Database: a light introduction (DRAFT)

wait until the end of the course for the improved version

Linda Anticoli
Dept. of Computer Science, Mathematics and Physics
University of Udine
linda.anticoli@uniud.it

Contents

1 Preliminaries and Notation

In this Section we will describe the role and main functionalities of a database system. Nowadays databases are widely used in many aspects of our daily lives: when we perform a bank deposit or withdrawal, we book a visit at the hospital, we use the library's electronic catalog to retrieve a book, and so on, we are implicitly relying on a database. The aforementioned operation use the so–called **traditional** database, i.e., the ones in which data are usually stored in a text or number format.

Many different kind of databases emerged, e.g., **multimedial** databases, in which data are combinations of text, images, video and sound (e.g., Shazam/Soundhound, to retrieve music from sound files). Another type of database are **GIS**, or geographic information systems, which are able to store and manipulate data from geographic maps, satellite images, meteorological data, and so on. Other database are **data warehouses**, **real–time** and **active** ones, mainly used in industry and for control processes.

How can we define a database?

> A database is an **organised collection** of **related data**.

Let us now clarify the term data and how a datum is related to the more abstract concept of *information*.

> **Data** refers to the set of values of qualitative or quantitative variables.

Data are not automatically converted into information, since a set of data is informative only if it is coherently organised for some purpose. Data can be any character, text, word, number, and, if not put into context, means little or nothing to a human. However, information is data formatted in a manner that allows it to be utilized by human beings in some significant way.

> **Example:**
>
> **Data**:
>
> `ITALY, 33100, Linda, Anticoli, Via delle Scienze, Udine, Room NN1`
>
> **Information**:
>
> Linda Anticoli
> Via delle Scienze Room NN1
> 33100 Udine
> ITALY

It is important to stress that the term database is often related to the software used to organise data, i.e., a database management system, or **DBMS**, which is an integrated set of computer software allowing users to interact with one or more databases and providing access to all of the data contained in the database.

The main functions that existing DBMS provide are the following:

1. **Data manipulation and update**: involves the creation, modification, removal, sharing and organisation of data.

2. **Retrieval**: through queries, the DBMS provides information in a form directly usable or for further processing by other applications.

3. **Administration**: protection and maintenance of existing data.

1.1 Database design: from abstraction to implementation

We will now address to the design of the database structure that will be used to store and manage data rather than the design of the DBMS software; by focusing on data we will remove all the non–required technical information about organisation, memory, etc, which are not relevant in the preliminary phase. The most important thing is the domain analysis, in which we identify in an abstract way the relevant data and the relationship occurring among them, that later will be fed to the DBMS.

In order to achieve a good abstraction of our domain we use a data model.

> A **data model** is a simplified abstraction of real–world events and objects. It describes the main elements of a domain and how they relate to one another.

Once the data model has been specified, we usually make a step further into the database design process by defining the database schema.

> A **database schema** is the description, through a formal language (e.g., chart, tables, etc.) of how the data should be organised into the database.

A schema, once designed, changes very little. Formally, a database schema defines the variables in data structures; the value of these variables at a moment of time is called the **database instance**.

Hence, the design of a database can be roughly divided into three main phases:

1. Conceptual: high-level description, close to the way humans perceive data. It identifies the potential objects from the initial requirements. It uses data models.

2. Logical: process of deciding how to arrange, by using a formal language, the attributes of the objects of the domain into database structures, such as the tables. It maps entities, relationships, and attributes into a logical schema.

3. Physical: actual instance of the logical schema; it involves implementing the details of how data are stored by using a DBMS.

The aforementioned phases usually involve different translation steps from one level to another, following a set of predefined rules which will be listed more in–depth in the following sections. In this lecture notes we will focus on *relational* databases. Other examples can be found in [1].

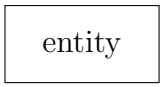
2 Conceptual Design: the Entity/Relationship model

The Entity/Relation (ER) model is an abstract data model widely used in database design, in particular for relational databases. It is a conceptual model, which describes entity types (e.g., the things of interest) and specifies relationships that can exist between instances of those entity types.

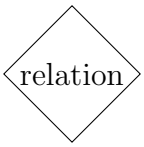
The ER diagrams represent the visual counterpart of the conceptual level of database design.

In the following we will describe the main concepts in the ER, together with their diagram representation:

- **Entity**: it is a thing of the real–world capable of an independent existence that can be *uniquely* identified. An entity is an abstraction from the complexities of a domain. An entity can be a physical object (e.g., a person, a car, a tree, a toy, etc.) or an abstract concept or institution (e.g., a university course, a job, an organisation, etc.). An entity is represented as a rectangle:



- **Relationship**: it describes how entities are related to one another. Relationships can be thought of as verbs, linking two or more concepts. A relationship is represented by a diamond shape as follows:

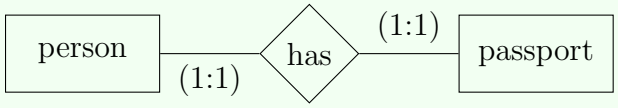


  There exist different kinds of binary relationship; in the following we list some of them which are based on cardinality and participation constraints:

  – $1:1$ relationship: an entity occurs a minimum of 1 and a maximum of 1 times in the relationship with another entity.

Example:

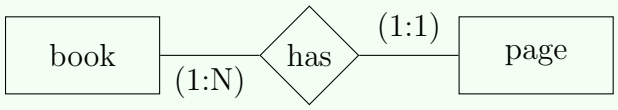
One person has one passport (from right to left: one passport belongs to one person).



  – $1:N$ or $N:1$ relationship: an entity occurs a minimum of 1 and a maximum of N (or vice–versa) times in the relationship with another entity.

Example:

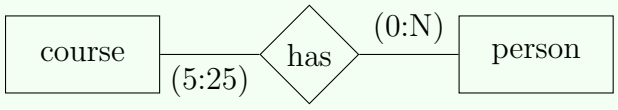
One book may have one or more pages (from right to left: a page only belongs to one specific book).



- $M : N$ relationship: an entity occurs a minimum of M and a maximum of N times in the relationship with another entity. If $M = 0$ then the relationship has a *partial participation*, while if $M > 0$ the relationship has a *total participation*

Example:

A course must be attended by at least 5 people and it is full when the participants are 25 (from right to left: a person may attend 0 or more courses).

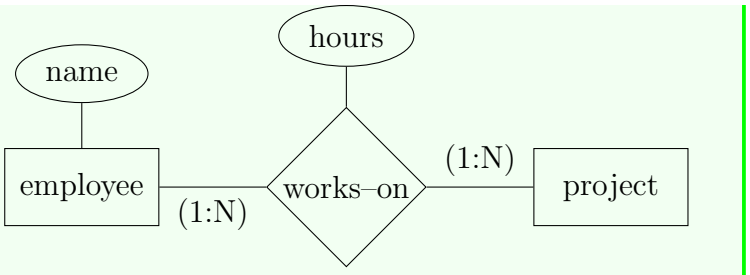


It is also possible to have ternary or n–ary relationships, but we will restrict to the binary case, which turns out to be more efficient.

- **Attribute**: both entity and relationships can have one or more attributes to describe particular properties. There are different types of attributes, and they are represented as ovals in which the border changes according to the type:
  - *Atomic* vs. *Composite*: atomic attributes are those which cannot be expanded in other attributes.

Atomic Attributes

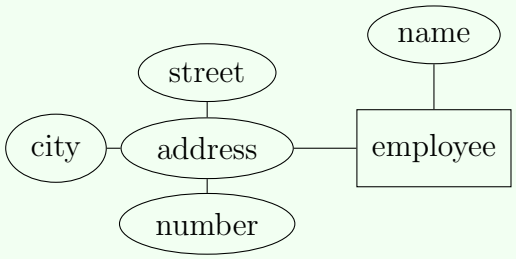
An employee has a name and works a given number of hours on (one or more) projects. *Name* is a simple attribute of *employee* and *hours* is a simple attribute of *work-on*.



Composite attributes can themselves have attributes and form a hierarchy.

Composite Attribute

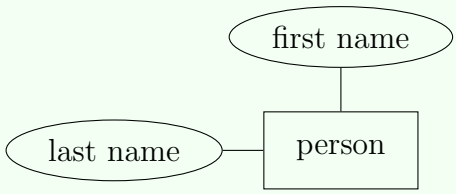
A person has a name and an *address*; the address is composed by a other attributes such as *city*, *street* and *number*.



– *Single*–valued vs. *Multi*–valued: an attribute is considered single–valued it has at most one value associated with it. They are represented as atomic attributes, in an oval with a single line for a border.

Single–valued Attribute

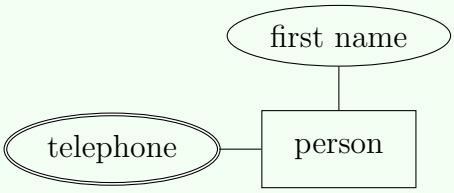
In Italy a person has just one first name and only one last name;



On the contrary, an attribute is considered multi–valued if it can have many different values associated with it. Multi–valued attributes are represented by using a double border.

Multi–valued Attribute

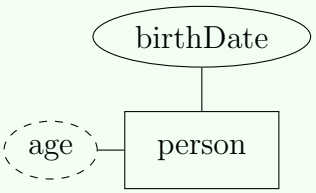
A person may have one or *more* telephone numbers.



– *Stored* vs. *Derived*: the value of a stored attribute cannot be determined from the values of other attributes. It is represented by a solid border (as atomic, single or multi–valued attributes), while if an attribute's value can be determined from the values of other attributes, then the attribute is called derived and it is represented by a dashed oval:

Stored and Derived Attributes

The birth date of a person cannot be derived from other attributes, while his/her age can be derived from the birth date.



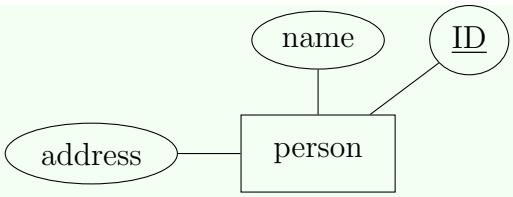
- **Key**: for each entity type, it is useful to have an attribute, or a set of attributes, to uniquely identify entities.

A key is an attribute, or a minimal set of attributes, that uniquely identify entities in an entity set. Minimal set in the sense that all of the attributes are required (and none of them can be omitted).

A key is usually represented by underlying the name of the key attribute(s).

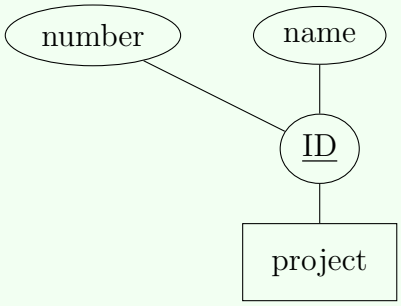
Atomic Key

A person is uniquely identified by a code, called National Insurance Number (ID), which is unique for each person



Composite Key

Many attributes can be grouped to form a key. A project is uniquely determined by its name and code number.



In the example neither *name* nor *number* are keys themselves, but their combination forms a key attribute that we have called *ID*. An entity can also have more than a key attribute.

Each attribute is associated with a set of values (domain of the attribute). Some attribute admit the value **NULL**; e.g., the entity *person* can have as attributes *first name*, *middle name* and *last name*, the attribute *middle name* is not guaranteed to exists, hence it can take value null, while *first name* and *last name* cannot (a person has always at least a name and a surname). Key values can never be NULL.

We recall that, according to the context, binary $1:1$ relationships can be more efficiently modelled as attributes.

We have mentioned some of the main ingredients of the Entity/Relationship model of data. For a more in–depth analysis of the features of this model we address to the following references: [1].

2.1 Exercises:

1. Model the following scenario: An actor has a name which uniquely identifies him/her, an age, and a nationality; an actor may have 0 or more telephone numbers and he/she works in 0 or more movies, for a defined period of time. Each movie casts one or more actors and it is identified by its title and year, moreover each movie has also a producer and a director.

2. Olympic games happen in a certain year at a certain place. Each year, there is at most one instance of Olympic games. In each discipline of an olympic game, there is exactly one gold medalist, one silver medalist, and one bronze medalist. All these medalists are athletes.

3. A person has a name and an age. Cities have a name and are located in a country. Every year, persons can form groups in order to travel together to a city. A person may be part of the same or a different group in different years, but may be part of at most one group in any given year. Furthermore, a group travels to the same or different city in different years, but travels to exactly one city in any given year.

4. Assume there is a library system with the following properties. The library contains one or several copies of the same book. Every copy of a book has a copy number and is located at a specific location in a shelf. A copy is identified by the copy number and the ISBN number of the book. Every book has a unique ISBN, a publication year, a title, an author, and a number of pages. Books are published by publishers. A publisher has a name as well as a location. Within the library system, books are assigned to one or several categories. A category can be a subcategory of exactly one other category. A category has a name and no further properties. Each reader needs to provide his/her family name, his/her first name, his/her city, and his/her date of birth to register at the library. Each reader gets a unique reader number. Readers borrow copies of books. Upon borrowing the return date is stored.

5. We must plan a University Database and we know the following information:

   - Professors have an ID, a name, an age, a rank, and a research specialty.
   - Projects have a project number, a sponsor name (e.g., NSF), a starting date, an ending date, and a budget. Each project is managed by one professor (known as the projects principal investigator) and it is worked on by one or more professors (known as the projects co-investigators).
   - Graduate students have an ID, a name, an age, and a degree program (e.g., M.S.or Ph.D.).
   - Professors can manage and/or work on multiple projects. Each project is worked on by one or more graduate students (known as the projects research assistants).
   - When graduate students work on a project, a professor must supervise their work on the project. Graduate students can work on multiple projects, in which case they will have a (potentially different) supervisor for each one.
   - Departments have a department number, a department name, and a main office. Departments have a professor (known as the chairman) who runs the department.Professors work in one or more departments, and for each department that they work in, a time percentage is associated with their job.
   - Graduate students have one major department in which they are working on their degree.Each graduate student has another, more senior graduate student (known as a student advisor) who advises him or her on what courses to take.

2.2 The Enhanced Entity/Relationship Model

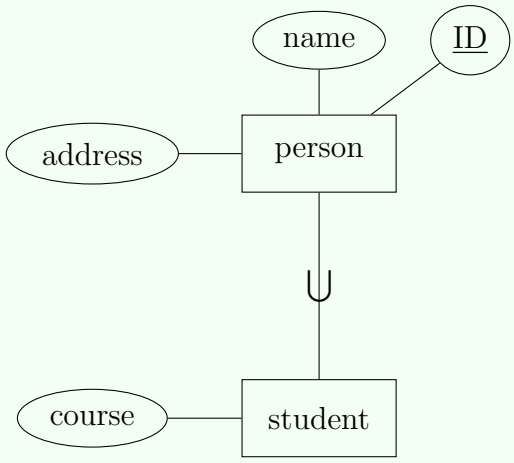
E/R notions are sufficient to represent in a conceptual way "traditional" databases, where there are few or none constraints on data. Nowadays, in particular *CAD/CAM* applications, telecommunications and geographic information systems *GIS* need more complex databases, hence an E/R diagram turns out to be not sufficient. To cope with this lack of a suitable data model many techniques has been proposed, from knowledge–representation to the object–oriented one; in this Section we will focus on a data model which uses the E/R as a starting point, namely the EER model, or Enhanced Entity Relationship.

The EER model uses all the basic concepts of the E/R one; moreover it adds the following notions:

- **Subclass and superclass**: subclass entity inherits all attributes and relationships of superclass. Subclasses are indicated by putting a set inclusion relation symbol ($\cup$) along the edge from the subclass to the superclass.

Subclass

The entity *student* is a subclass of the entity *person*, of which it inherits all the attributes; moreover *student* has its own attributes (course) which are not proper of all people.



- **Generalisation and specialisation**: Specialisation is the process of defining a set of subclasses of an entity type defined on the basis of some distinguishing characteristic of the entities in the superclass.

  Subclasses inherit all the attributes of the superclass but can also have specific attributes and specific relationships. The key is the same as the one of the parent class. Where different subclasses of a single superclass are present, then they can be disjointed, which is graphically depicted by a circle including the letter **d** or the can overlap, which is graphically depicted by a circle including the letter **d**, along the edges linking the subclasses to the parent class.

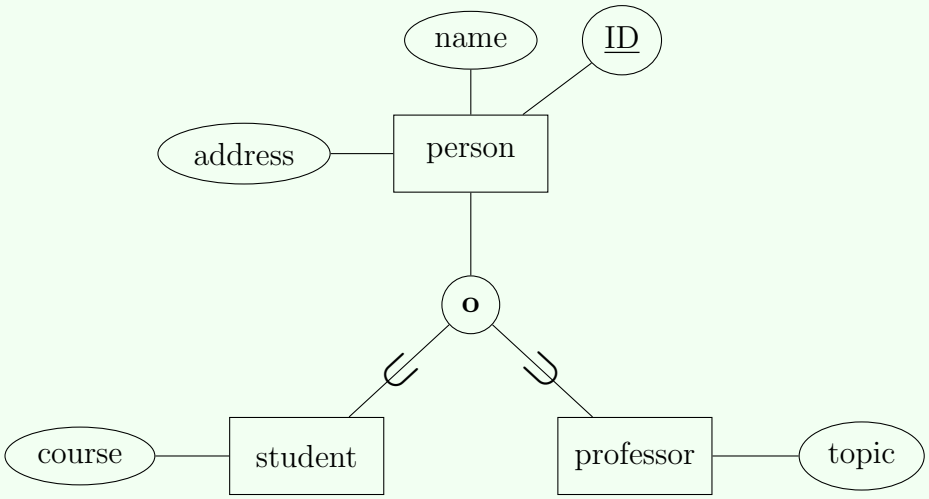
Specialisation can be **total** (each member of superclass must be in some subclass) or **partial** which are depicted by:

- Total: double line connecting superclass to specialisation circle;
- Partial: single line.

Some examples are visible in the following pictures.

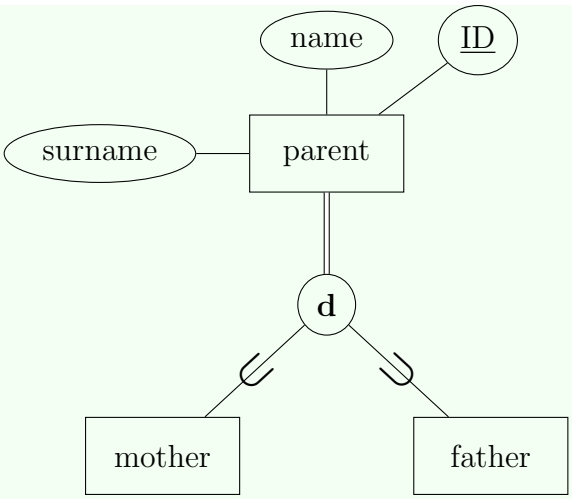
Specialisation 1

The entity *person* has been specialised into two subclasses *student* and *professor*. The following specialisation is partial, since a person can be a student, a professor, but also an electrician or can be unemployed. The subclasses can admit overlap, since a professor can also be a student (if, for example, he/she is taking his/her second degree in another topic).



Specialisation 2

The entity *parent* has been specialised into two subclasses *mother* and *father*. The following specialisation is total, since a parent can only be a mom or a dad. The subclasses are disjoined, since a mother cannot also be a father at the same time.



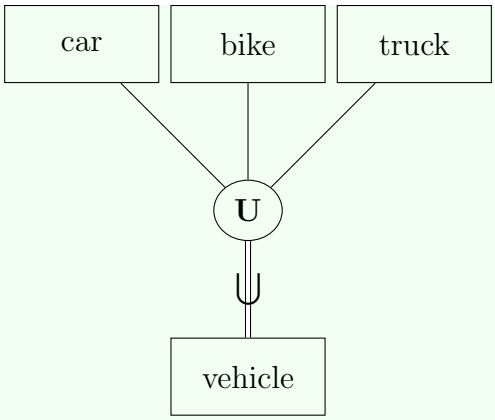
Generalisation is the reverse process: instead of specialise a parent class into many different subclasses, we generalise many subclasses which share common relationships and attributesv into a single superclass.

- **Union types**: a union type is a subclass related to a collection of superclasses. Each instance of the subclass belongs to one, not all, of the superclasses. An example can be seen in the following Figure, in which the entity *sponsor* is a union type of *team*, *department* and *club* superclasses. Union types usually have different keys from the parent superclass, since they can be entities on their own.

  A union can be **total**, every member of the sets that make up the union must participate (double line from union circle to subset), or **partial** not every member of the sets must participate (single line).

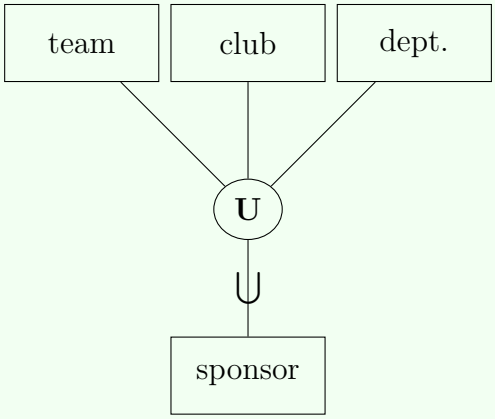
Total union

The entity *vehicle* results from the union of three superclass *car*, *bike* and *truck*. The union is total, since each *car*, *bike* and *truck* is surely a *vehicle*.



Partial union

The entity *sponsor* results from the union of three superclasse *club*, *team* and *dept.*. The union is partial, since each of them is not necessary a *sponsor*.



Further information and details about the EER model can be found in [1].

2.3 Exercises

1. You have gotten a job planning databases for the European Union. Your first on job assignment is to help the various countries maintain information about their inhabitants. Your model should capture the following information:

   - In each country, there are provinces, which contain towns. There cannot be two provinces with the same name in a single country. Similarly, there cannot be two towns with the same name in a single province.
   - People live in towns. Men and women work in a town. Children learn in a school in a town.

- A person can be a man, a woman, or a child, and has a first-name, last-name, id, and birthday. Children are any people under the age of 18.
- A man can be married to a woman (polygamy is not allowed, i.e.,one man can be married only to one woman). Divorce, and subsequent remarriage, is possible.
- For each marriage, store the date of the marriage and information about who are the children of the married couple.

2. Design a database to keep track of information for an art museum. Assume that the following requirements were collected:

   - The museum has a collection of ART_OBJECTs. Each ART_OBJECT has a unique IdNo, an ARTIST (if known), a Year (when it was created, if known), a Title, and a Description. ART_OBJECTs also have information describing their country/culture using information on country/culture of Origin (Italian, Egyptian, American, Indian, etc.), Epoch (Renaissance, Modern, Ancient, etc.).
   - ART_OBJECTs are categorized based on their type. There are three main types: PAINTING, SCULPTURE, and STATUE, plus another type called OTHER to accommodate objects that do not fall into one of the three main types (hence type description is needed).
   - A PAINTING has a PaintType (oil, watercolor, etc.), material on which it is DrawnOn (paper, canvas, wood, etc.), and Style (modern, abstract, etc.).
   - A SCULPTURE has a Material from which it was created (wood, stone, etc.), Height, Weight, and Style.
   - A STATUE also has a Style.
   - An art object in the OTHER category has a Type (print, photo, etc.) and Style.
   - ART_OBJECTs are also categorized as PERMANENT_COLLECTION that are owned by the museum (which has information on the DateAcquired, whether it is OnDisplay or stored, and Cost) or BORROWED, which has information on the Collection (from which it was borrowed), DateBorrowed, and DateReturned.
   - The museum keeps track of ARTISTs information, if known: Name, DateBorn, DateDied (if not living), CountryOfOrigin, Epoch, MainStyle, Description. The Name is assumed to be unique.
   - Different EXHIBITIONs occur, each having a Name, StartDate, EndDate, and is related to all the art objects that were on display during the exhibition.
   - Information is kept on other COLLECTIONs with which the museum interacts, including Name (unique), Type (museum, personal, etc.), Description, Address, Phone, and current ContactPerson. Draw an EER schema diagram.

3 Logical Design: the Relational Data Model

Since 1980s , the relational model has been implemented in a large number of commercial systems.

The relational model, or relational schema, is a method of structuring data using relations, which are table–like mathematical structures, consisting of columns and rows. Each row in the relation represents a collection of data values related among them.

The formal terminology, which will be explained in this section is the following:

Table → Relation
Column → Attribute
Row → Tuple

For each attribute appearing in a column we have a *domain* of possible values.

- **Domain**: is the set of values that an attribute can take. The domain can be specified by using a logical definition, or conversely a data–type definition (also called *format*).

Attribute	**Logical Definition of Domain**	**Format**
it_phone_number	set of 10 digits valid in Italy	$(nnnn)\ nnnnnn$
ID	string of 7 digits and 9 letters in a given order	$LLLLLddLddLdddL$

- **Attribute**: each attribute A_i is represented as a column header in the table, and it has a domain which is indicated by $\mathcal{D}(A_i)$. A_i can have a name or not; in the case in which A_i has a name then its position in the table is not important and can be changed, while if it does not have a name its position becomes crucial. In the same table there cannot be two attributes with the same name.

- **Relation**: it is a table which has a name and a list of attributes; we can calculate the *degree* of a relation, i.e., the number n of its attributes (columns). In this context is important to distinguish between:

  1. Relation Schema $R(A_1, \ldots, A_n)$, which is an abstract definition of the relation, in which we denote only the name of the relation together with the list of its attributes:

     PERSON(Name, ID, Birth_date) has degree 3 and can be represented as a table:

     PERSON

ID	Name	Birth_date

  2. Relation Instance (or state) $r(R)$, which is the set of tuples (row) of the table. The number of tuples is called *cardinality*. Each tuple τ is represented by $\tau = \{(\mathrm{A}_1, \mathrm{value}_1), \ldots, (\mathrm{A}_n, \mathrm{value}_n)\}$. Tuples do not have a particular order.

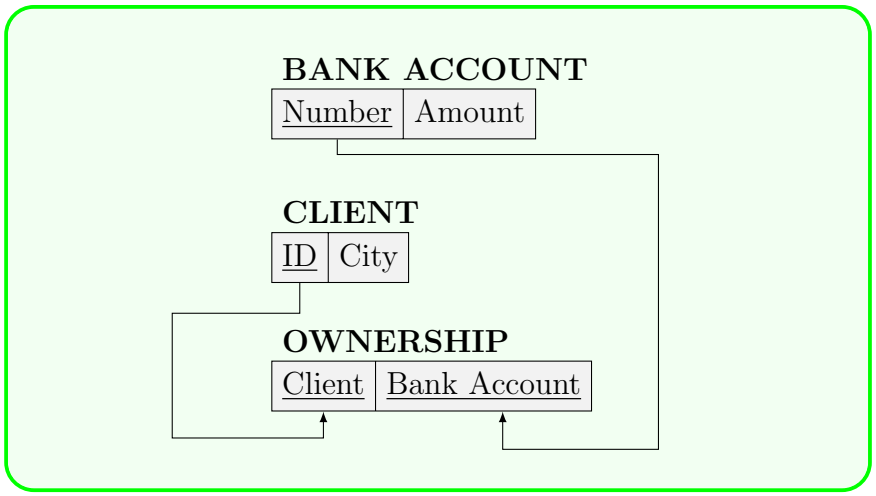
PERSON	ID	Name	Birth_date
	LND88	Linda	02/11/1988
	MRC87	Marco	04/12/1987
	GDA91	Giada	22/08/1991

In the above relation we have cardinality 3. Let's consider the first tuple, it can be represented as : $\{(\underline{\text{ID}}, \text{LND88}), (\text{Name}, \text{Linda}), (\text{Birth\_date}, 02/11/1988)\}$

We listed the main characteristics of single relations but, since in a relational database there will be many relations whose tuples are linked together by some kind of relationship, we should discuss the various restrictions, or *constraints*, on data that can be specified on a relational database. In the context of these lectures we will restrict to the integrity constraints; for a deeper understanding please see reference [1]. An **integrity constraint** is a constraint always true, e.g., the finiteness of a relational database. We can divide the integrity constraints into two main categories:

- **Intra–relational constraints**: are those constraints involving tuples belonging to the same relation. Some of these constraints are:
  - Domain constraints: each attribute has a domain, e.g. *name* is a string of characters, if a number is inserted it returns an error, or the *not_null* constraint, which prevents key attributes to be left blank.
  - Primary Key constraints: a key attribute is always not_null and no two distinct tuples in any state $r(R)$ can have the same key value (uniqueness of the key). In a relational schema there can be multiple keys, that we call *candidate key*. In general it is better to identify one of them (usually the key with the least number of attributes correlated) as *primary key*.

    The primary key is one or more columns that uniquely identifies the row.
- **Inter–relational constraints**: constraints that involve several relations, the most important one is the *Referential integrity* one: an attribute named in one relation must refer to tuples that describe it in another.



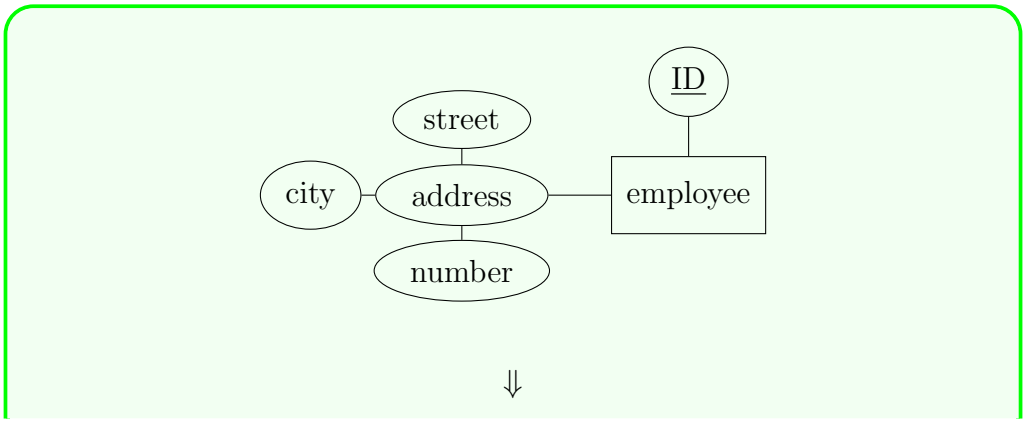
Attribute *Client* and *Bank Account* are foreign keys of *OWNERSHIP* referring to attribute *ID* in *CLIENT*, and *Number* in *BANK ACCOUNT* and it is always true that if the value of *Client* changes in *OWNERSHIP*, there is a tuple of *CLIENT* in which *ID* changes accordingly, the same happen with *Bank Account*.

It is important to remember that we can merge two or more tables by means of foreign keys.

3.1 Translation from E/R and EER Diagrams to Relational Schema

There are a set of rules that allow to translate from an Entity Relationship diagram into a Relational schema.

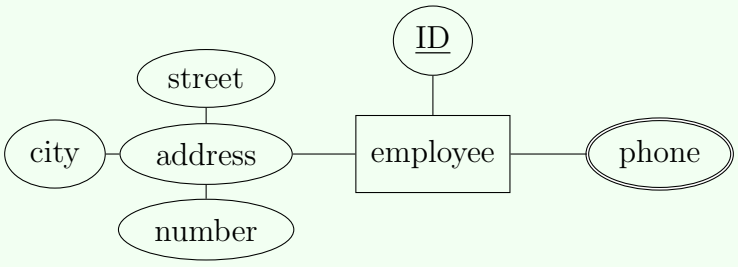
- **Entities and Simple Attributes**: entities in E/R diagrams are translated into relations (tables). It is useful to keep the names as similar as possible to the E/R model. Keys are listed in the first positions of the relation. Each atomic attribute becomes a column (attribute) in the relation. Each composite attribute is translated as the list of its component attributes.



EMPLOYEE

ID	Street	Number	City

- **Multi–Valued Attributes**: each multi–valued attribute is mapped to a new relation R. R includes an attribute corresponding to the multi–valued one, the primary key attribute of the parent entity as a foreign keys. The primary key of R is the composition of the multi–valued attribute and the foreign key. If the multi–valued attribute is composite, we include its simple components.



⇓

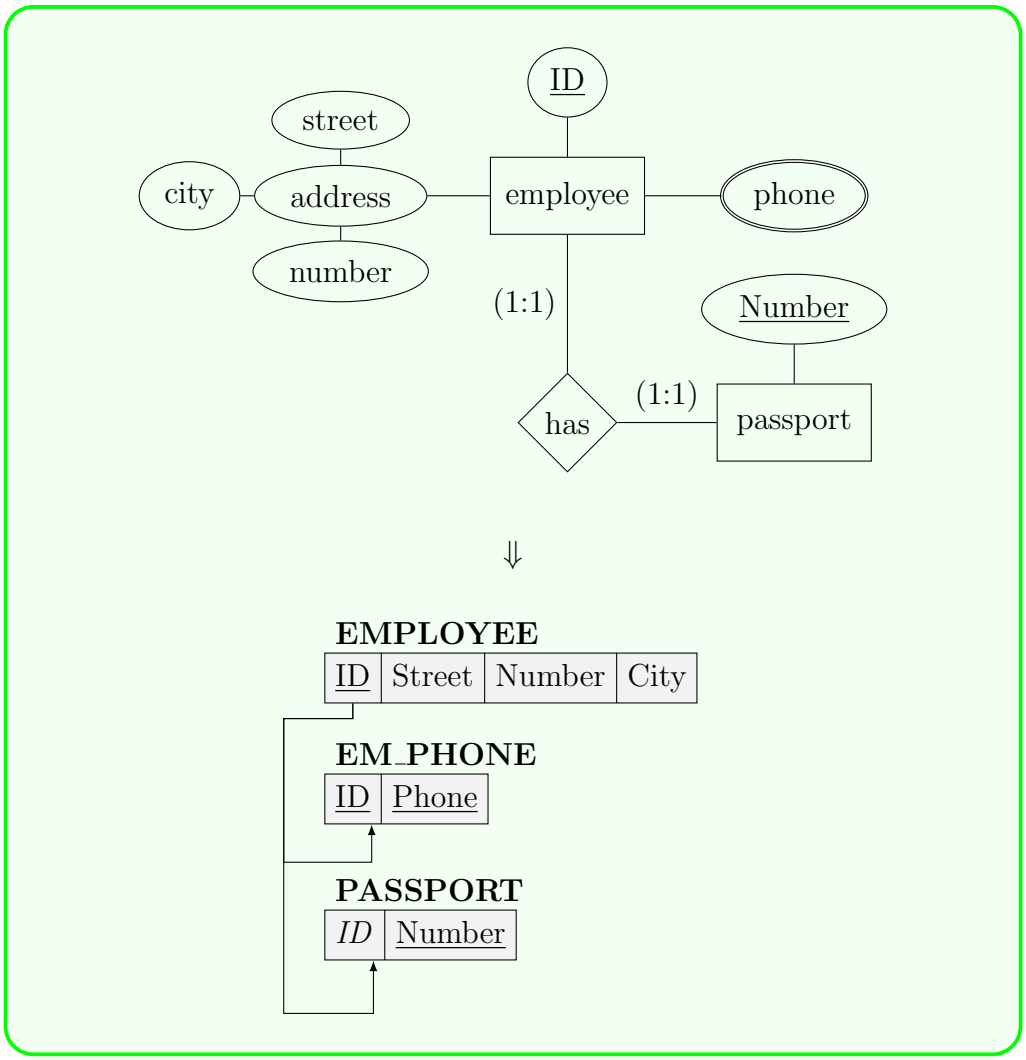
EMPLOYEE

ID	Street	Number	City

EM_PHONE

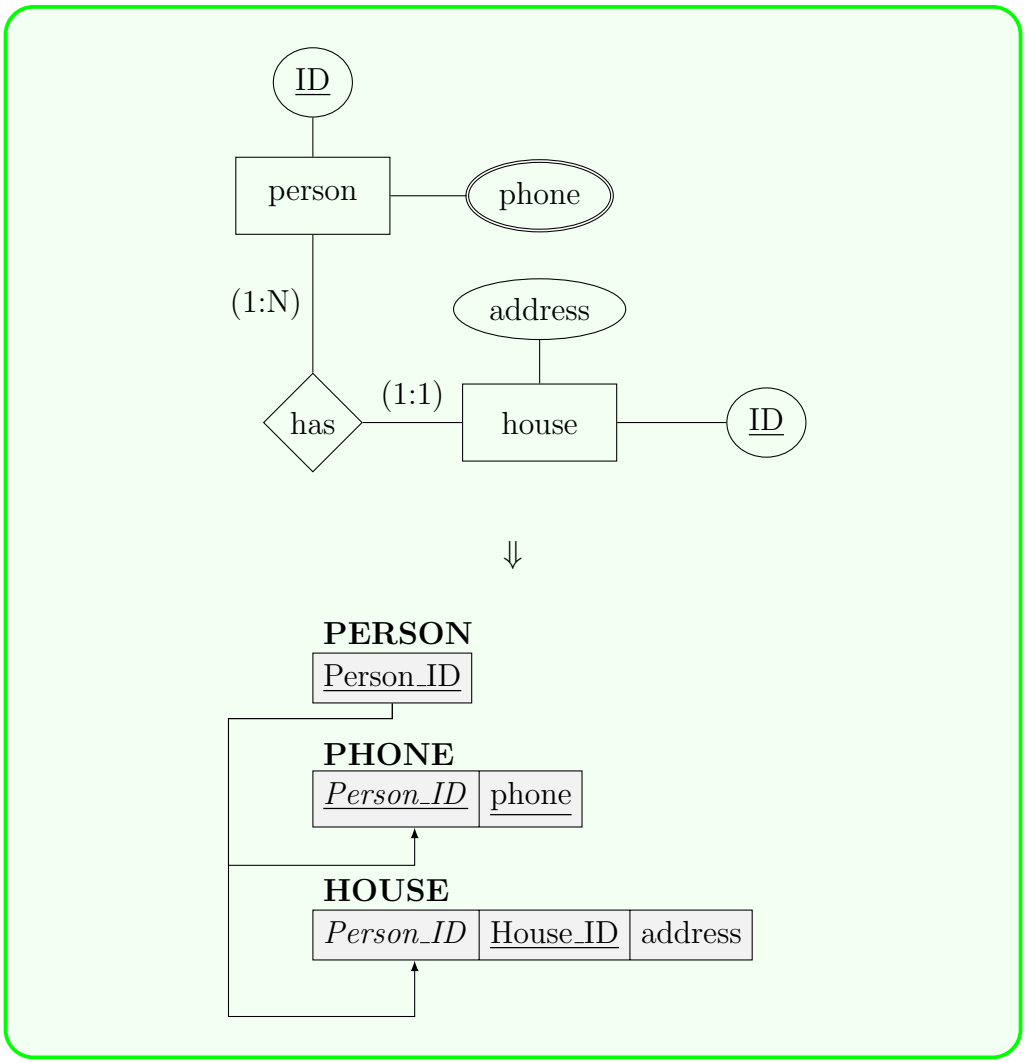
ID	Phone

- **Binary 1:1 Relationships**: for each binary 1:1 relationship in the E/R diagram we should identify the entities participating the relationship, then different approaches are possible. we will recall the simplest one, which is suitable to translate the wide majority of exercises. *Foreign key approach*: unless special cases, this represents the method on which we will rely for the translation. We choose one of the entity participating in the relationship (it is better to choose an entity with total participation) and include as a foreign key in it the primary key of the remaining entity. We then include all the simple attributes of the 1:1 relationship as attributes of the chosen entity.

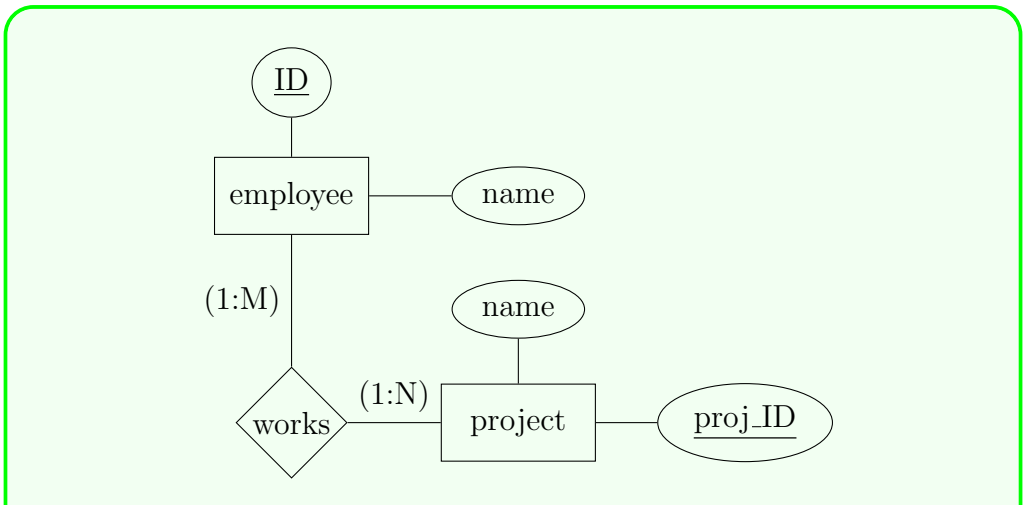


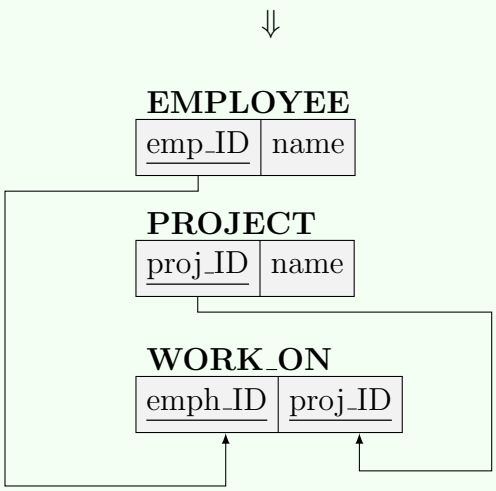
For other approaches see [1, 2].

- **Binary 1:N Relationships**: for each binary l:N relationship, we identify the entity E_1 from the N–side of the relationship. We then include as foreign key in E_1 the primary key of the other entity E_2 participating; in short, the primary key on the 'one side' of the relationship is added to the 'many side' as a foreign key.



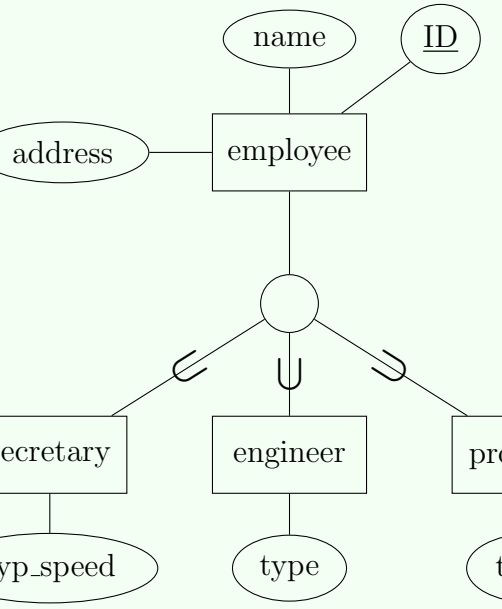
- **Binary M:N Relationships**: a new relation is produced which contains the primary keys from both sides of the relationship; these primary keys form a composite primary key.

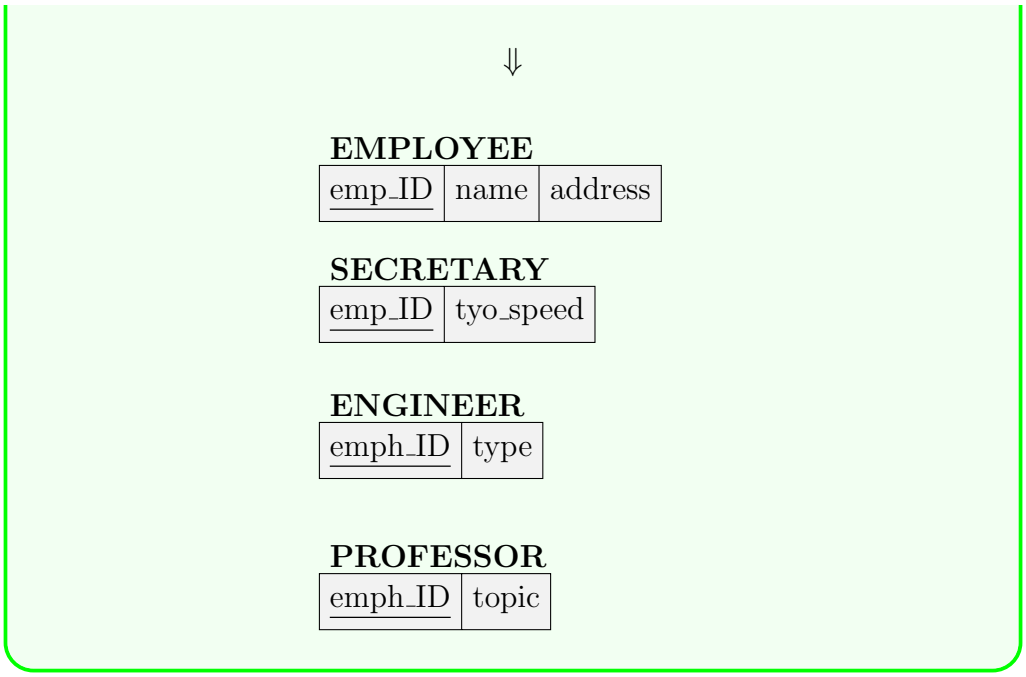




- **Specialisation and Generalisation**: there are many different methods in order to translate the specialisation hierarchy of an EER diagram into a relational schema; nevertheless, since a more in depth analysis is out of the scope of these introductory lectures, we will focus on a very general techniques that works for any specialisation constraint, overlapping, disjoint, partial or total.

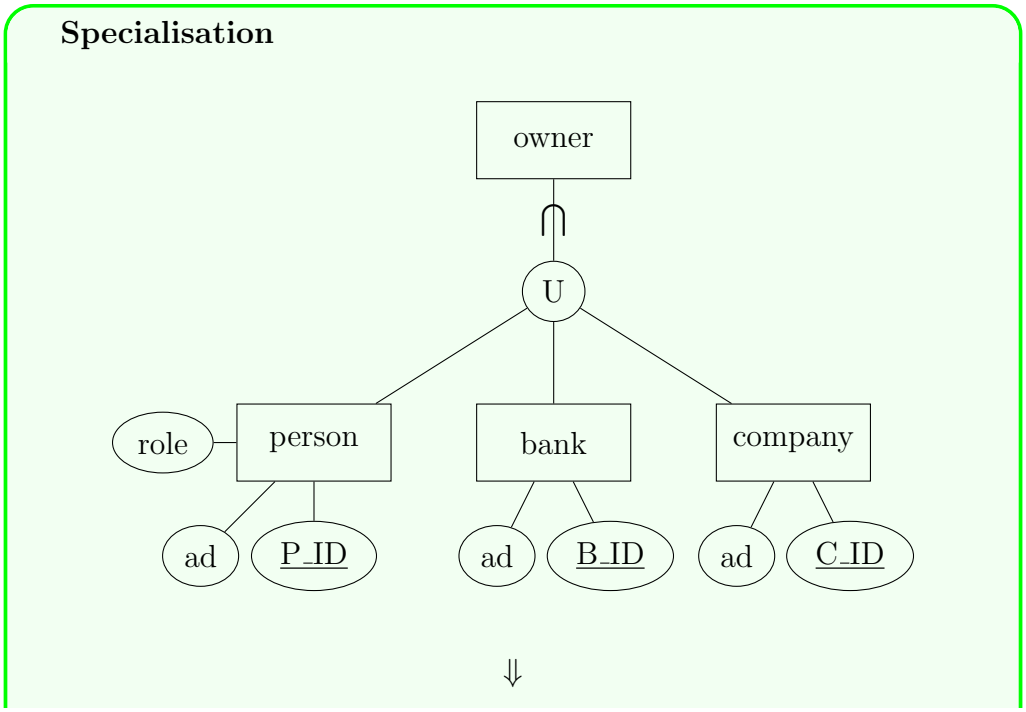
  Given a superclass E (parent entity) with many subclasses $e_1, \ldots, e_n$, the easiest technique is to create a relation for E where the attributes are only its own attributes; then a relation for each subclass of E is created. The attributes are the attributes of the subclass and the primary key of the superclass is mapped into each subclass and becomes the subclasses primary key.

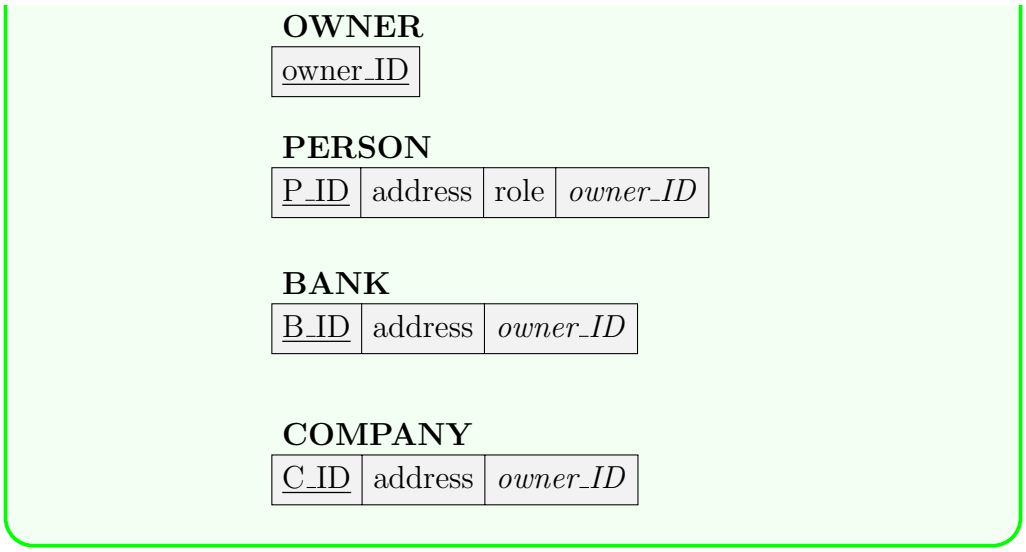




This example represents the simplest technique to translate from EER to Relational schema; it can be very bulky sometimes, and more sophisticated and elegant ways can be found for specific situations. These methods are illustrated in [1].

- **Union Types**: in order to map a union type whose superclasses have different keys, we should specify a new key attribute, called a *surrogate key*, when creating a relation to correspond to the category. This is because the keys of the defining classes are different, so we cannot use anyone of them exclusively to identify all entities in the union.





In this Section we briefly tackled how to model an EER schema into a logical, relational schema. Far from being an omni–comprehensive lecture, we recall the following references for a more in–depth analysis of more difficult cases and translations.

3.1.1 Exercises:

Translate all the exercises about the E/R and EER diagrams from the previous section to a relational schema.

3.2 Queries and Operations: Relational Algebra

We can access the data of a database through *queries*. There are many query–languages. In this Section we will introduce some basic concepts about *Relational Algebra*, a procedural query language, allowing to access and manipulate data in a relational schema. It is closed with respect to the set of relations, which means that it takes in input relations and produces as output relations.

Relational algebra has some limitations, since the relational model allows finite relations only; hence relational algebra must comply to this constraint making some operations (e.g., complement of a set) difficult to perform. Complementation is important, since relational algebra does not deal with negative information, which are obtained by complement of the positive ones.

Anyways, relational algebra is an integral part of the relational data model, and its operations can be divided into two groups:

1. set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples in the formal relational model. Set operations include union ($\cup$), intersection ($\cap$), set difference ($\setminus$), and Cartesian product ($\times$).

2. operations developed specifically for relational databases, e.g., SELECT, PROJECT, and JOIN, among others.

At first we will introduce unary relational operations, then we will proceed towards more difficult, binary and aggregate operations.

3.2.1 UNARY RELATIONAL OPERATIONS

SELECT: it is used to select a *subset of the tuples* from a relation satisfying the *selection condition*. SELECT operation acts as a filter that keeps only those tuples that satisfy the required condition. It may reduce the cardinality of a relation (less tuples), without altering the degree (same attributes); as a result we graphically get a table with the same number of columns, but with less rows.

The SELECT operation acting on a relation R, according to a Boolean condition, is indicated as follows:

$$\sigma_{<\text{CONDITION}>}(R)$$

and the Boolean condition is composed by:

$$< \text{attribute} >< \text{comparison op} >< \text{constant} >$$

Let's consider the following example in which we want to select from the table EMPLOYEE, all the people whose salary is less than $30k$ Euros. The query can be written as follows:

$$\sigma_{\text{Salary}<30k}(\text{EMPLOYEE})$$

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
LNDNTC	PHY	12k
MRCBSD	INF	15k
DRDBRL	ING	30k
GNLNDR	INF	31k
RSSGCM	ING	50k

(before)

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
LNDNTC	PHY	12k
MRCBSD	INF	15k

(after)

It is possible to notice that the degree (number of attributes) has not changed, but the number of tuples has decreased.

The SELECT operation is commutative it can be nested and applied on intermediate results;

$$\sigma_{\text{COND}_1}(\sigma_{\text{COND}_2}(R)) = \sigma_{\text{COND}_2}(\sigma_{\text{COND}_2}(R)) = \sigma_{\text{COND}_1\text{COND}_2}(R)$$

moreover SELECT can be composed by means of Boolean operators (*and* $\wedge$, *or* $\vee$, *not* $\neg$, *xor* $\oplus$, etc.) [1].

As an example, always considering the relation EMPLOYEE, we want to select all the people belonging to the department of Informatics, with a salary less than $30k$ Euros.

[1] For this reason we suggest a quick review of the truth tables for the mentioned Boolean operators.

$$\sigma_{\text{Dept=INF } \wedge \text{ Salary<30}k}(\text{EMPLOYEE})$$

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
LNDNTC	PHY	12k
MRCBSD	INF	15k
DRDBRL	ING	30k
GNLNDR	INF	29k
RSSGCM	ING	50k

(before)

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
MRCBSD	INF	15k
GNLNDR	INF	29k

(after)

PROJECT: selects certain columns from the table and discards the other ones. If we are interested in only a subset of attributes of a relation, we use the PROJECT operation to project the relation over these attributes. PROJECT reduces the degree (since it selects only a subset of the columns) of the relation. If we do not insert a key attribute among the projected ones, PROJECT may reduce also the cardinality, since tuples (rows) with identical values are eliminated and only one of them is kept (it removes duplicate tuples).

The PROJET operation acting on a relation R, selecting a list of attributes, is indicated as follows:

$$\pi< \text{ATT. LIST} >(R)$$

.

PROJECT operation is non–commutative, hence:

$$\pi_{\text{LIST1}}(\pi_{\text{LIST2}}(R)) \neq \pi_{\text{LIST1}}(\pi_{\text{LIST2}}(R))$$

moreover, a constraint is that the final attribute list must always be smaller than the first one:

$$\text{LIST}_1 \subseteq \text{LIST}_2$$

In the following examples we will present a projection with and without duplicate tuples removal. Consider a situation in which, from the relation EMPLOYEE, we are interested tin the attribute Dept. only. Since some departments have the same name, and there is no key to distinguish them, the duplicates are removed.

$$\pi_{\text{Dept.}}(\text{EMPLOYEE})$$

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
LNDNTC	PHY	12k
MRCBSD	INF	15k
DRDBRL	ING	30k
GNLNDR	INF	29k
RSSGCM	ING	50k

(before)

EMPLOYEE

Dept.
PHY
INF
ING

(after)

Consider a similar situation in which, from the relation EMPLOYEE, we are interested in the attribute Dept. but we also include the key attribute, Emp_ID. The resulting table behaves as follows:

$$\pi_{\text{Emp\_ID, Dept.}}(\text{EMPLOYEE})$$

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
LNDNTC	PHY	12k
MRCBSD	INF	15k
DRDBRL	ING	30k
GNLNDR	INF	29k
RSSGCM	ING	50k

(before)

EMPLOYEE

Emp_ID	**Dept.**
LNDNTC	PHY
MRCBSD	INF
DRDBRL	ING
GNLNDR	INF
RSSGCM	ING

(after)

PROJECT and SELECT can be combined in order to perform more complex queries. As an example, we want to select from the table EMPLOYEE the ID and the Department of those employees which earn more than $20k$ Euros.

$$\pi_{\text{Emp\_ID, Dept.}}(\sigma_{\text{Salary}>20k}(\text{EMPLOYEE}))$$

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
LNDNTC	PHY	12k
MRCBSD	INF	15k
DRDBRL	ING	30k
GNLNDR	INF	29k
RSSGCM	ING	50k

(select)

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
DRDBRL	ING	30k
GNLNDR	INF	29k
RSSGCM	ING	50k

(project)

EMPLOYEE

Emp_ID	**Dept.**
DRDBRL	ING
GNLNDR	INF
RSSGCM	ING

(final)

RENAME: operation that allows to rename either the relation or the attribute names. The general RENAME operation when applied to a relation R of degree n is usually denoted by:

$$\rho_S(R)$$

where S can be either the new name of the relation R, or a list $a_1/b_1, \ldots, a_n/b_n$ where b_i are attributes of R and the a_i their renaming.

$$\rho_{\text{UNI\_STAFF(ID/Emp\_ID, Scholarship/Salary)}}(\text{EMPLOYEE})$$

EMPLOYEE

Emp_ID	**Dept.**	**Salary**
LNDNTC	PHY	12k
MRCBSD	INF	15k
DRDBRL	ING	30k
GNLNDR	INF	29k
RSSGCM	ING	50k

(before)

UNI_STAFF

ID	**Dept.**	**Scholarship**
LNDNTC	PHY	12k
MRCBSD	INF	15k
DRDBRL	ING	30k
GNLNDR	INF	29k
RSSGCM	ING	50k

(before)

The RENAME is useful when we need to perform binary operations on relations, such as UNION or DIFFERENCE operations. Indeed, for set union and set difference, the two relations involved must be *union–compatible* meaning that the two relations must have the same set of attributes (with the same name as well).

3.2.2 BINARY SET OPERATIONS

The first set of binary operations are the set operations which are used to merge the elements of two relations in various ways, and are called UNION, INTERSECTION, and DIFFERENC. Each one of these operations is applied to two relations that must have the same type of tuples; we recall that this condition has been called union–compatibility. Two relations are said to be union–compatible if they have the same degree n (same number of attributes) and each corresponding pair of attributes has the same domain.

UNION: given two relations R and S the result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R,in S, or in both. Duplicate tuples are eliminated. Union is the only set operation that can increase cardinality.

As an example, we want to merge the following two relations, MOTHER and FATHER, in a relation called PARENT. At first the two relations are not union–compatible, since they don't have the same set of attributes; we can fix it by renaming the attributes Mom and Dad with a new name: Parent.

MOTHER

Mom	**Child**
Alice	Bob
Carol	James
Carol	Evelyn

FATHER

Dad	**Child**
Jim	Bob
Edward	James
Luke	James
John	Judy

$\rho_{\text{Parent/Mom}}(\text{MOTHER})$ and $\rho_{\text{Parent/Dad}}(\text{FATHER})$

MOTHER

Parent	**Child**
Alice	Bob
Carol	James
Carol	Evelyn

FATHER

Parent	**Child**
Jim	Bob
Edward	James
Luke	James
John	Judy

Now that the set of attributes is the same we can merge the two relations and rename the resulting one as PARENT.

$$\rho_{\text{PARENT}}(\text{MOTHER} \cup \text{FATHER})$$

PARENT

Parent	**Child**
Alice	Bob
Carol	James
Carol	Evelyn
Jim	Bob
Edward	James
Luke	James
John	Judy

INTERSECTION: the result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S. Let us consider the example below, in which

we want to take the intersection between the relation containing the students of UNIUD, and the one containing the students which are members of the UNIUD swimming pool.

STUDENT

FName	**LName**
Emma	Woodhouse
Harriet	Smith
Jane	Fairfax
George	Knightley
Philip	Elton

MEMBERS

FName	**LName**
Jane	Fairfax
Harriet	Smith
Frank	Churchill
Isabella	Woodhouse

Here we do not have the need of renaming attributes, since their are the same in both relations. We can perform the intersection as follows:

$$\rho_{\text{SWIMMING\_ST}}(\text{STUDENTS} \cap \text{MEMBERS})$$

SWIMMING_ST

FName	**LName**
Jane	Fairfax
Harriet	Smith

Both UNION and INTERSECTION are commutative, hence $R \cup S = S \cup R$ and $R \cap S = S \cap R$. Moreover, since they are both associative, they are n-ary operations, thus we can extend them to any number of relations, e.g., $R \cup S \cup T \cup \dots$.

DIFFERENCE: the result of this operation, denoted by R/S, is a relation that includes all tuples (rows) that are in R but not in S.

INTERSECTION is non–commutative. Let us consider the example used for the INTERSECTION operator, but now we want the set of NON_SW_ST (non swimming students).

$$\rho_{NON\_SW\_ST}(\text{STUDENT}/\text{MEMBERS})$$

NON_SW_ST

FName	**LName**
Emma	Woodhouse
George	Knightley
Philip	Elton

CARTESIAN PRODUCT: also known as CROSS PRODUCT, and it is denoted by $R \times S$. This is also a binary set operation, but without the constraint of union–compatibility. This operation is used to combine tuples from two relations in a combinatorial fashion; hence the result of $R(A_1 \ldots, A_n) \times S(B_1, \ldots, B_m)$ is a relation $Q(A_1, \ldots, A_n, B_1, \ldots, B_m)$ with degree $n + m$. The cardinality (number of rows) of Q is denoted as $|Q| = |R| * |S|$. An important requirement is that the attributes have different names in order to avoid clashes.

The CARTESIAN PRODUCT is generally meaningless if applied by itself, while it becomes useful when followed by a selection that matches values of attributes coming from the component relations.

Let us consider, as an example, a situation in which we have the following relations: EMPLOYEE(FN, LN, Emp_ID), PROJECT(PNum, PName) and WORK_ON(ID, Proj). We want to produce a relation in which the name and surname of each employee is associated to the name of project he/she is working on (while, by now, we just have the ID and the Project number). We can proceed as follows:

$$\pi_{\substack{\text{ID}\\\text{FN}\\\text{LN}\\\text{PName}}} \left(\sigma_{\text{Proj=PNum}} \left(\pi_{\substack{\text{ID}\\\text{FN}\\\text{LN}\\\text{Proj}}} \left(\sigma_{\text{Emp\_ID=ID}}(\text{WORK\_ON} \times \text{EMPLOYEE}) \right) \times \text{PROJECT} \right) \right)$$

Let us now decompose the query step by step; the initial situation can be the following:

EMPLOYEE

FN	**LN**	**Emp_ID**
John	Doe	JND
Harry	Smith	HRS

PROJECT

PNum	**PName**
1	P1
2	P2

WORK_ON

ID	**Proj**
JND	1
HRS	2

$\Downarrow$ first cartesian product

T←WORK_ON × EMPLOYEE

ID	**Proj**	**FN**	**LN**	**Emp_ID**
JND	1	John	Doe	JND
JND	1	Harry	Smith	HRS
HRS	2	John	Doe	JND
HRS	2	Harry	Smith	HRS

$\Downarrow$ select

T2← $\sigma_{\text{Emp\_ID=ID}}(T)$

ID	**Proj**	**FN**	**LN**	**Emp_ID**
JND	1	John	Doe	JND
JND	1	Harry	Smith	HRS
HRS	2	John	Doe	JND
HRS	2	Harry	Smith	HRS

$\Downarrow$ project

T3← $\pi_{\substack{\text{ID}\\\text{FN}\\\text{LN}\\\text{Proj}}}(T2)$

ID	**Proj**	**FN**	**LN**	**Emp_ID**
JND	1	John	Doe	JND
HRS	2	Harry	Smith	HRS

$\Downarrow$ second cartesian product

T4← T3 × PROJECT

ID	**Proj**	**FN**	**LN**	**PNum**	**PName**
JND	1	John	Doe	1	P1
JND	1	John	Doe	2	P2
HRS	2	Harry	Smith	1	P1
HRS	2	Harry	Smith	2	P2

$\Downarrow$ second select and project

$$\text{T5} \leftarrow \pi_{\substack{\text{ID}\\\text{FN}\\\text{LN}\\\text{PName}}} \left(\sigma_{\text{Proj=PNum}}(T4)\right)$$

ID	**Proj**	**PName**
JND	1	P1
HRS	2	P2

3.2.3 BINARY RELATIONAL OPERATIONS

JOIN : it is used to combine related tuples from two relations into single tuples. The JOIN operation can be thought as a CARTESIAN PRODUCT followed by a SELECT; it is used very frequently when specifying database queries. The general form of a JOIN on two relations $R(A_1, \ldots, A_n)$ and $S(B_1, \ldots, B_m)$ is the following:

$$R \bowtie_{\text{<CONDITION>}} S$$

and the condition has the following form:

$$\text{condition 1} \wedge, \ldots, \wedge \text{condition k}$$

The result is a relation Q of degree $n + m$ which has one tuple for each combination of tuples, whenever the combination satisfies the JOIN condition (hence the difference with the CARTESIAN PRODUCT). In JOIN, only combinations of tuples satisfying the join condition appear in the result, in the CARTESIAN PRODUCT all combinations of tuples are included in the result.

Given the relations EMPLOYEE and PROJECT from the previous example, the result of:

$$\text{EMPLOYEE} \bowtie_{\text{Emp\_ID=ID}} \text{WORK\_ON}$$
$$\equiv$$
$$\sigma_{\text{Emp\_ID=ID}}(\text{EMPLOYEE} \times \text{WORK\_ON})$$

is equivalent to the table T2 from the example above.

Since this operation may, in general, use different comparison operators between attributes $A_i \theta B_i$, it is also known with the name of THETA JOIN. For further references see [].

Nevertheless, the most common use of JOIN involves conditions with equality comparisons only. Such operation is called an EQUI JOIN.

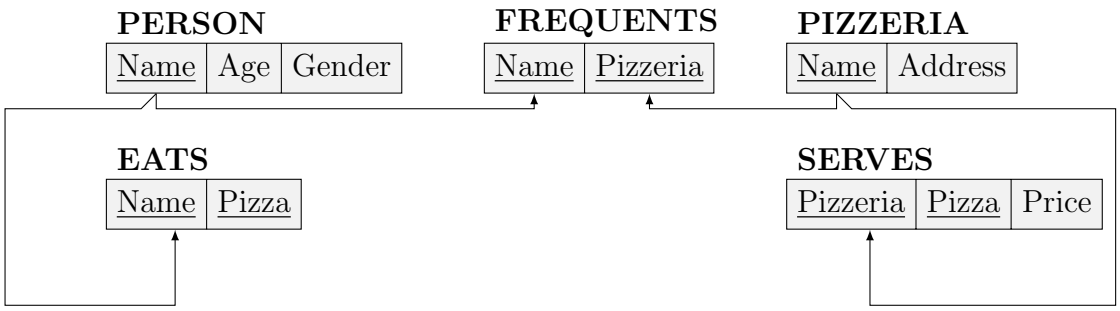
Both THETA and EQUI JOIN do not remove redundant columns (i.e., and attribute can be repeated); if we want to remove the duplicate attributes, then we should use a projection

π after the application of a JOIN; this operation is called NATURAL JOIN, it is indicated with the symbol $*$ and it equates all attribute pairs that have the same name.

Since other operations, such as DIVISION or SEMI JOIN, are out of the scope of these lectures, we will address to [] for further instructions.

3.2.4 Exercises:

1. Given the following relational schema:



   Write the following queries as relational algebra expressions:

   a) Find all pizzerias frequented by at least one person under the age of 18.

   b) Find the names of all females who eat either mushroom or pepperoni pizza (or both).

   c) Find the names of all females who eat both mushroom and pepperoni pizza.

   d) Find all pizzerias that serve at least one pizza that a girl eats for less than 10 Euro.

   e) Find all pizzerias that are frequented by only girls or only boys.

   f) For each person, find all pizzas the person eats that are not served by any pizzeria the person frequents. Return all such person (name) / pizza pairs.

   g) Find the names of all people who frequent only pizzerias serving at least one pizza they eat.

   h) Find the names of all people who frequent every pizzeria serving at least one pizza they eat.

   i) Find the pizzeria serving the cheapest pepperoni pizza. In the case of ties, return all of the cheapest-pepperoni pizzerias.

2. Given the following relational schema:

   - Author(author_id,FN,LN)
   - Author_Pub(author_id,ISBN)
   - Book(ISBN,book_title,month,year,editor)

   Answer the following questions, by using relational algebra expressions:

a) Find the names of all authors who are not book editors

b) Find the names of all authors who are book editors

c) Which authors authored a book that was published in July?

4 Physical design: DBMS

The goal of the last phase of database design, physical design, is to actually implement the database. At this phase one must know which database management system (DBMS) is used. During the course we will use MySQL workbench. In this phase we will create SQL clauses and idexes, and we will define the integrity constraints (rules) and the users' access rights. Finally, we will learns how to insert the data to test the database. You will see all these steps in laboratory. In reference [3] you can find a quick recap of the SQL queries that you will use.

References

[1] Elmasri R. and Navathe S. *Fundamentals of Database Systems* (2010), 6th edition, Addison-Wesley Publishing Company.

[2] Atzeni P., Ceri S., Paraboschi S., Torlone R. *Database systems: concepts, languages & architectures* (1999), McGraw Hill.

[3] Welling L., Thomson L., *MySQL Tutorial* (2004), Pearson Education Inc.