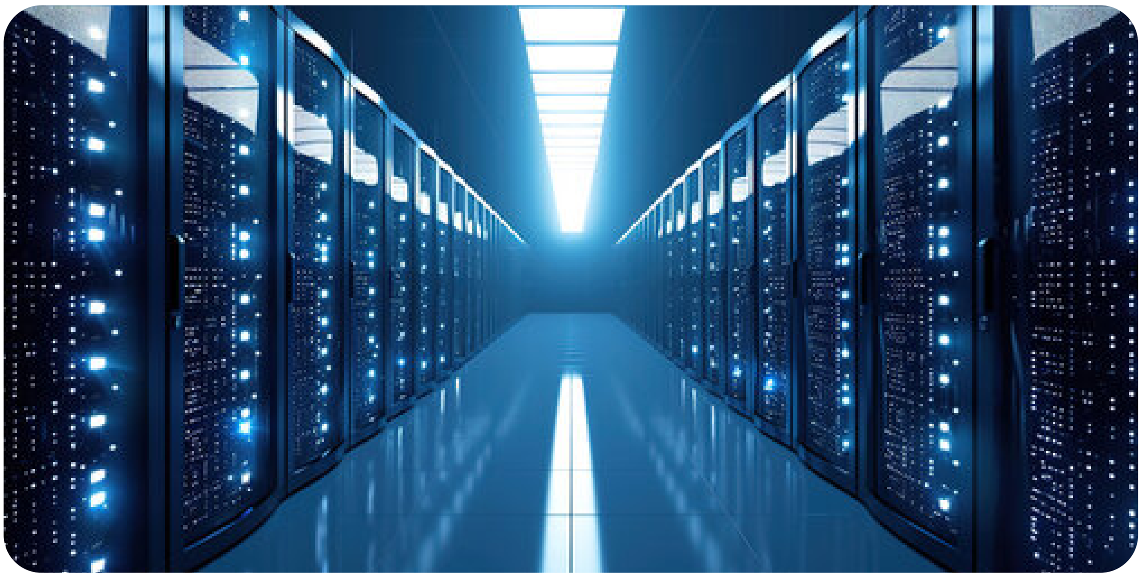
Sapienza, University of Rome
Course of Applied Computer Science and Artificial Intelligence (ACSAI)
2rd Year, 2nd Semester

Data Management and Analysis
Unit 1



Notes by Leonardo Biason
Course taught by prof. Giuseppe Perelli

About these notes

Those notes were made during my three years of university at Sapienza, and **do not** replace any professor, they can be an help though when having to remember some particular details. If you are considering of using *only* these notes to study, then **don't do it**. Buy a book, borrow one from a library, whatever you prefer: these notes won't be enough.

License

Bibliography & References

[1] J. D. Ullman (1990). *Principles of Database & Knowledge-Base Systems: Classical Database Systems Vol. 1*. Computer Science Press

The "*Data Management and Analysis Unit 1*" course was taught in the winter semester in 2024 by prof. Giuseppe Perelli (`perelli@di.uniroma1.it`)

I hope that this introductory chapter was helpful. Please reach out to me if you ever feel like. You can find my contacts on my website. Good luck[3]! Leonardo Biason
→ `leonardo@biason.org`

Contents

Chapter 1
Databases and management

In modern computing, **information** can be recorded in two ways:

- as **structured data**, where the objects are represented by strings and numbers;
- as **unstructured data**, like in plain text.

When storing data, it's better to follow a structure. Every organization has its own way to manage information, from the storing to the processing and retrieval of data. We call the systems that manage data **Information Systems**.

DEFINITION

Information System

An **Information System** is a **set of data** physically organized in secondary memory and managed in such a way as to allow its creation, updating and interrogation

Back in the days, each organization would have its own file system, its own application written in different languages with different core ideas and different data management systems. The disadvantages are mostly the following three:

- **redundancy**: while using the same data at the same time, two applications would create two copies of the same data;
- **inconsistency**: the update of an item would regard that only item, and not any other linked item;
- **data dependency**: its own organization would be on its own, since data would be stored in proprietary ways.

A solution was made with the creation of **databases**.

DEFINITION

Database (DB)

A **database** is a set of mutually linked files. Data is organized in a data structure that facilitates the creation, the processing, the updating and the access of the resources. In order to handle a database, a **D**ata**b**ase **M**anagement **S**ystem (**DBMS**) is used

Databases allow users to see in an abstract way the data that the users made. The goal of databases is to facilitate the processing of data, in relation to its properties. Generally, a database is an integrated resource shared by multiple components. It's important to establish integration and sharing between all users, but it's important to avoid concurrency (the simultaneous access to the same data)

When talking about databases a lot of terms come to our mind: one of them is **queries**. Queries can be done in multiple ways and with multiple languages (for instance SQL), but they all have a common basis: **relational algebra**. Queries are used to retrieve data from a database. All the properties shared between data are represented in a record structure.

SECTION 1.1 Data models

Data is conceptually organized in **aggregates of homogeneous information** (so the **files**) that constitute the **components of the information system**, and each **update operation** is targeted to a **single aggregate** (to avoid redundancy), while a **query** may involve **one or more aggregates** (since it's a read-only process). Queries usually take advantages of **indexes**: they are files that allow to quickly retrieve information from a large quantity of files.

There are various models that can be the basis of a database:

- **logical models**: they are independent of the physical structures, but they are available in DBMS (some examples are networks, hierarchical models, relational models or object-oriented models);
- **conceptual models**: they are independent of the modalities of realization; their scope is to represent entities of the real world and their relations.

1.1.1 Mesh model

A common model is the **mesh model**, where data is usually represented as a collection of **records** of homogeneous type. Any **relationship** between data is represented as a **link** (more practically, links are made through pointers).

A model is represented via a graph structure where:

- the **nodes** are the **records**;
- the **relationships** are the **links**;

The most popular mesh system is `CODASYL`.

1.1.2 Hierarchical model

Another common model is the **hierarchical model**. Such model is a restricted type of mesh model, where each node has only one parent, and where a mesh is composed by a collection of trees (thus a forest). This last point guarantees hierarchy.

1.1.3 Relational model

In a relational model, each data and relationship is represented as values: there are no explicit references, so we will never see explicit pointers between data. Usually, relational models are used to have a high-level representation. In such models, objects are used to represent records, while all the field and attributes of such objects refer to any information of interest.

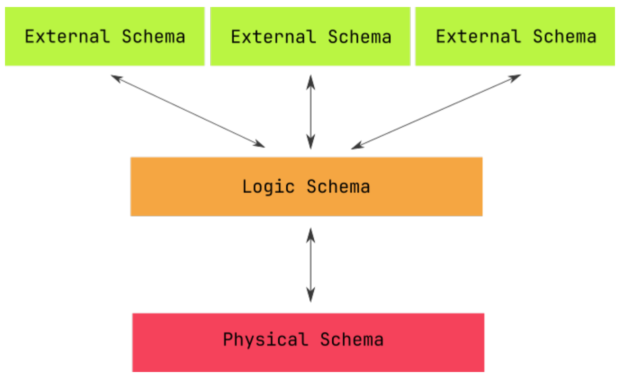
1.1.4 Object model

Such model is based on object and classes. Each attribute describes the state of an object, while methods describe the behaviour of an object. Each object encapsulates some attributes and behaviours. There is no universally recognised model yet.

SECTION 1.2 The three abstraction levels of a database

Each database has three separate levels:

- an **external schema**: it's the description of a portion of the database that allows different "views" of the database itself. The external schema depends on the access needs of each single user. The access to the database is granted only through the external schema, which may or may not coincide with the logical schema;
- a **logic schema**: it's the description of the entire database in the main logical models of the DBMS (for instance, the structure of the table);
- a **physical schema**: it's the representation of the logical schema by means of physical data structures.



In a database there must be **physical** and **logical** independence:

- **physical independence**: logical and external levels must be independent from the physical level. There must be the possibility to change the hardware without the need of changing also the software and the other two levels;
- **logical independence**: the external level is independent of the logical level, because the view of a user shouldn't affect the way data is organized. If one external level changes, it shouldn't change the logical organization.

Every database has its own **schema** (which must be **invariant** in time), which describes the database structure, and its own **instance**, which is an actual implementation of the schema. For instance, let the following table:

Name	Surname	Birth date
Marco	Togni	23/7/2003
Alessandro	Borghese	4/8/1998
Joe	Biden	6/11/1961

We consider the first row as the scheme of our database, while the other rows with the data are the instances.

While designing a database, it's important to keep **integrity**: there are, while designing a database, some constraints that have to be respected. For instance, a database used for university grades can't have negative values for the grades column, or in general it can't have any value outside the range $[18,31]$.

Section 1.3 Database languages

There are two types of languages that are used to interact with databases:

- **D**ata **D**efinition **L**anguage (**DDL**), which is used for the definition of schemes (be it a logical, an external or a physical scheme) and other general operations;
- **D**ata **M**anipulation **L**anguages (**DML**), which are used for querying and updating one or multiple instances of a database.

An example of language used with databases is the **S**tructured **Q**uery **L**anguage (**SQL**), which relies on the relational models. SQL integrates both DDL and DML in one unique language.

Section 1.4 Relational model in depth

A relational model is based on the mathematical definition of relation. Each relation can be easily translated into tables. The relationships/associations between data are expressed as values. All values exist within a **domain**, which is a possibly infinite set of values (for instance the st of all integer numbers is a domain, but also $\{0,1\}$ is a domain). Now, let $D_1, D_2, \ldots, D_k$ be not necessarily distinct domains, then the Cartesian product, denoted by

$$D_1 \times D_2 \times \ldots \times D_k$$

is the set

$$\{(v_1, v_2, \ldots, v_k) \mid v_1 \in D_1, v_2 \in D_2, \ldots, v_k \in D_k\}$$

A mathematical **relation** is any subset of the Cartesian product of one or more domains. A relation is said "of degree k" if such relation is a subset of a Cartesian product of k domains. Each element of a relation is called **tuple** (or n-uples), and the **cardinality** of the relation is the number of tuples. Each tuple is distinct, and with the notation $t[i]$ we can select one component of the tuple among all the components. Indexes in this case start from 1.

For instance, suppose that we have $k = 2$ domains. We could have for instance the following two domains:

$$D_1 = \{\text{white, black}\} \qquad D_2 = \{0,1,2\}$$

The Cartesian product of the two domain would be the following:

$$D_1 \times D_2 = \{(\text{white}, 0), (\text{white}, 1), (\text{white}, 2), (\text{black}, 0), (\text{black}, 1), (\text{black}, 2)\}$$

A possible relation could be for instance the following:

$$r = \{(\text{white}, 0), (\text{black}, 1), (\text{black}, 2)\}$$

Such relation has degree 2 and cardinality 3.

It's really east to translate a relation into a table: for instance, let's translate the previous relation in a table:

$$r = \{(\text{white}, 0), (\text{black}, 1), (\text{black}, 2)\}$$

white	0
black	1
black	2

We can interpret the data in the table by assigning names to the column and the table. The names are called **attributes**, which are defined by an **attribute name** A and the related **domain** (denoted as dom(A)). An attribute simply assigns a readable name to a column, making it easier to understand what a column represents. The set of attributes creates a **relation schema**, which is usually indicated as follows:

$$R(A_1, A_2, ..., A_k)$$

A set of relation schemas with different names is instead called **database schema**. From here, two other definitions can be made: first, the set of schemas $R_1, R_2, ..., R_k$ is called **relational database schema**; second, the set of relation instances $\{r_1, r_2, ..., r_k\}$ based on a schema R_n is called **relational database**. Now, given these definitions, we can rearrange the previous table, making it become something like this:

Relations:

Colors	Numbers
white	0
black	1
black	2

Mind that, once an order has been set, then it's not possible (or at least, not automatically) to change the order of the rows. If we change it, it doesn't really change regarding the content and the data, but mathematically, the order of the domains changed, thus it's a different table.

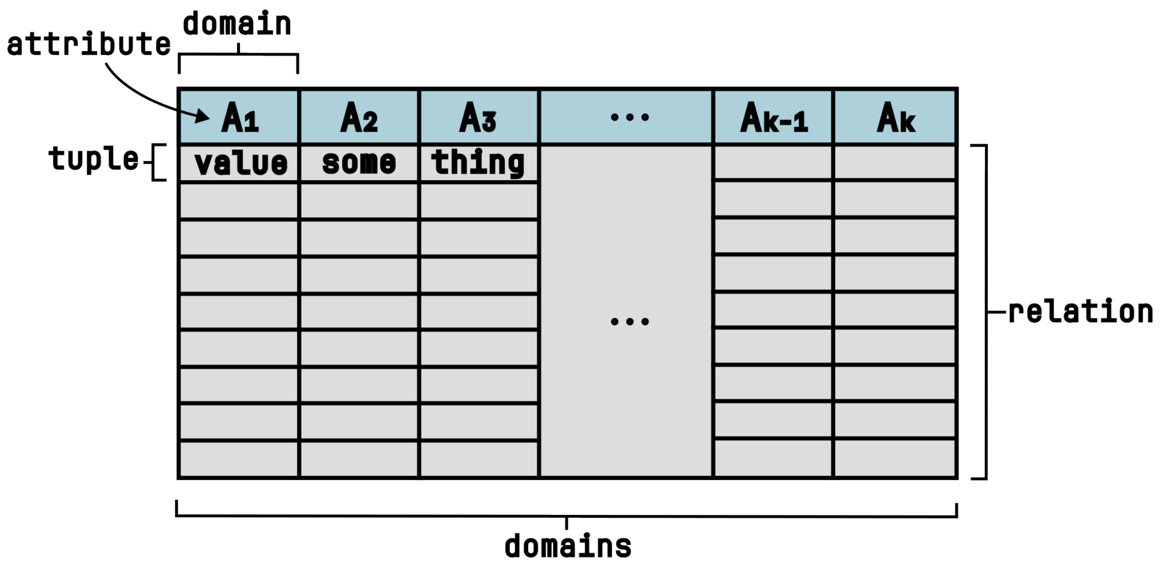
DEFINITION

Ennuple on R

Let R be a set of **attributes**: we call an **ennuple**, or tuple, **on** R (denoted as t) a **function** that associates for each attribute $A \in R$ an element of dom(A). The function $t(A)$ takes a value belonging to dom(A)

In a relational model, the schema of a relation does not variate over time, since it describes its structure. Such aspect is called **intentional**.

We call **instance** of a relation with a certain schema $R(X)$ a set r of tuples on a set X of attributes that contain some current values. More in general, it's an instance and practical application of the schema: it's a row of the database with its relative values. A **relation instance** is just the practical set of values that make a relation. Here follows an image with all the elements presented so far:



In addition to all the elements presented before, we call a **database schema** a set of relation schemas which all have different names. Applied to the context of relational databases, a **relational database schema** is a set of relation schemas R_1, R_2, ..., R_n, while a **relational database** is the instance of a relational database schema, where r_1, r_2, ..., r_n represent the relation instances or the corresponding relation schemas R_1, R_2, ...,R_n.

City	Region	Population
Rome	Lazio, Italy	3.000.000
Milan	Lombardia, Italy	2.000.000
Brussel	Belgium	1.220.000
Antwerpen	Belgium	506.000

An example of a relational database

If we want to define a precise item of a tuple, we can do it similarly on how dictionaries work in Python: given a tuple t, with $t[A_i]$ (where A_i is the attribute relative to the element that we are defining), we denote the element. So in the previous case, if t is the tuple of {"Brussel", "Belgium", 1.220.000}, then with t["Region"] we are pointing to "Belgium". This operation has a precise name, and it's called **restriction**.

DEFINITION

Restriction

Let Y be a subset of attributes of a schema X, such that $Y \subseteq X$, then $t[Y]$ is the subset of values of the tuple t that correspond to the attributes in Y. This operation is called **restriction**

In a relational model, whenever there is a reference between two different relations, then such references are represented by means of domain values that appear in the ennuples. So for instance, if we have the following three relational databases:

Students:

ID	Surname	Name
0001	Rossi	Mario
0002	Monari	Lucia
0003	Abili	Luca

Exams:

ID	Grade	Course
0001	29	CHEM1
0002	26	CALC1
0003	30	PROG1

then the references between the two tables are given by dom(ID).

1.4.1 The NULL value

When a value is missing, we can't leave it blank, nor we can use 0 as filler. We need a specific value that is treated as a missing value. Such value is the `NULL` value. Called the "*billion dollar mistake*" by Tony Hoare, its creator, the `NULL` value represents the **lack of information** or the **impossibility** to **apply a certain value to a certain domain**. In a database though, not all values should be able to be `NULL` (for instance the element ID). The `NULL` value doesn't belong to any specific domain, but it can replace any value in any domain (we call it **polymorphic value**).

It's always important to use it whenever some data is missing or can't be made: unused values might be used later, and may have a meaning in a second time. The `NULL` value instead has a fixed meaning. Even in the same domain though, two `NULL` values have a different meaning.

1.4.2 Constraints and keys

When making a database, we must ensure some **integrity constraints**: they are properties that must be ensured by every instance of the database; they describe specific properties related to the scope of the constraint itself, thus to the information contained in the database. We say that a database instance is **correct** if it satisfies **all the integrity constraints** related to its schemas.

If a constraint regards some elements in the same relation (thus either a single element of a tuple or some elements of the same tuple or relation), then such constraint is called **intra-relational constraint**, while if the constraint regards elements of two or more separate relations, then we call such constraint an **inter-relational constraint**.

When dealing with relations, we may want to have a quick way that allows us to refer to each tuple. For instance, in a database with all the students in an university, there must be a way that allows us to differentiate between each tuple. But not any value will do, only some attributes (so either one attribute or a set of attributes) are useful and can be used as relational keys. First, let's introduce the concept of key:

Keys · Part 1

DEFINITION

A **key** is a set X of attributes of a relation R that satisfies both the following conditions:

1) For each instance of R, there do not exist any two distinct tuples t_1 and t_2 that have the same values for all the attributes in X such that $t_1[X] = t_2[X]$;

2) There is no subset of X that doesn't respect the first condition.

Mind that a relation could have several keys, it's not important that it has more than one key: it's important though that it has **at least** one key. Why can't there be no keys? Because we can't have similar values, so each tuple should be distinct. When in a relation there are multiple keys, then either the key that is most used or the key that consists of the less possible attributes becomes the **primary key**. The keys are also useful while trying to refer to data from multiple relations.

Now that we explained the concept of key, we can introduce another constraint: the **referential integrity constraint**: following this constraint, portions of information from different relations are associated to the same key value, which means that the values in a first relation refer to the values of one attribute or set of attributes that should appear in the second relation. More formally, this constraint is written as follows:

DEFINITION

Referential Integrity Constraint

A **referential integrity constraint** between the attribute or set of attributes X or a relation R_1 and another relation R_2 is a property which forces the values of X in R_1 to appear as **primary keys** in the other relation R_2

For instance, let the following example:

Students:

ID	Surname	Name
0001	Rossi	Mario
0002	Monari	Lucia
0003	Abili	Luca

Exams:

ID	Grade	Course
0001	29	CHEM1
0002	26	CALC1
0003	30	PROG1

Courses:

Course	Course Name	Professor
CHEM1	Chemistry 1	Scotti, Gambarotti
CALC1	Calculus 1	Barelli
PROG1	Programming 1	Stegnardini

in this example, the **Course** key is the primary key in the Courses relation, and it's just a normal value in the Exams relation; also, the **ID** key is the primary key in the Students relation and in the Exams relation. Mind that the `NULL` value **does not violate** the referential integrity constraint.

Generically, relations between elements of the same tuples are considered a subset of the intra-relational constraints. Such relations are also called **functional dependencies** defined on the same schema.

DEFINITION

Functional dependencies · Part 1

A establishes a **semantic link** between two non-empty sets of attributes X and Y belonging to the same schema R: such link is represented via an **ordered pair** of sets. Written as

$$X \to Y$$

and read as "X determines Y". A relation with schema R satisfies the functional dependency if:

1) the functional dependency is applicable to R, in the sense that X and Y are subsets of R;

2) ennuples in R that are identical on X have to be identical on Y as well

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$$

An example is that if we had a database with all the flights of an airport, then given a day, a pilot and the flight time, we could get the flight code.

Chapter 2 Relational Algebra

We previously talked about databases and some possible types of databases. But how do we select and pick data from them? How can we structure efficient queries? We do it via the use of **relational algebra**. Relational algebra is a notation for specifying queries about the content of the relations, and eases the task of thinking about the way a query has to be structured. The `SQL` language is a query language based on the principles of relational algebra.

Definition — Relational algebra

Relational algebra is a **formal language** that allows to interrogate a **relational database** via the use of binary and unary operators that, if applied to one or two relation instances, returns a newly generated relation instance. Relational algebra is a **procedural language**, meaning that all the operations are executed in a precise order, which allow to get to the precise wanted result.

More precisely, what is an **algebra**? We can consider an algebra as a structure with a domain and a list of operators: for instance, arithmetic algebra is an algebra where the domain is a number set and the operators are the sum, the product, etc...; the set algebra has instead as a domain the sets, while the operators are the union, the intersection, the difference, etc... We say that an **algebraic expression** is a combination of values and operators.

Section 2.1 Possible operations

Relational algebra has the following properties: the **domain** is made out of the **relations**, which are set of tuples, and the **result** is as well made of a **set of tuples**. The expressions in relational algebra are what we call queries. There are four types of operators in relational algebra:

1) remove parts of a single relation: **projection** and **selection**;
2) usual set operations: **union**, **intersection**, **difference**;
3) combine the tuples of two relations: **Cartesian product** and **joins**;
4) **renaming**

2.1.1 Projection

A projection is similar to a "vertical cut" on the relation, **returning** a **subset** of the **relation's attributes**. It's represented with the symbol π. More specifically, having a relation R with attributes A_1, A_2, ..., A_n, the projection $\pi_{A_j,\ldots,A_k}(R)$ returns the attributes A_j, ..., A_k, where $1 \le j \le k \le n$. Let the following example:

R:

A_1	A_2	A_3	A_4	A_5	A_6
...	...	A	...	B	...
...	...	A	D	B	...
...	E	A	...	C	...

then $\pi_{A_3,A_5}(R)$:

A_3	A_5
A	B
A	C

In the previous example we can see how in the projection there are only 2 tuples: this is because row 1 and row 2 of the first table are, after the projection, identical, thus they get merged together. A projection follows the set rules, there **can't be two identical items**. If we want to save also the duplicates, we should include in the selection a domain that acts as primary key, which identifies each tuple, even if equal for most of the attributes. For instance, if our primary key was A_1 in the previous example, we would now get the following:

R:

A_1	A_2	A_3	A_4	A_5	A_6
1	...	A	...	B	...
2	...	A	D	B	...
3	E	A	...	C	...

then $\pi_{A_1,A_3,A_5}(R)$:

A_1	A_3	A_5
1	A	B
2	A	B
3	A	C

2.1.2 Selection

Similar to the projection, but instead of performing a "vertical cut", it performs an "horizontal cut" on the relation, and selects all the rows that respect a given constraint. Denoted with the symbol $\sigma_C(R)$, where C is the **condition** that has to be met. The **condition** is a **composite Boolean expression** (which uses the binary operators $\wedge$, $\vee$ and $\neg$. The operators are respectively the **AND**, the **OR** and the **NOT**), where the terms are in the following form:

$$A\theta B \quad \text{or} \quad A\theta\text{'}a\text{'} \qquad \text{where } \theta \in \{\le, <, =, >, \ge\}$$

In the previous formula:

- θ is called **comparison operator**, and can be one of the operators listed above;
- A and B are attributes of the same domain ($A = B$);
- a is an item of A, thus $a \in A$

For instance, let us consider the following example:

R:

A_1	A_2	A_3	A_4
1	Mark	Host	Milan
2	Joe	Host	Rome
3	Will	Client	Milan

$\theta_{A_3=\text{"Host"}}(R)$:

A_1	A_2	A_3	A_4
1	Mark	Host	Milan
2	Joe	Host	Rome

When using the selection, we don't lose any tuple, since we take all the content of the tuples, without discarding any value.

2.1.3 Union

A union between two relations is represented as

$$r_1 \cup r_2$$

and results in a relation that contains all the tuples belonging to at least one of the operand instances. Union can't be performed on every relation though: it can be performed only on what we call **union compatible** operands. Union compatible operands have the **same number of attributes**, and each corresponding attribute has the **same domain**. It's not necessary that the attributes names are the same, but their domain must be functionally identical. For instance, let the following two tables:

R_1:

Name	Surname	Code	Course
Mark	Hike	001	Chemistry
Joe	Hill	002	Calculus
Will	Smith	004	Programming

R_2:

Name	Surname	Code	Course
Mark	Hike	001	Algorithms
Edith	Jones	003	Linear Algebra
Joe	Hill	002	Physics

The union of R_1 and R_2 is equal to:

$R_1 \cup R_2$:

Name	Surname	Code	Course
Mark	Hike	001	Chemistry
Joe	Hill	002	Calculus
Will	Smith	004	Programming
Mark	Hike	001	Algorithms
Edith	Jones	003	Linear Algebra
Joe	Hill	002	Physics

If in two relations two tuples are equal (so $t_1 = t_2$ if $t_1 \in R_1$ and $t_2 \in R_2$), then by making the union of R_1 and R_2 we are **merging** also all the **double values**. The result of a union is still a relation, which means that it will still follow the rule that two equal tuples can't exist in the same relation. If we want to make the union between two non-compatible relations, we can first use the projection to make them compatible, and then make the union.

Moreover, the union **must make sense**. If we had two relations like R_1 = {name, code, department} and R_2 = {name, code, office}, then they would theoretically be union compatible, even though the union wouldn't make sense, since "department" and "office" have different meanings.

2.1.4 Difference

Since the difference is also a **set operation**, it follows some of the rules of the **union**: for instance, it can be applied only to **union compatible** operands. The result of the difference is a relation containing the tuples of the first relation that do not appear in the second relation. It's denoted as $r_1 - r_2$. For example:

R_1:

Name	Surname	Code	Course
Mark	Hike	001	Chemistry
Joe	Hill	002	Calculus
Will	Smith	004	Programming

R_2:

Name	Surname	Code	Course
Mark	Hike	001	Chemistry
Edith	Jones	003	Linear Algebra
Joe	Hill	002	Physics

The difference $R_1 - R_2$ would be then equal to:

$R_1 - R_2$:

Name	Surname	Code	Course
Joe	Hill	002	Calculus
Will	Smith	004	Programming

While instead $R_2 - R_1$ would be equal to:

$R_2 - R_1$:

Name	Surname	Code	Course
Edith	Jones	003	Linear Algebra
Joe	Hill	002	Physics

Similarly to the union, if two relations are not union compatible, then we can use the projection to make them alike and then use the difference between them

2.1.5 Intersection

The **intersection** is the last of the three set operations that we can apply to a relation. As for the other two operations, it can be applied only to **union compatible** operands. The intersection creates a relation containing the items that are common to both the relations, and it's denoted as $r_1 \cap r_2$. The intersection is given by this expression here:

$$r_1 \cap r_2 = (r_1 - (r_1 - r_2))$$

An example follows:

R_1:

Name	Surname	Code	Course
Mark	Hike	001	Chemistry
Joe	Hill	002	Calculus
Will	Smith	004	Programming

R_2:

Name	Surname	Code	Course
Mark	Hike	001	Chemistry
Edith	Jones	003	Linear Algebra
Joe	Hill	002	Physics

The intersection $R_1 \cap R_2$ would be equal to:

$R_1 \cap R_2$:

Name	Surname	Code	Course
Mark	Hike	001	Chemistry

Unlike the difference, the intersection is a commutative operand. And similarly to the union and the difference, in order to make the intersection of two non-compatible relations, we can use first the projection to select the domains that we're interested into and then use the intersection.

2.1.6 Cartesian product

Sometimes a database doesn't contain one single relation, but it has multiple relations that contain different data, even if each relation is linked one to the other. Normally, the main information are stored in multiple relations, so in order to retrieve such information we have to combine the multiple tuples from the multiple relations before doing any operation on them.

One possible way of uniting two or more relations is with the **Cartesian product**. The Cartesian product creates a relation with **tuples** from the **first relation combined** with **tuples** from the **second relation**. A Cartesian product is denoted as $r_1 \times r_2$. Let us make an example:

R_1:

Name	CCode	Town
Mark	C01	Rome
Joe	C02	Milan
Will	C03	Florence
Jack	C04	Rome

R_2:

Order	CCode	AssetCode	Pieces
O01	C01	A01	200
O02	C02	A02	150
O03	C03	A01	300
O04	C02	A03	150
O01	C01	A02	200

We want to make the Cartesian product between R_1 and R_2. Before doing that, we have to rename the "CCode" attribute of R_2, since otherwise we would lose that data. The renaming can be done via the operator ρ. So, before doing the product, we do the renaming:

$$R_{2\text{New}} = \rho_{\text{CopyCCode} = \text{CCode}}(R_2)$$

Now we can proceed with the product. This will result in the following:

Customer × Order:

Name	CCode	Town	Order	CopyCCode	AssetCode	Pieces
Mark	C01	Rome	O01	C01	A01	200
Mark	C01	Rome	O02	C02	A02	150
Mark	C01	Rome	O03	C03	A01	300
Mark	C01	Rome	O04	C02	A03	150
Mark	C01	Rome	O01	C01	A02	200
Joe	C02	Milan	O01	C01	A01	200
Joe	C02	Milan	O02	C02	A02	150
Joe	C02	Milan	O03	C03	A01	300
...	...	...	...	...	...	...
Will	C03	Florence	O03	C03	A01	300
...	...	...	...	...	...	...
Jack	C04	Rome	O01	C01	A02	200

All the tuples with the same Name, Code and Town are all marked with the same colour. There is only one problem though: some orders were never made by some customers. For instance, user Mark never did an order of 150 pieces of the asset A02, but it appears on the product. Here is the table with the right orders appearing in green and the wrong ones in red:

Customer × Order:

Name	CCode	Town	Order	CopyCCode	AssetCode	Pieces
Mark	C01	Rome	O01	C01	A01	200
Mark	C01	Rome	O02	C02	A02	150
Mark	C01	Rome	O03	C03	A01	300
Mark	C01	Rome	O04	C02	A03	150
Mark	C01	Rome	O01	C01	A02	200
Joe	C02	Milan	O01	C01	A01	200
Joe	C02	Milan	O02	C02	A02	150
Joe	C02	Milan	O03	C03	A01	300
...	...	...	...	...	...	...
Will	C03	Florence	O03	C03	A01	300
...	...	...	...	...	...	...
Jack	C04	Rome	O01	C01	A02	200

This can be solved by checking the Customers Codes: we have both CCode and CopyC-Code, so if we select all the tuples where these two are equal, we get the right product with the right data:

$\sigma_{\text{CCode}=\text{CopyCCode}}(\text{Customer} \times \text{Order})$:

Name	CCode	Town	Order	CopyCCode	AssetCode	Pieces
Mark	C01	Rome	O01	C01	A01	200
Mark	C01	Rome	O01	C01	A02	200
Joe	C02	Milan	O02	C02	A02	150
Joe	C02	Milan	O04	C02	A03	150
Will	C03	Florence	O03	C03	A01	300

A more complete and elegant solution would consist of making an ulterior projection to eliminate the CopyCCode attribute:

$$\pi_{\text{Name, CCode, Town, Order, AssetCode, Pieces}}\left(\sigma_{\text{CCode}=\text{CopyCCode}}(\text{Customer} \times \text{Order})\right)$$

Let's try to structure a query based on the table that we just got: we want the data of the customers and of the orders that are more than 150 pieces. We already have the table, so we don't have to worry about making the Cartesian product again. We just need to select the rows of the orders with more than 150 pieces:

$$\pi_{\text{Name, CCode, Town, Order, AssetCode, Pieces}}\left(\sigma_{\text{CCode}=\text{CopyCCode} \wedge \text{Pieces}>150}(\text{Customer} \times \text{Order})\right)$$

2.1.7 Natural join

The **natural join** simplifies the process of taking away the incorrect tuples, as its definition is that if there are two relations r_1 and r_2, then the natural join between those two (denoted as $r_1 \bowtie r_2$) is equal to a relation where the tuples respect the following condition:

$$R_1.A_1 = R_2.A_1 \wedge R_1.A_2 = R_2.A_2 \wedge \ldots \wedge R_1.A_k = R_2.A_k$$

where A_1, A_2, ..., A_k are the **attributes in common** between the two relations r_1 and r_2. All the **duplicate attributes** are, moreover, **automatically dropped**. In synthesis, the natural join is equal to the projection of the selection of the following Cartesian product:

$$r_1 \bowtie r_2 = \pi_{XY}\big(\sigma_C(r_1 \times r_2)\big)$$

where:

- the condition C is equal to the condition

$$R_1.A_1 = R_2.A_1 \wedge R_1.A_2 = R_2.A_2 \wedge \ldots \wedge R_1.A_k = R_2.A_k$$

- X is the set of attributes in r_1;
- Y is the set of attributes in r_2 that are not in r_1;

It's important to say that the attributes in the condition C have all the same name, and that only the tuples having the same values for the attributes in common are then merged and returned from the query.

There are two special cases when dealing with the **natural join**:

1) the two relations have some attributes in common, but the two attributes have no values in common: the result of such join is an **empty set**;
2) the two relations do not have attributes with the same name, so the condition

$$R_1.A_1 = R_2.A_1 \wedge R_1.A_2 = R_2.A_2 \wedge \ldots \wedge R_1.A_k = R_2.A_k$$

   cannot be evaluated, making the natural join result into a normal Cartesian product.

When doing the natural join, it's important to remember that attributes with the same name share also the same meaning. This is important because after the natural join the attributes that share the same name will be merged. It's important to check before doing the join what is the meaning of each attribute, and if necessary, we should rename it. Another solution is to use a θ-join.

2.1.8 θ-join

A θ-join selects the tuples resulting from the Cartesian product that satisfy the following condition:

$$A\,\theta\,B$$

where:

- θ is the comparison operator (such that $\theta \in \{\leq, <, =, >, \geq\}$);
- A is an attribute of the first relation;
- B is an attribute of the second relation;
- $A = B$.

The final formula of a θ-join is the following:

$$r_1 \bowtie r_2 = \sigma_{A\,\theta\,B}(r_1 \times r_2)$$

SECTION 2.2 Universal quantification

Up until now, we always made examples and statements considering that there **exist** some tuples for some properties that will satisfy our query or our operation: we always used the **existential quantification** $\exists$. The way an existential quantification works is by checking all the tuples and returning the ones that correspond to our query.

There exists another type of quantification: **universal quantification** ($\forall$). Such quantification checks if a condition holds true for all the elements on a set, which is different from the existential quantification: the existential checks each tuple independently, while the universal considers all tuples at the same time.

Let us make an example: suppose that we have two tables, one with a list of customers and one with a list of orders. We want to make a table that collects all the customers that ordered more than 100 pieces of any item. The tables would be similar to the following ones:

Customers:

Name	CCode	Town
John	C1	Rome
John	C2	Foggia
Smith	C3	Rome
Jack	C4	Milan

Orders:

CCode	ACode	Pieces
C1	A1	100
C2	A2	200
C3	A2	150
C4	A3	200
C1	A2	200
C1	A3	100

Now, let's perform the following query:

$$\pi_{\text{Name, CCode, Town}}(\sigma_{\text{Pieces} > 100}(\text{Customers} \bowtie \text{Orders}))$$

The result would be the following:

Query result:

Name	CCode	Town
John	C1	Rome
John	C2	Foggia
Smith	C3	Rome
Jack	C4	Milan

But what if we wanted to know the customers that **always** ordered more than 100 items? Let's start and do it step by step: first, we do the natural join between the two tables Customers $\bowtie$ Orders:

Customers $\bowtie$ Orders:

Name	CCode	Town	ACode	Pieces
John	C1	Rome	A1	100
John	C1	Rome	A2	200
John	C1	Rome	A3	100
John	C2	Foggia	A2	200
Smith	C3	Rome	A2	150
Jack	C4	Milan	A3	200

Now, how can we do the selection? We can use $\sigma_{\text{Pieces} > 100}$, but that would include the second tuple: such tuple can't be included, since the customer C1 hasn't always ordered more than 100 pieces of any item. There is a method that we can exploit, which involves the logical negations.

Let's suppose that we have the following statement:

$\forall$ elements the condition is **true**

what would be the negation of such statement? It **can't** be

$\forall$ elements the condition is **false**

because in order to make the first statement not true we don't need **all** the elements to not respect the condition, but rather we need just one element for which the statement doesn't hold. Thus, the correct negation is

$\exists$ an element for which the condition is **false**

So analogously, if we say "*All* of my orders is of at least 100 items", the negation is **not** "*None* of my orders is of at least 100 items", but rather "*None* of my orders is *not* of at least 100 items". We can exploit this concept of the double negation in order to make our query work: we can find the items that satisfied the inverted condition, and then we can use them in order to remove the related items in the "for all" condition. We thus proceed this way: first, we find all the customer which ordered a number of items which is less or equal to 100 from the natural join Customers $\bowtie$ Orders:

$\sigma_{\text{Pieces} \leq 100}(\text{Customers} \bowtie \text{Orders})$:

Name	CCode	Town	ACode	Pieces
John	C1	Rome	A1	100
John	C1	Rome	A3	100

$\pi_{\text{Name, CCode, Town}}\Big(\sigma_{\text{Pieces} \leq 100}(\text{Customers} \bowtie \text{Orders})\Big)$:

Name	CCode	Town
John	C1	Rome

Now we can subtract this last query from the natural join Customers $\bowtie$ Orders and get the customers that always ordered more than 100 items. Why don't we subtract it directly from the Customers relation? Because if there was a customer that made no orders, then we wouldn't be able to detect him. If we do the natural join between Customers and Orders, all the customers that made no order will disappear, thus having a table with all the customers that made at least one order, be it of more than 100 items or not. The final query will then be:

$$\pi_{\text{Name, CCode, Town}}(\text{Customers} \bowtie \text{Orders}) - \pi_{\text{Name, CCode, Town}}\Big(\sigma_{\text{Pieces} \leq 100}(\text{Customers} \bowtie \text{Orders})\Big)$$

Let us observe another example query: we want to know the name and the code of all the employees that have an higher salary than their supervisors. The beginning table is the following (where `SCode` stands for "Supervisor's Code"):

Employees:

Name	CCode	Section	Salary	SCode
Jack	C1	B	1000$	C3
Joe	C2	A	2000$	C3
Mark	C3	A	5000$	NULL
Buck	C4	B	2000$	C2
Jimmy	C5	B	1500$	C1
Jonathan	C6	B	1000$	C1

By itself the table can't give us any concrete data, so how can we proceed? We could for instance make a join of the table with itself: this way we would be able to have on a same tuple both the supervisor and the employee. Let's first compute the Cartesian product Employees × Employees:

$$\text{EmployeesProd} = \rho_{\text{Name, CCode, Section, Salary, SCode, CName, CCCode, CSection, CSalary, CSCode}}(\text{Employees} \times \text{Employees})$$

$$\sigma_{\text{SCode}=\text{CCode}}(\text{EmployeesProd}):$$

Name	CCode	Section	Salary	SCode	CName	CCCode	CSection	CSalary	CSCode
Jack	C1	B	1000$	C3	Mark	C3	A	5000$	NULL
Joe	C2	A	2000$	C3	Mark	C3	A	5000$	NULL
Buck	C4	B	2000$	C2	Joe	C2	A	2000$	C3
Jimmy	C5	B	1500$	C1	Jack	C1	B	1000$	C3
Jonathan	C6	B	1000$	C1	Jack	C1	B	1000$	C3

Now we can make a further selection and pick all the rows where Salary is bigger than CSalary, and finally make a projection and pick only the Name and CCode of each tuple:

$$\text{Result} = \sigma_{\text{Salary}\geq\text{CSalary}}\Big(\sigma_{\text{SCode}=\text{CCode}}(\text{EmployeesProd})\Big):$$

Name	CCode	Section	Salary	SCode	CName	CCCode	CSection	CSalary	CSCode
Buck	C4	B	2000$	C2	Joe	C2	A	2000$	C3
Jimmy	C5	B	1500$	C1	Jack	C1	B	1000$	C3
Jonathan	C6	B	1000$	C1	Jack	C1	B	1000$	C3

$\pi_{\text{Name, CCode}}(\text{Result})$:

Name	CCode
Buck	C4
Jimmy	C5
Jonathan	C6

If we wanted to get for instance the name and the code of all the employees that have a lower salary than their supervisors, then we have multiple ways of getting to such solution. Here we are talking as well about a ∀ condition, so we should be careful. We can use what we got previously: `Result` is equal to the tuples of the employees that have an higher salary than their supervisors, thus we can use it and subtract it from the original Employees relation:

$$\pi_{\text{SCode}}(\text{Employees}) - \pi_{\text{CCode}}(\text{Result})$$

Now we have the codes of the employees (which represent the supervisors) that have an higher salary than their employees: we can make a join with the Employees table and get the data of each employee listed in the operation done earlier. After it, we can make a projection and take only Name and CCode:

$$\pi_{\text{Name, CCode}}\Big((\pi_{\text{SCode}}(\text{Employees}) - \pi_{\text{CCode}}(\text{Result})) \bowtie \text{Employees}\Big)$$

SECTION 2.3 Designing a relational database

Now that we saw all the possible operations, we can start describing some techniques that we can use to build a database. Let us suppose that we want to build a database for a university: we would start with a database with a single schema `Curriculum` with the following attributes: `StudentID`, `TC` (Tax Code), `Surname`, `Name`, `BirthDate`, `City`, `Province`, `Credits`, `TitleCourse`, `Professor`, `Date`, `Grade`.

We have some problems though. For example, consider the following instance of the database:

ID	TC	Sname	Name	BD	City	Prov	C	Course	Prof	Date	Grade
01	...	Rossi	Mario	...	Roma	Roma	10	CHEM	De Marco	...	...
02	...	Bianchi	Paolo	...	Roma	Roma	10	CHEM	De Marco	...	...
01	...	Rossi	Mario	...	Roma	Roma	10	CALC	Checchi	...	...

We have a problem though: **redundancy**: the students appear again for every course. We don't need to rewrite the same data for all the times that a student does an exam. **Redundancy** has to be avoided, since it just occupies memory. From this problem, we can formulate 3 problems, or **anomalies**:

- **update anomaly**: if the information regarding one course has to be updated, then we have to update each time each row where the course appears;
- **insertion anomaly**: a student's data can't be inserted until the students attends and registers at least one exam. The same goes for the course: if no students are attending one course, then it will never be registered;
- **deletion anomaly**: if we want to delete a student's row, then we might delete also the courses' data.

Another solution is to split the schema into multiple schemas: we now want to make 3 schemas, made of the following attributes:

- **Students**: `StudentID`, `TC`, `Surname`, `Name`, `BirthDate`, `City`, `Province`;
- **Courses**: `CourseCode`, `Title`, `Professor`;
- **Exams**: `StudentID`, `CourseCode`, `Date`, `Grade`.

The database would look like this:

Students:

ID	TC	Surname	Name	BirthDate	City	Province
01	...	Rossi	Mario	...	Roma	Roma
02	...	Bianchi	Paolo	...	Roma	Roma

Course:

CCode	Title	Professor
C01	CHEM	De Marco
C02	CALC	Checchi

Exams:

ID	CCode	Date	Grade
01	C01	...	28
02	C01	...	27
01	C02	...	23

We still have some **redundancy** though: the City and Province could be repeated multiple times, and we can avoid it. Moreover, let us suppose that a municipality changes the province, then we would have to edit all the rows of all the students. Finally, we wouldn't be able to memorize the correspondence City - Province if there are no students with a specific correspondence, and deleting the only row with such a record would delete all the information regarding the match of a City with its Province. We can solve this by making a new relation:

Students:

ID	TC	Surname	Name	BirthDate	City
01	...	Rossi	Mario	...	Roma
02	...	Bianchi	Paolo	...	Roma

Course:

CCode	Title	Professor
C01	CHEM	De Marco
C02	CALC	Checchi

Exams:

ID	CCode	Date	Grade
01	C01	...	28
02	C01	...	27
01	C02	...	23

Municipality:

City	Province
Roma	Roma
Frascati	Roma
Veglie	Lecce

We define a good database schema as a schema that presents **no redundancies** and **no update**, **insertion** and **deletion anomalies**. Most of the times we might think that it's more immediate to store all the data in a same table, but we will surely encounter at a point the problems that we saw earlier. Via the use of relational algebra, we can easily recover all the data that we need. The way to design a good schema is then to represent each concept into a different table, and the way to identify such concepts is to create **keys**.

Keys

DEFINITION

A **key** is an **attribute** or a **set of attributes** which define a particular **functional dependence**. A key can be seen as a particular **constraint**.

When creating a schema, some data may follow some conditions: in an exams schema, we can't register grades outside the range [18, 31] (if we count 31 as 30 cum laude); the student ID must be different for each student; a salary can't be negative; etc...

Such conditions are called **constraints**:

Constraint

DEFINITION

A **constraint** is the representation in a database schema of a **condition** that is valid in the reality of interest. A database schema is said to be **legal** if it respects all the given constraints.

DBMSs allow for an automatic control of the constraints, so that all the domains, keys and the containment of domains are correct. We'll see how functional dependencies represent constraints between attributes or subsets of attributes of the schema itself, and such constraints have to be respected at all times in order for the schema to be said legal.

SECTION 2.4 Exercises

2.4.1

EXERCISE

We have the following three relations:

- `VEHICLE` {Plate, Km, Type, Model}
- `MODEL` {Name, Engine, Seats}
- `RENT` {Plate, TaxC, PickupD, DropoffD}

Perform the following queries:

1) Plate and number of seats of passenger vehicles that were rented (picked up) on January 25, 2016;
2) Plates of diesel vehicles that were not leased (picked up and returned) in January 2016. Find for each user the plate of the first electric vehicle they rented

1.1.1

1) The solution query is

$$\pi_{\text{Plate, Seats}}\left(\sigma_{\text{Type = "People"} \wedge \text{PickupD = 25/1/2016}}(A)\right)$$

$$A = \left((\text{VEHICLE} \bowtie_{\text{VEHICLE.Model = MODEL.Name}} \text{MODEL}) \bowtie \text{RENT}\right)$$

2) The solution query is

$$\pi_{\text{Plate}}\left(\sigma_A(B)\right)$$

$$A = \neg(\text{"31/12/2015"} \le \text{ PickupD } \le \text{"1/2/2016"} \vee \implies$$

$$\implies \text{"31/12/2015"} \le \text{ DropoffD } \le \text{"1/2/2016"}) \wedge \text{Engine = "Diesel"}$$

$$B = \left((\text{VEHICLE} \bowtie_{\text{VEHICLE.Model = MODEL.Name}} \text{MODEL}) \bowtie \text{RENT}\right)$$

3)

Chapter 3 Functional Dependencies

In the previous chapter we had a look over relational databases and relation schemas. Let us recall that a **relational schema** is a **set of attributes** $\{A_1, A_2, ..., A_n\}$, and is denoted as

$$R = A_1, A_2, ..., A_n$$

Usually, we use the first letters of the alphabet to denote single attributes, while with the last letters of the alphabet we denote **sets** of attributes. If for instance we had two sets of attributes X and Y and we wanted to take their **union**, we would then denote it as XY. Another concept that we want to recall is that a **tuple** is a **function** that associates to each attribute $A_i \in \mathbb{R}$ a value $t[A_i]$ in the corresponding domain dom(A_i). Let us consider the following relation as an example:

Name	Surname	StudID	Department
Mark	Doe	001	INF
John	Doe	002	INF

If we consider X = {Surname, Department} and the two tuples in the relation as t_1 and t_2, then there is a property that can be verified: we say that t_1 and t_2 **coincide on** X (so $t_1[X] = t_2[X]$) if $\forall A \in X : t_1[A] = t_2[A]$

In the past we also talked about **functional dependencies**: semantic links between two non-empty sets of attributes belonging to the same relation R. The complete definition can be seen here.

Definition

Functional dependencies · Part 2

A establishes a **semantic link** between two non-empty sets of attributes X and Y belonging to the same schema R: such link is represented via an **ordered pair** of sets. Written as

$$X \to Y$$

and read as X *functionally determines* Y; vice versa, it can be said that Y *is functionally determined by* X. X is called **determinant**, while Y is called **dependent**. A relation with schema R satisfies the functional dependency if:

1) the functional dependency is applicable to R, in the sense that X and Y are subsets of R;

2) ennuples in R that are identical on X have to be identical on Y as well

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$$

It's not necessary that two tuples t_1 and t_2 exist in order to satisfy the functional dependency, but if they do exist and if they are equal, then the implication must be true.

Note the second condition that must be respected by the functional dependency: we say that

$$\forall\, t_1, t_2 \in r : t_1[X] = t_2[X] \quad \Longrightarrow \quad t_1[Y] = t_2[Y]$$

which is completely different of saying instead

$$\forall\, t_1, t_2 \in r : t_1[X] \neq t_2[X] \quad \Longrightarrow \quad t_1[Y] \neq t_2[Y]$$

This last proportion is **not** the same as the first one, because if $t_1[X] \neq t_2[X]$, then we don't care about the values of $t_1[Y]$ and $t_2[Y]$: for what we care, they could be the same. Functional dependencies just express **constraints** on data, such as the matching of a student ID with one and only one student. We can't use for instance functional dependencies in the following case:

Grade → Honour (Is it possible?)   Exams:

StudID	Grade	Honour
001	30	Yes
002	27	No
003	27	No

Can we use the functional dependency Grade → Honour? If a student gets 30, then it could receive the honour, but this doesn't imply that all the students who get 30 should have the honour. Also because, the honor could be given also to a student who got 27, but if we have two tuples (such as t_2 and t_3) which both have the same grade, then it would imply that both got the honour, which could not be the case. We can't, following the inverse reasoning, say that Honour → Grade.

Some examples of functional dependencies are the following:

Is the functional dependency $AB \to C$ valid in both tables?

r_1:

A	B	C
A_1	B_1	C_2
A_2	B_3	C_1
A_1	B_1	C_2

r_2:

A	B	C
A_1	B_1	C_2
A_2	B_3	C_1
A_1	B_1	C_3

In the relation r_1 the functional dependency is valid, because t_1 and t_3 respect the logic $t_1[AB] = t_3[AB] \implies t_1[C] = t_3[C]$, while the second relation doesn't, since $t_1[C] \neq t_3[C]$. Given a relation instance R and a set of functional dependencies valid on it, we say that the relation instance is **legal** if it satisfies all the functional dependencies in F. Always referring to the previous example, if $F = \{A \to B\}$, then both relations satisfy the functional dependency in F: they wouldn't satisfy $AB \to C$, but since such functional dependency isn't in F, it won't matter whether r_1 and r_2 respect it. If our set of functional dependencies was $F = \{A \to B, B \to C\}$, then only r_1 would be a legal instance in respect to F.

Section 3.1 Closure of F (F^+) and keys

Let us consider again a set of functional dependencies such as $F = \{A \to B, B \to C\}$: each legal instance of r would satisfy these two functional dependencies, but would also satisfy the functional dependence $A \to C$. Can we directly include $A \to C$ into F? We actually can't, since F is delimited only to certain functional dependencies. There could be in a relation some other functional dependencies that are not belonging to F. We could consider F as a subset of all the possible functional dependencies on a relation $r \in R$.

In general, a relation r might have more functional dependencies than the ones defined in F; an example is the one that we said earlier: if $F = \{A \to B, B \to C\}$, then $A \to C$ is a possible functional dependency, because it's **deduced** (or inferred) from F.

DEFINITION

Inferred functional dependencies

A **functional dependency** $X \to Y$ is said to be **inferred from** or **implied by** a set of functional dependencies if $X \to Y$ holds for **every legal instance** r of a relation schema R where F is applied.

In other terms, if each relation r is legal according to F, then $X \to Y$ also holds in r.

Formally, we refer to the set of all the possible dependencies that can be inferred from F as the **closure** of F.

DEFINITION

Closure of F (F^+)

Formally, we denote the **closure of** F as the set of all the dependencies that can be inferred from F. F is a subset of the closure ($F \subseteq F^+$). It is indicated with F^+

We also talked in the past about the concept of **key**: a set of attributes that allows to identify each tuple. There is always at least one key in each relation, and there are multiple types of keys, depending on their meaning. Keys can also be defined in term of functional dependencies:

DEFINITION

Keys · Part 2

Given a relation schema R and a set F of functional dependencies defined on it, we can define K, a subset of the relation schema R, as a **key** of R if:

1) the functional dependence $(K \to R) \in F^+$ always holds;

2) there is no subset $K' \subseteq K$ such that the functional dependence $K' \to R$ holds.

Taking again the example of a university database with all the students, the matriculation number is the identifier for each student, and is thus the key of the relation. There are multiple types of keys, and in general a table might have multiple keys: in that case, when multiple keys exist, each key is called **candidate key**. In `SQL` in particular, a key must be selected as **primary key**: a key that can't assume a `null` value and that should determine all the items inside the relation.

There is a particular type of functional dependency, called **trivial functional dependency**: let us consider two non-empty sets X and Y of R, such that $Y \subseteq X$. Suppose now that we impose the functional dependency $X \to Y$, even if not part of a specific set of functional dependencies F. We would have for instance $X = ABC$ and $Y = AB$. As a consequence, the functional dependency $X \to Y$ belongs to the closure of F (thus $(X \to Y) \in F^+$).

There are some properties connected to the functional dependencies, and one of them is the following: considering a relation schema R and a set of functional dependencies F, we can say that given two non-empty sets X and Y belonging to R, if the functional dependency $X \to Y$ is **included** in F^+, then for each element S belonging to the set Y, also the functional dependency $X \to S$ belongs to F^+. It can be also written with mathematical terms as follows:

$$(X \to Y) \in F^+ \iff \forall\, S \in Y, (X \to S) \in F^+$$

So for instance, let us have the following situation:

$$X = A \qquad Y = BC$$

then, if we let $X \to Y$, for the property we would have $A \to B$, $A \to C \in F^+$.

SECTION 3.2 F^A and Armstrong's axioms

When trying to compute the closure of F, we might realize that it's an hard task: F^+ does comprehend all the functional dependencies in F, and we can derive some of the inferred ones from F, but there might be other ones that are not so obvious, and for large schemas it may be hard to compute all the functional dependencies. It's for this reason that in order to compute all the functional dependencies in F^+ we introduce a smaller set F^A, which is easier to compute, although still being time consuming. We consider F^A as a **set** of **functional dependencies on** R. By default F^A is an **empty set**, which can be then populated.

In order to compute all the functional dependencies in F^A, we'll use the **Armstrong's axioms**. Some of them are here listed:

1) if $(X \to Y) \in F$, then $(X \to Y) \in F^A$
2) if $(Y \subseteq X) \in R$, then $(X \to Y) \in F^A$ **(reflexivity)**
3) if $(X \to Y) \in F^A$, then $(XZ \to YZ) \in F^A$, $\forall\, Z \in R$ **(augmentation)**
4) if $\{X \to Y, Y \to Z\} \in R$, then $(X \to Z) \in F^A$ **(transitivity)**

Let's go through the various axioms:

- **reflexivity**: if for instance $X = \{A,B\}$ and $Y = \{A\}$, then it's obvious that if a tuple has $t_1[A] = t_2[A]$ and $t_1[B] = t_2[B]$ then the value for the attribute A in both tuples is equal;
- **augmentation**: let us have something like TaxCode → Name, then if two tuples have the same TaxCode value then they must obviously have the same Name value. Now, let us consider, together with TaxCode and Name, the attribute Address. We then get TaxCode, Address → Name, Address. It's also obvious that if two tuples have the same TaxCode and Address values, then they have the same Name and Address values;

- **transitivity**: let the two functional dependencies $A \to B$ and $B \to C$: if two tuples have the same value for the attribute A, then they must have the same value also for the attribute B; similarly, if two tuples have the same value for B, then also the value on C will be the same. If both functional dependencies are respected, then whenever two tuples have the same value for A, then they will have the same value for B.

Why did we introduce another set F^A though? Because we can recursively apply the Armstrong axioms in order to reach the point where $F^+ = F^A$. With the previously introduced axioms we can introduce three new rules: the **union**, **decomposition** and **pseudotransitivity** rules:

- **Union rule**: if $(X \to Y) \in F^A$ and $(X \to Z) \in F^A$, then $(X \to YZ) \in F^A$;
- **Decomposition rule**: if $(X \to Y) \in F^A$ and $Z \subseteq Y$, then $(X \to Z) \in F^A$;
- **Pseudotransitivity rule**: if $(X \to Y) \in F^A$ and $(WY \to Z) \in F^A$, then $(WX \to Z) \in F^A$.

Now, such rules can be proven through a theorem, which is here shown:

Theorem

Implications regarding the functional dependencies

Let F be a **set** of **functional dependencies**, then the following 3 implications hold:

1) if $(X \to Y) \in F^A$ and $(X \to Z) \in F^A$, then $(X \to YZ) \in F^A$;
2) if $(X \to Y) \in F^A$ and $Z \subseteq Y$, then $(X \to Z) \in F^A$;
3) if $(X \to Y) \in F^A$ and $(WY \to Z) \in F^A$, then $(WX \to Z) \in F^A$.

Proof

1) Demonstration of the **union rule**:
We start with $(X \to Y) \in F^A$: with the axiom of **augmentation** we add an X by both sides:

$$(XX \to XY) \in F^A \quad \Longrightarrow \quad (X \to XY) \in F^A$$

We simplify XX to X because we are dealing with sets. Now, we do the same with the other functional dependency, $(X \to Z) \in F^A$, reaching then

$$(XY \to YZ) \in F^A$$

We now have $(X \to XY) \in F^A$ and $(XY \to YZ) \in F^A$: we can apply the **transitivity** axiom and finally obtain $(X \to YZ) \in F^A$.

2) Demonstration of the **decomposition rule**:
At the beginning we have $Z \subseteq Y$, which for the **reflexivity** axiom can be translated as $(Y \to Z) \in F^A$. Now, we have

$$(X \to Y) \in F^A \quad \text{and} \quad (Y \to Z) \in F^A \quad \Longrightarrow \quad (X \to Z) \in F^A$$

This last passage is given by the axiom of **transitivity**.

PROOF

3) Demonstration of the **pseudotransitivity rule**:
Starting with $(X \to Y) \in F^A$, we can use the axiom of **augmentation** to reach $WX \to WY \in F^A$. Now, we have

$$(WX \to WY) \in F^A \quad \text{and} \quad (WY \to Z) \in F^A$$

We can use the **transitivity** axiom and reach finally $(WX \to Z) \in F^A$.

Regarding the **union rule**: it's obvious that if we have a functional dependency such as $(X \to A_i) \in F^A$ where $i = 1, 2, ..., n$, then $(X \to A_1 A_2 \dots A_n) \in F^A$; we can go back to the previous state by using the **decomposition rule**. We can say then that

$$(X \to A_1 A_2 \dots A_n) \in F^A \quad \text{if and only if} \quad (X \to A_i) \in F^A \text{ with } i = 1, 2, ..., n$$

When finding other functional dependencies that are not defined on F, a typical strategy is to repeatedly apply the Armstrong's axioms. Another way is to find, given a set of attributes X, all the attributes that depend on X given the functional dependencies on F. Such attributes are part of a set denoted as X_F^+, which is the **closure of X with respect to F**.

DEFINITION

Closure of X with respect to F (X_F^+)

Let R be a **relational schema**, F a **set of functional dependencies** on R and X a **subset** of R. The **closure of X with respect to F** is the set of all the **functionally determined** attributes of X, which are determined by the functional dependencies defined in F. In other words, all the attributes of X that are determined by the functional dependencies in F. It is denoted as X_F^+, and is defined as:

$$X_F^+ = \{A \mid (X \to A) \in F^A\}$$

Trivially, $X \subseteq X_F^+$, by the reflexivity axiom.

Regarding the closure of X with respect to F, we can have a lemma, which is the following:

LEMMA

Closure of X with respect to F

Let R be a schema and F a set of functional dependencies on R, then the following holds:

$$(X \to Y) \in F^A \iff Y \subseteq X_F^+$$

MMA

PROOF

Let Y be a set of attributes such that $Y = A_1 A_2 \ldots A_n$. Now, the lemma has **two points** that have to be proven: that if $Y \subseteq X_F^+$ then $(X \to Y) \in F^A$ and its opposite, since the original statement is **bidirectional**. We start with the first condition:

1) if $Y \subseteq X_F^+$ then $(X \to Y) \in F^A$

   Since Y is a subset of X^+, then for each $i = 1, 2, \ldots, n$ we can say that $(X \to A_i) \in F^A$ equals to $(X \to Y) \in F^A$ for the **union** rule;

2) if $(X \to Y) \in F^A$ then $Y \subseteq X_F^+$

   Similarly to the previous point, we use the **decomposition** rule: for every $i = 1, 2, \ldots, n$, then $(X \to Y) \in F^A$ is equal to $(X \to A_i) \in F^A$. This is allowed because all attributes A_i are a subset of Y, thus the possibility of using the decomposition rule. By the definition of X_F^+, we can say that $A_i \in X_F^+$, $\forall\, i \in \{1, 2, \ldots, n\}$, thus $Y \subseteq X^+$

We said earlier that the scope of F^A was to introduce an easier set into the equation that would allow us to compute the closure of F (F^+). Now we have all the necessary tools to introduce and prove the theorem that enunciates this very equivalence:

$F^+ = F^A$

THEOREM

Let R be a schema and F a set of functional dependencies on R, then the following holds:

$$F^+ = F^A$$

PROOF

The theorem can be proven by **double inclusion**. For such proof, we will show that both $F^+ \supseteq F^A$ and $F^+ \subseteq F^A$. If both statements will be true, then obviously $F^+ = F^A$.

First inclusion: $F^+ \supseteq F^A$

We want to prove, as a first step, that $F^+ \supseteq F^A$ (so F^+ **contains** F^A): let us consider a generic functional dependency $(X \to Y) \in F^A$, then we can prove that such functional dependency is also in F^+ (our goal is then $(X \to Y) \in F^+$) by **induction**. In order to do so, we apply i times one of the Armstrong's axioms.

Let us start with $i = 0$: in such a case, the functional dependency $X \to Y$ is in F^A, but is also in F. This is because at $i = 0$ we haven't yet used any axiom, and if F^A has an item at that time it's because it contains only the functional dependencies of F, which in this case is only $X \to Y$. This means that $(X \to Y) \in F^+$.

We formulate the following **hypothesis**: if we apply Armstrong's axioms for a

PROOF

number of times equal to $i-1$, then all the functional dependencies that were obtained from F are now in F^+. We now have to prove that such hypothesis is valid also if we apply the axioms for a number of times equal to i. We can prove that this hypothesis is true for each of the Armstrong's axioms:

1) **Application of the reflexivity axiom**:

   The functional dependency $X \to Y$ could have been inserted into F^A because of the **reflexivity** axiom, which would imply that $Y \subseteq X$. Now, in a **legal** instance of R (which will be called r), let two tuples t_1 and t_2. Clearly, since the instance is legal, we have that

$$t_1[X] = t_2[X] \quad \Longrightarrow \quad t_1[Y] = t_2[Y]$$

   As a consequence, we have that $(X \to Y) \in F^+$.

2) **Application of the augmentation axiom**:

   If the functional dependency $(X \to Y) \in F^A$ was obtained via **augmentation**, then it must've come from a functional dependency such as $(V \to W) \in F^A$, which has been obtained in turn by the application of Armstrong's axioms for $n-1$ times, such that $X = ZV$ and $Y = ZW$ for some $Z \subseteq R$. By our induction hypothesis, we have that $(V \to W) \in F^+$. Now, let r be a **legal** instance of R, and let the two tuples t_1 and t_2 belong to R. The following holds:

$$t_1[X] = t_2[X] \quad \Longrightarrow \quad t_1[V] = t_2[V]$$

   because $V \subseteq X$. Similarly, because of $(V \to W) \in F^+$, we have:

$$t_1[V] = t_2[V] \quad \Longrightarrow \quad t_1[W] = t_2[W]$$

   Now, there is also $(X \to Z) \in F^+$. Now, since X determines both W and Z, we reach that $X \to WZ$, and since $WZ = Y$, we have that $(X \to Y) \in F^+$.

3) **Application of the transitivity axiom**:

   By the transitivity axiom, if we have two functional dependencies such as $(X \to Z, Z \to Y) \in F^A$, then $(X \to Y) \in F^A$. Now, this tells us that in order to arrive to $X \to Y$ with i applications of the Armstrong's axioms, we should arrive at most at the step $i-1$ to a situation where we have $X \to Z$, $Z \to Y$. For our induction hypothesis, these two functional dependencies are in F^+, because they originated from F. Now, let be r a legal instance of R, and t_1 and t_2 be two tuples belonging to R. We have that

$$t_1[X] = t_2[X] \quad \Longrightarrow \quad t_1[Z] = t_2[Z] \quad \Longrightarrow \quad t_1[Y] = t_2[Y]$$

   Therefore $(X \to Y) \in F^+$.

PROOF

Now that our hypothesis has been verified, we just need to show the second inclusion.

Second inclusion: $F^+ \subseteq F^A$

We want to prove this second step by **contradiction**: suppose that there exists a functional dependency $(X \to Y) \in F^+$ such that $(X \to Y) \notin F^A$. Let us also consider the following instance of R:

$$r = \begin{array}{c} \begin{array}{|c|c|c|c|c|c|c|c|}\hline 1 & 1 & ... & 1 & 1 & 1 & ... & 1 \\ \hline 1 & 1 & ... & 1 & 0 & 0 & ... & 0 \\ \hline \end{array} \\ \underbrace{\qquad\qquad}_{X^+} \qquad \underbrace{\qquad\qquad}_{R-X^+} \end{array}$$

We make the assumption that r is a **legal instance** of R. Suppose that it's wasn't legal, so there would be at least one functional dependency in the form $(V \to W) \in F$ (which is automatically also in F^A) that is not satisfied by the values in r. The fact that such dependency would exist implies that there must be at least two tuples t_1 and t_2 that don't respect such functional dependency, so that they are equal on V and different on W. In symbols, that would be

$$V \subseteq X^+ \qquad W \cap (R - X^+) = \emptyset$$

Since $V \subseteq X^+$, it means that V is an attribute which is functionally determined by some functional dependency. We can use the previously shown lemma (denoted as **Lemma 1**), and determine that a functional dependency such as $(X \to V) \in F^A$ exists. We can use X because, by definition of X^+, X is a subset of X^+. By applying the axiom of transitivity, we get that since $\{X \to V, V \to W\}$ then $X \to W \in F^A$. This tells us that $W \subseteq X^+$, which contradicts our original statement. This determines that r is a legal instance of R.

Now, let's go back to the original assumption, so that $(X \to Y) \in F^+$ and $(X \to Y) \notin F^A$. We know that r is a legal instance, so by definition not only r satisfies all the functional dependencies in F^+, but it also satisfies $X \to Y$. Now, we know that $X \subseteq X^+$ because of the definition of X^+. There are two tuples in r that are identical on X, so they must be identical on Y too. This leads us to $Y \subseteq X^+$, since Y is determined by X. By the lemma, we get that $X \to Y \in F^A$, so we have a **contradiction**, and we can now say that $F^+ \subseteq F^A$.

With this theorem we have a way to finally identify all the functional dependencies in F^+. The only drawback is that it takes exponential time to compute all the functional dependencies: let us just consider the axiom of reflexivity, it would analyse each possible subset of R, and there are $2^{|R|}$ possible subsets of R; We would discover, moreover, that $|F^+| \gg 2^{|R|}$ (so the number of elements in F^+ is much greater than the number of subsets of R).

But why do we have to know how to compute F^+? The definition of **Third Normal Form (3NF)** is based on such concept. In order to obtain a schema in 3NF from a schema that is not in such form, we have to decompose such schema. While decomposing, we want to preserve the dependencies in F^+.

SECTION 3.3 Third Normal Form (3NF)

We know that a relation R is said to be legal if it respects all the given functional dependencies in a set F, so for instance, if we had a database for storing the Students' data, we would have the following:

Students:

Matr	TaxCode	Surname	Name	BDate	Town
...					

$$F = \{\,\text{Matr} \rightarrow \text{Matr, TaxCode, Surname, Name, BDate, Town}\,\}$$

The relation *Students* would be said **legal** if F was valid for each tuple of *Students*. If F was equal to

$$F = \{\,\text{TaxCode} \rightarrow \text{TaxCode, Matr, Surname, Name, BDate, Town}\,\}$$

then *Students* would still be considered as a legal relation. We can't say the same if for instance we had a set F such as $F = \{\,\text{Surname} \rightarrow \text{Name}\,\}$: there could be for instance two students with the same name and surname, but with different matriculation numbers. We see a pattern here: all the functional dependencies of the type $K \rightarrow X$ have the attribute K that behaves like a **key**.
We come thus to a very important concept: the only **non-trivial** functional dependencies that a relation R must satisfy in order to be said **legal** are of the form

$$K \rightarrow X$$

where K represents a **key** belonging to R
We can make another example regarding this aspect of keys: suppose that we have a relation *Exams* with the following attributes:

$$\text{Exams} = \{\,\text{Matr, CCode, DatePassing, Grade}\,\}$$

In such a system, a student with a given matriculation ID can pass the exam of a specific course only once. We derive from such property that there is only one functional dependency that respects the previously said property:

$$\text{Matr, CCode} \rightarrow \text{DatePassing, Grade}$$

Why is it so? Let's suppose for instance that the functional dependencies were one of the following:

- Matr → DatePassing and Matr → Grade: if the matriculation ID determined the date on which an exam was passed, then if a student would not be able to make more than one exam, since the relation wouldn't be anymore legal. Same thing goes for the grade: a student would always have the same grade for all the exams, otherwise the instance won't be legal anymore;
- CCode → DatePassing and CCode → Grade: the first functional dependency won't allow for an exam of a specific course to be taken in multiple dates, while the second doesn't allow for an exam of a specific course to have different grades.

Logically, we would think of *Matr* and *CCode* as two independent keys, but they actually form together one single key, since if they were considered as independent then we would have the previously described two cases. This property that we just used on these two cases is the fundamental property of the **third normal form** (**3NF**):

Definition

Third Normal Form (3NF)

A relation schema R is said to be in **T**hird **N**ormal **F**orm (**3NF**) if

$$\forall\,(X \to A) \in F^+, \quad A \notin X$$

then one of the two applies:

- A **belongs** to a **key** (A will be called **prime**);
- X **contains** a **key** (X will be called **super key**)

In other words, R is in **3NF** if the only non-trivial functional dependencies that must be satisfied by every legal instance are of the type

$$K \to X$$

where either K **contains** a **key or** X **is contained** in a **key**

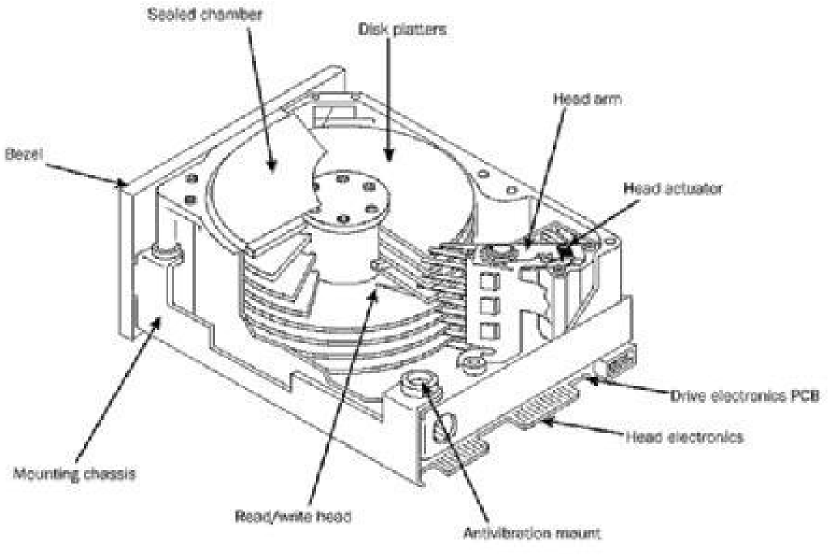
CHAPTER 4

Physical Organization

We said at the beginning that a DBMS is made out of multiple layers: an **external** layer, a **logic** layer and a **physical** layer. In the previous chapters we always dealt with the logical later, now we'll focus on the physical layer.

We know that in a computer there are two types of memory: **volatile** and **non-volatile**. Some examples of volatile memory are the CPU registers, the cache and the central storage, while some examples of non-volatile memory are HDDs (Hard Disks Drive) or SSDs (State Solid Drive). The examples that we gave for volatile memory represent elements with a high-speed memory, but they are costly and limited in the capacity; regarding non-volatile memory instead, we have slower memory, but it's cheaper and has less limits regarding the capacity.

In a DBMS, the physical data is stored on the secondary storage such as HDDs and SSDs, while the buffer and the code of the DBMS is stored in a buffer, which is considered as part of the primary memory.



View of an HDD

How do HDDs work? They have a set of **disks** (or **platters**), which are covered in a magnetic dust. The platters rotate on a spindle at a constant speed. Via the use of some arms and some heads, the data can be read and write from/to the hardisk. Hardisks are also called **D**irectly **A**ccessible **S**torage **D**evices (**DASDs**).

Whenever we have to read from a disk, we have to locate the sector on which the data is written: this implies that we need some time in order to position the actuator (called **seek time**) and wait until the disk rotates enough to make the sector arrive under the head (we call such delay the **rotation delay**).

DEFINITION

Seek time and Rotation delay

We call **seek time** the time needed to position the head on the correct track of the disk.

The **rotation delay** (**ROT**) is the delay given by the waiting that the disk rotates until the required disk sector is brought under the read/write head.

There are some other timings that we consider:

1) **transfer time** (**TR**): it's the time needed to transfer a quantity of data from/to the disk. It's given by the **block size**, by the **density** of the **magnetic particles** and on the **rotation speed** of the **disks**;

2) **service time**: it's the sum between the **seek time**, the **rotational delay** and the **transfer time**;

3) **response time**: it's the sum between **service time** and **queuing time**.

Some other two important timings are:

1) **Random Block Access Time** (T_{rba}): it's the expected time to read/write on a disk regardless of the previous read/write operations

$$T_{\text{rba}} = \text{Seek} + \frac{\text{ROT}}{2} + \frac{\text{BS}}{\text{TR}}$$

2) **Sequential Block Access Time** (T_{sba}): it's the expected time to sequentially retrieve the disk block with the read/write head already in place

$$T_{\text{sba}} = \frac{\text{ROT}}{2} + \frac{\text{BS}}{\text{TR}}$$

With **BS** we mean the **B**lock **S**ize. How can we have the most optimal storing result? We need a system for which the organization on the various tracks and cylinders minimizes as much as possible the **seek time** and, partially, the **rotation delay**; we also want to minimize as much as possible the accesses to different blocks. We can't have everything all at once, some trade-offs must be made. We will focus here on the physical organization of structured, relational data.

SECTION 4.1 Primary File Organization Methods

At a physical level, the database consists of a set of various **files**; each file can be organized as a set of different **pages**, where each page has a fixed length (of usually 4KB). Each page can store several **records**, where a record consists of different **fields**, with either a fixed or variable size. Each field represents an attribute value. There are various ways for which we can find the records in the disk:

- via **linear search**, where each record is stored and retrieved depending on a key;
- via **hashing** or **indexing**, where the location on the disk is specified by its relationship with the record search key. Such method has an improved performance in contrast to the linear search, since we have a direct access to the location where the record is stored;
- via the organization and the **type of files**: there could be **heap files**, **sequential files**, **random files**, **indexed sequential files** or **lists of data**.

4.1.1 Heap file organization

Such type of file is the basic of primary files organization. Each time a new record has to be inserted in a file, it gets added at the **end** of such file. Here we have **no relationship** between the record's attributes and their physical location, so the only option for retrieval is **linear search**. For a file with a number of blocks (**NBLK**), it usually takes $\text{NBLK}/2$ time to find the record according to the given key. If the search key is not unique, then the whole file has to be scanned.

4.1.2 Sequential file organization

In this kind of file, the data is stored in an **ascending/descending order** depending on the search key. It's way more efficient to access the records in a specified order rather than not knowing where we have to look for. Although the data is stored in such way, we still access to the records with a linear search, but we have a **stopping** criterion: we stop whenever a key is higher/lower than the one needed. In order to have a more effective search method, we can use **binary search**: for a unique key K, with values K_i, we retrieve the record with the key value K_μ.

```
• Selection criterion:  record with search value $k_\mu$
• Set $l = 1$ and $h$ as the number of blocks in a file, supposing that the records
  are stored in an ascending order with respect to the search key $K$
• Repeat until $h \geq 1$:
    – Set $i = \frac{(1+h)}{2}$, rounded to the nearest integer
    – Retrieve block $i$ and examine key values $K_j$ of records in block $i$
• if any $K_j = K_\mu$, then the record is found and we can exit
• else if $K_\mu >$ all $K_j$, then continue with $l = i+1$
• else if $K_\mu <$ all $K_j$, then continue with $h = i-1$
• else the record is not in the file.
```

SECTION 4.2 Exercises

4.2.1

EXERCISE

Assume we have a file with **250.000** records, each of them of size **300** bytes, **75** of which for the key field. Each block is of size **1024** bytes. A block pointer is **4** bytes.

A) If we use a hash organization with 1200 buckets, how many blocks do we need for the bucket directory?

B) How many blocks do we need for the buckets, assuming a uniform distribution of records into buckets?

C) Still assuming a uniform distribution of records, what is the average number of accesses needed to find a record in the file?

D) How many buckets should we create to have instead an average number of accesses less or equal 10, still assuming a uniform distribution of records in the buckets?