Databases Essentials

Antonio Albano

Department of Computer Science
University of Pisa

February 13, 2015

CONTENTS

Chapter 1

PRELIMINARIES

1.1 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 on-going 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

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.

maintain the correspondence between the database and the real-world situation it is modeling as events occur in the real world.

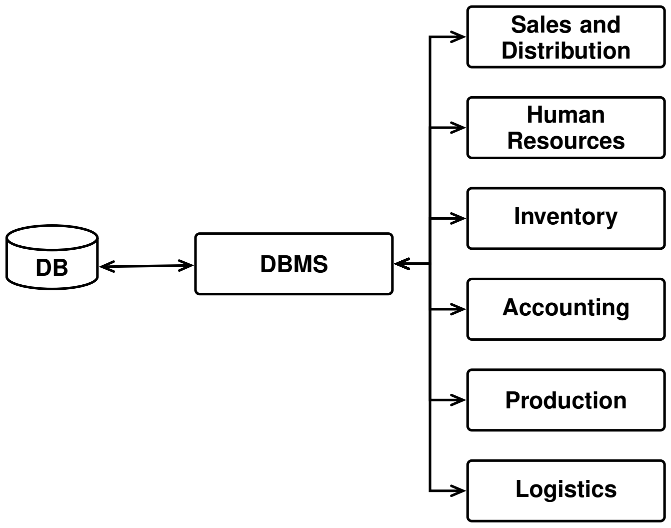


Figure 1.1: Transaction processing system

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 *reports*, *multidimensional data analysis*, and *exploratory data analysis* 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 that every computer scientist and information systems professional should be familiar with. The emphasis is on the concepts underlying database languages, systems and design. The discussion is organized into three main parts: *Database from the designer perspective*, *DBMS from a user perspective*, and *DBMS from a system perspective*.

Part I, database from the designer perspective, 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 report. 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.

Part II, 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.

Part III, DBMS from a system perspective, presents DBMS architectures and the features of the basic functional components: the *memory manager*, which manages the allocation of space on disk storage and the data structures to store and retrieve data efficiently; the *query processor*, which attempts to transform a user's request into an equivalent but more efficient form, thus finding a good strategy for executing the query; the *recovery manager*, which ensures that the database remains in a correct state despite hardware and software failures; the *concurrency controller*, which ensures that concurrent interactions with the database can proceed without conflicting with one another.

Part I

Database: The Designer Perspective

Chapter 2

INFORMATION MODELING

2.1 Introduction

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 2.1**

A *symbolic model* is a subjective formal representation of ideas and knowledge about some aspects of the real world (*domain of discourse*), designed to serve an explicit purpose.

Three fundamental aspects of this definition must be underlined:

1. A model is a representation of only some aspects of the real world to serve a purpose of some kind.
2. The representation is given by a formal language.
3. The model is the result of an interpretation process which depends on the knowledge possessed by the designer about the portion of the real world modeled.

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.

2.2 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 2.2**

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 2.3**

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 2.4**

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.

2.3 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 2.5**

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.

2.4 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.

2.4.1 Object

An object is the computer representation of certain facts about an entity of the observed 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.

2.4.2 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*), but in the following they will not be considered because are beyond the scope of this notes.

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

The following example shows a graphic representation of types.[1]

Example 2.1
Attributes are represented by the pair (Name : Type). Attributes can be *multivalued* (have a type seq T), structured (have a type $[A_1 : T_1, \ldots, A_n : T_n]$); they can be optional, meaning that the value can be left unspecified.

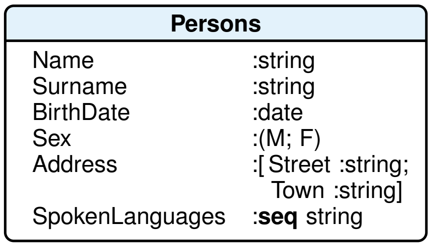

Persons
Name :string
Surname :string
BirthDate :date
Sex :(M; F)
Address :[Street :string; Town :string]
SpokenLanguages :**seq** string

Figure 2.1: A graphic representation of an object type

2.4.3 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. 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).

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.
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.

To simplify the graphical representation of a database conceptual schema we will not use a new graphical notation for classes, but we will assume that

– the only object types modeled are those which are also elements of a class, and
– the object type name is also the class name.

Therefore, in the following, a definition such as that in Figure 2.1 will be used for a the class Persons with the element structure the one shown.

2.4.4 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 examples.

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), or as a class (Figure 2.3c). 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 2.3d).

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.

2.4.5 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

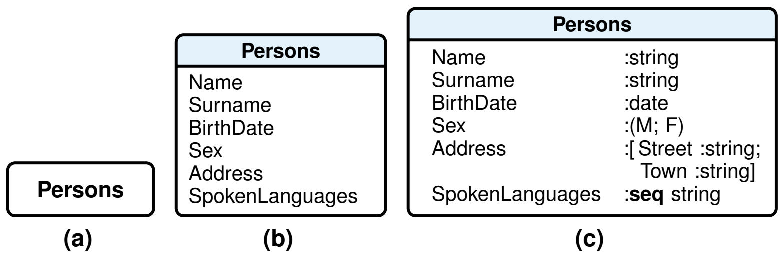


Figure 2.2: Graphic representation of classes

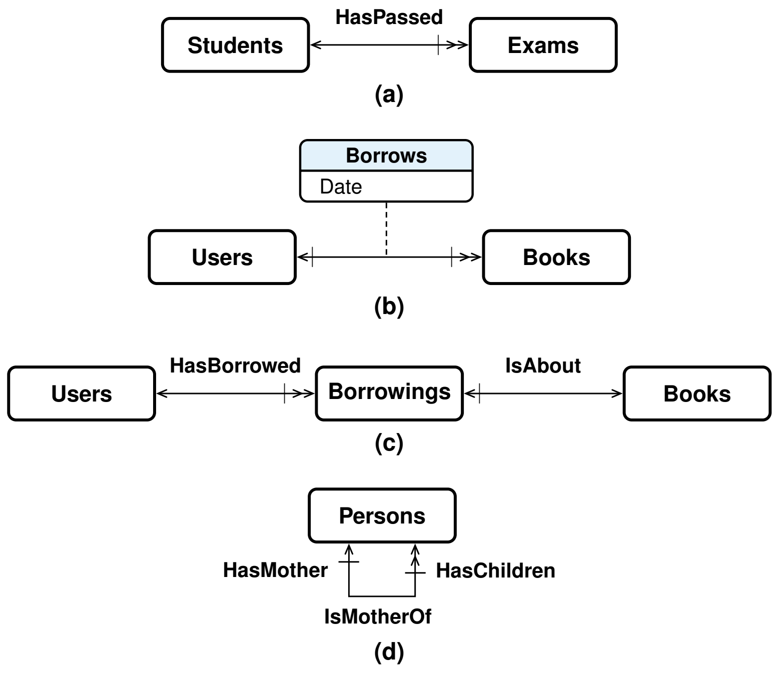


Figure 2.3: Graphic representation of relationships

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*).

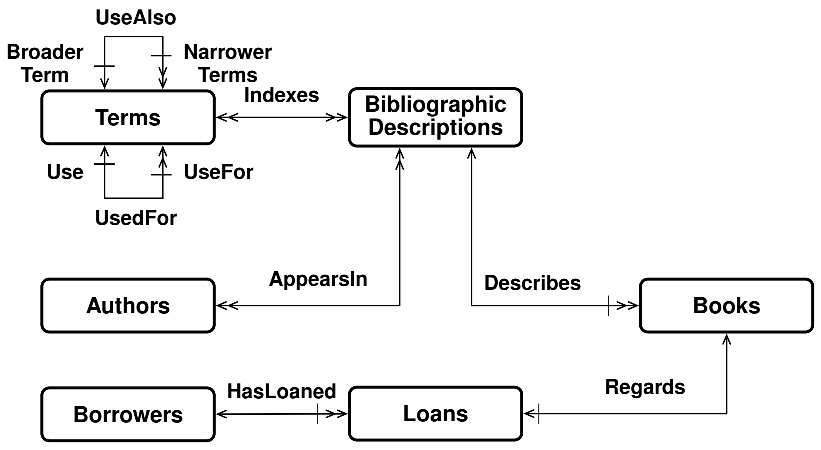


Figure 2.4: A schema for a library

2.4.6 Class Hierarchies

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

- The type of the elements of C_1 is a subtype of the type of the elements of C_2 (*intensional constraint*).
- 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 *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* Students is a *subclass* of the *class* Persons) (Figure 2.5).

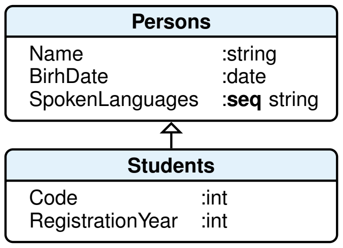


Figure 2.5: A subclass example

Note that in object-oriented programming languages, such as Java or C++, an object type is called *class* and so the graphical notation shown in the figure is used to model an object subtype defined by inheritance, called a *subclass* (*intensional* *aspect*). Instead the notion of *class* of the database ODM has also an *extensional aspect* and denotes a modifiable sequence of the elements of an object type currently in the database, while a *subclass* is a subset of a class.

A subclass can be defined from a single superclass (*simple inheritance*) or from several superclasses (*multiple inheritance*) (Figure 2.6).

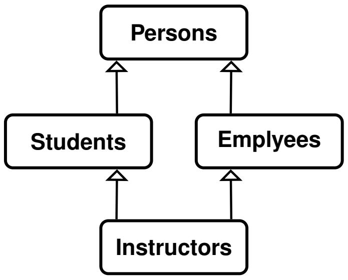


Figure 2.6: Subclasses with multiple inheritance

Moreover on subclasses of the same superclass can be defined two kinds of constraints: *overlap* and *covering*.

A *no overlap* (*disjoint*) *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).

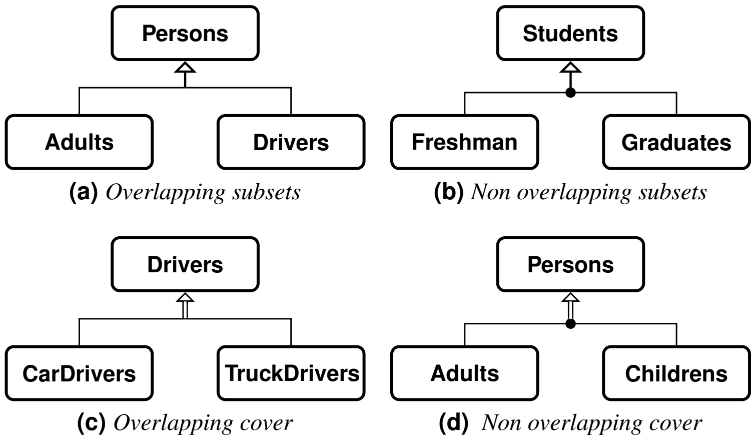


Figure 2.7: Kinds of subclasses

A refined library schema using subclasses is shown in Figure 2.8 with the class attributes. Elements of classes can be constrained to be uniquely identified by certain attributes, called *keys*. Attributes of a key are marked with a ≪K≫ or are underlined. Attributes of different keys are marked with a ≪Ki≫ or are underlined differently.

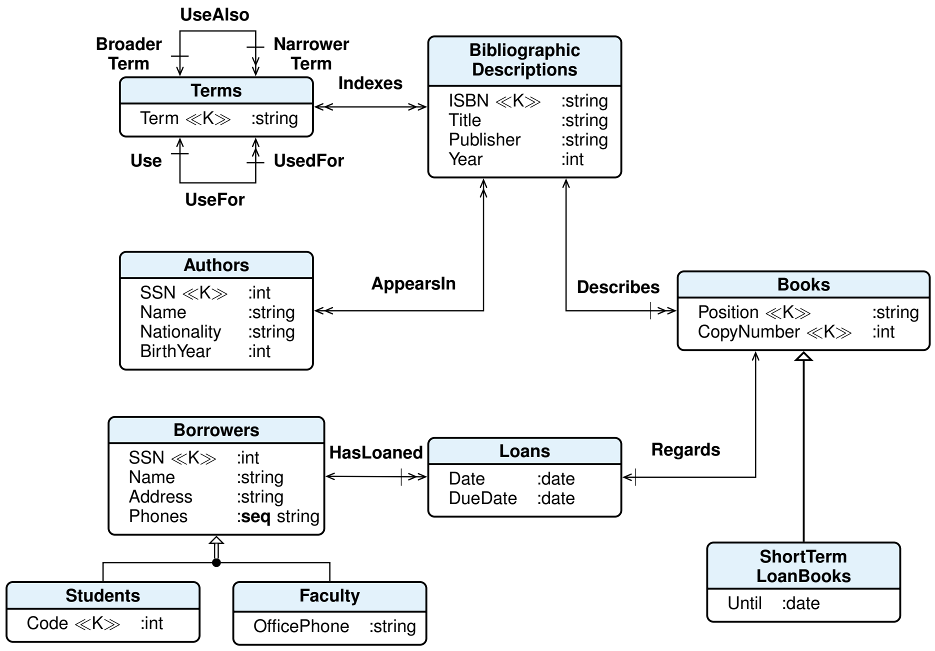


Figure 2.8: A refined schema for a library with class attributes

Usually there are two ways to populate 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 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 2.8 is shown in Figure 2.9 using the ER notation.

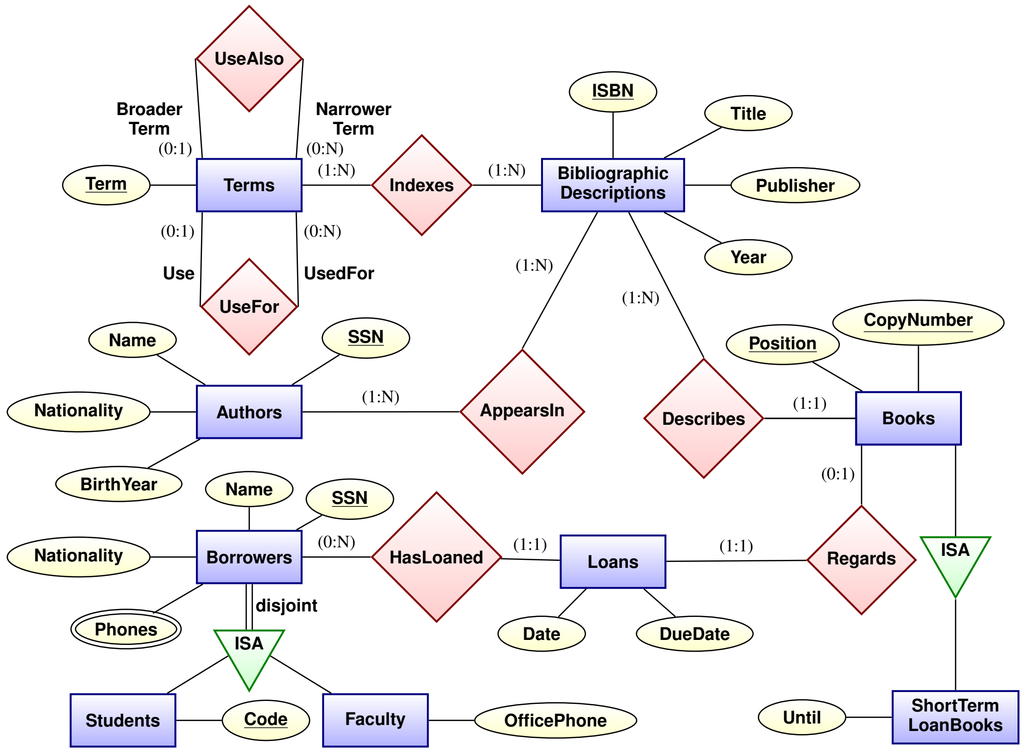


Figure 2.9: The library schema with an Entity-Relationship diagram

2.5 Exercises

1. 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.
2. 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.
3. 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.

4. 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.
5. Let us assume that a company uses the following worksheet to store data about its computers.

Inventory

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.

6. We would like to design a database to maintain information for an administrator of condominium (i.e. a block of flats). Each condominium has a code (the key), an address and the number of the checking account where should be payed the supported expenses. A condominium is made of flats, and we are interested in the flat number, the number rooms, the surface, the state (free or occupied). The flats can be rented, and we are interested in tenants name, the social-security numbers (the key), the telephones (more than one) and the balance, that is the amount that tenant must pay for running expenses. Some rented flats can have been noticed, and in this case we are interested in the date of the notice. A flat can have several owners, and an owner can possess several flats. Of every owner we are interested in the name, the social-security numbers (the key), the address, the telephones (more than one) and the balance, that is the sum that the owner must pay for supported expenses. Maintenance expenses for the condominium are described by the code of identification, the nature (light, cleaning, elevator, etc), the date and the amount. (Optional) The expenses are classified as extraordinary, to be paid by owners, or as ordinary to be paid by tenants. Ordinary expenses must be paid in one installment,

while extraordinary expenses can be paid in more installments, and for each of them it is necessary to remember the date and the amount.

7. 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	**DATE**: 07/02/12
LOAN DATE OF EXPIRY: 01/01/20	
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, 1 Pisa

Payment No	Payment Date of Expire	Amount	Payment Date
1	01/07/10	3 500	29/08/10
2	01/01/11	3 500	30/12/10
3	01/07/11	3 500	30/08/11
4	01/01/12	3 500	30/12/11

Figure 2.10: A loan repor

8. 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

THE RELATIONAL DATA MODEL

3.1 Introduction

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

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

■ Definition 3.2

A tuple $(A_1 := V_1, \dots, A_n := V_n)$ of type $T = (A_1 : T_1, \dots, A_n : T_n)$ 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 a 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. Two relations of the same type are equal if have the same sets of tuples.

The extension of a relational database is a collection of the extensions of its relation schemas.

■ Definition 3.3

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 $R : \{(A_1 : T_1, \dots, A_n : T_n)\}$, we will use as standard notation $R(A_1 : T_1, \dots, A_n : T_n)$ to denote a relation with name R and type $\{(A_1 : T_1, \dots, A_n : T_n)\}$, which will be further abbreviated to $R(A_1, \dots, A_n)$ 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 (the primary key attributes are underlined):

Authors(SSN, Name, Nationality, BirthYear)
BibliographicDescriptions(ISBN, Title, Publisher, Year)
Books(ISBN, Position, CopyNumber)
AuthorsAppearsIn(SSN, ISBN)

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

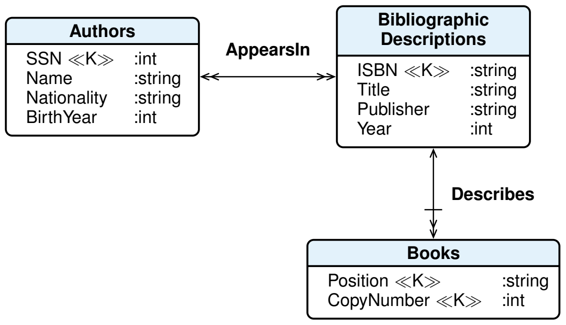
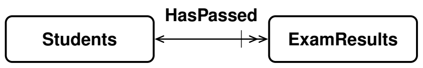
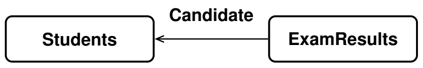


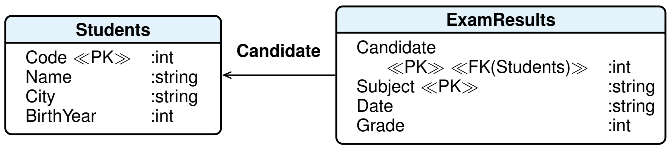
Figure 3.1: A database schema using the ODM



(a) *ODM schema*



(b) *Relational schema*



(c) *Relational schema with attributes*

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 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: $\boldsymbol{\rho}_{A_1 \leftarrow B_1, A_2 \leftarrow B_2, \ldots, A_m \leftarrow B_m}\ (E)$

$A_1, A_2, \ldots, A_m$ are attributes of E, and $B_1, B_2, \ldots, B_m$ are not attributes of E. The result is a relation with type $\{(B_1 : T_1, B_2 : T_2, \ldots, B_m : T_m)\}$ whose tuples are those of E with the attributes A_i renamed to B_i.

Project: $\boldsymbol{\pi}_{A_1, A_2, \ldots, A_m}\ (E)$

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

$A_1, A_2, \ldots, A_m$ are attributes of E. 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 E with only the attributes $A_1, A_2, \ldots, A_m$. Since the result is a set, any duplicate tuples are eliminated.

Select: $\boldsymbol{\sigma}_{\text{Condition}}\ (E)$

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

Set union: $E_1 \cup E_2$

E_1 and E_2 are relations of the same type $\{T\}$. The result is a relation with type $\{T\}$ whose tuples are those which are in E_1 or E_2 or both.

Set difference: $E_1 - E_2$

E_1 and E_2 are relations of the same type $\{T\}$. The result is a relation with type $\{T\}$ whose tuples are those which are in E_1 but not in E_2.

Product: $E_1 \times E_2$

E_1 and E_2 are relations of type $\{(A_1 : T_1, \ldots, A_n : T_n)\}$ and $\{(A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})\}$ *with disjoint set of attributes*. The result is a relation of type $\{(A_1 : T_1, \ldots, A_n : T_n, A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})\}$ whose tuples are all possible tuples whose first n components form a tuple in E_1 and whose last m components form a tuple in E_2.

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.

$$\boldsymbol{\pi}_{\text{Name, StudentCode}}\ (\boldsymbol{\sigma}_{\text{City} = \text{'Pisa'}}\ (\text{Students}))$$

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	Student	City	Birth	Subject	Candidate	Date	Grade
	Code		Year				
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.

$R := \boldsymbol{\sigma}_{\text{StudentCode = Candidate}}(T)$

Name	Student	City	Birth	Subject	Candidate	Date	Grade
	Code	City	Year	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:

$\boldsymbol{\pi}_{\text{Name, Date}}(\boldsymbol{\sigma}_{\text{Subject = 'DA'} \wedge \text{Grade = 30}}(R))$

The same result might have been obtained with the expression

$$\boldsymbol{\pi}_{\text{Name, Date}}(\boldsymbol{\sigma}_{\text{Subject = 'DA'} \wedge \text{Grade = 30} \wedge \text{StudentCode = Candidate}}(\text{Students} \times \text{ExamResults}))$$

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 six basic operators above are *intersect*, *join*, *natural join* and *division*.

Set intersection: $E_1 \cap E_2$

E_1 and E_2 are relations of the same type $\{T\}$. The result is a relation with type $\{T\}$ whose tuples are those which are both in E_1 and in E_2.

Join: $E_1 \underset{A_i\,=\,A_j}{\bowtie} E_2$

E_1 and E_2 are relations of type $\{(A_1 : T_1, \ldots, A_n : T_n)\}$ and $\{(A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})\}$ *with disjoint set of attributes*, A_i an attribute of E_1 and A_j an attribute of E_2. The join $E_1 \underset{A_i\,=\,A_j}{\bowtie} E_2$ is equivalent to

$\boldsymbol{\sigma}_{A_i\,=\,A_j}\,(E_1 \times E_2)$.

Natural Join: $(E_1 \bowtie E_2)$

The natural join is only applicable when both E_1 and E_2 have attributes with the same name. Let $A_1, \ldots, A_n$ be the common attributes of E_1 and E_2, Y be only attributes of E_1 and Z be only attributes of E_2. The natural join can be defined as follows.

1. Let us change the types of E_1 and E_2 in order to make different the common attributes:

   $E'_1 = \boldsymbol{\rho}_{A_1 \leftarrow E_1A_1, \ldots, A_n \leftarrow E_1A_n}(E_1)$ and

   $E'_2 = \boldsymbol{\rho}_{A_1 \leftarrow E_2A_1, \ldots, A_n \leftarrow E_2A_n}(E_2)$.

2. Let $T = \boldsymbol{\sigma}_{E_1A_1\,=\,E_2A_1\,\wedge\,\ldots\,\wedge\,E_1A_n\,=\,E_2A_n}\,(E'_1 \times E'_2)$.

3. The natural join of E_1 and E_2 is the relation

   $\boldsymbol{\rho}_{E_1A_1 \leftarrow A_1, \ldots, E_1A_n \leftarrow A_n}\,(\boldsymbol{\pi}_{E_1A_1, \ldots, E_1A_n, YZ}\,(T))$.

   Note that

– if E_1 and E_2 have not common attributes, $E_1 \bowtie E_2 \equiv E_1 \times E_2$;
– if E_1 and E_2 have the same type, $E_1 \bowtie E_2 \equiv E_1 \cap E_2$.

Division: $E_1 \div E_2$

Let XY be the attributes of E_1 and Y be the attributes of E_2. Then $W = E_1 \div E_2$ is a relation with attributes X such that for every tuple $t \in E_2$, there is a tuple $w \in W$ such that $w \circ s \in E_1$.

The division operation is expressible in relational algebra as follows:

$$E_1 \div E_2 \equiv \boldsymbol{\pi}_X(E_1) - \text{“the tuples in } \boldsymbol{\pi}_X(E_1) \text{ that are not in } E_1 \div E_2\text{”}$$
$$E_1 \div E_2 \equiv \boldsymbol{\pi}_X(E_1) - \boldsymbol{\pi}_X((\boldsymbol{\pi}_X(E_1) \times E_2) - E_1)$$

Let W and S be two relations *with disjoint set of attributes* and R be a relation with the union of their attributes. The product and division operators satisfy the following properties:

$$(W \times S) \div S = W \quad \text{and} \quad (R \div S) \times S \subseteq R$$

Let us see an example of query which requires the division operator. Given Enrolls(StudentCode, Course) and Teaches(Teacher, Course), find the codes of students who take every course taught by 'Tao'. The desired answer is given by the expression:

$$\text{Enrolls} \div \pi_{\text{Course}}(\sigma_{\text{Teacher = 'Tao'}}(\text{Teaches}))$$

Two useful extended relational algebra operations

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

where each of $e_1, \ldots, e_n$ is an arithmetic expression involving constants and attributes in the type of E, and $\mathsf{Ide}_1, \ldots, \mathsf{Ide}_n$ is a set of different attributes. The result is a relation with type $\{(\mathsf{Ide}_1 : T_{e_1}, \ldots, \mathsf{Ide}_n : T_{e_n})\}$

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

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

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

where $A_1, \ldots, A_n$ is a list of attributes of E on which to group; each f_i is an *aggregate function* applied to attributes of E, which takes a collection of values and return a single value as result. Common aggregate functions include *min*, *max*, *count*, *sum*, and *avg*. For example, the aggregate function *sum* takes a collection of values and returns the sum of the values. Thus, the function *sum* applied on the collection $\{1, 1, 4, 4\}$ returns the value 10.

The result is a relation with type $\{(A_1 : T_1, \ldots, A_n : T_n, f_1 : T_{n+1}, \ldots, f_m : T_{n+m})\}$

The meaning of the operation is as follows (Figure 3.3):

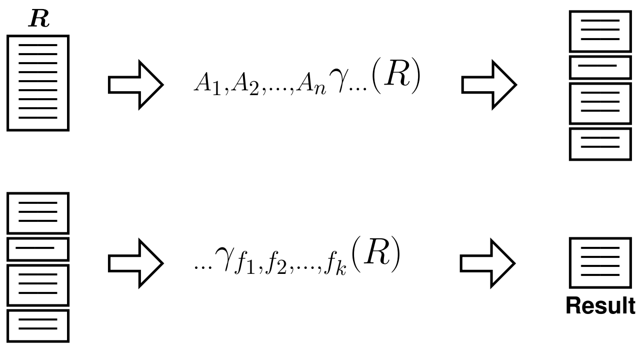

R
$_{A_1,A_2,\ldots,A_n}\gamma_{\ldots}(R)$
$\ldots\gamma_{f_1,f_2,\ldots,f_k}(R)$
Result

Figure 3.3: Grouping evaluation

1. The tuples of R 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

$$(A_1 := a_1, \ldots, A_n := a_n, f_1 := v_1, \ldots, f_k := v_k)$$

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.

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:

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

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

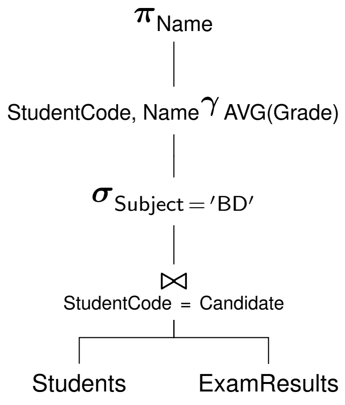
$$_{A_1}\gamma_{\mathsf{max(A_2)}\ \textbf{AS}\ \mathsf{M,\ sum(A_3)}\ \textbf{AS}\ \mathsf{S}}\ (R)$$

Logical Query Plan

To make a relational algebra expression more readable, it is usually represented as an expression tree of relational algebra operators, called a *logical query plan* or a *query tree*. For example, the expression

$$\pi_{\mathsf{Name}}(\ _{\mathsf{StudentCode,\ Name}}\gamma_{\mathsf{AVG(Grade)}}\ (\ \sigma_{\mathsf{Subject\ =\ 'BD'}}\ (\ \mathsf{Students}\ \underset{\mathsf{StudentCode\ =\ Candidate}}{\bowtie}\ \mathsf{ExamResults})))$$

is represented as the logical query plan in Figure 3.4.



π_{Name}
$_{\mathsf{StudentCode,\ Name}}\gamma_{\mathsf{AVG(Grade)}}$
$\sigma_{\mathsf{Subject = 'BD'}}$
$\underset{\mathsf{StudentCode\ =\ Candidate}}{\bowtie}$
Students
ExamResults

Figure 3.4: Logical query plan

3.3 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. *Cascading of selections*

$$\boldsymbol{\sigma}_{\psi_X}\ (\boldsymbol{\sigma}_{\psi_Y}(E)) = \boldsymbol{\sigma}_{\psi_X\ \wedge\ \psi_Y}(E)$$

2. *Commutativity of selection and projection*

$$\boldsymbol{\pi}_{\mathsf{Y}}(\boldsymbol{\sigma}_{\psi_X}(E)) = \boldsymbol{\sigma}_{\psi_X}(\boldsymbol{\pi}_{\mathsf{Y}}(E))$$

if $X \subseteq Y$, otherwise

$$\boldsymbol{\pi}_{\mathsf{Y}}(\boldsymbol{\sigma}_{\psi_X}(E)) = \boldsymbol{\pi}_{\mathsf{Y}}(\boldsymbol{\sigma}_{\psi_X}(\boldsymbol{\pi}_{\mathsf{XY}}(E)))$$

3. *Commutativity of selection and join*

$$\boldsymbol{\sigma}_{\psi_X}(E_1 \bowtie E_2) = \boldsymbol{\sigma}_{\psi_X}(E_1) \bowtie E_2$$

if X are attributes of E_1.

$$\boldsymbol{\sigma}_{\psi_X \wedge \psi_Y}(E_1 \bowtie E_2) = \boldsymbol{\sigma}_{\psi_X}(E_1) \bowtie \boldsymbol{\sigma}_{\psi_Y}(E_2)$$

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

$$\boldsymbol{\sigma}_{\psi_X \wedge \psi_Y \wedge \psi_Z}(E_1 \bowtie E_2) = \boldsymbol{\sigma}_{\psi_Z}(\boldsymbol{\sigma}_{\psi_X}(E_1) \bowtie \boldsymbol{\sigma}_{\psi_Y}(E_2))$$

if X are attributes of E_1, Y are attributes of E_2 and Z are attributes of both E_1 and E_2.

4. *Cascading of projections*

$$\boldsymbol{\pi}_Z(\boldsymbol{\pi}_Y(E)) = \boldsymbol{\pi}_Z(E)$$

if $Z \subseteq Y$.

5. *Commutativity of projection and join*

$$\boldsymbol{\pi}_{XY}(E_1 \bowtie E_2) = \boldsymbol{\pi}_X(E_1) \bowtie \boldsymbol{\pi}_Y(E_2)$$

where X are attributes of E_1, Y are attributes of E_2 and the join condition involves only attributes in XY.
If the join condition involves attributes not in XY, then

$$\boldsymbol{\pi}_{XY}(E_1 \bowtie E_2) = \boldsymbol{\pi}_{XY}(\boldsymbol{\pi}_{XX_{E_1}}(E_1) \bowtie \boldsymbol{\pi}_{YX_{E_2}}(E_2))$$

where X_{E_1} are attributes of E_1 that are involved in the join condition, but are not in XY, and X_{E_2} are attributes of E_2 that are involved in the join condition, but are not in XY.

6. *Commutativity of join*

$$E_1 \bowtie E_2 = E_2 \bowtie E_1$$

7. *Associativity of join*

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

8. *Commutativity of set union and intersection*

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$
$$E_1 - E_2 \neq E_2 - E_1$$

9. *Associativity of set union and intersection*

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

10. *Selection distributes over set operations*

$$\boldsymbol{\sigma}_{\phi_X}(E_1 - E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) - \boldsymbol{\sigma}_{\phi_X}(E_2)$$

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

$$\boldsymbol{\sigma}_{\phi_X}(E_1 - E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) - E_2$$

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

11. *Projection distributes over set operations*

$$\boldsymbol{\pi}_X(E_1 \cup E_2) = \boldsymbol{\pi}_X(E_1) \cup \boldsymbol{\pi}_X(E_2)$$

12. *Selection distributes over grouping*

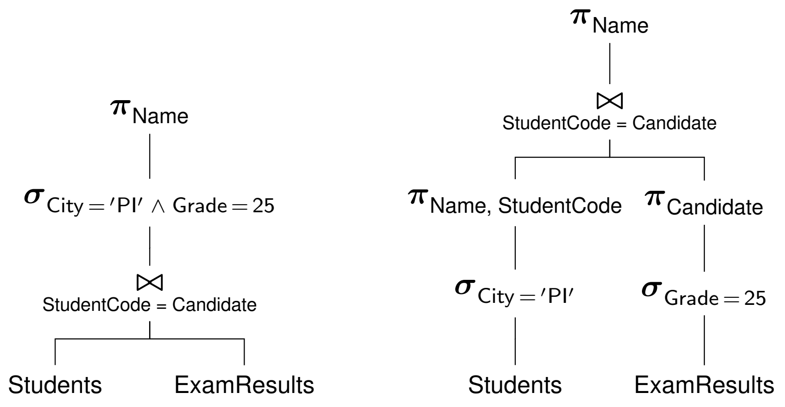
$$\sigma_X({}_Y\gamma_F(E)) = {}_Y\gamma_F(\sigma_X(E))$$

if X uses only attributes from Y.

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

$$\pi_{\text{Name}}(\sigma_\phi\,(\text{Students} \underset{\text{StudentCode = Candidate}}{\bowtie} \text{ExamResults}))$$

where $\phi = (\text{City = 'Pisa'} \wedge \text{Grade = 30})$, can be represented as the initial query tree in Figure 3.5a, and then it can can be represented also with the transformed query tree, taking into account the previous equivalence rules.



π_{Name}
$\sigma_{\text{City='PI'} \wedge \text{Grade}=25}$
$\bowtie$ StudentCode = Candidate
Students
ExamResults
π_{Name}
$\bowtie$ StudentCode = Candidate
$\pi_{\text{Name, StudentCode}}$
$\pi_{\text{Candidate}}$
$\sigma_{\text{City='PI'}}$
$\sigma_{\text{Grade}=25}$
Students
ExamResults

(a) *Initial query tree* **(b)** *Query tree after transformations*

Figure 3.5: Expression trees

3.3.1 Exercises

Consider the following database schema (the primary key attributes are underlined)

Students(<u>StudentCode</u>, Name, City, BirthYear)
Exams(<u>Subject</u>, <u>Candidate</u>, Grade, Date)

Write the logical query plan for the following queries:

1. Find the number of students who have passed the DB exam with grade 30.
2. Find the name and the student code of students who have passed 3 exams.
3. Find the name and the student code of students who have passed some exam.
4. Find the name and the student code of students who have not passed some exam.
5. Find the name and the student code of students who have passed all exams.

3.4 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.

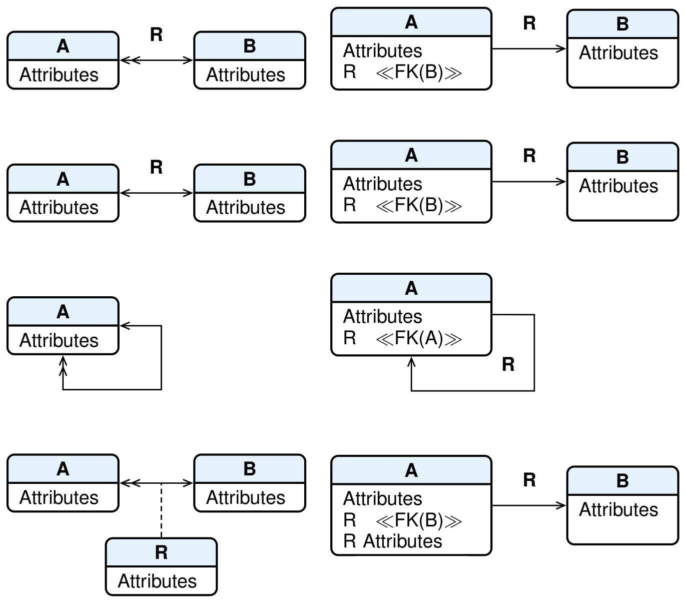
In this section the steps of an algorithm are described to design a relational schema from a conceptual design. For simplicity only structural aspects are considered:

STEP 1 Representation of 1:N and 1:1 associations with the rules in Figure 3.6.

STEP 2 Representation of N:M associations with the rules in Figure 3.7.

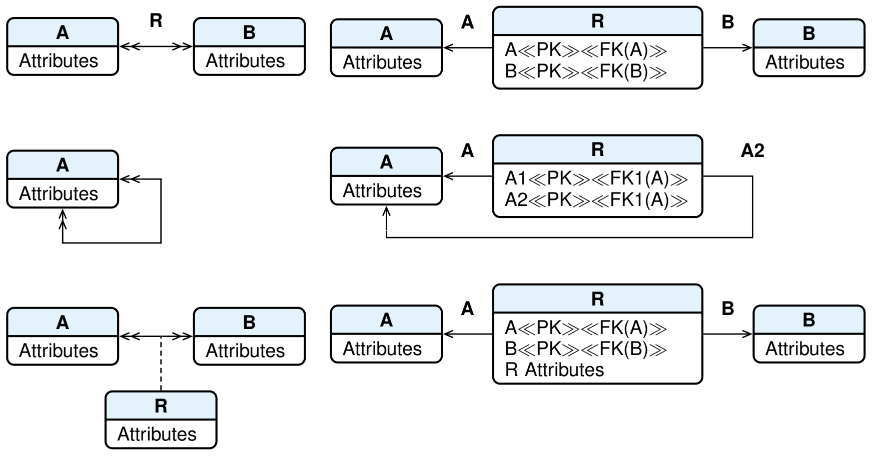
STEP 3 Representation of class hierarchies with the rules in Figure 3.8. For simplicity we assume that the subclasses are disjoint and that the class attributes are not redefined in subclasses. Three main options are possible:

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



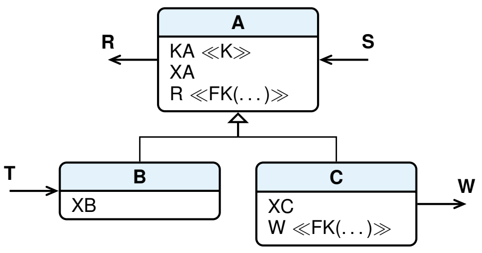
(a) *ODM associations* **(b)** *Relational representation of ODM associations*

Figure 3.6: Rules for STEP 1

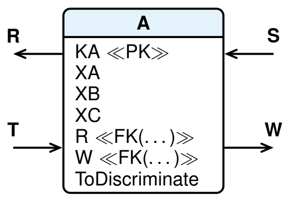


(a) *ODM associations* **(b)** *Relational representation of ODM associations*

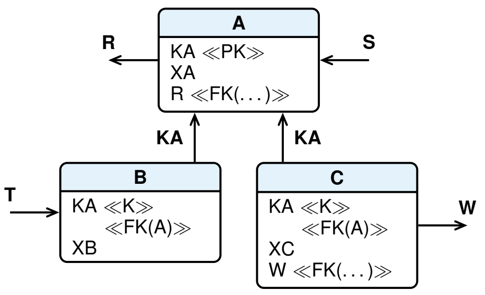
Figure 3.7: Rules for STEP 2



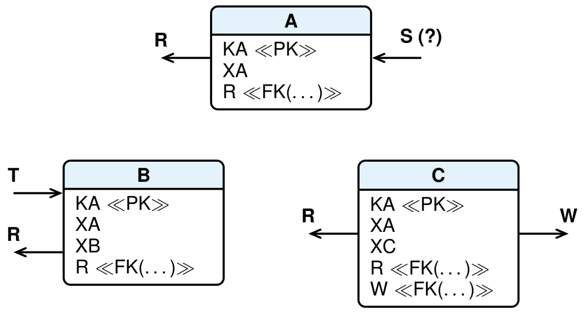
(a) *ODM subclasses*



(b) *Single relation representation of ODM subclasses*



(c) *Relational representation of ODM subclasses by vertical partitioning*



(d) *Relational representation of ODM subclasses by horizontal partitioning*

Figure 3.8: Rules for STEP 3

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 3.9.

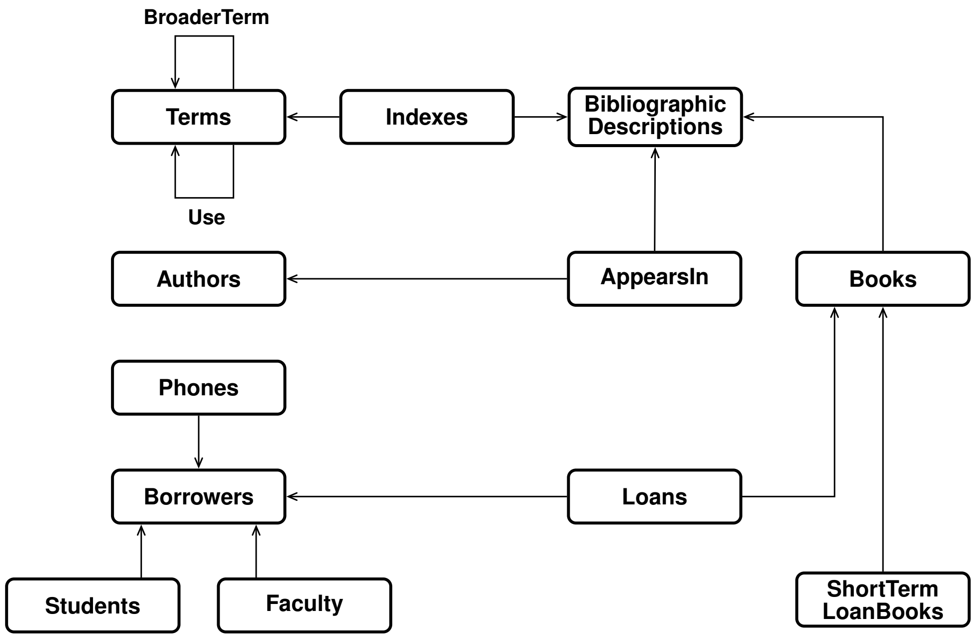


Figure 3.9: A relational schema for a library

3.4.1 Exercises

1. Convert the following conceptual schemas to a relational database schema.
   (a) Your solution to Exercise 2.5(3).
   (b) Your solution to Exercise 2.5(5).
   (c) Your solution to Exercise 2.5(6).

3.5 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 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 (*CallNumber*). 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:

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	61338	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:

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

UserName	**Address**	**Tel**
Rossi Carlo	Carrara	75444
Paolicchi Luca	Avenza	59729
Pastine Maurizio	Dogana	61338
Paolicchi Laura	Avenza	59729

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

CallNumber	Author	Title	Date	Tel
XY188A	Boccaccio	Decameron	07-07	75444
XY256B	Verga	Novelle	07-07	59729
XY090C	Petrarca	Canzoniere	01-08	61338
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:

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

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 $\{A\} \cup X$.
- $A_1A_2 \ldots A_n$ is a shorthand for $\{A_1, A_2, \ldots, A_n\}$.
- 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.

3.5.1 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 [Codd, 1970] proposed a particular kind of formalism based on the notion of functional dependency:

Definition 3.4

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

$$\forall t_1, t_2 \in r \text{ such that } t_1[X] = t_2[X] \text{ implies } t_1[Y] = t_2[Y]. \tag{3.1}$$

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 3.5

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.

3.5.2 Inference Rules

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

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

Definition 3.6

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

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

$\{X \to Y, X \to Z\} \vdash X \to YZ$ (*union rule*)
$Z \subseteq Y\ \{X \to Y\} \vdash X \to Z$(*decomposition rule*)
$\{\} \vdash X \to X$
$Z \subseteq Y\{X \to Y\} \vdash XZ \to Y$
$W \subseteq Z, V \subseteq Y\ \{X \to Y\} \vdash XZ \to VW$

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 3.1

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.

3.5.3 Closure of a Set of FDs

Definition 3.7

Given a set F of FDs, the closure of F, denoted by F^+, is:

$$F^+ = \{X \to Y | F \vdash X \to Y\}$$

Therefore, to test whether an FD $V \to W$ 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 $F = \{A \to B_1, \ldots, A \to B_n\}$, then F^+ will includes all the dependencies $A \to Y$, 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.

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

Definition 3.8

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 3.2

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

Instead of computing F^+, compute X^+ and then test whether $Y \subseteq X^+$. The following simple algorithm can be used to compute X^+.

Algorithm 3.1 *Computing the Closure of an Attribute Set X*

```
$X^+ = X$
while (changes to $X^+$) do
    for each $W \rightarrow V$ 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 [Beeri and Bernstein, 1979].[3]

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

Definition 3.9

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

1. $W \rightarrow T \in F^+$
2. $\forall V \subset W, V \rightarrow T \notin F^+$

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.

Example 3.3 Given a relational schema $R\langle T, F\rangle$, to find all the candidate keys of R is simple when the following properties hold:

1. If an attribute A of T does not appear on the right-hand side of some FDs, then A must be in any candidate key of R (see Exercise 1).

3. For an example see the application implemented by R. Orsini available at this URL:http://dblab.dsi.unive.it:8080

2. If an attribute A of T appears on the right-hand side of some FDs, but it does not appear on the left-hand side of some non trivial FDs, then A is not not in any candidate key of R.

Let X a set of attributes which do not appear on the right-hand side of some FDs. According to property (1), if $X^+ = T$, then X is the *only candidate key* for R.

For example, let $R\langle T, F\rangle$ a schema with $T = \{A, B, C, D, E, G\}$ and F = $\{BC \to AD, D \to E, CG \to B\}$. C and G do not appear on the right-hand side of some FDs, therefore they must be in any candidate key of R. Since $CG^+ = T$, CG is the only candidate key of R.

Instead, if $X^+ \neq T$, then we must add other attributes to X. According to property (2), we consider only attributes W of T which appear both on the right-hand side and on the left-hand side of some FDs. At every step we must avoid adding attributes that are already in the closure of X, since these attributes are clearly redundant, or attributes that produce a set X' that contains a key found earlier. Then we calculate the closure of each X', until it is different from T. At the end of the process $X' = T$, and so X' is a candidate key.

For example, Let $R\langle T, F\rangle$ a schema with $T = \{A, B, C, D, E, G\}$ and F = $\{AB \to C,\ BC \to AD,\ D \to E,\ CG \to B\}$. G does not appear on the right-hand side of some FDs, but $G^+ = G$. Let us add attributes from $W = \{A, B, C, D\}$ to G and compute their closure to find all candidate keys:

$GA^+ = GA \neq T$.
$GB^+ = GB \neq T$.
$GC^+ = T$. GC is a candidate key of R.
$GD^+ = GDE \neq T$.

Let us add now attributes from W to GA, GB and GD, considering only sets of attributes that do not contain the key GC:

$GAB^+ = T$. GAB is a candidate key of R.
$GAD^+ = GADE \neq T$.
$GBD^+ = GBDE \neq T$.

Finally, we try to add other attributes from W to GAD and GBD, but we find that there not sets of attributes that do not contain the keys GC and GAB, therefore we conclude that there are no other candidate keys.

3.5.4 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 3.10

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 3.11 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 3.3

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 3.4 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 3.5.4 has time complexity $O(a^2p^2)$.

3.5.5 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 3.12

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 3.13

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**

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

$$r \subseteq (\boldsymbol{\pi}_{T_1}(r)) \bowtie (\boldsymbol{\pi}_{T_2}(r)) \bowtie \cdots \bowtie (\boldsymbol{\pi}_{T_k}(r))$$

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 3.5 Let us consider the following instance r 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 $r \subset (\boldsymbol{\pi}_{A,B}(r)) \bowtie (\boldsymbol{\pi}_{B,C}(r))$.

$\boldsymbol{\pi}_{\text{A, B}}(r) =$

A	B
a_1	b
a_2	b

$\boldsymbol{\pi}_{\text{B,C}}(r) =$

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 3.5**

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

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

3.5.6 Dependency Preserving Decomposition

■ **Definition 3.14**

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

$$\boldsymbol{\pi}_{T_i}(F) = \{X \to Y \in F^+ | X, Y \subseteq T_i\}$$

Proposition 3.1

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.

The following simple algorithm can be used to compute $\boldsymbol{\pi}_{T_i}(F)$.

Algorithm 3.2 *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 $Y \subset T_i$ do
        begin
            $Z := Y_F^+ - Y$;
            return $Y \to (Z \cap T_i)$
        end
end
```

Definition 3.15

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

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 [Ullman, 1989].

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 3.16

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.

3.5.7 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 3.17**

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

Example 3.6 A schema which is not in 3NF is

Employees(#Employee, NameOfEmployee, NameOfDept, InformationOnDept)
#Employee $\to$ NameOfEmployee NameOfDept InformationOnDept
NameOfDept $\to$ 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 3.2**

$R < T, F >$ 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 3.3**

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

Example 3.7 Let us consider the following schema *ZipCodes*(*City, Street, Zip*), with FDs

City Street $\to$ Zip
Zip $\to$ 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 3.18

$R < T, F >$ 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 3.7 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

$R < T, F >$ 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 3.5

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.

3.5.8 Polynomial Algorithms to Normalize in 3NF and BCNF

A Synthesis Algorithm for 3NF

The best known synthesis algorithm was proposed by Bernstein [Bernstein, 1976]. The basic steps are the followings:

STEP 1 A canonical cover G of the FDs is computed.

STEP 2 G is 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.

STEP 3 Each G_i 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 3.8 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 [Bernstein, 1976] where it is also shown that its complexity is $O(a^2p^2)$. In [Biskup et al., 1979], the following 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*.

STEP 4 If the final set of relation schemas does not include a relation whose attributes are a superkey of the relation R which contains all the attributes in the initial FDs, which are the inputs to the synthesis algorithm, a relation schema is added with attributes W, where W is a key of R.

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 the following.

■ **Algorithm 3.3** *A Decomposition Algorithm for BCNF*

```
$\rho := \{R_1\langle T_1, F_1\rangle\}; n := 1;$
while exists in $\rho$ a $R_i\langle T_i, F_i\rangle$ not in BCNF because of the FD $X \to A$ do
    $n := n + 1;$
    $T' := X\,A;$
    $F' := \pi_{T'}(F_i);$
    $T'' := T_i - A;$
    $F'' := \pi_{T''}(F_i);$
    $\rho := \rho - R_i\langle T_i, F_i\rangle + \{R_i\langle T', F'\rangle, R_n\langle T'', F''\rangle\}$
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.

[Tsou and Fischer, 1982] 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.

3.5.9 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) has been introduced and a new normal form has been defined, a generalization of a Boyce-Codd normal form, called *fourth normal form* (4NF), that applies to relation schemas with functional and multivalued dependencies.

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 [Maier, 1983] for a fuller discussion of dependency theory, including other topics which have not been addressed here.

3.5.10 Exercises

1. Prove that for a schema $R < T, F >$, with F a canonical cover, if an attribute A_i does not appear on 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 $X \to A \in F$, 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 $AB \to C$, $C \to D$, and $D \to A$.
   (b) $R(A, B, C, D)$ with functional dependencies $A \to B$, and $A \to C$.
   (c) $R(A, B, C, D)$ with functional dependencies $A \to B$, and $B \to C$.

   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.

Part II

DBMS: The User Perspective

Chapter 4

OBJECTIVES OF A DBMS

4.1 Introduction

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

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 4.2

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.

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.

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*.

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 a later chapter.

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 how the database is queried by users or by 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 queries or 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 4.3**

A *transaction* is a sequential program with embedded database operations and the following properties properties:

Atomicity	Only transactions terminated normally (*committed transactions*) change the database; if a transaction execution is interrupted because of a failure (*aborted transaction*), the state of the database should remain unchanged as if no operations of the interrupted transaction had occurred.
Durability	The effects of a committed transaction are permanent and must survive system and media failures, i.e. commitment is an irrevocable act.
Isolation	When a transaction is executed concurrently with others, the final effect must be the same as if it were executed alone.

The acronym ACIDis sometimes used to refer to the following four properties of transactions: *Atomicity*, *Consistency*, *Isolation*, and *Durability*. Among these properties, atomicity, durability and isolation are provided by a DBMS. Consistency cannot be ensured by the system when the integrity constraints are not declared in the schema. However, assuming that each transaction program maintains the consistency of the database, the concurrent execution of the transactions by a DBMS also maintain consistency due to the isolation property.

The isolation property is sometime called the *serializability* property: when a transaction is executed concurrently with others, the final effect must be the same as a *serial* execution of committed transactions, i.e. the DBMS behaves as if it executes the transactions one at a time.

■ **Definition 4.4**

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 4.5**

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.

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 4.6**

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.

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

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.

Chapter 5

SQL: A RELATIONAL DATABASE LANGUAGE

5.1 Introduction

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.

5.2 The Data Definition Sublanguage

A relation schema is specified using the **CREATE TABLE** command of SQL. This command 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 command 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** command and deleted with the **DROP TABLE** command.

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** command. A view can be queried as an ordinary table, but its content does not physically exist 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** command 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';
```

5.3 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 provides the **GRANT** and **REVOKE** commands 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** command:

```
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 5.1 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;
```

5.4 The Query Sublanguage

SQL provides the **SELECT** command to define a query to retrieve data from a database. A query expresses in a declarative way *what* we are looking for rather than *how* to compute the result, as it happens with a relational algebra expression (logical plan). Another important aspects of SQL is that the tables of a database may be without keys and so they are not *set* but *multiset* (*bags*). Therefore, in order to understand the semantics of an SQL query in terms of a relational algebra expression, the relational algebra is extended on *multisets* as follows.

Relation operations on multisets

- ***Multiset projection***: $\boldsymbol{\pi}^{\boldsymbol{b}}_{A_1, A_2, \ldots, A_m}(E)$

  where $A_1, A_2, \ldots, A_m$ are attributes of E.
  The operator returns the tuples of E projected onto the attributes $A_1, A_2, \ldots, A_m$, without duplicates elimination, as it happens with the π operator of relational algebra, which is not any longer available.
  The result is a multiset with type $\{\{(A_1 : T_1, A_2 : T_2, \ldots, A_n : T_n)\}\}$.

- ***Duplicate elimination***: $\boldsymbol{\delta}(E)$

  The operator returns the tuples of E without duplicates.

– ***Sort***: $\boldsymbol{\tau}_{\mathsf{A_1,A_2,\ldots,A_m}}(E)$

  where $A_1, A_2, \ldots, A_m$ are attributes of E.
  The result is a *list* with the tuples of E sorted in ascending order on the attributes $A_1, A_2, \ldots, A_m$ and type $\{\{(A_1 : T_1, A_2 : T_2, \ldots, A_n : T_n)\}\}$. To sort in descending order the attributes becomes pairs $A_i\, d$, where d stands for "descending".
  Since the the operator returns a *list* of tuples, rather than a multiset, it is meaningful as the root of a logical plan only.

– ***Multiset union, intersection and difference***: $E_1 \cup^b E_2$, $E_1 \cap^b E_2$, $E_1 -^b E_2$

  Note that in SQL the order of the relational attributes is *important*: two relations R and S have the same type if (a) they have the same set of pairs (attribute, type), and (b) the order of the attributes is the same for both relations.

  If an element t appears n times in E_1 and m times in E_2, then

  – t appears $n + m$ times in the multiset $E_1 \cup^b E_2$:

$$\{1, 1, 2, 3\} \cup^b \{2, 2, 3, 4\} = \{1, 1, 2, 3, 2, 2, 3, 4\}$$

  – t appears *min*(n, m) times in the multiset $E_1 \cap^b E_2$:

$$\{1, 1, 2, 3\} \cap^b \{2, 2, 3, 4\} = \{2, 3\}$$

  – t appears *max*$(0, n - m)$ times in the multiset $E_1 -^b E_2$:

$$\{1, 1, 2, 3\} -^b \{1, 2, 3, 4\} = \{1\}$$

The extension of the other relational algebra operators (selection, grouping, product, join) from sets to bags is obvious.

The SELECT syntax

Let us consider a simplified version of the command syntax.

```
SELECT     DISTINCT  Attributes
FROM       Tables
WHERE      Condition
ORDER BY   Attributes;
```

where

Attributes ::= $*$ | *Attribute* {, *Attribute* }
Tables ::= *Table* [*Ide*] {, *Table* [*Ide*]}

The asterisk means to retrieve all attributes; alternatively, the desired attributes are listed separated by commas. The **DISTINCT**, **WHERE** and **ORDER BY** clauses only are optional.

The semantics of the **SELECT** command is given with the following equivalence:

SELECT $*$
FROM $R_1, \ldots, R_n$
WHERE C; $\quad \equiv \boldsymbol{\sigma}_C(R_1 \times \ldots \times R_n)$

SELECT **DISTINCT** $A_1, \ldots, A_n$
FROM $R_1, \ldots, R_n$
WHERE C; $\quad \equiv \boldsymbol{\delta}(\boldsymbol{\pi}^{\boldsymbol{b}}_{A_1,\ldots,A_n}(\boldsymbol{\sigma}_C(R_1 \times \ldots \times R_n)))$

SELECT **DISTINCT** $A_1, \ldots, A_n$
FROM $R_1, \ldots, R_n$
WHERE C
ORDER BY A_1, A_2; $\equiv \boldsymbol{\tau}_{\mathsf{A_1,A_2}}(\boldsymbol{\delta}(\boldsymbol{\pi}^{\boldsymbol{b}}_{A_1, \ldots, A_n}(\boldsymbol{\sigma}_C\ (R_1 \times \ldots \times R_n))))$

Example 5.2 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 command:

```
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 command terminator.

Set Queries: Union, Intersection, Difference

Set queries are expressed by the following forms:

(<subquery>)**UNION** [**ALL**] (<subquery>)
(<subquery>) **INTERSECT** [**ALL**](<subquery>)
(<subquery>) **EXCEPT** [**ALL**] (<subquery>)

The set operators have *set semantics*, adding the **ALL** keyword forces *bag semantics* (duplicates allowed).

The semantics of the set queries is given with the following equivalence:

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

SELECT   *
FROM     R
  INTERSECT
SELECT   *
FROM     S;                  ≡ R ∩ S

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

5.5 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.

5.6 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 with the following version of the command syntax.

```
SELECT     DISTINCT $S_A, S_{AF}$
FROM       $T$
WHERE      $W_C$
GROUP BY   $G_A$
HAVING     $H_C$
ORDER BY   $O_A$;
```

where (a) S_A are the select attributes and S_{AF} are the select aggregation functions; (b) T are the **FROM** tables; (c) W_C is the **WHERE** condition; (d) G_A are the grouping attributes, with $S_A \subseteq G_A$; (e) H_C is the **HAVING** condition with aggregation functions H_{AF}; (f) O_A are the sorting attributes; (g) the **DISTINCT**, **WHERE**, **HAVING** and **ORDER BY** clauses are optional.

The command semantics with tables R and S, and all the optional clauses specified, in terms of the extended relational algebra is shown in Figure 5.1.

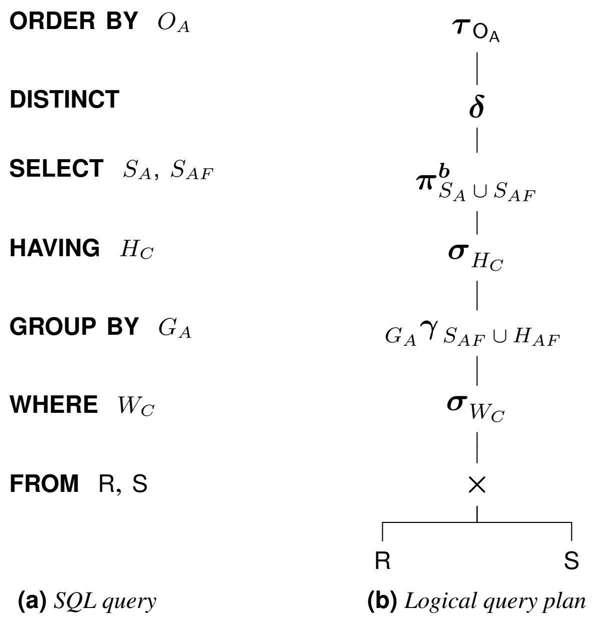


Figure 5.1: SQL query with **GROUP BY** semantics

For example

```
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 value of Name. The aggregate function AVG(BirthYear) 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 AVG(BirthYear). The important point is that *each attribute 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 only used in conjunction with **GROUP BY**. The **HAVING** condition (unlike the **WHERE** condition) is applied to groups, not to individual tuples, 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.

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

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;
```

5.7 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) ;
```

SQL nested queries cannot be expressed easily into in terms of the extended relational algebra, since there is no relational algebra operation equivalent to the subquery construct. Extensions of relational algebra have been proposed for this task, but they are beyond the scope of this report.

5.8 Queries that Require Universal Quantifiers

One major disadvantage of SQL is its lack of a universal quantification construct to express the *for all* and *every* phrases in a natural and intuitive way.

For example, let us consider the following database

```
Students(StudentCode, Name, City, YearOfBirth)
Exams(Subject, Candidate, Grade, Date)
```

The grade of an exam has one of the following values: A, B, C, D.

A query that requires universal quantification is "*Find the name and student code of the students who have passed all their exams with top grade A*".

The following method is proposed to be helpful to write in SQL this kind of queries: Let us assume that we have two non-SQL clause **FOR ALL**, **FOR SOME** to express an universal or existential quantification, and then we give a rule to translate the query in standard SQL.

ForAllPredicate ::= **FOR ALL** *TableReferenceList* [*WhereClause*] **:** *Cond*
ForSomePredicate ::= **FOR SOME** *TableReferenceList* [*WhereClause*] **:** *Cond*

TableReferenceList ::= [*Ide* **IN**] *Table* [, [*Ide* **IN**] *Table*]

FOR ALL evaluates to *true* if and only if the condition *Cond* evaluates to *true for all* the tuples of the cross product of the tables referenced in the *TableReferenceList* that satisfy the *WhereClause*.

FOR SOME evaluates to *true* if and only if the condition *Cond* evaluates to *true for some* of the tuples of the cross product of the tables referenced in the *TableReferenceList* that satisfy the *WhereClause*.

A *ForAllPredicate* can then be converted into an equivalent predicate involving *ForSomePredicate* instead, by virtue of the following identity:

$$\forall z\,(\exists y\, p(z,y)) \leftrightarrow \neg\, \exists z\,(\neg \exists y\, p(z,y))$$

A *ForSomelPredicate* is represented easily in SQL by an expression of the form EXISTS (SELECT * FROM ...).

Example 5.3 Let us see how to write in SQL the query "*Find the name and student code of the students who have passed all their exams with top grade A*".

First, let us write the query with the *ForAllPredicate*:

```
SELECT    StudentCode, Name
FROM      Students s
WHERE     FOR ALL e IN Exams
               WHERE e.Candidate = s.StudentCode
                    : e.Grade = 'A';
```

Then, let us rewrite the *ForAllPredicate* using the *ForSomePredicate*:

```
SELECT    StudentCode, Name
FROM      Students s
WHERE     NOT FOR SOME e IN Exams
              WHERE e.Candidate = s.StudentCode
                   : NOT (e.Grade = 'A');
```

Hence the SQL formulation is:

```
SELECT    StudentCode, Name
FROM      Students s
WHERE     NOT EXISTS (
          SELECT    *
          FROM      Exams e
          WHERE     e.Candidate = s.StudentCode AND NOT (e.Grade = 'A') );
```

The way in which has been written the initial query with the *ForAllPredicate* presents a problem: if there is a student that *has not passed* any exam, *he will appear in the result*, because the universal quantification evaluates to *true* if the set of tuples of the cross product of the tables referenced in the *TableReferenceList* that satisfy the *WhereClause is empty*.

To exclude this case, the query should be expressed as "*Find the name and student code of the students with some exam, who have passed all their exams with top grade A*".

```
SELECT    StudentCode, Name
FROM      Students s
WHERE     FOR SOME (e IN Exams
              WHERE e.Candidate = s.StudentCode)

          AND FOR ALL e IN Exams
              WHERE e.Candidate = s.StudentCode
                 : e.Grade = 'A';
```

The SQL formulation becomes:

```
SELECT    StudentCode, Name
FROM      Students s
WHERE     EXISTS (
          SELECT     *
          FROM       Exams e
          WHERE      e.Candidate = s.StudentCode )

          AND NOT EXISTS (
          SELECT     *
          FROM       Exams e
          WHERE      e.Candidate = s.StudentCode AND NOT (e.Grade = 'A') );
```

The first **EXISTS** can be replaced by a join in the external **SELECT**:

```
SELECT    StudentCode, Name
FROM      Students s, Exams
WHERE     Candidate = s.StudentCode
          AND NOT EXISTS (
          SELECT     *
          FROM       Exams e
          WHERE      e.Candidate = s.StudentCode AND NOT (e.Grade = 'A') );
```

5.9 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 command 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** command can be used in place of the **VALUES** clause of the **INSERT** command 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  StudentCode = '01234';
```

This sets to ‘Florence’ the city for the student with StudentCode = '01234'. 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 command.

DELETE is quite similar to **UPDATE**. The following command 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.

5.10 Executing SQL Commands 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 commands 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 tables, i.e. multisets 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 command. To evaluate an SQL command, 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 commands 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 commands and loops, variables, constants, and types, structured data, and customized error handling. The language compiler can control completely that SQL commands 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 5.2) use the standard cursor, while the second example use a special construct FOR with an implicit cursor (Figure 5.3).

```
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 5.2: 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 5.3: 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 commands 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 5.4).
3. *Embedded SQL*. SQL commands can be used within a host language program. Before the program can be compiled by the host language compiler, the SQL commands 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 5.5).

```
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 5.4: An example of API

```
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 5.5: An example of embedded SQL

Figure 5.6 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 command #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.

```
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 5.6: An example of SQLJ

5.11 Exercises

1. Give a relational schema in SQL for the following databases:
   (a) Your solution to Exercise 3.4.1(1).
   (b) Your solution to Exercise 3.4.1(2).
2. Give a relational schema in SQL for your solution to Exercise 3.4.1(3), and write the following queries:
   (a) Retrieve the birth-date and name of the female employees.
   (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.

Part III

DBMS: The System Perspective

Chapter 6

DBMS ARCHITECTURE

6.1 Introduction

As a programming language transforms a computer into an *abstract machine* whose characteristics and functionalities are mainly determined by those of the programming language, so will a language to define and use databases transform a computer, and in particular its file management system, into an *abstract database machine*, called the *database management system*, whose characteristics and functionalities will depend mainly on those of the data model adopted (Figure 6.1).

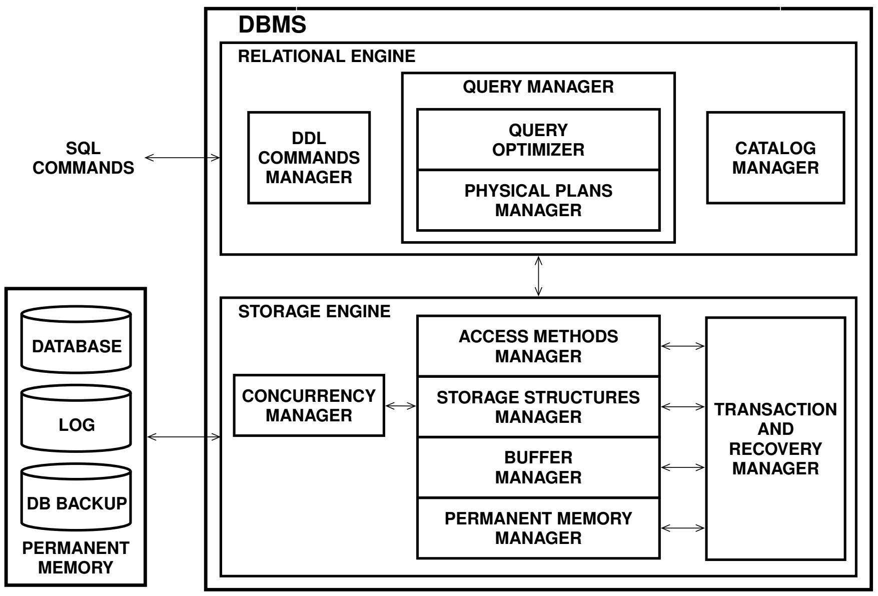


Figure 6.1: Architecture of a DBMS

An abstract database machine is normally divided into two: an *abstract machine for the logical data model* (from now on called the *Relational Engine*) and an *abstract machine for the physical data model* (the *Storage Engine*).

The *Relational Engine* includes modules to support the following facilities:

– The *Data Definition Language (DDL) Manager*, which process a user's database definition of a logical schema, an external schema, and a physical schema.
– The *Query Manager*, which process a user's query by transforming into an equivalent but more efficient form, thus finding a good strategy for executing it.
– The *Catalog Manager*, which manage special data, called *metadata*, about the schemas of the existing databases (views, storage structures and indexes), and security and authorization information that describes each user's privileges to access specific database, relations and views, and the owner of each of them. The catalog is stored as a database which allows the other DBMS modules to access and manipulate its content.

The relational engine interacts with the *Storage Engine*, which includes modules to support the following facilities:

– The *Permanent Memory Manager*, which manages the page allocation and deallocation on disk storage.
– The *Buffer Manager*, which manages the transfer of data pages between the permanent memory and the main memory.
– The *Storage Structures Manager*, which manages the data structures to store and retrieve data efficiently.
– The *Access Methods Manager*, which provides the storage engine operators to create and destroy databases, files, indexes, and the data access methods of sequential scan, and index scan.
– The *Transaction and Recovery Manager*, which ensures that database consistency is maintained despite transaction and system failures.
– The *Concurrency Manager*, which ensures that there is no conflict between concurrent accesses to the database.

Normally the DBMS storage engine is not accessible to the user, who will interact with the relational engine. An example of a system in which this structure is clearly shown is *System R*, a relational DBMS prototype developed at the IBM scientific center in San Josè, from which DB2 was then produced. This system has a relational engine called *Relational Data System* (RDS) and a storage engine called *Relational Storage System* (RSS).

While the interface with the relational engine depends on the data model features, the interface with the storage engine will be affected by the way the data are organized in permanent memory. In the following sections, we will first present the main issues affecting storage management and then discuss how query processing, recovery management, and concurrency control can be implemented.

6.2 Storing Collections of Records

We shall assume that a physical database is stored as a collection of *records*. Each record consists of one or more *attributes* (or *fields*) of an elementary type, such as integers, character strings, or pointers to other records. A collection of homogeneous records will be called a *file*.

For each file, there will be an attribute, called the *primary key*, whose value is unique for each record occurrence. Although accessing records via their primary keys is very common, other accesses may also be needed. At times, for instance, it is necessary to retrieve all records containing a given value for some non-primary key attribute, called a *non-key attribute*.

When records are stored in permanent memory, the unit used for data transfer between permanent storage and main memory is not normally a record, but a page. Pages are assumed to be of a fixed size, between 1 to 4 Kbyte, and to contain several records. The unit of cost for data access is a page access (read or write), and we assume that the costs of computation in main memory of the data in a page are negligible compared with the cost of a page access.

6.2.1 Page Structure

When a record is stored in the database, it is identified internally by a *record identifier* or *tuple identifier* (RID), which is then used in all data structures as a pointer to the record. The exact nature of a RID can vary from one system to another. An obvious solution is to take its address (*Page number, Beginning of record*) (Figure 6.2a). But this solution is not satisfactory because a record that contain variable-length attributes of type varchar are themselves variable-length strings within a page; so updates to data records can cause growth and shrinkage of these byte strings and may thus require the movement of records within a page, or from one page to another. When this happens all the references to the record in other data structures, most notably for indexing purposes, must be updated.

To avoid this problem, another solution is usually adopted: the RID consists of two parts (*Page number, Slot number*), where the slot number is an index into an array stored at the end of the page, called *slot array*. containing the full byte address of a record (Figure 6.2b). All records are stored contiguous, followed by the available free space.

If an updated record moves within its page, the local address in the array only must change, while the RID does not change. If an updated record cannot be stored in the same page because of lack of space, then it will be stored in another page, and the original record will be replaced by a forwarding pointer (another RID) to the new location. Again, the original RID remains unaltered. A record is split into smaller records stored in different pages only if it is larger than a page.

Each page has a *page header* (HDR) that contains administrative information, such as the number of free bytes in the page, the reference at the beginning of the free space, and the reference to the next not empty page of the file.

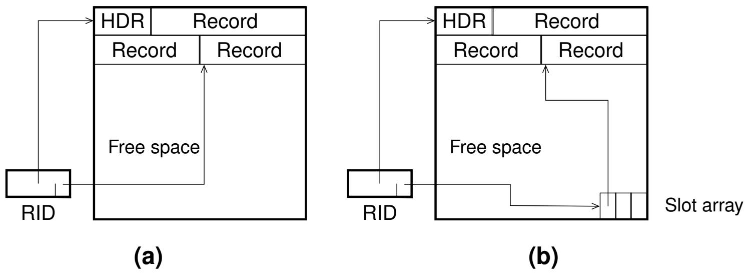


Figure 6.2: Pointers to records

6.2.2 Table Organizations

In the following sections, we shall first consider table organizations which support primary key and non-key attribute retrievals. Attention will be focussed on the typical solutions currently adopted in DBMSs. Of course, many other types of organization have been studied for specific applications (e.g., organizations for geometric data, pictures, and text), but they are outside the scope of this book.

A table organization is identified by the following two properties.

■ **Definition 6.1**

A table organization is called a *primary organization* if it determines the way the records are physically stored, otherwise it is called a *secondary organization*.

■ **Definition 6.2**

A table organization is said to be *static* if it has been designed to handle a given table size and, if the table exceeds this size, is not able to preserve efficiency; this implies a degradation of performance which entails a periodic reorganization. In contrast, an organization is said to be *dynamic* if it gradually evolves as records are added, thus preserving efficiency without the need for reorganization.

6.3 Heap and Sequential Organizations

The simplest way to organize data is store them in pages in the arrival order rather than in some special order. In this way, the pages allocated to data can be contiguous or can be linked in a list. This type of file organization is called a *heap organization*. Instead, when the records are stored in the order of a primary key, we have a *sequential organization*. Commercial DBMSs use the *external merge sort* algorithm for sorting a file.

6.4 Primary Key Organizations

The goal of a primary key organization is to be able to retrieve a record with a specified primary key value in as few accesses as possible, one is the optimum. In order to achieve this, a mapping from the set of primary keys to the set of record is defined. The mapping can be implemented as a *function*, implemented with an *hashing* technique or with a tree structure.

In the first case a hash function h is used that maps the key value k to the value $h(k)$. The value $h(k)$ is used as the address of the page in which the record is stored.

In the second case a tree structure called B^+-tree is used, with all records stored ordered in leaf nodes, and the leaf nodes are linked together. This solution is called *index-sequential organization*.

With a *secondary organization* the mapping from a key to the record is implemented with *tabular method*, listing all inputs and outputs, commonly known as an *index*. A *secondary organization* helps answer queries but do not affect the location of the data.

An index in this context has a role similar to that of a book. The pages of a book are ordered, and to find information about a particular subject, we use the index in the back of the book, in which we look up a keyword to get the list of one or more pages on which the keyword appears. In a similar way, in the case of the set of records of a table, to find a record with a given attribute value, we first look at the index defined on the attribute to get the location of the records in the table, and then the records are retrieved. As in the book index analogy, the index is ordered on the attribute values.

■ Definition 6.3

Let R be a table of records with an attribute A. An index I on a key A is a sorted table $I(A, \text{RID})$ on A, with $(N_{\text{rec}}(I) = N_{\text{rec}}(R))$. An entry of the index is a tuple $(A := a_i, \text{RID} := r_i)$, where k_i is a A value for a record, and r_i is a reference (RID) to the corresponding record in R. The records R are stored with an independent organization.

To access a record through an index, the index is first accessed to obtain the reference to the record, and then the record is retrieved. The index is stored in permanent memory using a primary organization.

An index can be defined on a *key attribute*, on a *non-key attribute* or on a *set of attributes*. In the last case, the index, called *composite*, contains an element for each combination of values of the attributes in the table, and can be used to execute efficiently queries that specify a value for each of these attributes or for a prefix of them.

Figure 6.3 shows two example of indexes on two attributes of the relation R, the key K and a non-key attribute A, assuming for simplicity that the RIDs are integers that represent the position of the record in the relation.

R

RID	**K**	**A**	**...**
1	k5	d	...
2	k3	b	...
3	k7	a	...
4	k6	c	...
5	k2	b	...
6	k4	g	...
7	k1	c	...
...	...	...	...

IdxK

K	**RID**
k1	7
k2	5
k3	2
k4	6
k5	1
k6	4
k7	7
...	...

IdxA

A	**RID**
a	3
b	2
b	5
c	4
c	7
d	1
g	6
...	...

Figure 6.3: Example of a relation with two indexes

An index can also be defined on a set of attributes. In this case, the index contains a record for each combination of values of the attributes in the relation, and can be used to execute efficiently queries that specify a value for each of these attributes.

6.5 Solutions for Relational DBMS

DBMSs normally use a combination of these basic techniques to store data, depending on the abstraction mechanism of the data model and the associated operators. We shall now describe the solutions adopted by some relational DBMSs.

In INGRES, released by Relational Technology, when a relation $R(A_1 : T_1, \dots, A_n : T_n)$ is created, it is organized as a heap in which tuples are not sorted and duplicates are not removed. However, once data have been loaded, the structure can be converted into a sequential, a static hashing, or a primary index-sequential organization with a static index, using a *modify* statement of the form:

MODIFY R **TO HEAPSORT ON** A_i **ASC**;
MODIFY R **TO HASH ON** A_i;
MODIFY R **TO ISAM ON** A_i;

The modify command has other options, which permit the specification of a primary key organization (e.g., **hash unique on** A_i), or a compressed version of the previous organizations (e.g., **cheapsort, chash**, and **cisam**).

It is also possible to define a secondary index, with the command:

CREATE INDEX Name **ON** $R(A_i)$;

An index is treated as binary relations with tuples (A_i, RID). A secondary index can be defined on a combination of at most six attributes. A secondary index is initially created as a heap can and then be modified in the same way as any other relation.

In Oracle a primary key organization implemented with a tree structure is called IOT (*index organized table*), is created with the command:

CREATE TABLE R(Pk Type **PRIMARY KEY**, ...) **ORGANIZED INDEX**;

In SQL Server a primary key organization implemented with a tree structure is created with the definition of a **CLUSTERED INDEX** on the primary key of a relation with the commands:

CREATE TABLE R(Pk Type **PRIMARY KEY**, ...);
CREATE CLUSTERED INDEX RTree **ON** R(Pk);

6.6 Query Processing

We have considered different types of organizations which can be adopted to store databases. When a query is presented to a system, it is necessary to identify the best strategy to find the answer using the existing data structures. As a very simple example, let us suppose that we have a B^+-tree primary index on *Name* for the *Student* table and that we want to find all the students whose *Name* comes after "Smith" in the alphabet: should the system use the primary B^+-tree index, or should it perform a sequential read of the file? Since the most interesting optimization techniques have been studied for the relational data model, we shall assume that a query is expressed in SQL, to specify *what* the user wants to get from the database.

As it has been shown, in SQL a query may be expressed in different ways. Since we do not expect users to write their queries in a way that suggests the best strategy for finding the answer, it is responsibility of the *Query Manager* module of the system to figure out *how* to find the most efficient way to get the data the user wants.

Query processing is usually accomplished top-down in distinct phases, with each phase solving a well-defined subproblem. First, the query is checked whether it is syntactically and semantically correct and then it is transformed into a relational algebra query (i.e., the passage from *what* the user wants to *how* to get the data goes through the relational algebra). Second, logical transformations at the relational algebra level are applied to standardize and simplify the query. An attempt is made to find an equivalent reformulation of the query that can be more optimizable (e.g. elimination of subqueries and views). The next phase is to select a detailed strategy to access the data, and also the algorithms to perform the necessary database operations. The result of this process is the *physical query plan*: a tree of physical operators that implement algorithms to execute the relational algebra operators.

Each relational algebra operator can be implemented using several algorithms (*physical operators*). For example, the join of two tables $R \underset{p_k = f_k}{\bowtie} S$, with p_k the primary key of R and f_k the foreign key of S for R, can be implemented using one of the following algorithms, among others:

– *nested loops*:

```
for each $r \in R$ do
    for each $s \in S$ do
        if $r.p_k = s.f_k$ then add $< r, s >$ to result;
```

– *page nested loops*:

```
for each page $p_r$ of $R$ do
    for each page $p_s$ of $S$ do
        for each $r \in p_r$ do
            for each $s \in p_s$ with $r.p_k = s.f_k$ do
                add $< r, s >$ to result;
```

– *index nested loops*:

```
for each $r \in R$ do
    for each $s \in S$ with $s.f_k = r.p_k$ do
        add $< r, s >$ to result;
```

where for each record $r \in R$ the records $s \in S$ **with** $s.f_k = r.p_k$ are retrieved using the index on the join attribute $s.f_k$ of S.

– *merge-join*. The operator requires that
  – the join is an *equi-join*;
  – the records of the operands are sorted on the join attributes;
  – the join attribute of the external operand R is a primary key.

Under these assumptions, the records $r \in R$ and $s \in S$ of the join are found by scanning the two tables once as follows.

```
while there is a record $r \in R$ and a record $s \in S$ do
    if $r.p_k = s.f_k$
    then ( add $< r, s >$ to result;
            let $s$ the next record in $S$)
    else let $r$ the next record in $R$;
```

Each physical operator is implemented as an *iterator*, an object with the following public methods implemented using the operators on storage structures and the access methods provided by the storage engine:

1. *open*, to initialize the process of getting records,
2. *next*, to return the next record in the result and adjust data structures to allow subsequent records to be obtained. In getting the next record of its result it usually call the *next* method on its arguments
3. *isDone*, to signal if there are no more records to be produced,
4. *close*, to end the iteration after all records have been obtained.

Figure 6.4 shows a possible set of physical operators.

Logical Operator	Physical Operators	Description
Table R	**TableScan** (R)	Full table scan of R.
	IndexScan (R, I)	Ordered scan of R on the index *I* attributes.
	SortScan $(R, \{A_i\})$	Sorted scan of R on the attributes $\{A_i\}$.
Projection π^b	**Project** $(O, \{A_i\})$	Projection of O records without duplicate elimination.
Duplicate elmination δ	**Distinct** (O)	Duplicate elimination from O records *sorted* on the attributes $\{A_i\}$.
	HashDistinct (O)	Duplicate elimination from O records.
Sort τ	**Sort** $(O, \{A_i\})$	Sort O records on $\{A_i\}$.
Selection σ	**Filter** (O, ψ)	Selection of the O records that satisfy the condition ψ.
	IndexFilter (R, I, ψ)	Selection of the R records using the index *I* defined on the attributes in ψ.
	IndexOnlyFilter $(R, I, \{A_i\}, \psi)$	Selection of the R records attributes available in the index *I* and used in ψ, without any access to R. The attributes $\{A_i\}$ are a subset of those in *I*.
Grouping γ	**GroupBy** $(O, \{A_i\}, \{f_i\})$	Grouping of O records *sorted* on the attributes $\{A_i\}$ using the aggregate functions in $\{f_i\}$. The operator returns records with attributes A_i and the aggregate functions in $\{f_i\}$, sorted on the attributes $\{A_i\}$.
	HashGroupBy $(O, \{A_i\}, \{f_i\})$	Grouping of O records on the attributes $\{A_i\}$ using the aggregate functions in $\{f_i\}$.
Join $\bowtie$	**NestedLoop** (O_E, O_I, ψ_J)	*Nested-loop* join.
	PageNestedLoop (O_E, O_I, ψ_J)	*Page nested-loop* join.
	IndexNestedLoop (O_E, O_I, ψ_J)	*Index nested-loop* join. O_I uses an index on the join attributes.
	MergeJoin (O_E, O_I, ψ_J)	*Merge join* join. The operand O_E and O_I records are sorted on join attributes. The external operand join attribute is a key.
Set Operators $\cup, -, \cap$	**Union, Except, Intersect** (O_E, O_I)	Set operations with the operand records sorted and without duplicates..
	UnionAll(O_E, O_I)	Union without duplicates elimination.

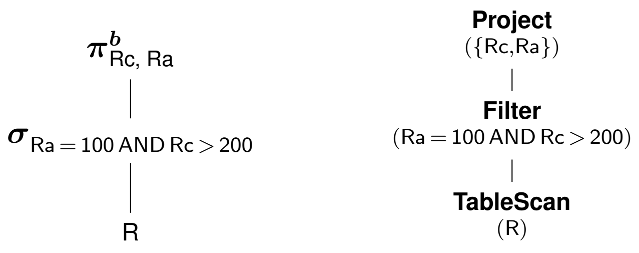
Figure 6.4: Examples of physical operators for query plans

Let us consider some examples of logical and physical plans for SQL queries on the database with relations $R(RPk, Ra, Rb, Rc)$ and $S(SPk, SFk, Sa)$, with all attributes of type integer.

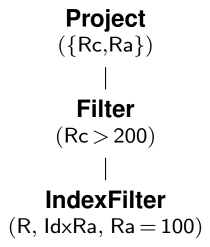
Example 6.1

```
SELECT      Rc, Ra
FROM        R
WHERE       Ra = 100 AND Rc > 200;
```

If there are not indexes, a possible *physical query plan* is



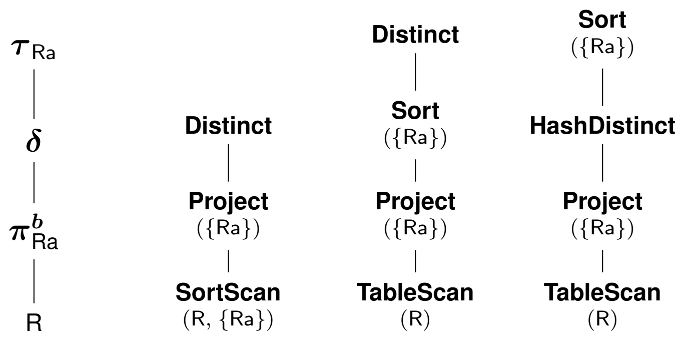
If there is an index on Ra, a possible *physical query plan* is



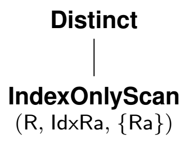
Example 6.2

```
SELECT      DISTINCT Ra
FROM        Rn
ORDER BY    Ra;
```

If there are not indexes, possible *physical query plans* are



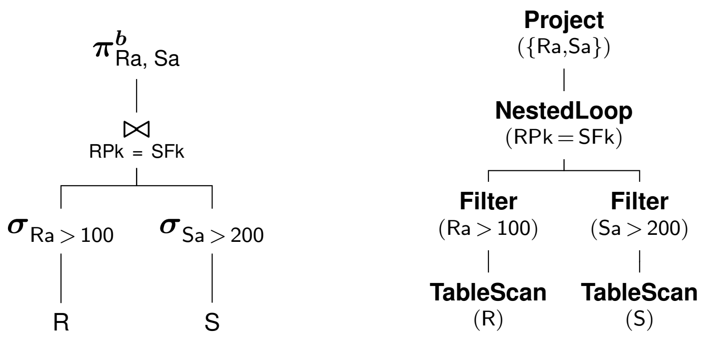
If there is an index on Ra, a possible *physical query plan* is



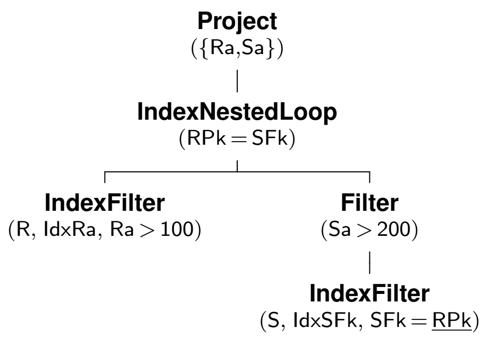
Example 6.3

SELECT Ra, Sa
FROM R, S
WHERE RPk = SFk **AND** Ra > 100 **AND** Sa > 200;

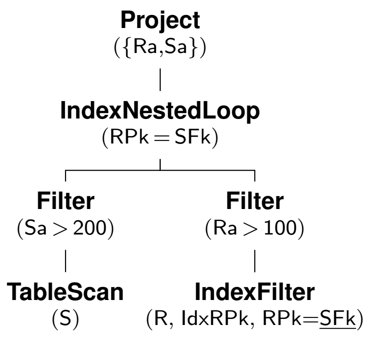
If there are not indexes, a possible *physical query plan* is



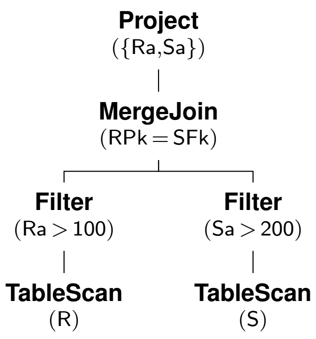
Let us assume that there are two indexes on Ra and on SFk, and to execute the join with the **IndexNestedLoop**:



If there is also an index on RPk, an **IndexNestedLoop** can be used with R as internal operand:

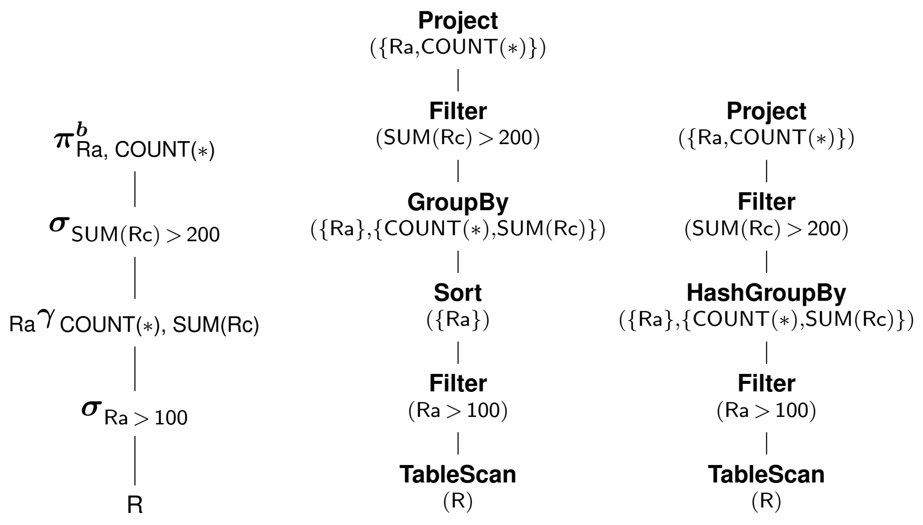


If the relations are sorted on the join attributes, the join can be execute with the **MergeJoin** as follows:



Example 6.4

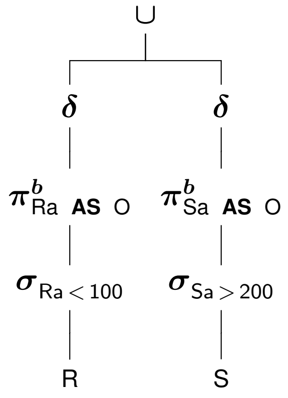
```
SELECT     Ra, COUNT(∗)
FROM       R
WHERE      Ra > 100
GROUP BY   Ra
HAVING     SUM(Rc) > 200;
```

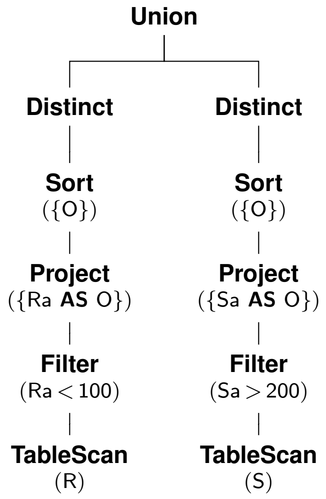

$\pi^b_{\text{Ra, COUNT}(*)}$
$\sigma_{\text{SUM(Rc)} > 200}$
$_{\text{Ra}}\gamma_{\text{COUNT}(*),\text{ SUM(Rc)}}$
$\sigma_{\text{Ra} > 100}$
R
Project
({Ra,COUNT(∗)})
Filter
(SUM(Rc) > 200)
GroupBy
({Ra},{COUNT(∗),SUM(Rc)})
Sort
({Ra})
Filter
(Ra > 100)
TableScan
(R)
Project
({Ra,COUNT(∗)})
Filter
(SUM(Rc) > 200)
HashGroupBy
({Ra},{COUNT(∗),SUM(Rc)})
Filter
(Ra > 100)
TableScan
(R)

Note that (a) the aggregation functions of both γ and **GroupBy** are those different that appear in the **SELECT** and in the **HAVING** clauses, (b) the condition of **HAVING** clause becomes a **Filter** on the **GroupBy** and (c) a **Project** is required to produce the final result.

Example 6.5

```
SELECT  Ra AS O
FROM    R
WHERE   Ra < 100
    UNION
SELECT  Sa AS O
FROM    S
WHERE   Sa > 200;
```





6.7 Concurrency and Recovery

The aim of this section is to provide a very brief and schematic introduction to concurrency control, i.e. the ways in which a DBMS ensures that simultaneously executed transactions do not interfere with each other, and also to present the recovery algorithms used to protect the database from transaction, system, and media failures. Much research has been done on these topics and in the bibliographic notes we refer the interested reader to some of the most relevant literature.

6.7.1 Strict Two-Phase Locking

When two or more transactions execute concurrently, their database operations are *interleaved*, that is, operations from one program can execute in between operations from another program. This interleaving can cause programs to produce unpredictable results, or to *interfere*, as has been shown in Example 4.2. One way to avoid interference problems is not to allow transactions to be interleaved in any way, and to adopt serial execution. An execution is said to be *serial* if, for every pair of transactions, all the operations of one transaction execute before any of the operations of the other. However, only in the simplest systems is serial execution a practical way to avoid interference. In general, since concurrency means interleaved executions, it is sufficient that the system guarantees that the resulting execution will have the same effect as serial ones. Such executions are called *serializable*.

■ Definition 6.4

An execution of a set of transactions is *serializable* if its effect is exactly the same as a serial execution of the transactions.

The DBMS module that controls the concurrent execution of transactions is called the *scheduler*. There are many scheduling algorithms to obtain serializability; a very simple one is the *strict two-phase locking* protocol (*strict 2PL*). This is the most popular protocol of this type in commercial products.

The idea behind locking is intuitively simple: each data item used by a transaction has a lock associated with it, a *read* (*shared*) or a *write* (*exclusive*) lock, and strict 2PL protocol follows two rules:

1. If a transaction wants to *read* (respectively, *write*) a data item, it first request a *shared* (respectively, *exclusive*) lock on the data item.
   Before a transaction T_i can access a data item, the scheduler first examines the lock associated with the data item. If no other transaction holds the lock, then the data item is locked. If, however, another transaction T_j holds a lock in *conflict* (two actions on the same data item conflict if at least one of them is a write), then T_i must wait until T_j releases it. In some systems, whole files are locked while a transaction is accessing them, but this solution does not allow sufficient concurrency, i.e. the lock *granularity* is too coarse. Record-level locking or page-level locking is used instead in large time-sharing systems.
2. All locks held by a transaction are released when the transaction is completed.

A strict two-phase locking protocol is so named because in locking operations there is a phase in which the transaction acquires all the looks it needs; when the transaction is completed, then it enters into its unlocking phase. Request to acquire and release locks are usually automatically inserted into transactions by the DBMS.

The following theorem shows the importance of strict 2PL.

■ Theorem 6.1

A strict 2PL protocol ensures serializability.

Two-phase locking is simple, but the scheduler needs a strategy to detect *deadlocks*. Consider a situation in which transaction T_i has locked item A and needs a lock on item B, while at the same time transaction T_j has locked item B and now needs a lock on item A. Deadlock occurs because neither transaction can proceed. A strategy to detect deadlocks uses a dependency graph indicating which transactions are waiting until which others are completed. A cycle in the graph indicates deadlock. Another strategy uses *timeout*: if a transaction has been waiting too long for a lock, the the scheduler simply presumes that deadlock has occurred and aborts the transaction.

6.7.2 Recovery Algorithms

Unpredictable results can also occur after a failure. For example, suppose that the system failed while transaction T_1 is transferring money from John's checking account into John's saving account; in particular suppose failure occurred when T_1 has taken the money out of the checking account, but not yet deposited it in the savings account.

To ensure that the database is protected against failures, the system must be able to "undo" operations made by aborted transactions. To do this, the system keeps a *log* on the disk which contains the information necessary to reconstruct the most recent database state before the occurrence of a failure. For each write in the database, the log will contain the identifier of the transaction that performed the write, a copy of the newly written page (called the *after-image*), and a copy of the page in the database that was overwritten by the write (called the *before-image*). Recovery algorithms differ in the information they store in the log, in how they structure that information and, more important, in the time at which they write pages in the stable database.

We say that a recovery algorithm requires an *undo* if an update of some uncommitted transaction is recordered in the stable database. Should a transaction or a system failure occur, the recovery algorithm must *undo* the updates by copying the before-image of the page from the log to the stable database.

We say that a recovery algorithm requires *redo* if a transaction is committed before all of its updates are installed in the stable database. Should a system failure occur after the transaction commits but before the updates are installed in the stable database, the recovery algorithm must *redo* the updates by copying the after-image of the page from the log to the stable database.

Thus, we can classify recovery algorithms into four categories: (1) those that require both undo and redo; (2) those that require undo but not redo; (3) those that require redo but not undo; and (4) those that require neither undo nor redo. Every recovery algorithm must observe two fundamental rules: the *commit rule* and the *log-ahead rule*.

The *commit rule*, called also the *redo rule*, requires that before a transaction can commit, the after-images produced by the transaction must be in stable storage (e.g., in the stable database or log). If this rule is not followed, a system failure shortly after the transaction commit will lose the last after-images, making it impossible to redo the transaction.

The *log-ahead rule*, called also the *undo rule*, requires that if a database page is modified before the end of a transaction, its before-image must have been previously recorded in the log. This rule enables a transaction to be undone in case of abort by reconstituting the before-image from the log.

We shall describe just one category of recovery algorithms: the *No-Undo/Redo Algorithm*. The algorithm uses three lists of transactions that are assumed persistent (they can be included in the log): the *list of active transactions*, the *list of aborted transactions*, and the *list of committed transactions*. We assume that no locks are released by a transaction until after a commit or abort has been processed. This ensures that transactions do not read uncommitted data. For simplicity, we ignore the recovering from media failure situations[1]. Here is how the *No-Undo/Redo Algorithm* works.

To perform a write operation, the system enters the transaction identifier, the data-item identifier, and the new value to be written (*after-image*) in the log. Therefore a "write" does not actually write into the database, it just creates an entry in the log.

To perform a read, the system will get the value from the database if the transaction has not previously written the same item, otherwise the after-image from the log is used.

To commit, the system (a) adds the transaction identifier to the commit list, (b) writes the after-images from the log to the database, and (c) takes off the active list the transaction identifier.

1. Recovery algorithms for media failure generally require redo. Most such algorithms keep a stable copy of the database, called the *backup* copy, which is almost certainly out-of-date. So the recovery algorithm must redo committed updates that occurred after the backup was created.

To abort, the system adds the transaction identifier to the abort list and removes it from the active list. Since the database has not been changed, nothing further needs to be done (this is the no-undo part of the algorithm).

To restart after a system failure, the log is read backwards. For each transaction whose identifier is in the list of active transactions, the transaction will be committed if the identifier is also in the list of committed transactions, otherwise it will be aborted.

With this algorithm, the pages written by a transaction are not written into the database until after the transaction commits. Thus, the algorithm does not have to *undo* a write, but the restart may require to *redo* the write by copying the after-image of the page from the log to the database.

6.8 Exercises

1. Give a physical query plan for the following queries based on the database schema

```
Books(BookCode, Title, EditorName, Year)
Authors(BookCode, AuthorName, Nationality)
Editors(EditorName, EAddress, EPhone)
BookVolumes(BookCode, LibraryCode, NoCopies)
Libraries(LibraryCode, LibAddress, LibName)
Loans(BookCode, LibraryCode, UserCode, LoanData, DueData)
Users(UserCode, UserName, UAddress, UPhone)
```

considering two cases: without using indexes and using indexes that you believe are useful to speed-up the query.

(a)
```
SELECT     DISTINCT Title
FROM       Books b, Editor e
WHERE      b.EditorName = e.EditorName
           AND Year > 1970 AND EditorName = 'Pearson'
ORDER BY   Title;
```

(b)
```
SELECT     Title, EditorName, Year
FROM       Books b, Authors a
WHERE      b.BookCode = a.BookCode
           AND Year > 1970 AND a.AuthorName = 'Codd'
ORDER BY   Year;
```

(c)
```
SELECT     Nationality, COUNT(*)
FROM       Books b, Authors a
WHERE      b.BookCode = a.BookCode
           AND Year > 1970
GROUP BY   Nationality
```

(d)
```
SELECT     b.LibraryCode, LibAddress, COUNT(*)
FROM       Libraries b, Loans p
WHERE      b.LibraryCode = p. LibraryCode
           AND LibName = 'National' AND DueData = '01/03/2005'
GROUP BY   b.LibraryCode , LibAddress
HAVING     COUNT(*) < 100
ORDER BY   LibAddress;
```

Chapter 7

FINAL REMARKS

7.1 Introduction

We have discussed the major aspects of database systems: a) the modeling of the information about the world that the database represents; b) the languages and facilities provided by database management systems; c) the formalism, theory, and algorithms used in relational database design; d) the data structures to store and access efficiently sets of data and relationships between them. In addition, we have presented some of the techniques used in database management systems to process and optimize queries specified in a declarative language, and to implement concurrency control and recovery. The evolution of database systems is recalled in Figure 7.1.

Before leaving this tutorial, we would like to mention three other topics that, for reasons of space, it has been impossible to consider: *distributed database systems*, *information retrieval systems*, and *deductive databases*.

The main objective of a *distributed* DBMS is to provide the capabilities to access data, which are physically located at several different sites, without needing to know at which site a particular piece of data is stored. This property is known as *location transparency*, i.e. to the user a distributed DBMS will appear exactly the same as a centralized DBMS. Location transparency permits data to be moved between sites without affecting existing applications. A typical application in which this type of situation is found is a bank in which checking accounts are stored at the branch nearest to where the account holders live, while at the same time it must be possible to access these accounts from automated teller machines located at all branches of the bank.

Distributed systems are usually made up of similar relational DBMSs, and will also provide facilities for distributed query optimization, distributed concurrency control, and distributed transaction management. In addition, such systems must cope with multiple copies of the same data at different sites, ensuring their mutual consistency at all times.

Examples of such systems are the distributed versions of *System R* and INGRES, renamed respectively *System R** and INGRES-STAR. Research is still under way on how heterogeneous DBMSs can share data.

Information retrieval systems (IRSs) deal with document representation, storage, and access. Since the input information will include the natural language text of documents or of document abstracts, these systems are also known as *document* or *text retrieval* systems. The field of information retrieval has studied the problem of searching collections of text documents since the 1950s and developed independently of

Year	Result
1962	C. Bachman develops the Integrated Data Management System (IDMS) network DBMS.
1966	Indexed Sequential Access Method (ISAM) is developed at IBM.
1968	IBM releases the Information Management System (IMS) hierarchical DBMS, designed to manage large bills of material for the construction of the spacecraft for the Apollo program.
1970	E. F. Codd introduces the Relational Model along with the Relational Algebra and Relational Calculus.
1971	CODASYL publishes the Data Base Task Group (DBTG) report on the network model.
1972	R. Bayer and E. McCreight publish paper on B-tree, the basic indexing mechanism in modern database systems.
1972	Boyce-Codd normal form for database design is introduced.
1973	C. Bachman receives the Turing Award for his work on network databases.
1974	The University of California at Berkeley distributes the Ingres DBMS using the QUEL query language.
1974	The theory of functional dependencies and relational normalization theory taking shapes.
1975	IBM develops System R, an experimental relational DBMS that introduces the Structured Query Language (SEQUEL, later called SQL).
1975	IBM develops Query By Example (QBE), the first graphical query language.
1976	Eswaran, Gray, Lorie and Traiger define isolation levels, serializability, and two-phase locking.
1976	P. Chen introduces the Entity-Relationship model.
1977	A. Makinouchi describes a nested relational model, a precursor of the object-relational model.
1977	Relational Software Inc., later to become Oracle Corporation, is founded and is the first company to release a relational DBMS based on the IBM System R model and utilizing SQL.
1979	H. Gallaire and J. Minker introduce logic-based database, also known as deductive databases.
1981	E. Codd receives the Turing Award for his contributions to database theory.
1983	IBM releases the DB2 relational DBMS.
1985	Active databases are introduced.
1985	Object-oriented and object-relational databases are introduced.
1986	Ingres releases IngresStar, the first distributed relational system.
1986	LDL, a logical-based database language is implemented at MCC Corporation.
1986	GemStone releases the first object-oriented DBMS.
1992	Open Database Connectivity (ODBC) is developed allowing machines to trasparently communicate with multiple DBMSs.
1995	Datacube OLAP operators are introduced.
1995	The semi-structured data model is developed.
1998	Unified Modeling Language (UML) is standardized as a modeling tools for software and data design.
1998	J. Gray receives the Turing Award for his contributions to the fields of databases and transaction processing.
1998	eXtensible Markup Language (XML) is developed as a standard for information interchange, particularly among DBMS.
1999	The SQL3 standard is published.

Figure 7.1: The time table of database principles and applications

database field. The Web has made the document search an everyday operation for most people and led to renewed research on the topic.

The goal of an IRS is to find documents that contain information which meets the specifics of a search request. Since the documents stored will generally be on a range of topics and be written by different authors, and users are seeking particular information, they will find some of the documents retrieved useful, or *relevant*, and others not. The problem is how should a collection of documents be organized so that a user can find all the relevant documents and only those.

Another issue to which much attention is being given is how traditional data processing records can be managed together with text data, graphic data, such as drawings and pictures, and semistructured data and how these new requirements will affect the DBMS architecture.

Finally, the objective of *deductive databases* is the integration of logic programming and databases. In this approach, a database schema, usually a relational one, constraints and queries are formulated in a subset of first-order logic formulas, called *Horn clauses*, using PROLOG as the logic programming language. A query is considered as a theorem to be demonstrated, and the main problem in deductive DBMSs is the implementation of inference methods that can be executed efficiently. Several alternative architectures have been proposed for extending current relational DBMSs with deductive capabilities, and references on this topic appear in the bibliographic notes.

7.2 Bibliographic Notes

Even a brief bibliography on database systems would contain hundreds of entries. The following list is intended only to indicate some of the most relevant publications which discuss the major topics presented in this report. Most of the references cited include extensive bibliographies, which can be used by those wishing to explore a particular topic in depth.

Database: The Designer Perspective

The growing use of computerized information systems based on database technology has led to greater attention being paid to database design methodologies and automated tools to support them. Examples of such methodologies and tools are described in [Batini et al., 1992], [Maciaszek, 2001] and [Teorey, 1990].

The *entity-relationship model* was the first example of a data model for conceptual modeling, and it has found wide acceptance in database design [Chen, 1976]. In [Hull and King, 1987] we have surveys of the data models for conceptual modeling considered by many to be the most influential ones.

DBMS: The User's Perspective

[Albano et al., 2005] [Date, 1995] [Elmasri and Navathe, 2000] [O'Neil and O'Neil, 2000] [Ramakrishnan and Gehrke, 2003] [Silberschatz et al., 2010] [Ullman and Widom, 1997] [van der Lans, 1993] [Kifer et al., 2006] and [Celko, 1996] are general sources for a discussion of the objectives and functions of DBMSs, and for a presentation of classical data models, data manipulation languages and SQL.

The theory of relational databases is presented in all the previous books and in detail in [Maier, 1983] [Abiteboul et al., 1995]. The proposal of the relational data model and the first discussion of relational database design theory appeared in [Codd, 1970].

Extensions to the relational data model to support some aspects of object orientation have been made recently. Examples of this approach are extension of the relational data model with the support of abstract data types and hierarchical objects [Abiteboul et al., 1995].

DBMS: The System Perspective

There are several papers describing the overall system structure of specific DBMS. For example, Ashtrahan et al. [Ashtrahan et al., 1976] discusses *System R*. Chamberlin et al. [Blasgen et al., 1981] reviews *System R* in retrospect. Stonebraker et al. [Stonebraker et al., 1976] describes the implementation of *Ingres*, and Stonebraker [Stonebraker, 1980] reviews *Ingres* in retrospect.

Basic file organizations are discussed in [Albano, 2001] [Ramakrishnan and Gehrke, 2003] [Garcia-Molina et al., 1999]. A discussion of several dynamic hashing techniques is reported in [Enbody and Du, 1988]. B-tree indexes were presented in [Bayer and Creight, 1972].

Query optimization is discussed in depth in [Albano, 2001] [Ramakrishnan and Gehrke, 2003] [Garcia-Molina et al., 1999] but the topic is discussed in all text books on DBMSs. The seminal paper by Selinger et al. [Selinger et al., 1979] describes access path selection in *System R*.

The main source for concurrency and recovery is the excellent book [Bernstein et al., 1987], but the topic is discussed in all text books on DBMSs, and in particular [Gray and Reuter, 1997].

The major issues concerning distributed database systems are discussed in [Ceri and Pelagatti, 1984] and [Ozsu and Valduriez, 1991]. Standard references for information retrieval systems are [Salton, 1989] [Van Rijsbergen, 1979]. Deductive databases are discussed in [Ullman and Widom, 1997]. A book dedicated to deductive databases is [Ceri et al., 1990].

Advanced Topics

There are several books about designing and implementing a data warehouse, a decision support database with historical, nonvolatile data, pulled together primarily from operational business systems, structured and tuned to facilitate analysis of the performance of key business processes, worthy of improvement. Three excellent sources are [Kimball and Ross, 2002], [Ponniah, 2001], and [Ballard et al., 2006]. A useful source of information about data warehouse is the page www.ondelette.com/OLAP/dwbib.html.

Object database languages are discussed in [Albano et al., 1997] [Bancilhon et al., 1992] [Ullman and Widom, 1997] [Elmasri and Navathe, 2000], and examples of approaches to the design of object DBMSs appear in [Kim, 1990] [Kim, 1995] [Cattel, 1994a] [Cattel, 1994b]. The definition of the Galileo language is given in [Albano et al., 1985] and [Albano et al., 2000]. SQL:1999 is discussed in [Melton and Simon, 2000]. A useful source of information about object databases is the page www.odbmsfacts.com at Barry & Associatiates Inc. Web and databases are discussed in [Morrison and Morrison, 2000]. An excellent reference that takes the database view information processing on the Web and covers recent works on semistructured data, including XML, is [Abiteboul et al., 2000].

Acknowledgments

I am very grateful to G. Ghelli, and R. Orsini who reviewed parts of the material and made many valuable observations.

BIBLIOGRAPHY

Abiteboul, S., Buneman, P., and Suciu, D. (2000). *Data on the web. From Relations to semistructured data and XML*. Morgan Kaufmann Publishers, San Mateo, California. 97

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Database Foundations*. Addison-Wesley, Reading, Massachusetts. 96

Albano, A. (2001). *Costruire sistemi per basi di dati*. Addison-Wesley, Milano. 96

Albano, A., Antognoni, G., and Ghelli, G. (2000). View operations on objects with roles for a statically typed database language. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):548–567. 97

Albano, A., Cardelli, L., and Orsini, R. (1985). Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260. Also in S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990. 97

Albano, A., Ghelli, G., and Orsini, R. (1997). *Basi di dati relazionali e a oggetti*. Zanichelli, Bologna. 97

Albano, A., Ghelli, G., and Orsini, R. (2005). *Fondamenti di basi di dati*. Zanichelli, Bologna. 96

Ashtrahan, M., Blasgen, M., Chamberlin, D., Eswaran, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., and Watson, V. (1976). System R: A relational approach to data base management. *ACM Transactions on Database Systems*, 1(2):97–137. 96

Ballard, C., Farrell, D. M., Gupta, A., Mazuela, C., and S.Vohnik (2006). *Dimensional Modeling: In a Business Intelligence Environment*. IBM, www.redbooks.ibm.com/redbooks/pdfs/sg247138.pdf. 97

Bancilhon, F., Delobel, C., and Kanellakis, P., editors (1992). *Building an Object-oriented Database System. The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, California. 97

Batini, C., Ceri, S., and Navathe, S. (1992). *Conceptual Database Design. An Entity-Relationship Approach*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California. 96

Bayer, R. and Creight, E. M. (1972). Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189. 96

Beeri, C. and Bernstein, P. (1979). Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4(1):30–59. 39

Bernstein, P. (1976). Synthesizing third normal form relations from functional dependencies. *ACM Transactions on Database Systems*, 1(4):277–298. 45

Bernstein, P., Goodman, N., and Hadzilacos, V. (1987). *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, Menlo Park, California. 96

Biskup, J., Dayal, U., and Bernstein, P. (1979). Synthesizing independent database schemas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 143–152. 45

Blasgen, M., Chamberlin, D., Gray, J., King, W., Lindsay, B., Lorie, R., Mehl, J., Price, T., Putzolu, G., Schkolnick, M., Selinger, P., Slutz, D., Traiger, I., Wade, B., and Yost, R. (1981). System R: an architectural overview. *IBM System Journal*, 20(1):41–62. 96

Cattel, R. (1994a). *Object Data Management. Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, Reading, Massachusetts. 97

Cattel, R. (1994b). *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Mateo, California. 97

Celko, J. (1996). *Joe Celko's SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann Publishers, San Mateo, California. 96

Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic Programming and Data Bases*. Springer-Verlag, Berlin. 97

Ceri, S. and Pelagatti, G. (1984). *Distributed Databases: Principles and Systems*. McGraw-Hill, New York. 97

Chen, P. (1976). The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36. 96

Codd, E. (1970). A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387. 37, 96

Date, C. (1995). *An Introduction to Database Systems*. Addison-Wesley, Reading, Massachusetts, sixth edition. 96

Elmasri, R. and Navathe, S. (2000). *Fundamentals of Database Systems*. Addison-Wesley, Reading, Massachusetts, third edition. 96, 97

Enbody, R. and Du, H. (1988). Dynamic hashing schemes. *ACM Computing Surveys*, 20(2):85–113. 96

Garcia-Molina, H., Ullman, J. D., and Widom, J. (1999). *Database System Implementation*. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 96

Gray, J. and Reuter, A. (1997). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, third edition. 96

Hull, R. and King, R. (1987). Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260. 96

Kifer, M., Bernstein, A., and Lewis, P. M. (2006). *Database Systems. An Application-Oriented Approach*. Pearson Addison Wesley, New York, second edition. 96

Kim, W. (1990). *Introduction to Object Oriented Databases*. MIT Press, Cambridge, Massachusetts. 97

Kim, W. (1995). *Modern Database Systems. The Object Model, Interoperability, and Beyond*. Addison-Wesley, Reading, Massachusetts. 97

Kimball, R. and Ross, M. (2002). *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling (Second Edition)*. Wiley, New York. 97

Maciaszek, L. A. (2001). *Requirements Analysis and System Design. Developing Information Systems with UML*. Addison-Wesley, Reading, Massachusetts. 96

Maier, D. (1983). *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland. 47, 96

Melton, J. and Simon, A. R. (2000). *SQL:1999 – Understanding Relational Language Components*. Morgan Kaufmann Publishers, San Mateo, California. 97

Morrison, M. and Morrison, J. (2000). *Database-Driven Web Sites*. Course Tecnology,, London. 97

O'Neil, P. and O'Neil, E. (2000). *Data Base. Principles, Programming, and Performance*. Morgan Kaufmann Publishers, San Mateo, California, second edition. 96

Ozsu, T. and Valduriez, P. (1991). *Principles of Distributed Database Systems*. Prentice Hall International, Inc., London. 97

Ponniah, P. (2001). *Data Warehousing Fundamentals*. Wiley, New York. 97

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill, New York, third edition. 96

Salton, G. (1989). *Automatic Text Processing. The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, Massachusetts. 97

Selinger, P., Ashtrahan, M., Chamberlin, D., Lorie, R. A., and Price, T. (1979). Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, MA. 96

Silberschatz, A., Korth, H., and Sudarshan, S. (2010). *Database System Concepts*. McGraw-Hill, New York, sixth edition. 96

Stonebraker, M. (1980). Retrospection on a database system. *ACM Transactions on Database Systems*, 5(2):225–240. 96

Stonebraker, M., Wong, E., Kreps, P., and Held, G. (1976). The design and implementation of Ingres. *ACM Transactions on Database Systems*, 1(3):189–222. 96

Teorey, T. (1990). *Database Modeling and Design. The Entity-Relationship Approach*. Morgan Kaufmann Publishers, San Mateo, California. 96

Tsou, D. and Fischer, P. (1982). Decomposition of a relation scheme into Boyce-Codd Normal Form. *ACM SIGACT News*, 14(3):23–29. 46

Ullman, J. (1989). *Principles of Database and Knowledge Base Systems*, volume I-II. Computer Science Press, Rockville, Maryland. 43

Ullman, J. D. and Widom, J. (1997). *A First Course in Database System*. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 96, 97

van der Lans, R. (1993). *Introduction to SQL*. Addison-Wesley, Reading, Massachusetts, second edition. 96

Van Rijsbergen, C. (1979). *Information Retrieval*. Butterworths, London, second edition. 97

INDEX