Chapter 2
Database Concepts

This chapter introduces the basic database concepts, covering modeling, design, and implementation aspects. Section **??** begins by describing the concepts underlying database systems and the typical four-step process used for designing them, starting with requirements specification, followed by conceptual, logical, and physical design. These steps allow a separation of concerns, where requirements specification gathers the requirements about the application and its environment, conceptual design targets the modeling of these requirements from the perspective of the users, logical design develops an implementation of the application according to a particular database technology, and physical design optimizes the application with respect to a particular implementation platform. Section **??** presents the Northwind case study that we will use throughout the book. In Sect. **??**, we review the entity-relationship model, a popular conceptual model for designing databases. Section **??** is devoted to the most used logical model of databases, the relational model. Finally, physical design considerations for databases are covered in Sect. **??**.

The aim of this chapter is to provide the necessary knowledge to understand the remaining chapters in this book, making it self-contained. However, we do not intend to be comprehensive and refer the interested reader to the many textbooks on the subject.

2.1 Database Design

Databases constitute the core component of today's information systems. A **database** is a shared collection of logically related data, and a description of that data, designed to meet the information needs and support the activities of an organization. A database is deployed on a **database management system** (DBMS), which is a software system used to define, create, manipulate, and administer a database.

Designing a database system is a complex undertaking typically divided into four phases, described next.

- **Requirements specification** collects information about the users' needs with respect to the database system. A large number of approaches for requirements specification have been developed by both academia and practitioners. These techniques help to elicit necessary and desirable system properties from prospective users, to homogenize requirements, and to assign priorities to them.
- **Conceptual design** aims at building a user-oriented representation of the database that does not contain any implementation considerations. This is done by using a **conceptual model** in order to identify the relevant concepts of the application at hand. The entity-relationship model is one of the most often used conceptual models for designing database applications. Alternatively, object-oriented modeling techniques can also be applied, based on the UML (Unified Modeling Notation) notation.
- **Logical design** aims at translating the conceptual representation of the database obtained in the previous phase into a **logical model** common to several DBMSs. Currently, the most common logical model is the relational model. Other logical models include the object-relational model, the object-oriented model, and the semistructured model. In this book, we focus on the relational model.
- **Physical design** aims at customizing the logical representation of the database obtained in the previous phase to a **physical model** targeted to a particular DBMS platform. Common DBMSs include SQL Server, Oracle, DB2, MySQL, and PostgresSQL, among others.

A major objective of this four-level process is to provide **data independence**, that is, to ensure as much as possible that schemas in upper levels are unaffected by changes to schemas in lower levels. Two kinds of data independence are typically defined. **Logical data independence** refers to immunity of the conceptual schema to changes in the logical one. For example, changing the structure of relational tables should not affect the conceptual schema, provided that the requirements of the application remain the same. **Physical data independence** refers to immunity of the logical schema to changes in the physical one. For example, physically sorting the records of a file on a disk does not affect the conceptual or logical schema, although this modification may be perceived by the user through a change in response time.

In the following sections, we briefly describe the entity-relationship model and the relational models, to cover the most widely used conceptual and logical models, respectively. We then address physical design considerations. Before doing this, we introduce the use case we will use throughout the book, which is based on the popular Northwind relational database. In this chapter, we explain the database design concepts using this example. In the next chapter, we will use a data warehouse derived from this database, over which we will explain the data warehousing and OLAP concepts.

2.2 The Northwind Case Study

The Northwind company exports a number of goods. In order to manage and store the company data, a relational database must be designed. The main characteristics of the data to be stored are the following:

- Customer data, which must include an identifier, the customer's name, contact person's name and title, full address, phone, and fax.
- Employee data, including the identifier, name, title, title of courtesy, birth date, hire date, address, home phone, phone extension, and a photo. Photos will be stored in the file system, and a path to the photo is required. Further, employees report to other employees of higher level in the company's organization.
- Geographic data, namely, the territories where the company operates. These territories are organized into regions. For the moment, only the territory and region description must be kept. An employee can be assigned to several territories, but these territories are not exclusive to an employee: Each employee can be linked to multiple territories, and each territory can be linked to multiple employees.
- Shipper data, that is, information about the companies that Northwind hires to provide delivery services. For each one of them, the company name and phone number must be kept.
- Supplier data, including the company name, contact name and title, full address, phone, fax, and home page.
- Data about the products that Northwind trades, such as identifier, name, quantity per unit, unit price, and an indication if the product has been discontinued. In addition, an inventory is maintained, which requires to know the number of units in stock, the units ordered (i.e., in stock but not yet delivered), and the reorder level (i.e., the number of units in stock such that, when it is reached, the company must produce or acquire). Products are further classified into categories, each of which has a name, a description, and a picture. Each product has a unique supplier.
- Data about the sale orders. The information required includes the identifier, the date at which the order was submitted, the required delivery date, the actual delivery date, the employee involved in the sale, the customer, the shipper in charge of its delivery, the freight cost, and the full destination address. An order can contain many products, and for each of them the unit price, the quantity, and the discount that may be given must be kept.

2.3 Conceptual Database Design

The entity-relationship (ER) model is one of the most often used conceptual models for designing database applications. Although there is general agreement about the meaning of the various concepts of the ER model, a number of different visual notations have been proposed for representing these concepts. Appendix **??** shows the notations we use in this book.

Figure **??** shows the ER model for the Northwind database. We next introduce the main ER concepts using this figure.

Entity types are used to represent a set of real-world objects of interest to an application. In Fig. **??**, Employees, Orders, and Customers are examples of entity types. An object belonging to an entity type is called an **entity** or an **instance**. The set of instances of an entity type is called its **population**. From the application point of view, all entities of an entity type have the same characteristics.

In the real world, objects do not live in isolation; they are related to other objects. **Relationship types** are used to represent these associations between objects. In our example, Supplies, ReportsTo, and HasCategory are examples of relationship types. An association between objects of a relationship type is called a **relationship** or an **instance**. The set of associations of a relationship type is called its **population**.

The participation of an entity type in a relationship type is called a **role** and is represented by a line linking the two types. Each role of a relationship type has associated with it a pair of **cardinalities** describing the minimum and maximum number of times that an entity may participate in that relationship type. For example, the role between Products and Supplies has cardinalities (1,1), meaning that each product participates exactly once in the relationship type. The role between Supplies and Suppliers has cardinality (0,n), meaning that a supplier can participate between 0 and n times (i.e., an undetermined number of times) in the relationship. On the other hand, the cardinality (1,n) between Orders and OrderDetails means that each order can participate between 1 and n times in the relationship type. A role is said to be **optional** or **mandatory** depending on whether its minimum cardinality is 0 or 1, respectively. Further, a role is said to be **monovalued** or **multivalued** depending on whether its maximum cardinality is 1 or n, respectively.

A relationship type may relate two or more object types: It is called **binary** if it relates two object types, and ***n*-ary** if it relates more than two object types. In Fig. **??**, all relationship types are binary. Depending on the maximum cardinality of each role, binary relationship types can be categorized into **one-to-one**, **one-to-many**, and **many-to-many** relationship types. In Fig. **??**, the relationship type Supplies is a one-to-many relationship, since one product is supplied by at most one supplier, whereas a supplier may supply several products. On the other hand, the relationship type OrderDetails is many-to-many, since an order is related to one or more products, while a product can be included in many orders.

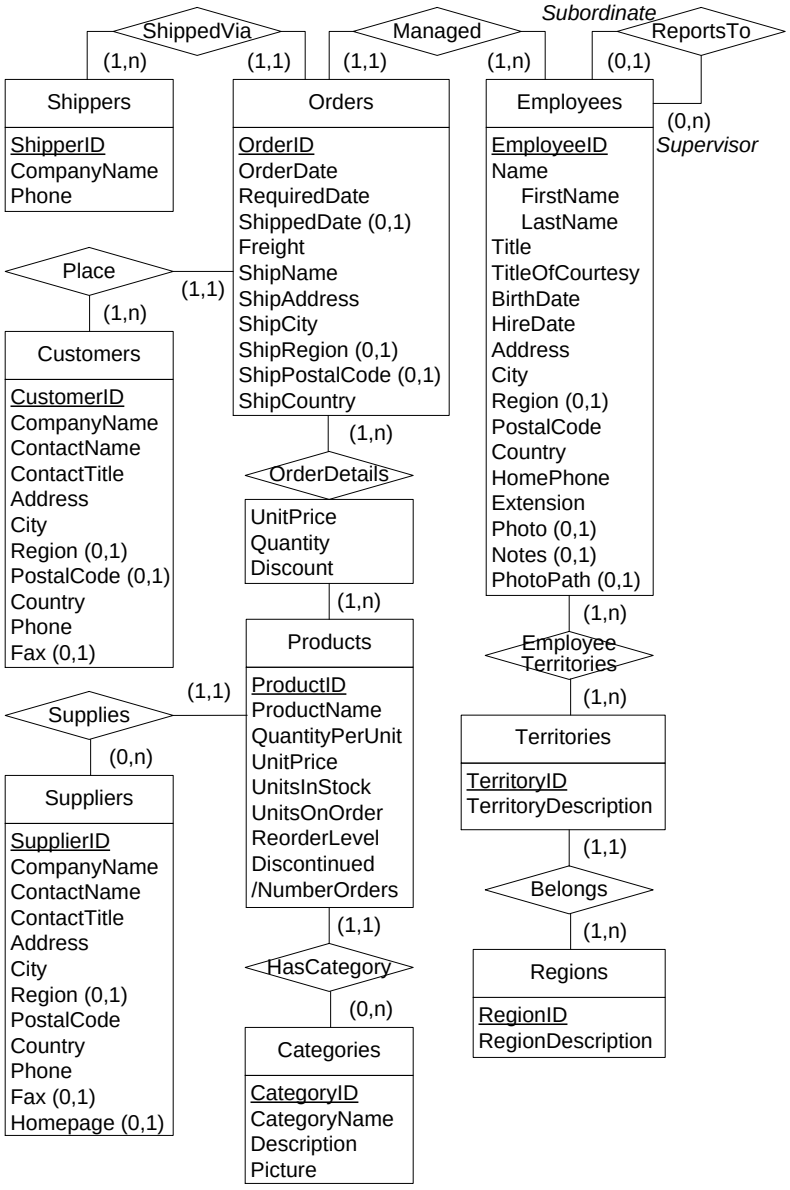

Fig. 2.1. Conceptual schema of the Northwind database

It may be the case that the same entity type is related more than once in a relationship type, as is the case for the ReportsTo relationship type. In this case, the relationship type is called **recursive**, and **role names** are

necessary to distinguish between the different roles of the entity type. In Fig. **??**, Subordinate and Supervisor are role names.

Both objects and the relationships between them have a series of structural characteristics that describe them. **Attributes** are used for recording these characteristics of entity or relationship types. For example, in Fig. **??** Address and Homepage are attributes of Suppliers, while UnitPrice, Quantity, and Discount are attributes of OrderDetails.

Like roles, attributes have associated **cardinalities**, defining the number of values that an attribute may take in each instance. Since most of the time the cardinality of an attribute is (1,1), we do not show this cardinality in our schema diagrams. Thus, each supplier will have exactly one Address, while they may have at most one Homepage. Therefore, its cardinality is (0,1). In this case, we say the attribute is **optional**. When the cardinality is (1,1) we say that the attribute is **mandatory**. Similarly, attributes are called **monovalued** or **multivalued** depending on whether they may take at most one or several values, respectively. In our example, all attributes are monovalued. However, if it is the case that a customer has one or more phones, then the attribute Phone will be labeled (1,n).

Further, attributes may be composed of other attributes, as shown by the attribute Name of the entity type Employees in our example, which is composed of FirstName and LastName. Such attributes are called **complex attributes**, while those that do not have components are called **simple attributes**. Finally, some attributes may be **derived**, as shown for the attribute NumberOrders of Products. This means that the number of orders in which a product participates may be derived using a formula that involves other elements of the schema, and stored as an attribute. In our case, the derived attribute records the number of times that a particular product participates in the relationship OrderDetails.

A common situation in real-world applications is that one or several attributes uniquely identify a particular object; such attributes are called **identifiers**. In Fig. **??**, identifiers are underlined; for example, EmployeeID is the identifier of the entity type Employees, meaning that every employee has a unique value for this attribute. In the figure, all entity type identifiers are simple, that is, they are composed of only one attribute, although it is common to have identifiers composed of two or more attributes.

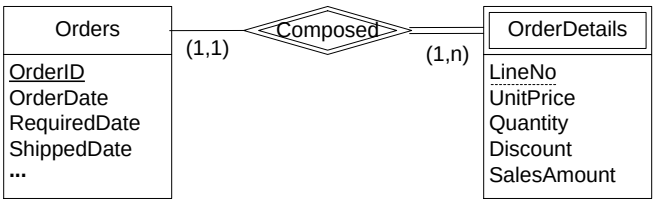

Fig. 2.2. Relationship type OrderDetails modeled as a weak entity type

Entity types that do not have an identifier of their own are called **weak entity types**, and are represented with a double line on its name box. In contrast, regular entity types that do have an identifier are called **strong entity types**. In Fig. **??**, there are no weak entity types. However, note that the relationship OrderDetails between Orders and Products can be modeled as shown in Fig. **??**.

A weak entity type is dependent on the existence of another entity type, called the **identifying** or **owner entity type**. The relationship type that relates a weak entity type to its owner is called the **identifying relationship type** of the weak entity type. A relationship type that is not an identifying relationship type is called a **regular relationship type**. Thus, in Fig. **??**, Orders is the owner entity type for the weak entity type OrderDetails, and Composed is its identifying relationship type. As shown in the figure, the identifying relationship type and the role that connects it to the weak entity type are distinguished by their double lines. Note that identifying relationship types have cardinality (1,1) in the role of the weak entity type and may have (0,n) or (1,n) cardinality in the role of the owner.

A weak entity type typically has a **partial identifier**, which is a set of attributes that uniquely identifies weak entities that are related to the same owner entity. An example is attribute LineNo of OrderDetails, which stores the line number of each product in an order. Therefore, the same number can appear several times in different orders, although it is unique within each order. As shown in the figure, partial identifier attributes are underlined with a dashed line.

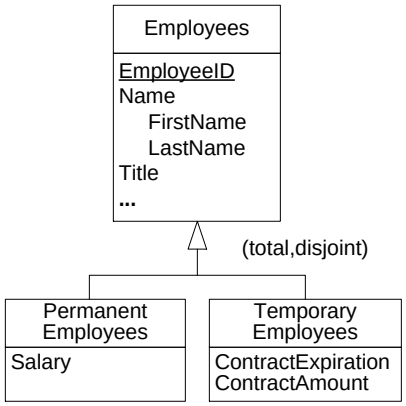

Fig. 2.3. Entity type Employees and two subtypes

Finally, owing to the complexity of conceptualizing the real world, human beings usually refer to the same concept using several different perspectives with different abstraction levels. The **generalization** (or **is-a**) **relationship** captures such a mental process. It relates two entity types, called the **super-**

type and the **subtype**, meaning that both types represent the same concept at different levels of detail. The Northwind database does not include a generalization. To give an example, consider Fig. **??**, in which we have a supertype Employees, and two subtypes, PermanentEmployees and TemporaryEmployees. The former has an additional attribute Salary, and the latter has attributes ContractExpiration and ContractAmount.

Generalization has three essential characteristics. The first one is **population inclusion**, meaning that every instance of the subtype is also an instance of the supertype. In our example, this means that every temporary employee is also an employee of the Northwind company. The second characteristic is **inheritance**, meaning that all characteristics of the supertype (e.g., attributes and roles) are inherited by the subtype. Thus, in our example, temporary employees also have, for instance, a name and a title. Finally, the third characteristic is **substitutability**, meaning that each time an instance of the supertype is required (e.g., in an operation or in a query), an instance of the subtype can be used instead.

Generalization may also be characterized according to two criteria. On the one hand, a generalization can be either **total** or **partial**, depending on whether every instance of the supertype is also an instance of one of the subtypes. In Fig. **??**, the generalization is total, since employees are either permanent or temporary. On the other hand, a generalization can be either **disjoint** or **overlapping**, depending on whether an instance may belong to one or several subtypes. In our example, the generalization is disjoint, since a temporary employee cannot be a permanent one.

2.4 Logical Database Design

In this section, we describe the most used logical data model for databases, that is, the relational model. We also study two well-known query languages for the relational model: the relational algebra and SQL.

2.4.1 The Relational Model

Relational databases have been successfully used for several decades for storing information in many application domains. In spite of alternative database technologies that have appeared in the last decades, the relational model is still the most often used approach for storing the information that is crucial for the day-to-day operation of an organization.

Much of the success of the relational model, introduced by Codd in 1970, is due to its simplicity, intuitiveness, and its foundation on a solid formal theory: The relational model builds on the concept of a mathematical relation,

which can be seen as a table of values and is based on set theory and first-order predicate logic. This mathematical foundation allowed the design of declarative query languages, and a rich spectrum of optimization techniques that led to efficient implementations. Note that in spite of this, only in the early 1980s the first commercial relational DBMS (RDBMS) appeared.

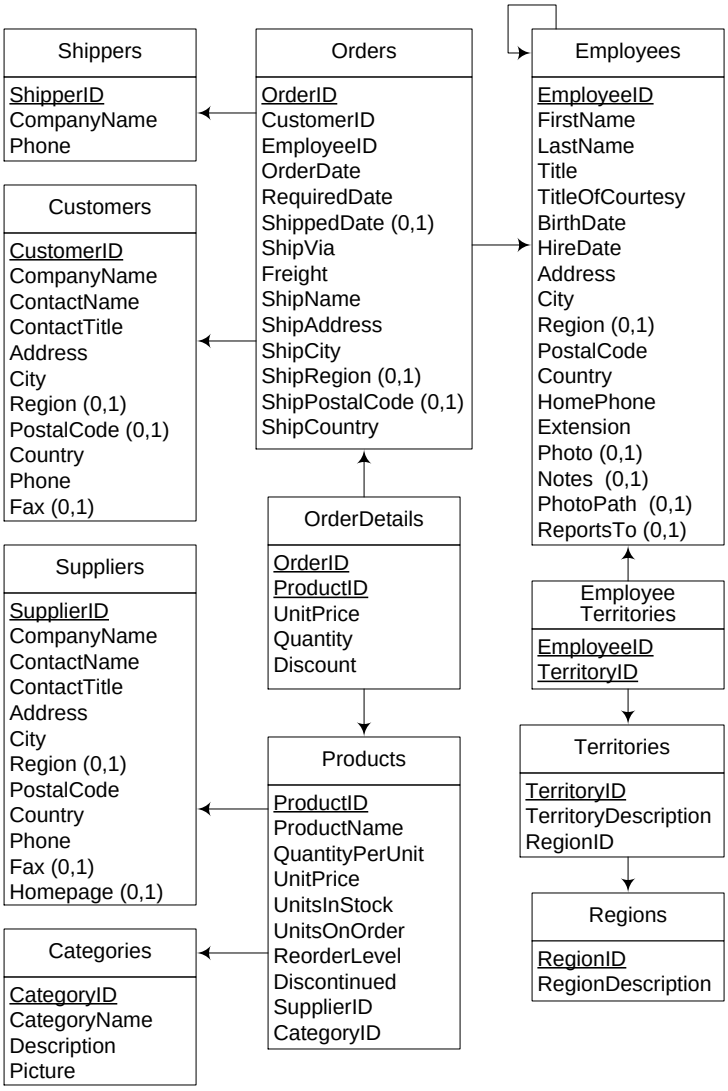

Fig. 2.4. A relational schema that corresponds to the conceptual schema in Fig. **??**

The relational model has a simple data structure, a **relation** (or **table**) composed of one or several **attributes** (or **columns**). Thus, a **relational schema** describes the structure of a set of relations. Figure **??** shows a relational schema that corresponds to the conceptual schema of Fig. **??**. As we will see later in this section, this relational schema is obtained by applying a set of translation rules to the corresponding ER schema. The relational schema of the Northwind database is composed of a set of relations, such as Employees, Customers, and Products. Each of these relations is composed of several attributes. For example, EmployeeID, FirstName, and LastName are some attributes of the relation Employees. In what follows, we use the notation $R.A$ to indicate the attribute A of relation R.

In the relational model, each attribute is defined over a **domain**, or **data type**, that is, a set of values with an associated set of operations, the most typical ones are integer, float, date, and string. One important restriction of the model is that attributes must be atomic and monovalued. Thus, complex attributes like Name of the entity type Employees in Fig. **??** must be split into atomic values, like FirstName and LastName in the table of the same name in Fig. **??**. Therefore, a relation R is defined by a schema $R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$, where R is the name of the relation, and each attribute A_i is defined over the domain D_i. The relation R is associated to a set of **tuples** (or **rows** if we see the relation as a table) $(t_1, t_2, \ldots, t_n)$. This set of tuples is a subset of the Cartesian product $D_1 \times D_2 \times \cdots \times D_n$, and it is sometimes called the **instance** or **extension** of R. The **degree** (or **arity**) of a relation is the number of attributes n of its relation schema.

The relational model allows several types of **integrity constraints** to be defined declaratively.

- An attribute may be defined as being **non-null**, meaning that **null values** (or blanks) are not allowed in that attribute. In Fig. **??**, only the attributes marked with a cardinality (0,1) allow null values.
- One or several attributes may be defined as a **key**, that is, it is not allowed that two different tuples of the relation have identical values in such columns. In Fig. **??**, keys are underlined. A key composed of several attributes is called a **composite key**, otherwise it is a **simple key**. In Fig. **??**, the table Employees has a simple key, EmployeeID, while the table EmployeeTerritories has a composite key, composed of EmployeeID and TerritoryID. In the relational model, each relation must have a **primary key** and may have other **alternate keys**. Further, the attributes composing the primary key do not accept null values.
- **Referential integrity** specifies a link between two tables (or twice the same table), where a set of attributes in one table, called the **foreign key**, references the primary key of the other table. This means that the values in the foreign key must also exist in the primary key. In Fig. **??**, referential integrity constraints are represented by arrows from the referencing table to the table that is referenced. For example, the attribute EmployeeID in table Orders references the primary key of the table Employees. This

ensures that every employee appearing in an order also appears in the table Employees. Note that referential integrity may involve foreign keys and primary keys composed of several attributes.

- Finally, a **check constraint** defines a predicate that must be valid when adding or updating a tuple in a relation. For example, a check constraint can be used to verify that in table Orders the values of attributes OrderDate and RequiredDate for a given order are such that OrderDate $\leq$ RequiredDate. Note that many DBMSs restrict check constraints to a single tuple: references to data stored in other tables or in other tuples of the same table are not allowed. Therefore, check constraints can be used only to verify simple constraints.

As can be seen, the above declarative integrity constraints do not suffice to express the many constraints that exist in any application domain. Such constraints must then be implemented using triggers. A **trigger** is a named event-condition-action rule that is automatically activated when a relation is modified. In this book, we shall see several examples of integrity constraints implemented using triggers. Notice that triggers can also be used to compute derived attributes, such as attribute NumberOrders in the table Products in Fig. **??**. A trigger will update the value of the attribute each time there is an insert, update, or delete in table OrderDetails.

The translation of a conceptual schema (written in the ER or any other conceptual model) to an equivalent relational schema is called a *mapping*. This is a well-known process, implemented in most database design tools. These tools use conceptual schemas to facilitate database design, and then automatically translate the conceptual schemas to logical ones, mainly into the relational model. This process includes the definition of the tables in various RDBMSs.

We now outline seven rules that are used to map an ER schema into a relational one.

Rule 1: A strong entity type E is mapped to a table T containing the simple monovalued attributes and the simple components of the monovalued complex attributes of E. The identifier of E defines the primary key of T. T also defines non-null constraints for the mandatory attributes. Note that additional attributes will be added to this table by subsequent rules. For example, the strong entity type Products in Fig. **??** is mapped to the table Products in Fig. **??**, with key ProductID.

Rule 2: Let us consider a weak entity type W, with owner (strong) entity type O. Assume W_{id} is the partial identifier of W, and O_{id} is the identifier of O. W is mapped in the same way as a strong entity type, that is, to a table T. In this case, T must also include O_{id} as an attribute, with a referential integrity constraint to attribute $O.O_{id}$. Moreover, the identifier of T is the union of W_{id} and O_{id}.

As an example, the weak entity type OrderDetails in Fig. **??** is mapped to the table of the same name in Fig. **??**. The key of the latter is composed

of the attributes OrderID and LineNo, where the former is a foreign key referencing table Orders.

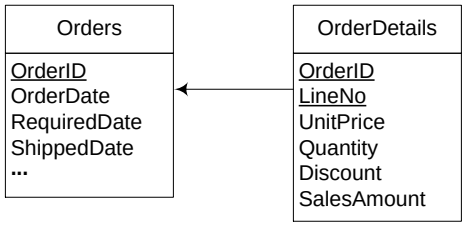

Fig. 2.5. Relationship translation of the schema in Fig. **??**

Rule 3: A regular binary one-to-one relationship type R between two entity types E_1 and E_2, which are mapped, respectively, to tables T_1 and T_2 is mapped embedding the identifier of T_1 in T_2 as a foreign key. In addition, the simple monovalued attributes and the simple components of the monovalued complex attributes of R are also included in T_2. This table also defines non-null constraints for the mandatory attributes.
Note that, in general, we can embed the key of T_1 in T_2, or conversely, the key of T_2 in T_1. The choice depends on the cardinality of the roles of R. In Fig. **??**, assume the relationship Supplies has cardinalities (1,1) with Products and (0,1) with Suppliers. Embedding ProductID in table Suppliers may result in several tuples of the Suppliers relation containing null values in the ProductID column, since there can be suppliers that do not supply any product. Thus, to avoid null values, it would be preferable to embed SupplierID in table Products.

Rule 4: Consider a regular binary one-to-many relationship type R relating entity types E_1 and E_2, where T_1 and T_2 are the tables resulting from the mapping of these entities. R is mapped embedding the key of T_2 in table T_1 as a foreign key. In addition, the simple monovalued attributes and the simple components of the monovalued complex attributes of R are included in T_1, defining the corresponding non-null constraints for the mandatory attributes.
As an example, in Fig. **??**, the one-to-many relationship type Supplies between Products and Suppliers is mapped by including the attribute SupplierID in table Products, as a foreign key, as shown in Fig. **??**.

Rule 5: Consider a regular binary many-to-many relationship type R between entity types E_1 and E_2, such that T_1 and T_2 are the tables resulting from the mapping of the former entities. R is mapped to a table T containing the keys of T_1 and T_2, as foreign keys. The key of T is the union of these keys. Alternatively, the relationship identifier, if any, may define the key of the table. T also contains the simple monovalued attributes

and the simple components of the monovalued complex attributes of R, and also defines non-null constraints for the mandatory attributes.

In Fig. **??**, the many-to-many relationship type EmployeeTerritories between Employees and Territories is mapped to a table with the same name containing the identifiers of the two tables involved, as shown in Fig. **??**.

Rule 6: A multivalued attribute of an entity or relationship type E is mapped to a table T, which also includes the identifier of the entity or relationship type. A referential integrity constraint relates this identifier to the table associated with E. The primary key of T is composed of all of its attributes.

Suppose that in Fig. **??**, the attribute Phone of Customers is multivalued. In this case, the attribute is mapped to a table CustomerPhone with attributes CustomerID and Phone both composing the primary key.

Rule 7: A generalization relationship between a supertype E_1 and subtype E_2 can be dealt with in three different ways:

Rule 7a: Both E_1 and E_2 are mapped, respectively, to tables T_1 and T_2, in which case the identifier of E_1 is propagated to T_2. A referential integrity constraint relates this identifier to T_1.

Rule 7b: Only E_1 is associated with a table T_1, which contains all attributes of E_2. All these attributes become optional in T_1.

Rule 7c: Only E_2 is associated with a table T_2, in which case all attributes E_1 are inherited in T_2.

As an example, the possible translations of the generalization given in Fig. **??** are shown in Fig. **??**.

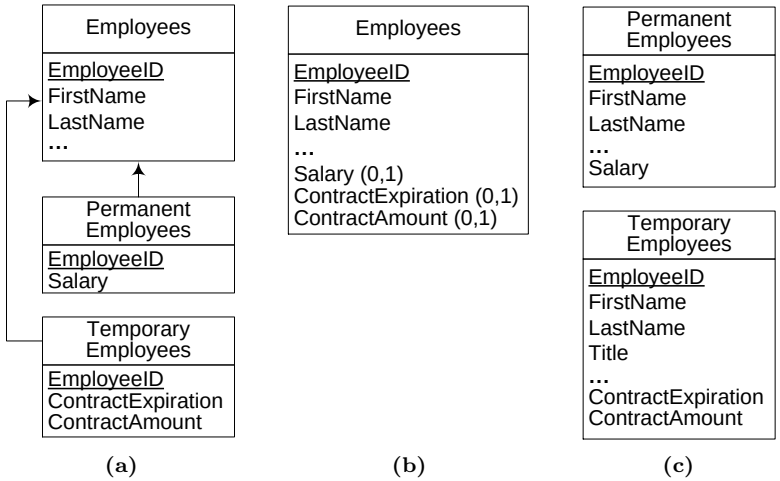

Fig. 2.6. Three possible translations of the schema in Fig. **??** (**a**) Using Rule 7a; (**b**) Using Rule 7b; (**c**) Using Rule 7c

Note that the generalization type (total vs. partial and disjoint vs. overlapping) may preclude one of the above three approaches. For example, the third possibility is not applicable for partial generalizations. Also, note that the semantics of the partial, total, disjoint, and overlapping characteristics are not fully captured by this translation mechanism. The conditions must be implemented when populating the relational tables. For example, assume a table T, and two tables T_1 and T_2 resulting from the mapping of a total and overlapping generalization. Referential integrity does not fully capture the semantics. It must be ensured, among other conditions, that when deleting an element from T, this element is also deleted from T_1 and T_2 (since it can exist in both tables). Such constraints are typically implemented with triggers.

Applying these mapping rules to the ER schema given in Fig. **??** yields the relational schema shown in Fig. **??**. Note that the above rules apply in the general case; however, other mappings are possible. For example, binary one-to-one and one-to-many relationships may be represented by a table of its own, using Rule 5. The choice between alternative representation depends on the characteristics of the particular application at hand.

It must be noted that there is a significant difference in expressive power between the ER model and the relational model. This difference may be explained by the fact that the ER model is a *conceptual* model aimed at expressing concepts as closely as possible to the users' perspective, whereas the relational model is a *logical* model targeted toward particular implementation platforms. Several ER concepts do not have a correspondence in the relational model, and thus they must be expressed using only the available concepts in the model, that is, relations, attributes, and the related constraints. This translation implies a semantic loss in the sense that data invalid in an ER schema are allowed in the corresponding relational schema, unless the latter is supplemented by additional constraints. Many of such constraints must be manually coded by the user using mechanisms such as triggers or stored procedures. Furthermore, from a user's perspective, the relational schema is much less readable than the corresponding ER schema. This is crucial when one is considering schemas with hundreds of entity or relationship types and thousands of attributes. This is not a surprise, since this was the reason for devising conceptual models back in the 1970s, that is, the aim was to better understand the semantics of large relational schemas.

2.4.2 Normalization

When considering a relational schema, we must determine whether or not the relations in the schema have potential redundancies, and thus may induce anomalies in the presence of insertions, updates, and deletions.

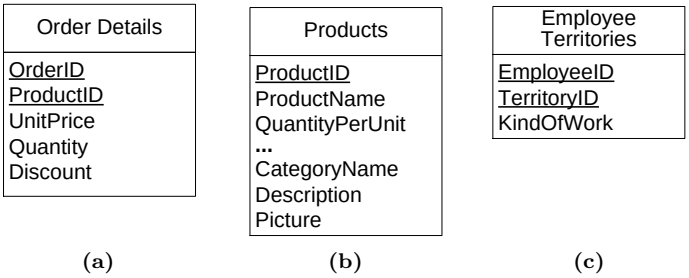

Fig. 2.7. Examples of relations that are not normalized

For example, assume that in relation OrderDetails in Fig. **??**, each product, no matter the order, is associated with a discount percentage. Here, the discount information for a product p will be repeated for all orders in which p appears. Thus, this information will be redundant. To solve this problem, the attribute Discount must be removed from the table OrderDetails and must be added to the table Products in order to store only once the information about the product discounts.

Consider now the relation Products in Fig. **??**, which is a variation of the relation with the same name in Fig. **??**. In this case, we have included the category information (name, description, and picture) in the Products relation. It is easy to see that such information about a category is repeated for each product with the same category. Therefore, when, for example, the description of a category needs to be updated, we must ensure that all tuples in the relation Products, corresponding to the same category, are also updated, otherwise there will be inconsistencies. To solve this problem, the attributes describing a category must be removed from the table and a table Category like the one in Fig. **??** must be used to store the data about categories.

Finally, let us analyze the relation EmployeeTerritories in Fig. **??**, where an additional attribute KindOfWork has been added with respect to the relation with the same name in Fig. **??**. Assume that an employee can do many kinds of work, independently of the territories in which she carries out her work. Thus, the information about the kind of work of an employee will be repeated as many times as the number of territories she is assigned to. To solve this problem, the attribute KindOfWork must be removed from the table, and a table EmpWork relating employees to the kinds of work they perform must be added.

Dependencies and normal forms are used to describe the redundancies above. A **functional dependency** is a constraint between two sets of attributes in a relation. Given a relation R and two sets of attributes X and Y in R, a functional dependency $X \rightarrow Y$ holds if and only if, in all the tuples of the relation, each value of X is associated with at most one value of Y. In this case it is said that X *determines* Y. Note that a key is a particular

case of a functional dependency, where the set of attributes composing the key functionally determines all of the attributes in the relation.

The redundancies in Fig. **??**,b can be expressed by means of functional dependencies. For example, in the relation OrderDetails in Fig. **??**, there is the functional dependency ProductID $\rightarrow$ Discount. Also, in the relation Products in Fig. **??**, we have the functional dependencies ProductID $\rightarrow$ CategoryName and CategoryName $\rightarrow$ Description.

The redundancy in the relation EmployeeTerritories in Fig. **??** is captured by another kind of dependency. Given two sets of attributes X and Y in a relation R, a **multivalued dependency** $X \rightarrow\rightarrow Y$ holds if the value of X determines a set of values for Y, which is independent of $R \backslash XY$, where '\' indicates the set difference. In this case we say that X *multidetermines* Y. In the relation in Fig. **??**, the multivalued dependency EmployeeID $\rightarrow\rightarrow$ KindOfWork holds, and consequently TerritoryID $\rightarrow\rightarrow$ KindOfWork. Functional dependencies are special cases of multivalued dependencies, that is, every functional dependency is also a multivalued dependency.

A **normal form** is an integrity constraint aimed at guaranteeing that a relational schema satisfies particular properties. Since the beginning of the relational model in the 1970s, many types of normal forms have been defined. In addition, normal forms have also been defined for other models, such as the entity-relationship model. We refer the reader to database textbooks for the definition of the various normal forms.

2.4.3 Relational Query Languages

Data stored in a relational database can be queried using different formalisms. Two kinds of query languages are typically defined. In a *procedural* language, a query is specified indicating the operations needed to retrieve the desired result. In a *declarative* language, the user only indicates *what* she wants to retrieve, leaving to the DBMS the task of determining the equivalent procedural query that is to be executed.

In this section, we introduce the relational algebra and SQL, which we will be using in many parts of this book. While the relational algebra is a procedural query language, SQL is a declarative one.

2.4.3.1 Relational Algebra

The relational algebra is a collection of operations for manipulating relations. These operations can be of two kinds: **unary**, which receive as argument a relation and return another relation, or **binary**, which receive as argument two relations and return a relation. As the operations always return relations, the algebra is *closed*, and operations can be combined in order to compute

the answer to a query. Further, another classification of the operations is as follows. **Basic** operations cannot be derived from any combination of other operations, while **derived** operations are a shorthand for a sequence of basic operations, defined in order to make queries easier to express. In what follows, we describe the relational algebra operations. We start first with the unary operations.

The **projection** operation, denoted $\pi_{C_1,\ldots,C_n}(R)$, returns the columns $C_1, \ldots, C_n$ from the relation R. Thus, it can be seen as a vertical partition of R into two relations: one containing the columns mentioned in the expression, and the other containing the remaining columns.

For the database given in Fig. **??**, an example of a projection is:

$\pi_{\mathsf{FirstName,LastName,HireDate}}(\mathsf{Employees})$.

This operation returns the three specified attributes from the Employees table.

The **selection** operation, denoted $\sigma_\phi(R)$, returns the tuples from the relation R that satisfy the Boolean condition ϕ. In other words, it partitions a table horizontally into two sets of tuples: the ones that do satisfy the condition and the ones that do not. Therefore, the structure of R is kept in the result.

A selection operation over the database given in Fig. **??** is:

$\sigma_{\mathsf{HireDate}\geq\text{`01/01/1992'}\wedge\mathsf{HireDate}\leq\text{`31/12/1994'}}(\mathsf{Employees})$.

This operation returns the employees hired between 1992 and 1994.

Since the result of a relational algebra operation is a relation, it can be used as input for another operation. To make queries easier to read, sometimes it is useful to use temporary relations to store intermediate results. We will use the notation $T \leftarrow Q$ to indicate that relation T stores the result of query Q. Thus, combining the two previous examples, we can ask for the first name, last name, and hire date of all employees hired between 1992 and 1994. The query reads:

$\mathsf{Temp1} \leftarrow \sigma_{\mathsf{HireDate}\geq\text{`01/01/1992'}\wedge\mathsf{HireDate}\leq\text{`31/12/1994'}}(\mathsf{Employees})$
$\mathsf{Result} \leftarrow \pi_{\mathsf{FirstName,LastName,HireDate}}(\mathsf{Temp1})$.

The **rename** operation, denoted $\rho_{A_1\rightarrow B_1,\ldots,A_k\rightarrow B_k}(R)$, returns a relation where the attributes $A_1, \ldots, A_k$ in R are renamed to $B_1, \ldots, B_k$, respectively. Therefore, the resulting relation has the same tuples as the relation R, although the schema of both relations is different.

We present next the binary operations, which are based on classic operations of the set theory. We first introduce the basic binary operations, namely, union, difference, and Cartesian product, and then discuss the most used binary operation, the join, and its variants inner and outer join.

The **union** operation, denoted $R_1 \cup R_2$, takes two relations with the same schema, and returns the tuples that are in R_1, in R_2, or in both, removing duplicates. If the schemas are compatible, but the attribute names differ, the attributes must be renamed before applying the operation.

The union can be used to express queries like "Identifier of employees from the UK, or who are reported by an employee from the UK," which reads:

$$\begin{aligned}
&\mathsf{UKEmps} \leftarrow \sigma_{\mathsf{Country}=\text{'UK'}}(\mathsf{Employees})\\
&\mathsf{Result1} \leftarrow \pi_{\mathsf{EmployeeID}}(\mathsf{UKEmp})\\
&\mathsf{Result2} \leftarrow \rho_{\mathsf{ReportsTo}\rightarrow\mathsf{EmployeeID}}(\pi_{\mathsf{ReportsTo}}(\mathsf{UKEmps}))\\
&\mathsf{Result} \leftarrow \mathsf{Result1} \cup \mathsf{Result2}.
\end{aligned}$$

Relation UKEmps contains the employees from the UK. Result1 contains the projection of the former over EmployeeID, and Result2 contains the EmployeeID of the employees reported by an employee from the UK. The union of Result1 and Result2 yields the desired result.

The **difference** operation, denoted $R_1 \setminus R_2$, takes two relations with the same schema, and returns the tuples that are in R_1 but not in R_2. As in the case of the union, if the schemas are compatible, but the attribute names differ, the attributes must be renamed before applying the operation.

We use the difference to express queries like "Identifier of employees who are not reported by an employee from the UK," which is written as follows:

$$\mathsf{Result} \leftarrow \pi_{\mathsf{EmployeeID}}(\mathsf{Employees}) \setminus \mathsf{Result2}.$$

The first term of the difference contains the identifiers of all employees. From this set we subtract the set composed of the identifiers of all employees reported by an employee from the UK, already computed in Result2.

The **Cartesian product**, denoted $R_1 \times R_2$, takes two relations and returns a new one, whose schema is composed of all the attributes in R_1 and R_2 (renamed if necessary), and whose instance is obtained concatenating each pair of tuples from R_1 and R_2. Thus, the number of tuples in the result is the product of the cardinalities of both relations.

Although by itself the Cartesian product is usually meaningless, it is very useful when combined with a selection. For example, suppose we want to retrieve the name of the products supplied by suppliers from Brazil. To answer this query, we use the Cartesian product to combine data from the tables Products and Suppliers. For the sake of clarity, we only keep the attributes we need: ProductID, ProductName, and SupplierID from table Products, and SupplierID and Country from table Suppliers. Attribute SupplierID in one of the relations must be renamed, since a relation cannot have two attributes with the same name.

$$\begin{aligned}
&\mathsf{Temp1} \leftarrow \pi_{\mathsf{ProductID},\mathsf{ProductName},\mathsf{SupplierID}}(\mathsf{Products})\\
&\mathsf{Temp2} \leftarrow \rho_{\mathsf{SupplierID}\rightarrow\mathsf{SupID}}(\pi_{\mathsf{SupplierID},\mathsf{Country}}(\mathsf{Suppliers}))\\
&\mathsf{Temp3} \leftarrow \mathsf{Temp1} \times \mathsf{Temp2}.
\end{aligned}$$

The Cartesian product combines each product with all the suppliers. We are only interested in the rows that relate a product to its supplier. For this, we filter the meaningless tuples, select the ones corresponding to suppliers from Brazil, and project the column we want, that is, ProductName:

$$\begin{aligned}
&\mathsf{Temp4} \leftarrow \sigma_{\mathsf{SupplierID}=\mathsf{SupID}}(\mathsf{Temp3})\\
&\mathsf{Result} \leftarrow \pi_{\mathsf{ProductName}}(\sigma_{\mathsf{Country}=\text{'Brazil'}}(\mathsf{Temp4})).
\end{aligned}$$

The **join** operation, denoted $R_1 \bowtie_\phi R_2$, where ϕ is a condition over the attributes in R_1 and R_2, takes two relations and returns a new one, whose schema consists in all attributes of R_1 and R_2 (renamed if necessary), and whose instance is obtained concatenating each pair of tuples from R_1 and R_2 that satisfy condition ϕ. The operation is basically a combination of a Cartesian product and a selection.

Using the join operation, the query "Name of the products supplied by suppliers from Brazil" will read:

$$\begin{aligned}&\mathsf{Temp1} \leftarrow \rho_{\mathsf{SupplierID}\rightarrow\mathsf{SupID}}(\mathsf{Suppliers})\\ &\mathsf{Result} \leftarrow \pi_{\mathsf{ProductName}}(\sigma_{\mathsf{Country}=\text{`Brazil'}}(\mathsf{Product} \bowtie_{\mathsf{SupplierID}=\mathsf{SupID}} \mathsf{Temp1})).\end{aligned}$$

Note that the join combines the Cartesian product in Temp3 and the selection in Temp4 in a single operation, making the expression much more concise.

There are a number of variants of the join operation. An **equijoin** is a join $R_1 \bowtie_\phi R_2$ such that condition ϕ states only equality comparisons. A **natural join**, denoted $R_1 * R_2$, is a type of equijoin that states the equality between *all* the attributes with the same name in R_1 and R_2. The resulting table is defined over the schema $R_1 \cup R_2$ (i.e., all the attributes in R_1 and R_2, without duplicates). In the case that there are no attributes with the same names, a cross join is performed.

For example, the query "List all product names and category names" reads:

$$\begin{aligned}&\mathsf{Temp} \leftarrow \mathsf{Products} * \mathsf{Categories}\\ &\mathsf{Result} \leftarrow \pi_{\mathsf{ProductName},\mathsf{CategoryName}}(\mathsf{Temp}).\end{aligned}$$

The first query performs the natural join between relations Products and Categories. The attributes in Temp are all the attributes in Product, plus all the attributes in Categories, except for CategoryID, which is in both relations, so only one of them is kept. The second query performs the final projection.

The joins introduced above are known as **inner joins**, since tuples that do no match the join condition are eliminated. In many practical cases we need to keep in the result all the tuples of one or both relations, independently of whether or not they verify the join condition. For these cases, a set of operations, called **outer joins**, were defined. There are three kinds of outer joins: left outer join, right outer join, and full outer join.

The **left outer join**, denoted $R_1 \mathbin{⟕}_\phi R_2$, performs the join as defined above, but instead of keeping only the matching tuples, it keeps every tuple in R_1 (the relation of the left of the operation). If a tuple in R_1 does not satisfy the join condition, the tuple is kept, and the attributes of R_2 in the result are filled with null values.

As an example, the query "Last name of employees, together with the last name of their supervisor, or null if the employee has no supervisor," reads in relational algebra:

$$\begin{aligned}\mathsf{Supervisors} &\leftarrow \rho_{\mathsf{EmployeeID}\rightarrow\mathsf{SupID},\mathsf{LastName}\rightarrow\mathsf{SupLastName}}(\mathsf{Employees})\\ \mathsf{Result} &\leftarrow \pi_{\mathsf{EmployeeID},\mathsf{LastName},\mathsf{SupID},\mathsf{SupLastName}}(\\ &\quad \mathsf{Employees} \mathbin{⟕}_{\mathsf{ReportsTo}=\mathsf{SupID}} \mathsf{Supervisors}).\end{aligned}$$

The resulting table has tuples such as $(2, \mathsf{Fuller}, \mathsf{NULL}, \mathsf{NULL})$, which correspond to employees who do not report to any other employee.

The **right outer join**, denoted $R_1 \ltimes\!\!\subset_\phi R_2$, is analogous to the left outer join, except that the tuples that are kept are the ones in R_2. The **full outer join**, denoted $R_1 \supset\!\!\ltimes\!\!\subset_\phi R_2$, keeps all the tuples in both R_1 and R_2.

Suppose that in the previous example we also require the information of the employees who do not supervise anyone. Then, we would have:

$$\pi_{\mathsf{EmployeeID,LastName,SupID,SupLastName}}(\mathsf{Employees} \supset\!\!\ltimes\!\!\subset_{\mathsf{ReportsTo=SupID}} \mathsf{Supervisors})$$

With respect to the left outer join shown above, the resulting table has in addition tuples such as $(\mathsf{NULL}, \mathsf{NULL}, 1, \mathsf{Davolio})$, which correspond to employees who do not supervise any other employee.

2.4.3.2 SQL

SQL (structured query language) is the most common language for creating, manipulating, and retrieving data from relational DBMSs. SQL is composed of several sublanguages. The **data definition language** (DDL) is used to define the schema of a database. The **data manipulation language** (DML) is used to query a database and to modify its content (i.e., to add, update, and delete data in a database). In what follows, we present a summary of the main features of SQL that we will use in this book. For a detailed description, we refer the reader to the references provided at the end of this chapter. the references provided at the end of this chapter.

Below we show the SQL DDL command for defining table Orders in the relational schema of Fig. **??**. The basic DDL statement is CREATE TABLE, which creates a relation and defines the data types of the attributes, the primary and foreign keys, and the constraints.

```
CREATE TABLE Orders (
        OrderID INTEGER PRIMARY KEY,
        CustomerID INTEGER NOT NULL,
        EmployeeID INTEGER NOT NULL,
        OrderDate DATE NOT NULL,
        RequiredDate DATE NOT NULL,
        ShippedDate DATE NOT NULL,
        ShippedVia INTEGER NOT NULL,
        Freight MONEY NOT NULL,
        ShipName CHARACTER VARYING (50) NOT NULL,
        ShipAddress CHARACTER VARYING (50) NOT NULL,
        ShipCity CHARACTER VARYING (50) NOT NULL,
        ShipRegion CHARACTER VARYING (50),
        ShipPostalCode CHARACTER VARYING (30),
        ShipCountry CHARACTER VARYING (50) NOT NULL,
        FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),
        FOREIGN KEY (ShippedVia) REFERENCES Shippers(ShipperID),
        FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID),
        CHECK (OrderDate ¡= RequiredDate) )
```

SQL provides a DROP TABLE statement for deleting a table, and an ALTER TABLE statement for modifying the structure of a table.

The **DML** part of SQL is used to insert, update, and delete tuples from the database tables. For example, the following INSERT statement

```
INSERT INTO Shippers(CompanyName, Phone)
VALUES ('Federal Express', '02 752 75 75')
```

adds a new shipper in the Northwind database. This tuple is modified by the following UPDATE statement:

```
UPDATE Shippers
SET    CompanyName = 'Fedex'
WHERE  CompanyName = 'Federal Express'
```

Finally, the new shipper is removed in the following DELETE statement.

```
DELETE FROM Shippers
WHERE CompanyName = 'Fedex'
```

SQL also provides statements for retrieving data from the database. The basic structure of an SQL expression is:

```
SELECT ⟨ list of attributes ⟩
FROM   ⟨ list of tables ⟩
WHERE  ⟨ condition ⟩
```

where ⟨ list of attributes ⟩ indicates the attribute names whose values are to be retrieved by the query, ⟨ list of tables ⟩ is a list of the relation names that will be included in the query, and ⟨ condition ⟩ is a Boolean expression that must be satisfied by the tuples in the result. The semantics of an SQL expression

```
SELECT R.A, S.B
FROM   R, S
WHERE R.B = S.A
```

is given by the relational algebra expression

$$\pi_{\mathsf{R.A,S.B}}(\sigma_{\mathsf{R.B=S.A}}(\mathsf{R} \times \mathsf{S})),$$

that is, the SELECT clause is analogous to a projection π, the WHERE clause is a selection σ, and the FROM clause indicates the Cartesian product $\times$ between all the tables included in the clause.

It is worth noting that an SQL query, opposite to a relational algebra one, returns a set *with duplicates* (or a bag). Therefore, the keyword DISTINCT must be used to remove duplicates in the result. For example, the query "Countries of customers" must be written:

```
SELECT DISTINCT Country
FROM   Customers
```

This query returns the set of countries of the Northwind customers, without duplicates. If the DISTINCT keyword is removed from the above query, then it would return as many results as the number of customers in the database.

As another example, the query "Identifier, first name, and last name of the employees hired between 1992 and 1994," which we presented when discussing the projection and selection operations, reads in SQL:

```
SELECT EmployeeID, FirstName, LastName
FROM   Employees
WHERE HireDate ¿= '1992-01-01' and HireDate ¡= '1994-12-31'
```

The binary operations of the relational algebra are supported in SQL: union, intersection, difference, and the different kinds of joins. Recall the query "Identifiers of employees from the UK, or who are reported by an employee from the UK." In SQL it would read:

```
SELECT EmployeeID
FROM   Employees
WHERE Country='UK'
    UNION
SELECT ReportsTo
FROM   Employees
WHERE Country='UK'
```

Notice that the UNION in the above query removes duplicates in the result, whereas the UNION ALL will keep them, that is, if an employee is from the UK and is reported by at least one employee from the UK, it will appear twice in the result.

The join operation can be, of course, implemented as a projection of a selection over the Cartesian product of the relations involved. However, in general, it is easier and more efficient to use the join operation. For example, the query "Name of the products supplied by suppliers from Brazil" can be written as follows:

```
SELECT ProductName
FROM   Products P, Suppliers S
WHERE P.SupplierID = S.SupplierID AND Country = 'Brazil'
```

An alternative formulation of this query is as follows:

```
SELECT ProductName
FROM   Products P JOIN Suppliers S ON P.SupplierID = S.SupplierID
WHERE Country = 'Brazil'
```

On the other hand, the outer join operations must be explicitly stated in the FROM clause. For example, the query "First name and last name of employees, together with the first name and last name of their supervisor, or null if the employee has no supervisor" can be implemented in SQL using the LEFT OUTER JOIN operation.

```
SELECT E.FirstName, E.LastName, S.FirstName, S.LastName
FROM   Employees E LEFT OUTER JOIN Employees S
       ON E.ReportsTo = S.EmployeeID
```

Analogously, we can use the FULL OUTER JOIN operation to also include in the answer the employees who do not supervise anybody.

```
SELECT E.FirstName, E.LastName, S.FirstName, S.LastName
FROM   Employees E FULL OUTER JOIN Employees S ON E.ReportsTo = S.EmployeeID
```

As shown in the examples above, SQL is a declarative language, that is, we tell the system *what* we want, whereas in relational algebra, being a procedural language, we must specify *how* we will obtain the result. In fact, SQL query processors usually translate an SQL query into some form of relational algebra in order to optimize it.

Aggregation is used to summarize information from multiple tuples into a single one. For this, tuples are grouped and then an aggregate function is applied to every group. In data warehouses, particularly in OLAP, aggregation plays a crucial role, as we will study in subsequent chapters of this book.

Typically, DBMSs provide five basic aggregate functions, namely, COUNT, SUM, MAX, MIN, and AVG. The COUNT function returns the number of tuples in each group. Analogously, the functions SUM, MAX, MIN, and AVG are applied over numeric attributes and return, respectively, the sum, maximum value, minimum value, and average of the values in those attributes, for each group. Note that all of these functions can be applied to the whole table considered as a group. Further, the functions MAX and MIN can also be used with attributes that have nonnumeric domains if a total order is defined over the values in the domain, as is the case for strings.

The general form of an SQL query with aggregate functions is as follows:

```
SELECT    ⟨ list of grouping attributes ⟩ ⟨ list of aggr_funct(attribute) ⟩
FROM      ⟨ list of tables ⟩
WHERE     ⟨ condition ⟩
GROUP BY  ⟨ list of grouping attributes ⟩
HAVING    ⟨ condition over groups ⟩
ORDER BY  ⟨ list of attributes ⟩
```

An important restriction is that if there is a GROUP BY clause, the SELECT clause must contain *only* aggregates or grouping attributes. The HAVING clause is analogous to the WHERE clause, except that the condition is applied over each group rather than over each tuple. Finally, the result can be sorted with the ORDER BY clause, where every attribute in the list can be ordered either in ascending or descending order by specifying ASC or DESC, respectively.

We next present some examples of aggregate SQL queries, more complex ones will be presented later in the book. We start with the query "Total number of orders handled by each employee, in descending order of number of orders. Only list employees that handled more than 100 orders."

```
SELECT    EmployeeID, COUNT(*) AS OrdersByEmployee
FROM      Orders
GROUP BY EmployeeID
HAVING    COUNT(*) ¿ 100
ORDER BY COUNT(*) DESC
```

Consider now the query "For customers from Germany, list the total quantity of each product ordered. Order the result by customer ID, in ascending order, and by quantity of product ordered, in descending order."

```
SELECT    C.CustomerID, D.ProductID, SUM(Quantity) AS TotalQty
FROM      Orders O JOIN Customers C ON O.CustomerID = C.CustomerID
          JOIN OrderDetails D ON O.OrderID = D.OrderID
WHERE     C.Country = 'Germany'
GROUP BY C.CustomerID, D.ProductID
ORDER BY C.CustomerID ASC, SUM(Quantity) DESC
```

This query starts by joining three tables: Orders, Customers (where we have the country information), and OrderDetails (where we have the quantity ordered for each product in each order). Then, the query selects the customers from Germany. We then group by pairs (CustomerID, ProductID), and for each group, we take the sum in the attribute Quantity.

A **subquery** (or a **nested query**) is an SQL query used within a SELECT, FROM, or WHERE clause. The external query is called the **outer query**. In the WHERE clause, this is typically used to look for a certain value in a database, and we use this value in a comparison condition through two special predicates: IN and EXISTS (and their negated versions, NOT IN and NOT EXISTS).

As an example of the IN predicate, let us consider the query "Identifier and name of products ordered by customers from Germany." The query is written as follows:

```
SELECT ProductID, ProductName
FROM   Products P
WHERE P.ProductID IN (
        SELECT D.ProductID
        FROM   Orders O JOIN Customers C ON O.CustomerID = C.CustomerID
               JOIN OrderDetails D ON O.OrderID = D.OrderID
        WHERE C.Country = 'Germany' )
```

The inner query computes the products ordered by customers from Germany. This returns a bag of product identifiers. The outer query scans the Products table, and for each tuple, it compares the product identifier with the set of identifiers returned by the inner query. If the product is in the set, the product identifier and the product name are listed.

The query above can be formulated using the EXISTS predicate, yielding what are denoted as **correlated nested queries**, as follows:

```
SELECT ProductID, ProductName
FROM   Products P
WHERE EXISTS (
```

```
        SELECT *
        FROM   Orders O JOIN Customers C ON
               O.CustomerID = C.CustomerID JOIN
               OrderDetails D ON O.OrderID = D.OrderID
        WHERE C.Country = 'Germany' AND D.ProductID = P.ProductID )
```

Note that in the outer query we define an alias (or variable) P. For each tuple in Products, the variable P in the inner query is instantiated with the values in such tuple; if the result set of the inner query instantiated in this way is not empty, the EXISTS predicate evaluates to true, and the values of the attributes ProductID and ProductName are listed. The process is repeated for all tuples in Products.

To illustrate the NOT EXISTS predicate, consider the query "Names of customers who have not purchased any product," which is written as follows:

```
SELECT C.CompanyName
FROM   Customers C
WHERE NOT EXISTS (
        SELECT *
        FROM   Orders O
        WHERE C.CustomerID = O.CustomerID )
```

Here, the NOT EXISTS predicate will evaluate to true if, when P is instantiated in the inner query, the query returns the empty set.

In SQl, a **view** is just a query that is stored in the database with an associated name. Thus, views are like virtual tables. A view can be created from one or many tables or other views, depending on the SQL query that defines it.

Views can be used for various purposes. They are used to structure data in a way that users find it natural or intuitive. They can also be used to restrict access to data such that users can have access only to the data they need. Finally, views can also be used to summarize data from various tables, which can be used, for example, to generate reports.

Views are created with the CREATE VIEW statement. To create a view, a user must have appropriate system privileges to modify the database schema. Once a view is created, it can then be used in a query as any other table.

For example, the following statement creates a view CustomerOrders that computes for each customer and order the total amount of the order.

```
CREATE VIEW CustomerOrders AS (
        SELECT     O.CustomerID, O.OrderID,
                   SUM(D.Quantity * D.UnitPrice) AS Amount
        FROM       Orders O, OrderDetails D
        WHERE      O.OrderID = D.OrderID
        GROUP BY O.CustomerID, O.OrderID )
```

This view is used in the next query to compute for each customer the maximum amount among all her orders.

```
SELECT    CustomerID, MAX(Amount) AS MaxAmount
FROM      CustomerOrders
GROUP BY CustomerID
```

As we will see in Chap. **??**, views can be materialized, that is, they can be physically stored in a database.

A **common table expression** (CTE) is a temporary table defined within an SQL statement. Such temporary tables can be seen as views within the scope of the statement. A CTE is typically used when a user does not have the necessary privileges for creating a view.

For example, the following query

```
WITH CustomerOrders AS (
          SELECT    O.CustomerID, O.OrderID,
                    SUM(D.Quantity * D.UnitPrice) AS Amount
          FROM      Orders O, OrderDetails D
          WHERE     O.OrderID = D.OrderID
          GROUP BY O.CustomerID, O.OrderID )
SELECT    CustomerID, MAX(Amount) AS MaxAmount
FROM      CustomerOrders
GROUP BY CustomerID
```

combines in a single statement the view definition and the subsequent query given in the previous section. It is worth noting that several temporary tables can be defined in the WITH clause. We will extensively use CTEs throughout this book.

2.5 Physical Database Design

The objective of **physical database design** is to specify how database records are stored, accessed, and related in order to ensure adequate performance of a database application. Physical database design is related to query processing, physical data organization, indexing, transaction processing, and concurrency management, among other characteristics. In this section, we provide a very brief overview of some of those issues that will be addressed in detail for data warehouses in Chap. **??**.

Physical database design requires one to know the specificities of the given application, in particular the properties of the data and the usage patterns of the database. The latter involves analyzing the transactions or queries that are run frequently and will have a significant impact on performance, the transactions that are critical to the operations of the organization, and the periods of time during which there will be a high demand on the database (called the **peak load**). This information is used to identify the parts of the database that may cause performance problems.

Various factors can be used to measure the performance of database applications. **Transaction throughput** is the number of transactions that can

be processed in a given time interval. In some systems, such as electronic payment systems, a high transaction throughput is critical. **Response time** is the elapsed time for the completion of a single transaction. Minimizing response time is essential from the user's point of view. Finally, **disk storage** is the amount of disk space required to store the database files. However, a compromise usually has to be made among these factors. From a general perspective, this compromise implies the following factors:

1. **Space-time trade-off:** It is often possible to reduce the time taken to perform an operation by using more space, and vice versa. For example, a compression algorithm can be used to reduce the space occupied by a large file, but this implies extra time for the decompression process.
2. **Query-update trade-off:** Access to data can be made more efficient by imposing some structure upon it. However, the more elaborate the structure, the more time is taken to build it and to maintain it when its contents change. For example, sorting the records of a file according to a key field allows them to be located more easily but there is a greater overhead upon insertions to keep the file sorted.

Further, once an initial physical design has been implemented, it is necessary to monitor the system and to tune it as a result of the observed performance and any changes in requirements. Many DBMSs provide utilities to monitor and tune the operations of the system.

As the functionality provided by current DBMSs varies widely, physical design requires one to know the various techniques for storing and finding data that are implemented in the particular DBMS that will be used.

A database is organized on **secondary storage** into one or more **files**, where each file consists of one or several **records** and each record consists of one or several **fields**. Typically, each tuple in a relation corresponds to a record in a file. When a user requests a particular tuple, the DBMS maps this logical record into a physical disk address and retrieves the record into main memory using the file access routines of the operating system.

Data are stored on a computer disk in **disk blocks** (or **pages**) that are set by the operating system during disk formatting. Transfer of data between the main memory and the disk and vice versa takes place in units of disk blocks. DBMSs store data on **database blocks** (or **pages**). One important aspect of physical database design is the need to provide a good match between disk blocks and database blocks, on which logical units such as tables and records are stored. Most DBMSs provide the ability to specify a database block size. The selection of a database block size depends on several issues. For example, most DBMSs manage concurrent access to the records using some kind of locking mechanism. If a record is locked by one transaction that aims at modifying it, then no other transaction will be able to modify this record (however, normally several transactions are able to read a record if they do not try to write it). In some DBMSs, the finest locking granularity is at the page level, not at the record level. Therefore, the larger the page size,

the larger the chance that two transactions will request access to entries on the same page. On the other hand, for optimal disk efficiency, the database block size must be equal to or be a multiple of the disk block size.

DBMSs reserve a storage area in the main memory that holds several database pages, which can be accessed for answering a query without reading those pages from the disk. This area is called a **buffer**. When a request is issued to the database, the query processor checks if the required data records are included in the pages already loaded in the buffer. If so, data are read from the buffer and/or modified. In the latter case, the modified pages are marked as such and eventually written back to the disk. If the pages needed to answer the query are not in the buffer, they are read from the disk, probably replacing existing ones in the buffer (if it is full, which is normally the case) using well-known algorithms, for example, replacing the least recently used pages with the new ones. In this way, the buffer acts as a **cache** that the DBMS can access to avoid going to disk, enhancing query performance.

File organization is the physical arrangement of data in a file into records and blocks on secondary storage. There are three main types of file organization. In a **heap** (or **unordered**) file organization, records are placed in the file in the order in which they are inserted. This makes insertion very efficient. However, retrieval is relatively slow, since the various pages of the file must be read in sequence until the required record is found. **Sequential** (or **ordered**) **files** have their records sorted on the values of one or more fields, called **ordering fields**. Ordered files allow fast retrieving of records, provided that the search condition is based on the sorting attribute. However, inserting and deleting records in a sequential file are problematic, since the order must be maintained. Finally, **hash files** use a **hash function** that calculates the address of the block (or **bucket**) in which a record is to be stored, based on the value of one or more attributes. Within a bucket, records are placed in order of arrival. A **collision** occurs when a bucket is filled to its capacity and a new record must be inserted into that bucket. Hashing provides the fastest possible access for retrieving an arbitrary record given the value of its hash field. However, collision management degrades the overall performance.

Independently of the particular file organization, additional access structures called **indexes** are used to speed up the retrieval of records in response to search conditions. Indexes provide efficient ways to access the records based on the **indexing fields** that are used to construct the index. Any field(s) of the file can be used to create an index, and multiple indexes on different fields can be constructed in the same file.

There are many different types of indexes. We describe below some categories of indexes according to various criteria.

- One categorization of indexes distinguishes between **clustered** and **non-clustered indexes**, also called **primary** and **secondary indexes**. In a clustered index, the records in the data file are physically ordered according to the field(s) on which the index is defined. This is not the case

for a nonclustered index. A file can have at most one clustered index and in addition can have several nonclustered indexes.

- Indexes can be **single-column** or **multiple-column**, depending on the number of indexing fields on which they are based. When a multiple-column index is created, the order of columns in the index has an impact on data retrieval. Generally, the most restrictive value should be placed first for optimum performance.
- Another categorization of indexes is according to whether they are **unique** or **nonunique**: unique indexes do not allow duplicate values, while this is not the case for nonunique indexes.
- In addition, an index can be **sparse** or **dense**: in a dense index there is one entry in the index for every data record. This requires data files to be ordered on the indexing key. Opposite to this, a sparse index contains less index entries than data records. Thus, a nonclustered index is always dense, since it is not ordered on the indexing key.
- Finally, indexes can be **single-level** or **multilevel**. When an index file becomes large and extends over many blocks, the search time required for the index increases. A multilevel index attempts to overcome this problem by splitting the index into a number of smaller indexes and maintaining an index to the indexes. Although a multilevel index reduces the number of blocks accessed when one is searching for a record, it also has problems in dealing with insertions and deletions in the index because all index levels are physically ordered files. A **dynamic multilevel index** solves this problem by leaving some space in each of its blocks for inserting new entries. This type of index is often implemented by using data structures called **B-trees** and **B$^+$-trees**, which are supported by most DBMSs.

Most DBMSs give the designer the option to set up indexes on any fields, thus achieving faster access at the expense of extra storage space for indexes, and overheads when updating. Because the indexed values are held in a sorted order, they can be efficiently exploited to handle partial matching and range searches, and in a relational system they can speed up join operations on indexed fields.

We will see in Chap. **??** that distinctive characteristics of data warehouses require physical design solutions that are different from the ones required by DBMSs in order to support heavy transaction loads.

2.6 Summary

This chapter introduced the background database concepts that will be used throughout the book. We started by describing database systems and the usual steps followed for designing them, that is, requirements specification, conceptual design, logical design, and physical design. Then, we presented the Northwind case study, which was used to illustrate the different concepts in-

troduced throughout the chapter. We presented the entity-relationship model, a well-known conceptual model. With respect to logical models, we studied the relational model and also gave the mapping rules that are used to translate an entity-relationship schema into a relational schema. In addition, we briefly discussed normalization, which aims at preventing redundancies and inconsistency in a relational database. Then, we presented two different languages for manipulating relational databases, namely, the relational algebra and SQL. We finished this introduction to database systems by describing several issues related to physical database design.

2.7 Bibliographic Notes

For a general overview of all the concepts covered in this chapter, we refer the reader to the textbook [**?**]. Other relevant database textbooks are [**?**, **?**]. An overall view of requirements engineering is given in [**?**]. Conceptual database design is covered in [**?**] although it is based on UML [**?**] instead of the entity-relationship model. Logical database design is covered in [**?**]. A thorough overview of the components of the SQL:1999 standard is given in [**?**, **?**], and later versions of the standard are described in [**?**, **?**, **?**]. Physical database design is detailed in [**?**].