically by the *recovery facility*, restoring the database to the same state it had at beginning of the transaction.

Definition 4.2.3 A *system failure* is an interruption (crash) of the system (either the DBMS or the computer) in which the contents of the temporary memory are lost, but the contents of the permanent memory remain intact.

Table	Type of Information
SYSTABLES	Information about the relational tables
SYSCOLUMNS	Information about the columns in tables and views
SYSVIEWS	Information about views
SYSINDEXES	Information about the indexes on tables
SYSKEYS	Information about the keys on tables

Figure 4.1 Examples of system catalog tables

When a system crash occurs, the DBMS is restarted (automatically or by an operator). The DBMS ensures that all transactions which were not completed at the time of the crash are undone, whereas all those which were completed have their effects reapplied to the database if necessary.

Definition 4.2.4 A *media failure*, or a *catastrophe*, is an interruption of the DBMS in which the contents of the permanent memory are lost.

When a media failure occurs, the recovery facility can use its historical data to reconstruct the current database contents starting from a prior version of the database.

Techniques used by DBMSs for concurrency control and data recovery will be considered later.

4.2.4 A User-Accessible System Catalog

The system *catalog* (*data dictionary* or *meta-data*, the 'data about data') is a special purpose database, maintained by the system, to store data that describe the structure of the objects in a database.

The catalog schema is designed by the DBMS vendor, and an instance of the catalog is created automatically whenever a new database is created. The catalog can be queried as any other databases. Examples of catalog tables for a relational DBMS are described in Figure 4.1.

4.2.5 Facilities for the Database Administrator

DBMSs provide important facilities for the data administrator; he needs tools to accomplish at least the following tasks:

- definition of the database schema;
- specification of integrity constraints;
- definition of external schemas for different applications;
- definition of data structures to improve the performance of the database operations;
- granting of data access authorization to the various users of the database;
- monitoring of DBMS performances and database tuning;
- restoring the database after a media failure and restructuring the database when the schema changes.

4.3 SQL: A Relational Database Language

SQL (*Structured Query Language*) is the most widely used relational database language. An initial version was proposed in 1975, the standard version is called SQL-92, and recently was completed the SQL:99 version for object databases. The purpose of the following sections is to introduce just some of its features, since a full treatment of the language is beyond the scopes of this report.

4.3.1 The Data Definition Sublanguage

A relation schema is specified using the CREATE TABLE statement of SQL. This statement has a rich syntax which we will not introduce here. As a bare minimum, CREATE TABLE specifies the typing constraint: the name of a relation and the names of the attributes with their associated types. However, the same statement can also specify primary and candidate keys, foreign key constraints, and other semantic constraints.

The Students relation is defined as follows:

```
CREATE TABLE Students (
    Name           CHAR(20) NOT NULL,
    StudentCode    CHAR(8) NOT NULL,
    City           CHAR(20),
    BirthYear      INTEGER NOT NULL,
PRIMARY KEY        (StudentCode),
UNIQUE             (Name, BirthYear)
CHECK              (BirthYear > 1900));
```

Null values are not allowed in keys. One additional feature to note is that a default value can be specified for an attribute. This value will be automatically assigned to the attribute of a tuple should the tuple be inserted without this attribute being given a specific value. Semantic constraints are specified using the CHECK clause.

A relation schema can be modified using the ALTER TABLE statement and deleted with the DROP TABLE statement.

In relational databases, it is common for tuples in one relation to reference tuples in the same or other relations to model associations. It is a violation of data integrity if the referenced tuple does not exist in the appropriate relation. For example, it makes no sense to have a ExamResults tuple with candidate 100 and not have the tuple with StudentCode = 100 in the relation Students. The requirement that the referenced tuple must exists is called *referential integrity*. One important type of referential integrity is the so-called *foreign key constraint*.

The following example shows how foreign key constraints are specified in SQL:

```
CREATE TABLE ExamResults (
    Subject        CHAR(20) NOT NULL,
    Candidate      CHAR(8) NOT NULL,
    Date           CHAR(8) NOT NULL,
    Grade          INTEGER NOT NULL,
PRIMARY KEY        (Subject, Candidate),
FOREIGN KEY        (Candidate)
    REFERENCES     Students
    ON DELETE NO ACTION);
```

The FOREIGN KEY clause has the option ON DELETE to specify what to do if a referenced tuple is deleted. NO ACTION means that any attempt to remove a Students tuple must be rejected outright if the student is referenced by a ExamResults tuple. The option ON DELETE CASCADE means that the referencing tuple is to be removed too. The option ON DELETE SET NULL means that the foreign key attributes in the references tuple must be set to NULL. Similar options are provided for the option ON UPDATE. NO ACTION is the default situation when ON DELETE or ON UPDATE is not specified.

More general remedial actions can be specified when a constraints is violated using the *trigger* mechanism: Whenever a specific *event* occurs, a specified *action* is executed.

Besides ordinary tables, also virtual tables (called *views*) can be defined with the CREATE VIEW statement. A view can be queried as an ordinary table, but its content does not physically exists in the database, instead, a definition of how to construct the view from ordinary database tables is given as a query with the CREATE VIEW statement and stored in the system catalog.

For example, the following view defines the students of Pisa:

```
CREATE VIEW PisaStudents AS
        SELECT  Name, StudentCode, BirthYear
        FROM    Students
        WHERE   City = 'Pisa';
```

Access Control

Since databases often contain sensitive information, a DBMS ensures that only those authenticated users who are authorized to access the database are allowed to and they are only allowed to access information that has been specifically made available to them.

SQL provide the GRANT and REVOKE statements to allow security to be set up on the tables in the database. When a user create a table he automatically becomes the owner of the table and receives full privileges for the table. To allow other users the access to the table, the owner must explicitly grant them the necessary privileges using the GRANT statements:

```
GRANT    { privilegeList | ALL PRIVILEGES } [(columnName [, columnName])]
ON       objectName
TO       { authorizationIdList | PUBLIC }
[ WITH GRANT OPTION ]
```

Privileges are the actions that a user is permitted to carry on a given base table or view (the objectName); examples are:

– SELECT: To retrieve data from a table.
– INSERT, MODIFY, DELETE: To insert, to modify or to delete rows.
– REFERENCES: To reference columns of a table in integrity constraints.

The INSERT, MODIFY, and REFERENCES privileges can be restricted to specific columns of a table. The WITH GRANT OPTION clause allows the users in the authorization list to pass the privileges that they have to others users.

Example 4.3 Granting and revoking privileges to users:

```
GRANT    ALL PRIVILEGES
ON       MyTable
TO       MyFriend WITH GRANT OPTION;
```

```
GRANT    SELECT, UPDATE(Grade)
ON       Exams
TO       Albano;

GRANT    SELECT
ON       Students
TO       PUBLIC;

REVOKE   SELECT
ON       Students
FROM     PUBLIC;
```

4.3.2 The Query Sublanguage

The SELECT statement is used to retrieve data from relations. Suppose you wanted to retrieve from the Students table the information on student named "Rossi". This is called making a query. To do it, you could issue the following statement:

```
SELECT   *
FROM     Students
WHERE    Name = 'Rossi';
```

SELECT is a keyword telling the database that this is a query. The asterisk means to retrieve all columns; alternatively, you could have listed the desired columns by name, separated by commas. The FROM Students clause identifies the table from which you want to retrieve the data.

WHERE Name = 'Rossi' is a predicate, and all rows that make the predicate TRUE are returned. This is an example of set-at-a-time operation. The predicate is optional, but in its absence the operation is performed on the entire table, so that, in this case, the entire table would have been retrieved. The semi-colon is the statement terminator.

The relationship between SQL and relation algebra is as follows:

Set union: $R \cup S$ is equivalent to

```
SELECT   *
FROM     R
  UNION
SELECT   *
FROM     S;
```

Set difference: $R - S$ is equivalent to

```
SELECT   *
FROM     R
   EXCEPT
SELECT   *
FROM     S;
```

Projection: $\Pi_{A_1, A_2, \ldots, A_m}(R)$ is equivalent to

```
SELECT   DISTINCT A1, A2, ..., Am
FROM     R;
```

Selection: $\sigma_{Condition}(R)$ is equivalent to

```
SELECT   *
FROM     R
WHERE    Condition;
```

Product: $R \times S$ is equivalent to

```
SELECT   *
FROM     R, S;
```

Join: $R \underset{R.A_i=S.A_j}{\bowtie} S$ is equivalent to

```
SELECT   *
FROM     R, S
WHERE    R.A_i = S.A_j;
```

Natural Join: $R \bowtie S$ is equivalent to

```
SELECT   *
FROM     R NATURAL JOIN S;
```

Nulls and Three-Valued Logic

With predicates, a three-valued logic is used. In SQL, the basic Boolean values of TRUE and FALSE are supplemented with another: NULL, also called UNKNOWN. This is because SQL acknowledges that data can be incomplete or inapplicable and that the truth value of a predicate may therefore not be knowable. Specifically, a column can contain a null, which means that there is no known applicable value. A comparison between two values using relational operators – for example, a = 5 – normally is either TRUE or FALSE. Whenever nulls are compared to other values, however, including other nulls, the boolean value is neither TRUE nor FALSE but itself NULL.

In most respects, NULL has the same effect as FALSE. The major exception is that, while NOT FALSE = TRUE, NOT NULL = NULL. In other words, if you know that an expression is FALSE, and you negate it, then you know that it is TRUE. If you do not know whether it is TRUE or FALSE, and you negate it, you still do not know. In certain cases, three-valued logic can create problems with your programming logic if you have not accounted for it. You can treat nulls specially in SQL with the IS NULL predicate.

Aggregation over Data

SQL provides five built-in functions, called *aggregate functions*, which operates on set of tuples. They are:

- COUNT([DISTINCT] Attr): Count the number of values in column Attr of the query result. The optional keyword DISTINCT indicates that each value should be counted only once, even if it occurs multiple times in different answer tuples. COUNT(*) counts the number of tuples of the query result.
- SUM([DISTINCT] Attr): Sum up the values in column Attr of the query result. DISTINCT indicates that each value should contribute to the sum only once, regardless of how often it occurs in column Attr.

– AVG([DISTINCT] Attr): Compute the average of the values in column Attr of the query result. Again DISTINCT means that each value should be used only once.
– MAX(Attr), MIN(Attr): Compute the maximum or the minimum value in the column Attr.

For example, the following query returns the number of students tuples:

```
SELECT    COUNT(*)
FROM      Students;
```

The following query returns the average birth year of students:

```
SELECT    AVG(BirthYear)
FROM      Students;
```

Note that it is not possible to mix an aggregate function and an attribute in this form of SELECT, as in

```
SELECT    Name, AVG(BirthYear)
FROM      Students;
```

To write such kind of SELECT the GROUP BY clause must be used:

```
SELECT      Name, AVG(BirthYear)
FROM        Students
GROUP BY    Name;
```

GROUP BY partition a set of tuples into groups whose membership is characterized by the fact that all of the tuples in a single group agree on the values in the specified set of attributes. The aggregate function then applies to the groups and produces a single value for each group. The result is a relation having two attributes, the student name and the average birth year. The important point is that each column in the SELECT clause either must be in the GROUP BY clause or must be the result of an aggregate function.

The HAVING clause is used in conjunction with GROUP BY: It is used to specify a condition that restricts which groups (specified in the GROUP BY clause) are to be considered for the final query result. Groups that do not satisfy the condition are removed before the aggregates are applied.

```
SELECT      Name, AVG(BirthYear)
FROM        Students
GROUP BY    Name
HAVING      COUNT(*) > 0;
```

The HAVING condition (unlike the WHERE condition) is applied to groups, not to individual tuples (Figure 4.2).

Finally, the order of tuples in the query result is generally unpredictable. If a particular ordering is desired, the ORDER BY clause can be used:

```
SELECT      Name, BirthYear
FROM        Students
ORDER BY    Name;
```

Ascending order is used by default, but descending order can also be specified:

```
SELECT      Name, BirthYear
FROM        Students
ORDER BY    DESC Name;
```

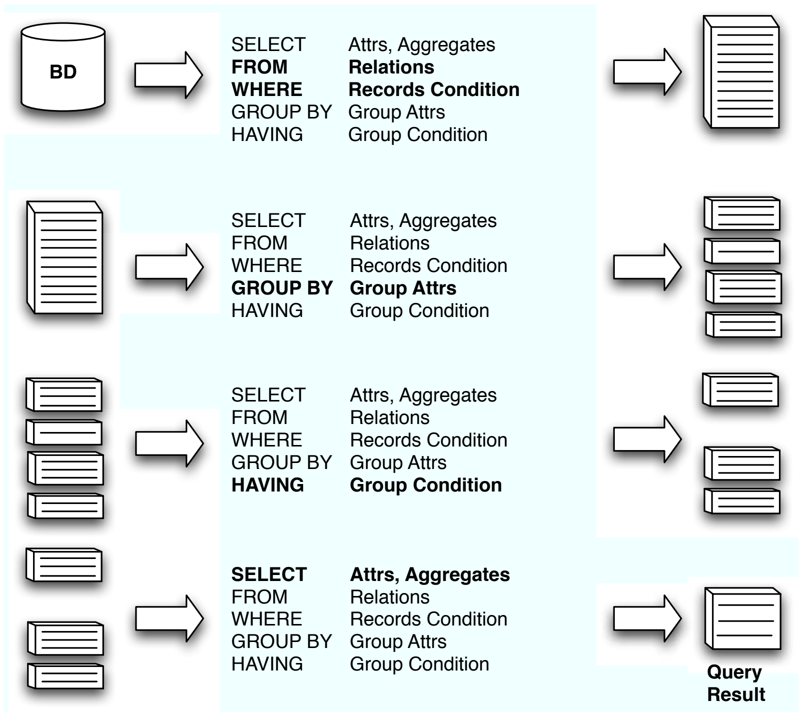


Figure 4.2 Query evaluation with GROUP BY

Nested Queries

Nested subqueries increase the expressive power of SQL, but are one of the most complex, expensive, and error-prone feature of SQL.

Consider the query *list the student code of the students who did not pass any exams*:

```
SELECT    StudentCode
FROM      Students
WHERE     StudentCode NOT IN (
          == Students who have passed an exam
          SELECT    Candidate
          FROM      ExamResults ) ;
```

Another useful operator is EXISTS to check if a nested subquery returns no answer. For example, here is another formulation of the above query:

```
SELECT    StudentCode
FROM      Students s
WHERE     NOT EXISTS (
          == All exams passed by a student
          SELECT    *
          FROM      ExamResults
          WHERE     Candidate = s.StudentCode) ;
```

4.3.3 Modifying Relation Instances

Relation instances are modified with the operators INSERT, UPDATE, and DELETE. INSERT places rows in a table, UPDATE changes the values they contain, and DELETE removes them.

For INSERT, you simply identify the table and its columns and list the values, as follows:

```
INSERT INTO   Students (Name, StudentCode, City, BirthYear)
              VALUES ('Rossi', '01234', 'Pisa', 1990);
```

This statement inserts a row with a value for every column but. If a value is specified for every column of the table, and the values are given in the same order as the columns in the table, the column list can be omitted. A SELECT statement can be used in place of the VALUES clause of the INSERT statement to retrieve data from elsewhere in the database.

UPDATE is similar to SELECT in that it takes a predicate and operates on all rows that make the predicate TRUE. For example:

```
UPDATE     Students
   SET     City = 'Florence'
   WHERE   Name = 'Rossi';
```

This sets to 'Florence' the city for the student named 'Rossi'. The SET clause of an UPDATE command can refer to current column values. "Current" in this case means the values in the column before any changes were made by this statement.

DELETE is quite similar to UPDATE. The following statement deletes all rows for students from 'Pisa':

```
DELETE  FROM     Students
        WHERE    City = 'Pisa';
```

You can only delete entire rows not individual values. To do the latter, use UPDATE to set the values to null. Be careful with DELETE that you do not omit the predicate; this empties the table.

4.3.4 Executing SQL Statements within Application Programs

In the previous sections, we discussed SQL as an interactive language: you type in a query and the see the results on your screen. In order to write application programs, SQL statements must included in some conventional language, such as C, COBOL, Java or Visual Basic. The main problem to solve is the fact that a mismatch exists between the data structures of the programming language, which operates on records, and those of SQL, which operates on relations, i.e. sets of records. Therefore, a mechanism is required to supply the result of an SQL expression to the programming language, one element at a time.

The standard solution is to declare a *cursor* for each query to be evaluated: a cursor is a "logical pointer" that ranges over all the tuples of the result of an SQL statement. To evaluate an SQL statement, the cursor is opened, and then, using a *fetch* operator, the "next" tuple of the result is retrieved, the components of each tuple are copied into a list of variables of the host language program, and the cursor is advanced to point to the next tuple. An exception is raised when a fetch is attempted beyond the last tuple of the result.

SQL statements can be included in an application program in three different ways:

1. *Extended language.* The language is a superset of SQL, supplementing it with standard programming-language features that include the following: block (modular) structure, flow-control statements and loops, variables,

constants, and types, structured data, and customized error handling. The language compiler can control completely that SQL statements are well formed. A notably example is Oracle PL/SQL.
Let us illustrate the approach by showing two programs which print the name and birth year of the students of Pisa. The first example (Figure 4.3) use the standard cursor, while the second example use a special construct FOR with an implicit cursor (Figure 4.4).

```
PROCEDURE Example1 (Cty IN Students.City%TYPE) IS
DECLARE
  CURSOR c IS
    SELECT  Name, BirthYear
    FROM    Students WHERE City = Cty;
  Stud_Rec c%ROWTYPE;
BEGIN
– retrieve a set of records
OPEN c
  LOOP
    FETCH   c INTO Stud_Rec;
    EXIT    WHEN c%NOTFOUND;
    PRINT   ... Stud_Rec.Name ... Stud_Rec.BirthYear ...
  END LOOP;
CLOSE c – cursor is released
```

Figure 4.3 A PL/SQL example with cursor

```
PROCEDURE Example2 (Cty IN Students.City%TYPE) IS
BEGIN
  FOR Stud_Rec IN (
    SELECT  Name, BirthYear
    FROM    Students WHERE City = Cty)
  LOOP
    PRINT   ... Stud_Rec.Name ... Stud_Rec.BirthYear ...
  END LOOP; – cursor is released
END
```

Figure 4.4 A PL/SQL example with implicit cursor

2. *Application programming interface (API).* Rather than design a new compiler, a standard programming language is used with a library of functions (API) which accept string SQL as parameter. Since SQL statements are passed to a function as strings, they cannot be controlled statically by the compiler, but are controlled dynamically by the DBMS. Microsoft's ODBC is the C/C++ standard API on Windows while Sun's JDBC is the Java equivalent. The API are DBMS-neutral and a *driver* traps the calls and translates them into DBMS-specific code.
   Let us illustrate the approach by showing a Java program which print the name and birth year of the students of Pisa using the JDBC API (Figure 4.5).
3. *Embedded SQL.* SQL statements can be used within a host language pro-

```
class PrintStudentsName{
public static void main(String argv[]){
Class.forName("DBMS driver");
Connection con = // connect
        DriverManager.getConnection("url", "login", "psw");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT   Name
                 FROM     Students
                 WHERE    City = "' + argv[0] + " "';
ResultSet iter = stmt.executeQuery(query);
System.out.println("Names retrieved:");
try      { // to handle exceptions
          // loop through result tuples
          while (iter.next()) {
               String name = iter.getString("Name");
               int year = iter.getInt("BirthYear");
               System.out.println(" Name: " + name + "; BirthYear: " + year);
          }
} catch(SQLException ex) {
System.out.println(ex.getMessage() + ex.getSQLState() + ex.getErrorCode());
}
stmt.close(); con.close();
}}
```

Figure 4.5 An example of API

gram. Before the program can be compiled by the host language compiler, the SQL statements must be processed by a pre-compiler, which check SQL syntax, the number and types of arguments and results, and replace them into calls to a library of functions. At runtime these functions communicate with the DBMS.
Let us illustrate the approach by showing a C program which prints the name and birth year of the students of Pisa (Figure 4.6).
Figure 4.7 shows the same example in SQLJ, is a dialect of embedded SQL that can be included in Java programs. The pre-compiler replace SQLJ constructs by call to a library which accesses a database using calls to a JDBC driver.
The statement #SQL iterator GetInfoStIte … in the figure tells the pre-compiler to generate a class GetInfoStIte which implements an iterator with the next() method. The class GetInfoStIte is used to store result sets in which each row has two columns: a string and an integer. The declaration gives a Java name to these columns, Name and Year, and implicitly defines the column accessor methods, Name() and Year(), which can be used to return data stored in the corresponding columns.

4.3.5 Exercises

1. Give a relational schema in SQL for the following databases:
   (a) Your solution to Exercise 3.3.1(1).
   (b) Your solution to Exercise 3.3.1(2).
2. Give a relational schema in SQL for your solution to Exercise 3.3.1(3), and write the following queries:
   (a) Retrieve the birth-date and name of the female employees.

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_BirthYear;
EXEC SQL END DECLARE SECTION
short c_City = "Pisa";
EXEC SQL DECLARE sinfo CURSOR FOR
        SELECT      S.name, S.BirthYear
        FROM        Students S
        WHERE       S.City = :c_City
        ORDER BY    S.name;
do {
        EXEC SQL FETCH sinfo INTO :c_sname, :c_BirthYear;
        printf("Name:%s; BirthYear: %s ", c_sname, c_BirthYear);
} while (SQLSTATE != 02000);
EXEC SQL CLOSE sinfo;
```

Figure 4.6 An example of embedded SQL

```
public static void main(String argv[]){
Oracle.connect("jdbc:oracle:oci8:@", "scott", "tiger");

#SQL iterator GetInfoStIter(String Name, int Year);
GetInfoStIter iter;

#SQL iter = {
        SELECT   Name, BirthYear AS Year
        FROM     Students
        WHERE    City =:(argv[0]) };

System.out.println("Students retrieved");
        while (iter.next()) {
            String name = iter.Name();
            int year = iter.Year();
            System.out.println(" Name = " + name + " Year = " + year);
        }

iter.close();
Oracle.close(); }
```

Figure 4.7 An example of SQLJ

(b) For each employee, retrieve the employee name and the name of the department where he works.
(c) Retrieve the distinct salary of every employee.
(d) Retrieve the names and the ages of female employees older than their supervisor.
(e) Retrieve the names of all employees who do not have supervisors.
(f) Retrieve the name and address of all employees who work for the "Research" department.
(g) For every project located in "Pisa", list the project number, the controlling department number, and the departament manager's name, address, and birthdate.
(h) Make a list of all projects numbers for projects that involve an employee whose last name is Smith, either as a worker or as a manager of the department that controls the project.
(i) Retrieve the names of employees who have no dependents.
(j) List the names of supervisors who have at least one dependent.
(k) For each employee, retrieve the employee's name and the name of his or her immediate supervisor.
(l) Retrieve the name of each employee who has a dependent with the same first name and sex as the employee.
(m) Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by name.
(n) Find the sum of the salaries of all the employees of the Research department, as well as the the maximum salary, the minimum salary, and the average salary in this department.
(o) For each department, retrieve the department number, the number of employees in the department, and their average salary.
(p) For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.
(q) For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.
(r) For each department having more than five employees, retrieve the department number, the number of employees making more than 40.000.
(s) Retrieve the name of each employee who has all dependents with the same sex as the employee.
(t) Retrieve the name of each employee who has all dependents with the same sex.
(u) Retrieve the names of the employees who work only to projects for 20 percent-time.
(v) Retrieve the name of each employee who work only on projects controlled by department number 5.
(w) Retrieve the name of each employee who work only on projects controlled by the same department.
(x) Retrieve the name of each employee who work on all the projects (and only those) to which the employee 100 participates.