Databases I Summary

Introduction to Database Design

Introduction to Database Design

Introduction to database modelling and design. Premises: motivations for information management. First example via tables. Levels of abstraction

---

1. Premises and Example

Q. How can we manage *information*, in any *functional context?* That is, how can we *model*, *normalize*, *access*, *restrict* and *extract* a database?
A. This is the goal of our course. Let us see a brief and heuristic example.

OBJECTIVE. Suppose we have a library where we store *books* where each *book* is identified by its ISBN, name and author. Each author is then identified by their name and similar information
APPROACH. Let us resolve in the most simple manner, that is we start making tables. Let us define a *first table*, where we store the *books*.

ISBN	TITLE	AUTHOR
12345432	Il Fu Mattia Pascal	Mattia Pascal
34564	Lo zar non è morto	Antonio Beltramelli, ..., Luciano Zuccoli

We immediately notice that storing *multiple authors* in a *single column* can reveal to be uncomfortable and lead to messy results. A possible solution is to delete the authors column, and to add a new table dedicated for storing *authors*:

IDAUTHOR	NAME	...
1	Mattia Pascal	...
2	Antonio Beltramelli	...
3	Luciano Zuccoli	...

And now, to *"link"* the books and authors, we can just make another table which connect the unique identifiers! (Which are *ISBN* and *IDAUTHOR*).

IDAUTHOR	ISBN
1	12345432
2	34564
3	34564

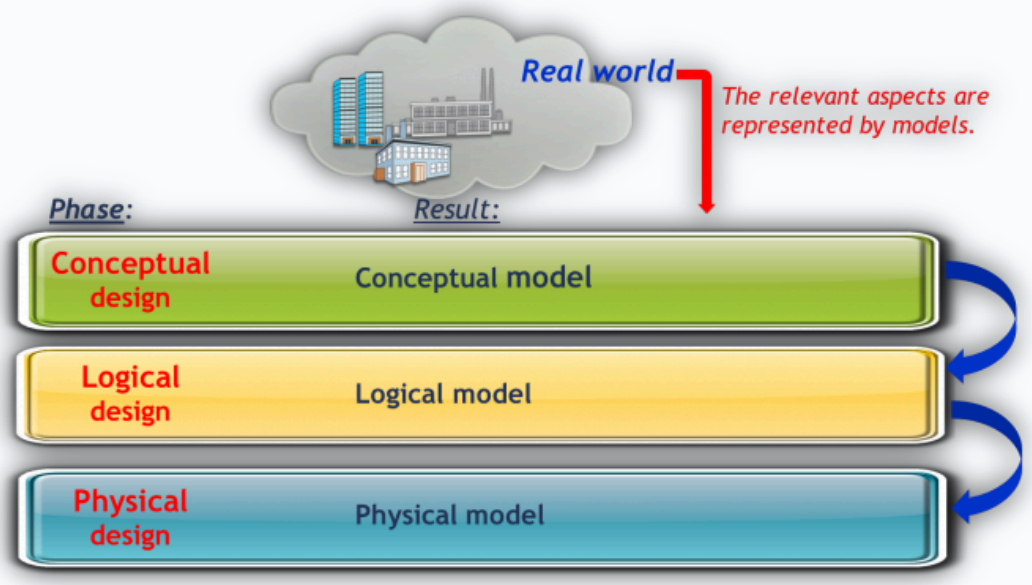
This is the core idea of data modelling. We will see to give precise definitions later on.

X

2. Levels of Abstraction

So, our core process is this: we have a functional context in the *real world*, and our goal is to make a database which *represents a model* of the context. Or well, at least the relevant aspects of it. By *modelling*, we start entering the *levels of abstraction*: first we start by *conceptual design* and then pass to the *physical design*. The transition from conceptual to physical is automatically done through software, such as *PowerDesigner*.

We will see specific models which belong to one of the two designs later on.



NOTE: Ignore the logical design and model. This step will *not* be covered in the course.

X

Entity-Relationship Model

Entity-Relationship Model

X

The entity-relationship model. Definition of entity, attributes, primary identifier. Definition of relationship and cardinality. Dependencies. Inheritance.

X

0. Voci correlate

- Introduction to Database Design

1. Introduction to ERM

Entity-Relationship Models is one of the possible forms of *conceptual data modelling*. This in particular allows *independence* from the *implementation* of the model. We have a lot of versions (syntaxes) of this, in this page we will focus on the *MARTIN/IE/Crow's Foot* version of the Entity-Relationship model.

X

2. Entities, Attributes

#Definizione

DEFINITION. (*Entity*)
In ERM, an *entity* is the representation of an object in a model. So, they're a class (or set) of things to be managed, and each of them has several instances (*objects*).

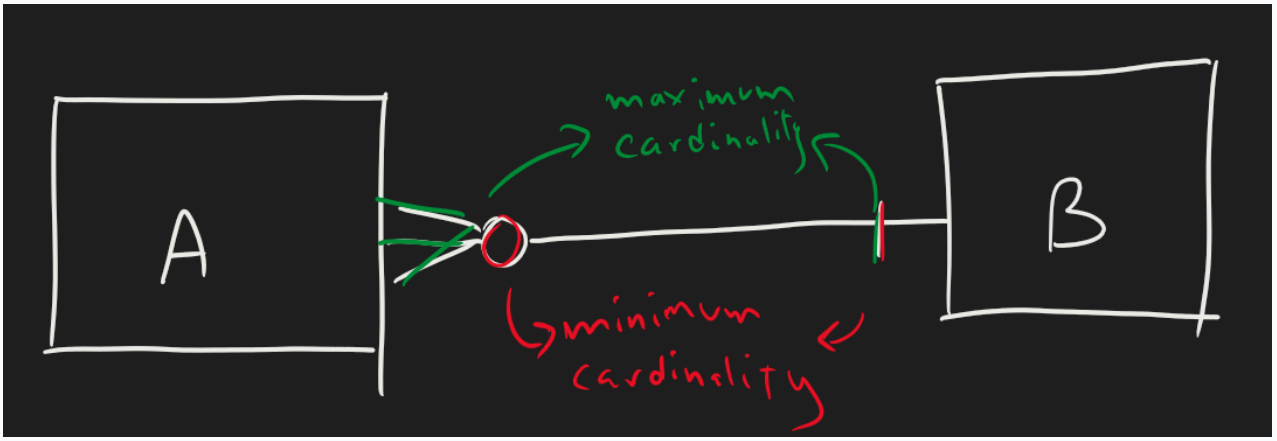
#Definizione

DEFINITION. (*Attributes, primary identifier*)
Attributes describe the properties of an entity. Every instance must have the attributes of the objects, and obviously they tend to be different. Each attribute has a *data type* or *domain*.
Moreover, one of the attributes can be chosen as the *primary identifier* of the entity.

3. Relationships between Attributes

#Definizione

DEFINITION. (*Regular relationship*)
Let A, B denote two entities. A regular *relationship* from A to B is a mathematical relation between the two sets that contain the instances. Their degree of relationship is defined by the *maximum* and *minimum cardinality*. In particular, the *maximum cardinality* is the maximum of the instances from one entity that could partecipate in a relationship; it could be 1 or many (> 1). The concept of *minimum cardinality* is analogous, but it can be only 0 (optional) or 1 (mandatory relation).



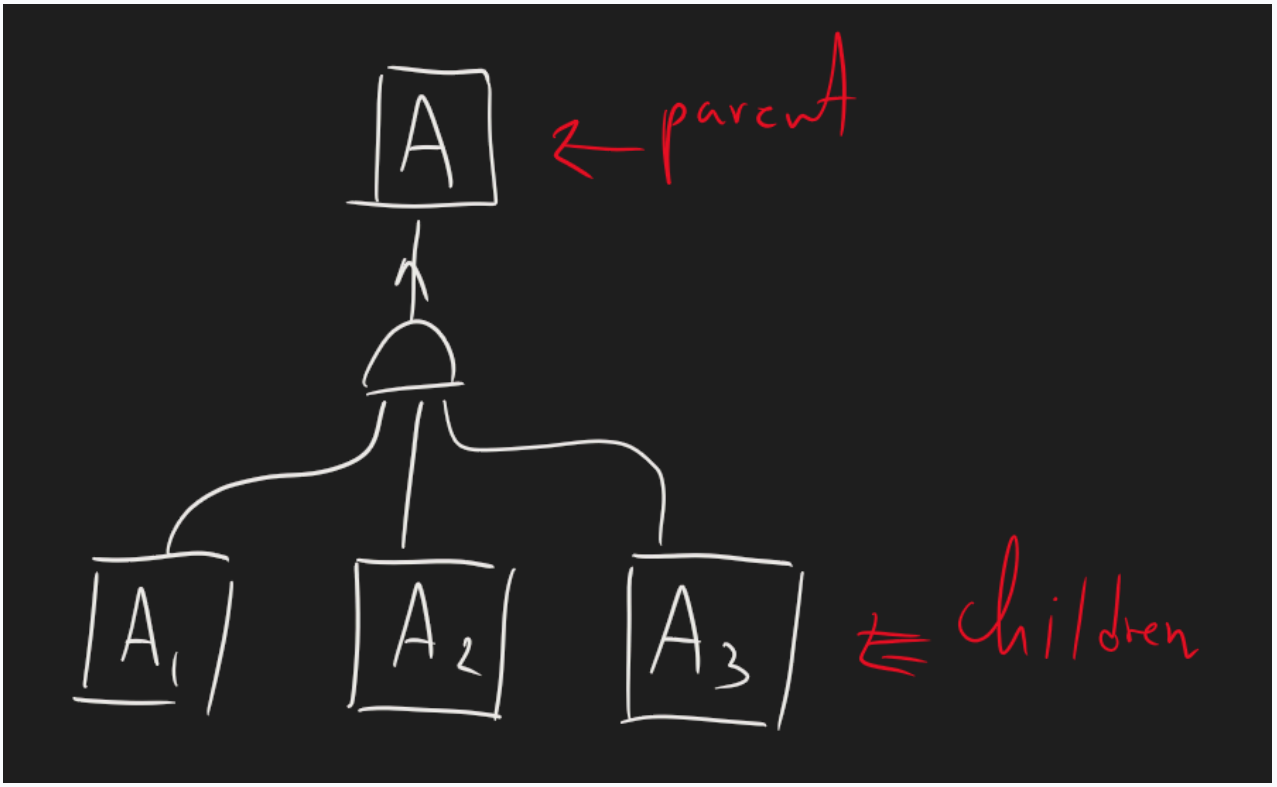
#Definizione

DEFINITION. (*Weak entities*)
A *weak entity* A *dependent on* B is an entity that *"inherits"* the primary identifier from its *"father"*. The maximum cardinality is always set to 1, as we have the *parent unicity*.

#Definizione

DEFINITION. (*Inheritance*)
In this case the *child inherits* the information from the *parent* and keep their *specialized attributes*, with their own unique primary identifier.
There are two types of inheritances:

- *Complete* if an instance in the parent entity *is always in one of the child entities* (note: they can be in more than one!)
- *Exclusive* if an instance in the parent entity can be *only in one of the child entities*; this is denoted with a cross in the inheritance gate symbol



X

Relational Model

Relational Model

X

A case of physical model: the relational model. Underlying mathematical model (relational algebra)

X

0. Voci correlate

- Introduction to Database Design

1. Introduction to Relational Model

The *relational model* has been invented in the *'70s*, and is still being widely used today. It is still used as it has the following advantages:

- Being built on top of a simple mathematical model, being *relational algebra*
- Enforcing data independence
- Using simple query languages, such as SQL.
  Let us see some preliminary mathematical definitions.

#Definizione

Definizione (relation and database).

Let $(A_n)_n$ be certain sets: R is said to be a *relation* for $(A_n)_n$ if an only if it is a *subset* of the cross product between those sets.

$$R \subseteq \bigtimes_n A_n$$

A *database* is a set of *relations*.

#Definizione

Definizione (attributes and rows).

Let R be a relation of $(A_n)_n$. The attributes $(A_n)_n$ are said to be *columns*, and the *singular tuples* $r = (a_{k_1}, \dots, a_{k_n}) \in R$ are said to be *rows*.

#Osservazione

Osservazione (need of primary keys).

Note that in a relation we can have *singular repeated values*. So, we can arbitrarily define a *column* to be a *"primary key"*, where in each row the primary key cannot be repeated.

#Definizione

Definizione (primary key).

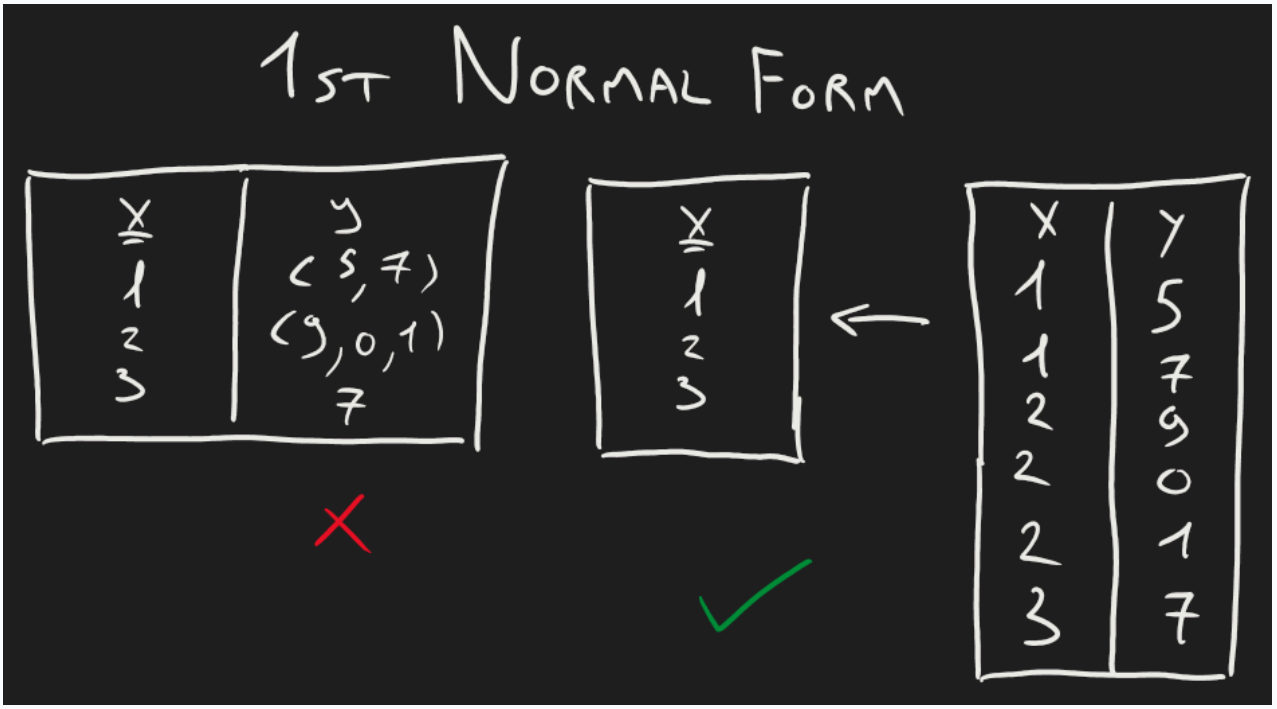
A *column* of a relation R is said to be a *primary key* iff its values are unique and cannot be non-existent.

2. Database Normalization

Note that with our mathematical models, *databases* can be in any form. However, we prefer them to be structured in a certain way (*"normalized"*) in order to manage it more easily.

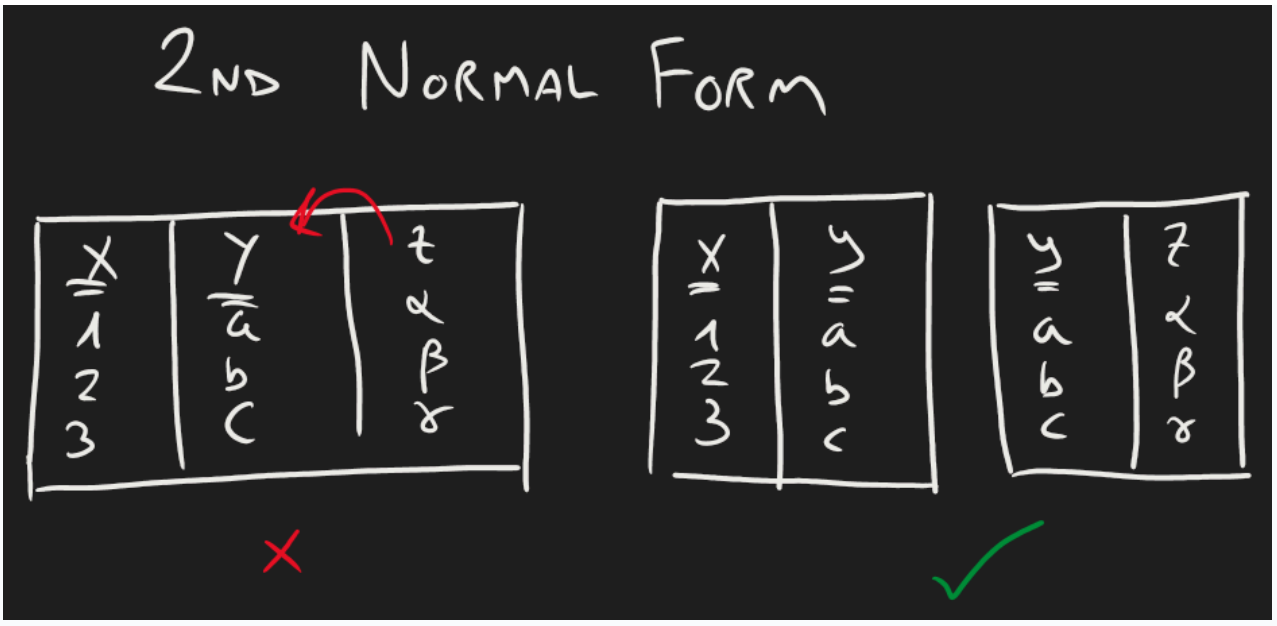
#Definizione

> Definizione (first normal form).
>
> A table is said to be in the *first normal form* if and only if each attribute domain contains only *atomic values* and the value of each attribute contains only *one value*.
>
> In other words, each value of a row must be one-dimensional only.



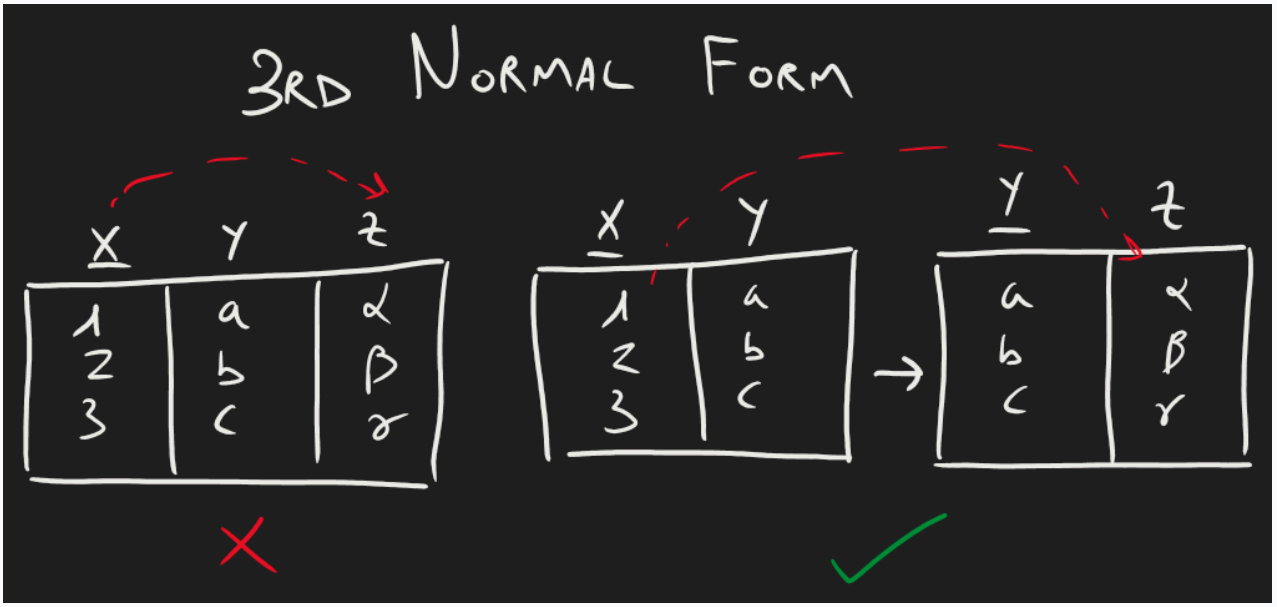
#Definizione

> Definizione (second normal form).
>
> A table is said to be in *second normal form* iff all non-key attributes are functional dependent of the all key. In other words, each non-key attribute should depend on every key attribute.



#Definizione

> Definizione (third normal form).
>
> A table is said to be in *third normal form* iff all non key attributes depend on the key attributes in a *non-transitive manner*; in other words, I cannot transitively obtain information.



Q. Why do we need database normalization?

A. Mainly to prevent *anomalies* as we *maintain the database*. In particular:

- We have *update anomalies* when tables are *not in the third normal form*; when we update a transitively dependent key, we have to update all rows.
- We have *insertion anomalies* when tables are *not in the second normal form*; suppose we want to insert a row with information about all the non-key values. Until we have the key value, we cannot insert that information.
- Same as before, we have *deletion anomalies*; when we delete an entry, we might delete relevant information that could have not been deleted..

Conversion from Entity-Relationship Model to Relational Model

Conversion from Entity-Relationship Model to Relational Model

---

X

Converting ERM diagrams to Relational diagrams (conceptual -> physical)

X

---

0. Voci correlate

- Entity-Relationship Model
- Relational Model

1. Simple Conversions

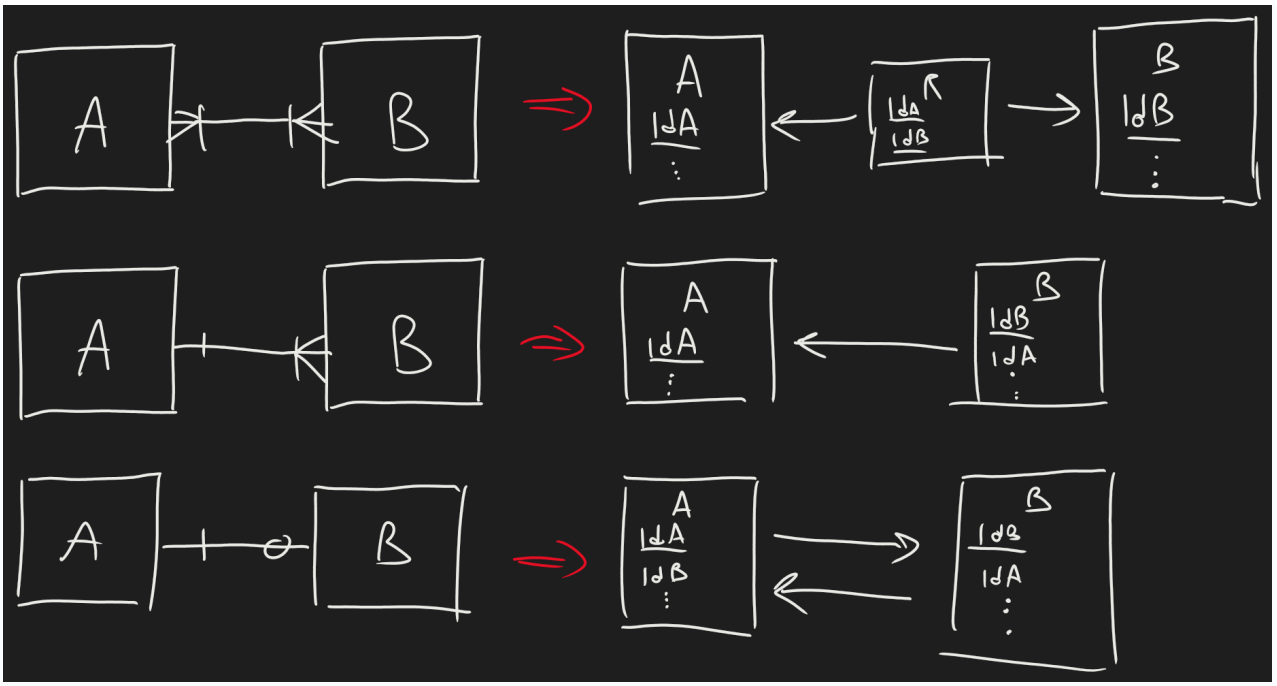
Q. The Entity-Relationship model is abstract for representing databases models, making it "easier" for us to model. However, how do we convert it into something more "concrete"?

#Teorema

> Teorema (entities to relations).
>
> *Entities* become *relations* (tables), where:
>
> - Attributes are called *fields*
> - Primary identifiers are converted to *primary keys*
>
> Sometimes it is necessary to add other fields representing *relationships*. We will see below

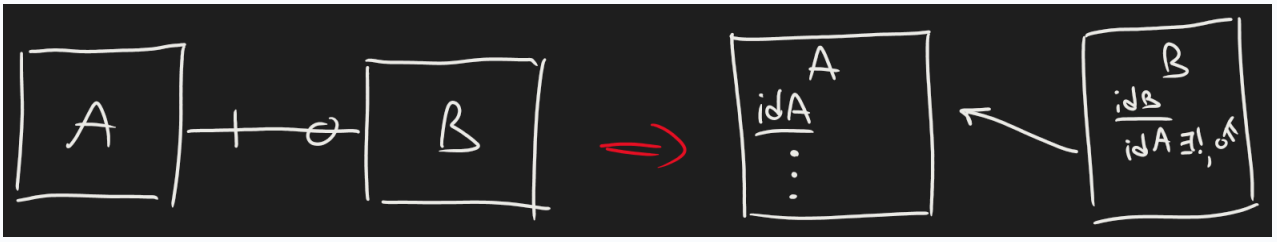
#Teorema

> Teorema (relationships to relations).
>
> Simple *relationships* also become (generally) *relations*; however, this depends on the cardinality of the relationships.
>
> 1. If it's a *many-to-many* type relationship, then we need to create an *auxiliar relation* containing keys to both of the referencing tables (which are primary and foreign keys at the same time)
> 2. If it's a *one-to-many* type relationship, then it's just needed to add a *foreign key field* in the second relation (the one that is on the "many" side)
> 3. If it's a *one-to-one* type relationship, then it's just necessary to add *foreign key fields* to both of the relations
>
> See the diagram below for a graphical representation.



#Osservazione

> Osservazione (alternative conversion of one-to-one relationships).
>
> Let it be noted that we can have an *alternative way* to convert one-to-one relationships.
>
> Instead of applying *foreign keys* to both of the tables, we can apply only one *alternative key* to one of the tables (with the UNIQUE constraint); this ensures that our design is simpler and better optimized.



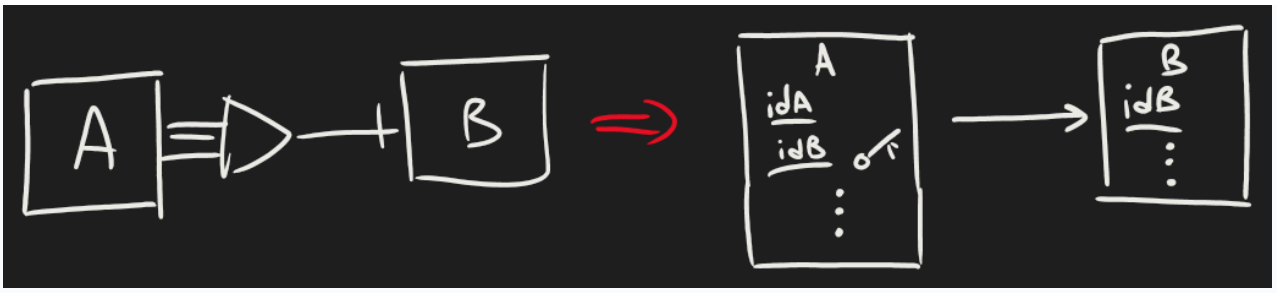
X

2. Weak Relationships and Inheritances

Q. How about more *"complex"* cases, such as weak relationships or inheritances?

#Teorema

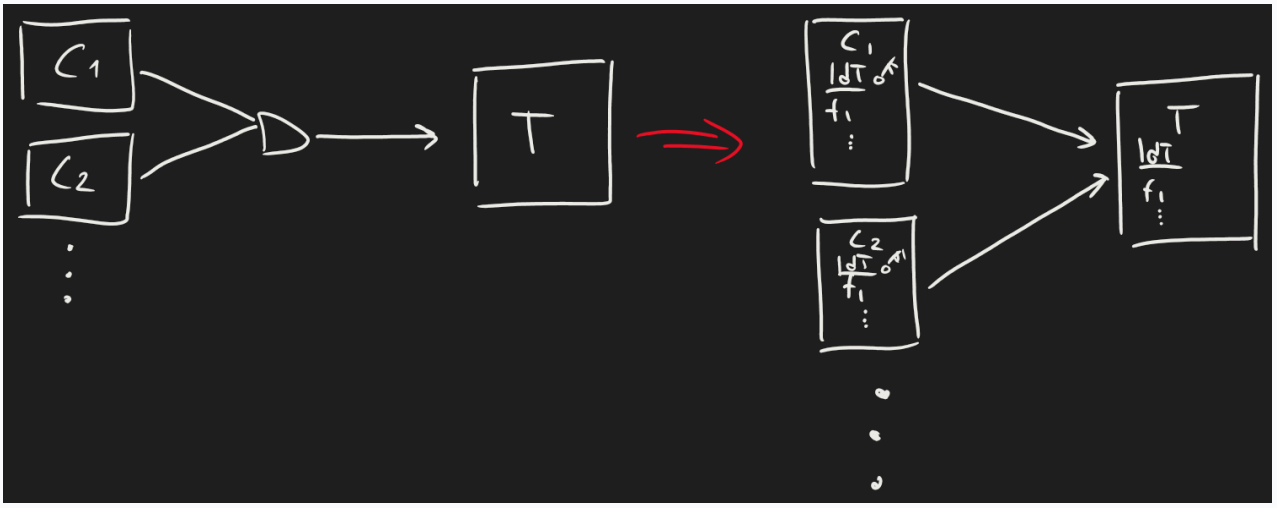
> Teorema (weak relationships to relations).
>
> In the case of *weak relationships*, we simply have a *"key inheritance"*, where basically the *weak entity* contains also the primary key of the pointed entity, except that it is also a *primary key*.
>
> See below for the graphical idea



However, for relationships the case is more complex as we might have more options to do this.

#Teorema

> Teorema (inheritance to relationships, 1).
>
> In the case of *inheritances*, we can use the first approach:
>
> - Every entity becomes a *relation*
> - The *"father"* contains the base attributes and his own identifier
> - The *"children"* inherits everything from the father (including his identifier as a primary key)
>
> In other words, it is even a *"broader case"* of weak relationships where we also include the other attributes.
>
> *When to use*: When a strict separation of entity types with well-defined, separate behaviors is needed; it's also better if we anticipate querying or working with each entity type frequently and independently.
>
> *Disadvantagse*: We have duplicated information, increasing the amount of space cost. Moreover, this won't ensure the *exclusivity* (if specified) between the children, and to do that we will need to use more complex back-end structures (triggers).
>
> This the most *common approach*



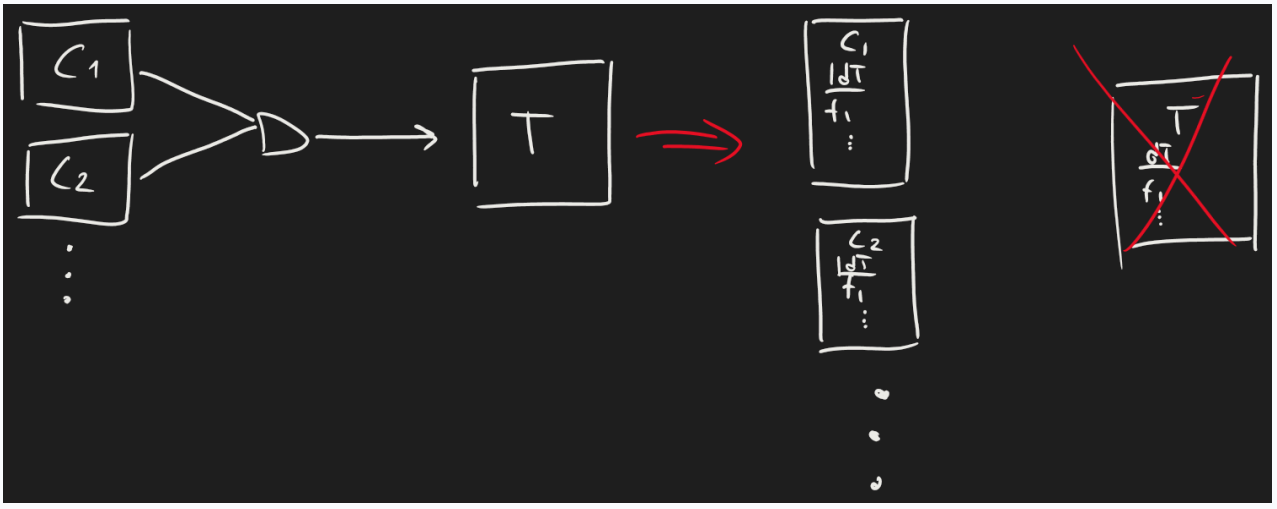
#Teorema

> Teorema (inheritance to relationships, 2).
>
> Another approach for inheritances is as follows:
>
> - Keep only the *"children"* as relations

- The children have their own primary identifiers
- Inherit the common attributes

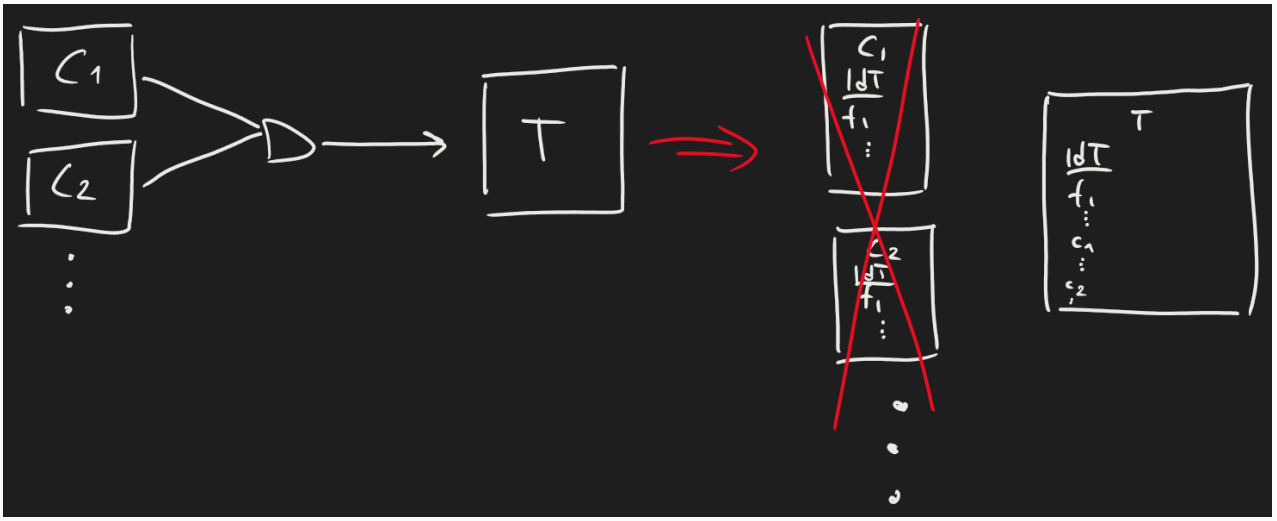
When to use: This approach is a good compromise if you don't need to frequently access all children together and prefer reducing redundant data.

Disadvantages: The children are not related to each other in any way and they need to be joined if we want to query them together



#Teorema

> Teorema (inheritance to relationships, 3).
>
> The last option is to keep the *parent entity* only as a relation, and make the childrens' fields optional (meaning we accept NULL values in some fields)
>
> *When to use*: This approach is best if your system often treats all children similarly, and the presence of NULL values is not a major concern (e.g., for performance reasons or specific types of analysis).



X

Practices for Database Modelling

Practices for Database Modelling

X

A short list of good practices for database modelling.

X

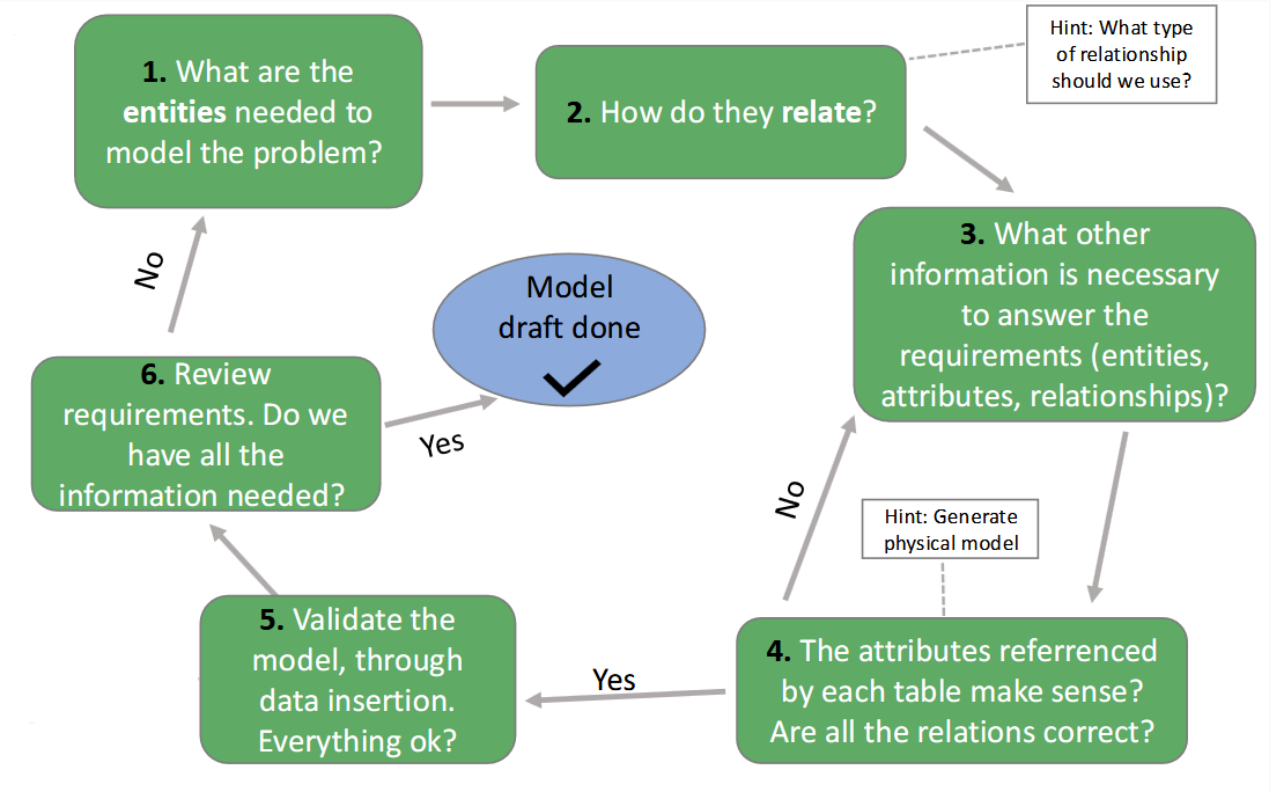
0. Voci correlate

- Introduction to Database Design
- Entity-Relationship Model

1. Database Modelling Pipeline

When having to deal with a *database modelling* assignment, it is reccomended to begin from the *conceptual model* and follow the following steps:

- Define the necessary *entities* for a basic representation
- Define the *relationship* between entities
- Define other *entities, attributes and relationships* for more specific requirements
- Check for *integrity* and *consistency* of the model
- If possible, test the model through insertion
- Review the requirements



Tips:

- If we need to *"store information"* between many-to-many relationships, we can add an auxiliary table with weak dependencies pointing both to original tables
- Avoid circularity as much as possible since it can lead to data inconsistencies

Introduction to SQL

X

Introduction to SQL

X

Introduction to the SQL language: definition of DDL, DML, paradigm of SQL and flavors. Mathematical model: extended relational algebra.

X

0. Voci correlate

- Relational Model

1. Introduction to SQL

The **SQL** language is the #1 programming language for *database modelling and manipulation*, as it is simultaneously:

- A *DDL* (*Data-Definition Language*), as it can define the structure of the database and impose particular restrictions on it
- A *DML* (*Data-Manipulation Language*) as it allows us to *query* and *insert/update/delete* data from the previously defined database
  So, it allows us to *define* and *interact* with a relational database at the same time.

Moreover, **SQL** is a *declarative* (based on logical statements) *high-level* language, so we have a high level of abstraction from our physical model.

There are a lot of variants of **SQL** (called *"flavours"*), the one we will use is *Transact-SQL* from *Microsoft / Sybase*

X

2. Extended Relational Algebra

Before looking at **SQL** in detail, we need to *slightly* modify our previous mathematical model of *relational database*. The model is called *extended relational algebra*, as we slightly weaken some notions.

#Definizione

> Definizione (bag).
>
> A *bag* is an *unordered list or a set of items* where the items can be duplicated. One can be denoted with the following symbol:
>
> $$\mathcal{A} = \overline{\{x, y, z, \dots, x\}}$$

Then our database model will be similar to our *relational model*, where we use *extended relational algebra* instead of traditional one (Relational Model). For instance, we can define a table as the following

#Definizione

Definizione (table or relation in extended relational algebra).

A *relation* or *table* between the bags $(A_n)_n$ is a *bag of n-tuples* where the attributes are specified by the relation schema.

$$\overline{\{(x_1, y_1, \dots, z_n), \dots, (x_1, y_1, \dots, z_n), \dots, (x_n, y_n, \dots, z_n)\}} \subseteq \bigtimes_n A_n$$

x

Database Definition in SQL

Database Definition in SQL

x

Basic DML statements in SQL for creating databases. Standard datatypes in SQL. Keywords to specify primary, alternate and foreign keys. Forcing mandatory values. Creating constraints. Deleting databases or columns.

x

0. Voci correlate

- Introduction to SQL

1. Structure Creation and Deletion

In the *SQL* language we have the following hierarchical structures:

- *Database* contains every *schema*, which contain every *table*.

$$\mathrm{DB} \supset \mathrm{S}_n \supset t_{n,k}$$

We can either *create* or *delete* them:

```SQL
-- Create
CREATE DATABASE <my_db>
CREATE SCHEMA <my_schema>
CREATE TABLE <my_table>
-- Delete
DROP DATABASE <my_db>
DROP SCHEMA <my_schema>
DROP TABLE <my_table>
```

In particular we have more detailed syntax for creating tables:

```SQL
CREATE TABLE <my_table>
(
    <col1> <col1_dtype> <keywords>,
    ...
    <coln> <coln_dtype> <keywords>
)
```

We will see *data types* and *keywords* in detail later (or soon).

X

2. Standard Datatypes in SQL

We have the following standard datatypes in SQL:
Character strings: We can use `CHAR(n)` for *fixed-length strings* or `VARCHAR(n)` for *variable-length strings*.
Bits: We can use `BIT(n)` or `BIT VARYING(n)`.
Booleans: We can use `TRUE`, `FALSE` or `UNKNOWN`. We will se how this *three-valued logic* works in detail later.
Integers: We can use `INTEGER` (or `INT`), or alternatively `SHORTINT` for less digits.
Floats: We can use `FLOAT` (or `REAL`), or alternatively `DOUBLE PRECISION` for more precise floats. If we have *fixed-point numbers* (e.g. we already know the number of digits), we can use `DECIMAL(n,d)` where n is the number of digits and d the number of decimal places.
Dates: `DATE` or `TIME`.

X

3. Keywords in SQL

Q. How to specify particular characters of certain columns? e.g. how do we specify if a column is a primary key, or has mandatory values, et ecetera...
A. With *keywords* or *constraints*.

Let us see the first case.

Primary Key: `PRIMARY KEY` specifies that a column is a primary key. It can be either put as a keyword of a column, or at the end as a statement of type `PRIMARY KEY (<key1>, ..., <keyn>)`.
Alternative Key: `UNIQUE` allows us to emulate the characteristic of *alternative keys*, e.g. they have to be unique but can assume empty values.
Foreign Key: `FOREIGN KEY(<fk1>, ..., <fkn>) REFERENCES TABLE <table> (<ptr1>, ..., <ptrn>` allows us to specify for *foreign keys* pointing to *certain "pointers"* of a certain table.
Mandatory: `NOT NULL` allows us to specify a column being *mandatory*.

X

4. Constraints in SQL

Let us define some sort of entity, called *constraints*. They are non-anonymous (e.g. have their own name) and are associated to a keyword (example: `CONSTRAINT x PRIMARY KEY (key)` is a constraint). They can be treated just like objects, so they can be *created* and *dropped*.

General syntax for restrictions:

```SQL
CONSTRAINT <KEYWORD> <*KEYWORD ARGUMENTS>
```

Manipulating constraints:

```SQL
-- Adding constraint
CREATE TABLE my_table
(
    MY_VAR <DTYPE> CONSTRAINT <...>
    ...
    CONSTRAINT my_constraint <...>
) -- or

ALTER TABLE my_table
    ADD CONSTRAINT my_constraint <...>

-- Removing constraint
ALTER TABLE my_table
    DROP CONSTRAINT my_constraint
```

X

Basic Queries in SQL

Basic Queries in SQL

X

Simple querying in SQL. Basic anatomy of a query. Wildcard ` and aliases. Operators on conditions. Type casting on variable selection. Keyword for selecting unique values. Sorting queries. Observation.*

X

0. Voci correlate

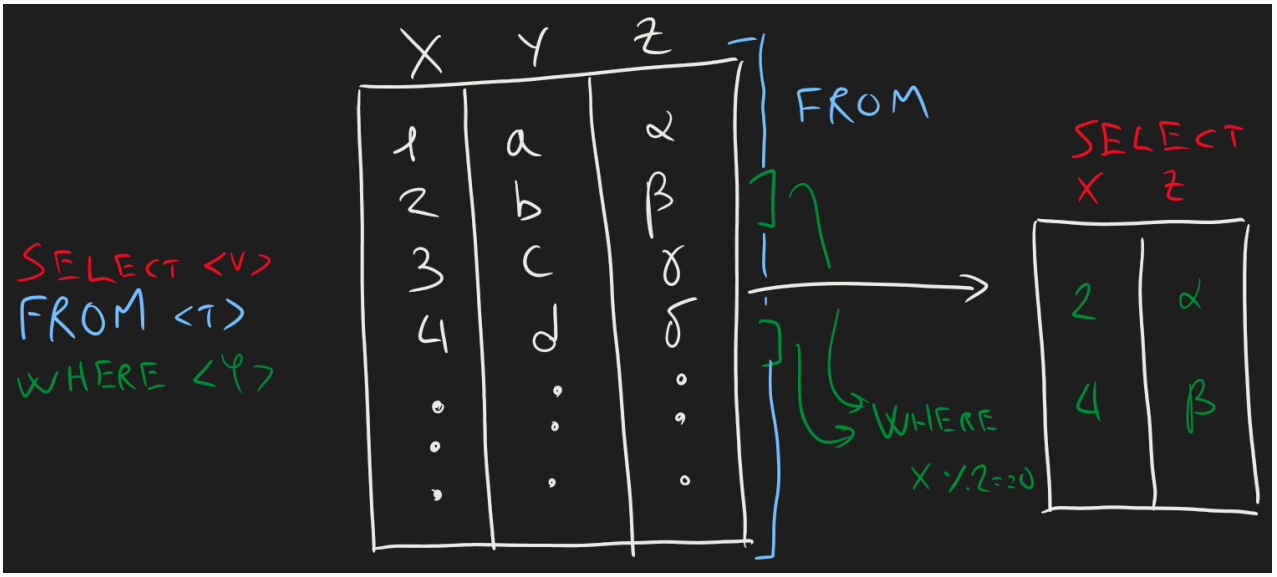
- Introduction to SQL

1. Anatomy of a Query

Querying is when we attempt to gain information from one (or more) tables, in form of another table. The basic anatomy of a query is as follows

```SQL
SELECT <columns>
FROM <table1>, ..., <tablen>
WHERE <conditions>
```

- The `SELECT` statement allows us to select the variables of our query.
- The `FROM` statement specifies the *source* of the query.
- The `WHERE` statement allows us to specify conditions to select certain instances (rows).



X

2. Details of the Query Anatomy

We can add more details to each statement.

- `SELECT`:
  - To select all variables, instead of specifying every column, we can just put the wildcard `*`.
  - To rename columns in the resulting queries, we can use the `AS` alias operator after the selection
  - To change data type, we can either do *implicit casting* (like in the *C* language) or *explicit casting*. To do explicit casting we have `cast(<VAR> AS <DTYPE>)`.
  - As we have the *bags approach*, we can have duplicates. To remove them, we can specify the `DISTINCT` keyword before specifying the selected columns.
- `FROM`
  - Analogous to before, we can add aliases after specifying the table with `AS`. Note that the `AS` keyword is optional
- `WHERE`
  - It can use several operators, such as:
  - Boolean operators: `AND`, `OR` and `NOT`
  - Comparison operators: `=`, `<>`, `<`, `>`, `<=`, `>=`
  - String matcher: `LIKE 'PATTERN'`
  - Wildcard `_` is any character (or more) replaced in its stead
  - Wildcard `%` marks the beginning or end of pattern. If placed twice, ir specifies in any position

- Null checker: `IS NULL`

Additionally, we can add the `ORDER BY <attributes> <ASC/DESC>` to sort the resultant query.

OBSERVATION. The `ORDER BY` statement does not require the specified attribute to be in the `SELECT` statement; it can be done as long as it is present in the table.

X

Joins in SQL

Joins in SQL

X

Joins in SQL: cross product, self joins, set operations. Preserving (or removing) uniqueness from joins. Explicit joins.

X

---

0. Voci correlate

- Introduction to SQL
- Data Integration

1. Cartesian Product in SQL.

Q. We want to select and combine information together. Are simple queries enough?
A. No, we need to integrate data.

A first form of *joins* is the *cartesian product*, where we simply take the cartesian product of our variables.

We have two syntaxes to define a cartesian product: implicit and explicit form.

SQL

```
-- Implicit form
FROM <tab1>, ..., <tabn>

-- Explicit form
FROM <tab1> CROSS JOIN <tab2> ... CROSS JOIN <tabn>
```

Note that the *cartesian product* gives us all of the possible combinations between instances; in fact our resulting query would be of size m^n. This implies that we would have to *select rows*, where their attribute equate in a certain way. For example, if a table references to another as a foreign key, we need to equate the identifiers.

OBSERVATION. What if one of the sets (or bags) involved is empty ($\emptyset$)? Simply we treat it as a cartesian product of the remaining sets. However, if the conditions imposed has anything to do with the empty set, it will be always false.

OBSERVATION. Is it possible to do a self-join? Theoretically, yes. Moreover, it is also a *powerful tool* to compare *duplicate values*.

---

2. Set Operations

Suppose we have *two tables*, which can be both resulting from a query. Suppose they have the same variables; can we treat them as *a set of n-tuples* and apply the set operators ($\cup$, $\cap$, $\setminus$)? Of course we can. In SQL the syntax is as follows

```SQL
<table1>
UNION / INTERSECT / EXCEPT
<table2>
```

Note that here we're no longer in the field of *extended relational algebra*, as we're treating everything as sets. To reintroduce duplicates, we can add the `ALL` keyword after our set operator.

---

3. Explicit Joins

Having always to deal with *cartesian products* can be quite messy sometimes. So, as some sort of *"syntactic sugar"*, we have the *explicit joins*. In particular, they replicate specific type of data merging and integration (Data Integration). The syntax is as follows:

```SQL
FROM <col1> <TYPE> JOIN <col2>
    ON <condition>
```

Note that in this case it is *obligatory* to specify a common column.

Ternary Logic

Ternary Logic

Ternary logic (SQL): heuristic meaning of values, definition of `AND`, `OR` and `NOT` for ternary logic.

---

0. Voci correlate

- Introduction to SQL

1. Ternary Logic

OBSERVATION. In *SQL*, empty values are admissible if we do not have any particular constraints. So what happens if we try to apply logical conditions on *empty values*? We will see how it works with *ternary logic*.

#Definizione

> Definizione (three-valued boolean variable).
>
> x is a three-valued boolean variable iff it assumes one of the values:
>
> - $x = 0$: It means that x is *FALSE*
> - $x = 0.5$: It means that x is *UNKNOWN*, as it might be the result of querying a missing value
> - $x = 1$: It means that x is *TRUE*

Q. How do we handle logical operators on the empty values?
A. As we *only select* columns with *satisfied expressions*, we can define the following operators in the mathematical sense.

#Definizione

> Definizione (logical operators).
>
> Let x, y be three-valued boolean variables.
>
> Then we define the following operations:
>
> $$x \wedge y := \min(x, y)$$
> $$x \vee y := \max(x, y)$$
> $$\neg x := 1 - x$$
>
> This makes sense for unknown values: assume $x = 0.5$, which would mean:
>
> - $\forall y, x \wedge y \neq 1$; it will have always a *"negative result"*; we either can make sure it's false, with $y = 0$ or we still cannot conclude anything with $y = 1$.
> - $\exists y, x \vee y = 1$; the only way to make sure our statement is true, is to make the *non-unknown* true.
> - $\neg x = x$; negating unknown still brings to unknown

Let us do a simple example
#Esercizio

> Esercizio (exercise).
>
> Determine x, y, z such that the following expression is satisfied (i.e. when evaluted with the variables it returns 1):

$$\varphi(x, y, z) = x \wedge (y \vee \neg z)$$

Reasoning with traditional boolean logic, we have that a possible solution is $(x, y, z) = (1, 1, 1)$. However, with three-valued logic, we can also admit z to be unknown: in fact $\varphi(1, 1, z) = 1, \forall z$.

X

Subqueries in SQL

Subqueries in SQL

X

Preliminary observations: queries are always relations. Cases for placing subqueries: in the `WHERE` *clause, in the* `FROM` *clause and in the* `SELECT` *clause.*

X

---

0. Voci correlate

- Basic Queries in SQL

1. Preliminary Observation for Subqueries

OBSERVATION. The result of a *query* in *SQL* is always *a table*, *i.e.* a *relation*. This means I can apply further *queries* on *queries*, having some sort of composition:

$$Q_1(Q_2(\dots(Q_n(T))))$$

The question is, *where* can I do these subqueries? We can do these everywhere in our *query anatomy*, each part has its own purpose.

X

---

2. Subqueries in SQL

2.1. Subqueries in the `WHERE` clause

A first place where we can make *subqueries* is the `WHERE` clause; as we have to do with *boolean conditions*, we need to use the following operators:

- `<column> IN <subquery>`: is something in a certain relation? `
- `EXISTS <subquery>`: do we have elements in a certain relations (subqueries)?
- `<column> <condition> ALL <subquery>`: does a certain numerical value satisfy a statement for each element in our subquery?
- `<column> <condition> ANY <subquery>`: same as before, but we need only one satisfied statement to let the element be included in our total query.

OBSERVATION. The `IN` is a *set operator*, so we do not have any duplicates in our resultant subquery.

2.2. Subqueries in the `FROM` clause

Another place where we can make subqueries is the `FROM` clause; this allows us to simply make further queries on our query.

2.3. Subqueries in the `SELECT` clause

If I want to select *correlated variables* in a table, we can make subqueries in the `SELECT` clause.

X

Aggregations in SQL

Aggregations in SQL

X

Aggregations in SQL: preliminary idea, anatomy of a query with aggregation, main aggregators in SQL.

X

---

0. Voci correlate

- Basic Queries in SQL

1. Anatomy of an Aggregation in SQL

Suppose we have some *data*, and we want to *aggregate* them by a certain group; then as we have the grouped data, we can extract more information. In SQL, this concept is known as *aggregation*.

An aggregation is always included in a *query*, so the anatomy of a query with aggregations is as follows:

```SQL
SELECT <grouped attributes>, <aggregators>
FROM <...>
WHERE <...>
GROUP BY <grouped attributes>
HAVING <grouped condition>
```

- **NOTE.** Here we can only put the *grouped attributed* in the `SELECT` clause.

X

---

2. Aggregator Functions

The *aggregator functions* are applied over *each group*.
The common ones are as follow:

- `COUNT(*)` or `COUNT(<...>)` return the number of non-null tuples (or values of certain columns). To apply uniqueness, use `DISTINCT` keyword.

- `SUM(<...>)`, `AVG(<...>)`, `MIN(<...>)` and `MAX(<...>)` return a statistic of numerical values

---

X

Database Modifications in SQL

Database Modifications in SQL

X

Database modifications in SQL. Inserting, removing and modifying data.

X

0. Voci correlate

- Introduction to SQL

1. Inserting Data

There are mainly two ways to *insert data* into a table:

1. Specify tuples by hand

```SQL
INSERT INTO <my_table(<atr_1>,..., <atr_n>)>
VALUES (<val_1>, ..., <val_n>), (<vâl_1>, ..., <vâl_n>), ...
```

- Note that `(<atr_1>,..., <atr_n>)` can be omitted, but every attribute must have a value.

2. Insert from a subquery

```SQL
INSERT INTO <my_table> (<subquery>)

-- Specific Example: copy tables
INSERT INTO <my_table>
    SELECT * FROM <other_table> WHERE 1=1
```

X

2. Delete Data

Notice that we're deleting *instances of data*, so we always delete the *singular rows*.

We mainly have two ways:

1. Delete by specify a criterion in the same table

```SQL
DELETE FROM <my_table>
WHERE (<condition>)
```

2. Delete by specifying a criterion which involves *other tables*

```SQL
DELETE <my_table>
FROM <tables>
    WHERE (<condition>)
```

- This is very flexible, as our condition can involve *other subqueries mentioning other tables*.

Example. Delete all applicants which have an account in 'Setubal'

```SQL
DELETE APPLICANT
    FROM ACCOUNT a, BRANCH b
    WHERE
        a.BRANCH = b.NAME AND
        b.CITY = 'Sebutal' AND
        APPLICANT.ACCOUNT = a.ACCOUNT
            -- Remember to join the deleting table with the querying one!
Or else I will delete everything...
```

X

---

3. Update Data

Now I want to modify attributes.

1. Simple way

```SQL
UPDATE <my_table>
SET <atr1=..., ..., atr_n=...>
WHERE <condition>
```

2. Complex way (involving other tables)

```SQL
UPDATE <my_table>
SET <atr1=..., ..., atr_n=...>
FROM <tables>
WHERE <condition>
```

Note: If we're updating conditions with something that resembles IF-ELSE statements, we can use the following syntax:

```SQL
UPDATE <my_table>
    SET
        atr_1 = ...(CASE WHEN <cond1> THEN <val1> ELSE <val2> END)
```

X

Integrity Restrictions in SQL

Integrity Restrictions in SQL

X

Integrity restrictions in SQL: motivation. Implementations of integrity restrictions: `NOT NULL`, `UNIQUE` *and* `CHECK <condition>`.

X

---

0. Voci correlate

- Database Definition in SQL

1. Definition and Motivation of Restriction

In *SQL*, a *restriction* is an object which ensures that *changes* to the *database* do not result in *data inconsistency*. In other words, they are to prevent *accidental* database damage.

Example. In a *banks database*, the name of a client cannot be NULL; or two branches cannot have the same name; or balance must be always greater than zero; or foreign keys must exist in the pointing table

---

X

---

2. Implementation of a Restriction

We can define an *integrity restriction* with two main methods:

- In `CREATE TABLE` statements (Database Definition in SQL)
- Using the `ALTER <TABLE> ADD CONSTRAINT` DML statement

We have three main types of restrictions:

- `NOT NULL` makes a value *mandatory*
  - Example: Primary keys
- `UNIQUE` makes a value an *alternate key*
- `CHECK <condition>` allows to be a value to be modified if and only if the condition is satisfied. This generalizes the *concept of foreign key*; foreign keys check for existence in another table, while `CHECK` constraints determine the valid values of a logical condition.
  - Note: the condition will apply in all relation tuples
  - Note: the deep difference from foreign key is that `CHECK` acts only on the same table it is specified upon
- `PRIMARY KEY` makes value a *primary key*

Let us see the anatomy of an integrity restriction:

```
-- Case 1: DDL;
CREATE TABLE T(
    V1 int,
    ...,
    CONSTRAINT my_constraint
        CHECK (...)
)

-- Case 2: DML;
ALTER TABLE T
    ADD CONSTRAINT other_constraint CHECK (...);
```

X

Domain Restrictions in SQL

Domain Restrictions in SQL

X

Domain restrictions in SQL: implementing default values and custom domains.

X

0. Voci correlate

- Database Definition in SQL

1. Domain Restrictions in SQL

Domain restrictions is the most simple form of *integrity constrain*; the core idea is to *test the values entered in the database* and in the *queries* and ensure that such values make sense.

We have two methods to implement *domain restrictions*.

1.1. Default Values

Default values allow the assignment of a *default value* to an attribute, when it is not specified. For example, when I insert a tuple without specifying a value for a certain attribute, its default value gets inserted.

To implement this in SQL, we will simply use the keyword `DEFAULT` in our DDL language. Alternatively, we can specify it as a *costraint* (Integrity Restrictions in SQL).

```
-- DDL;
CREATE TABLE T(
    ...,
    my_var INT DEFAULT 0,
    ...
    other_var INT,
    CONSTRAINT default_other_var other_var DEFAULT 0,
    ...
);

-- DML;
MODIFY TABLE T
    ADD CONSTRAINT defaulter <VAR> DEFAULT 0;
```

1.2. Custom Datatypes

We have seen the default datatypes with the *DDL* part of SQL (Database Definition in SQL). We can create *"custom domains"* of those existing datatypes by applying some restrictions (e.g. `NOT NULL` and default values assigned)

To implement a custom domain we can use the `CREATE` statement:

```
CREATE TYPE <my_type> FROM <dtype> <restrictions>;

-- Example
CREATE TYPE SSN FROM varchar(11) NOT NULL;
```

X

Referential Integrity in SQL

Referential Integrity in SQL

X

Referential integrity in SQL. Cases for referential integrity: inserting, deleting, updating data. Formal procedures.

X

---

0. Voci correlate

- Database Modifications in SQL

1. Definition of Referential Integrity

Referential integrity is a when a *foreign key* matches the table *primary key* to which it points.

Formally:

- Let R_1, R_2 be the relations with attribute sets K_1, K_2 as their primary keys (respectively)
- Let α be the of attributes of R_2 a foreign key referencing K_1
- Referential integrity is when the following condition is held:

$$\forall t_2 \in R_2, \exists t_1 \in R_1 : t_1(K_1) = t_2(\alpha)$$

  Let us see how such condition is held.

X

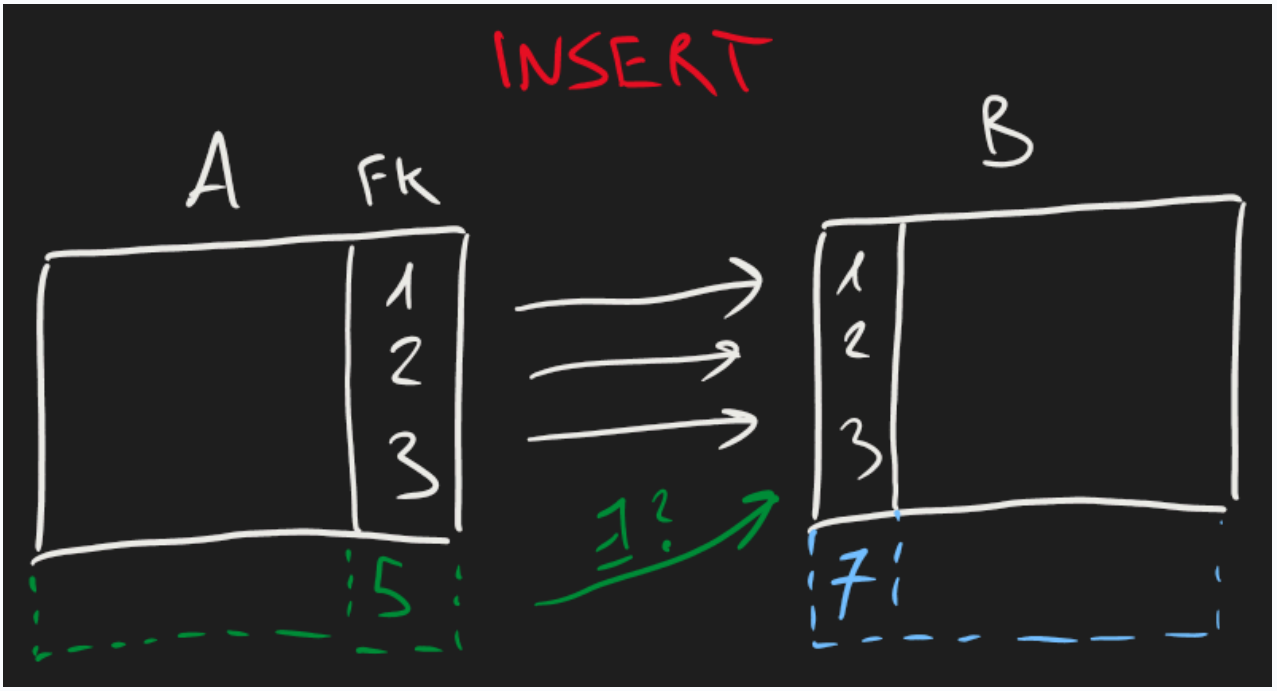
---

2. Ways to ensure Referential Integrity

2.1. Insertion

Let A, B be tables, where A holds a foreign key pointing to B. Then:

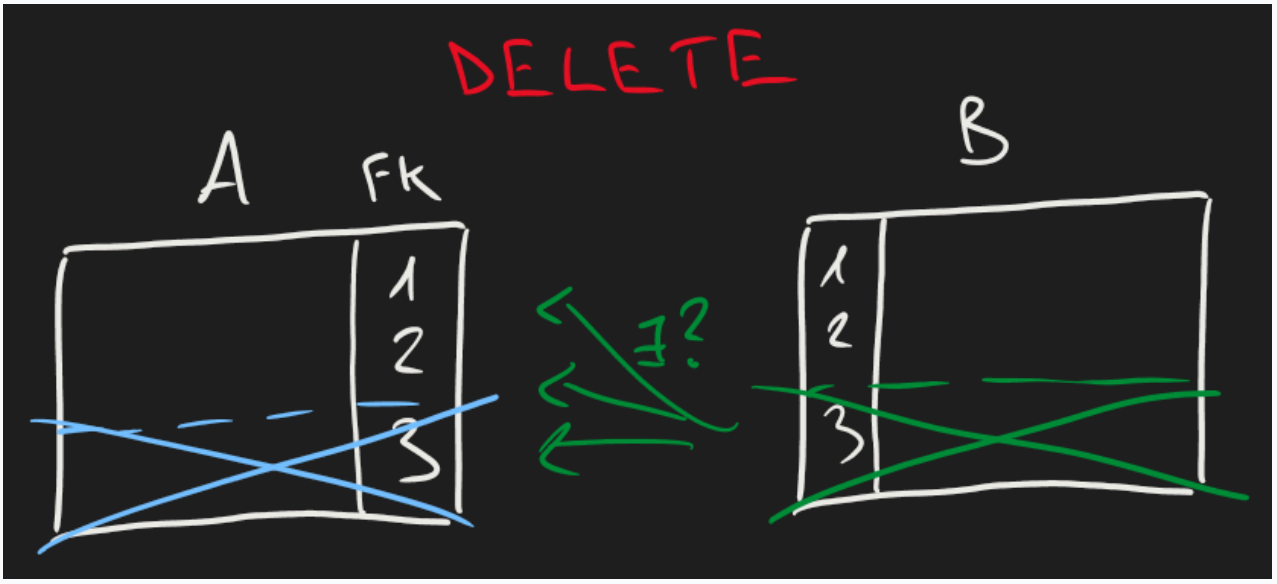
- If we add a *new row* to A, then we have to check for *existence* of A's foreign key in B
  - If it does not exist, rollback the transaction
- If we add a new row to B, then we have to do nothing.



2.2. Delete

Let A, B be tables, where A holds a foreign key pointing to B. Then:

- If we *delete a row* from A, then we have to do nothing.
- If we *delete a row* from B, then we have to check if there are *foreign keys* in A which are pointing to the deleted pointer key in B
    - If there are such rows, we have three ways to resolve it:
        - Rollback the transaction with an *error* (default behaviour)
        - Delete the rows in A as well (cascade)
        - Set `NULL` or default values

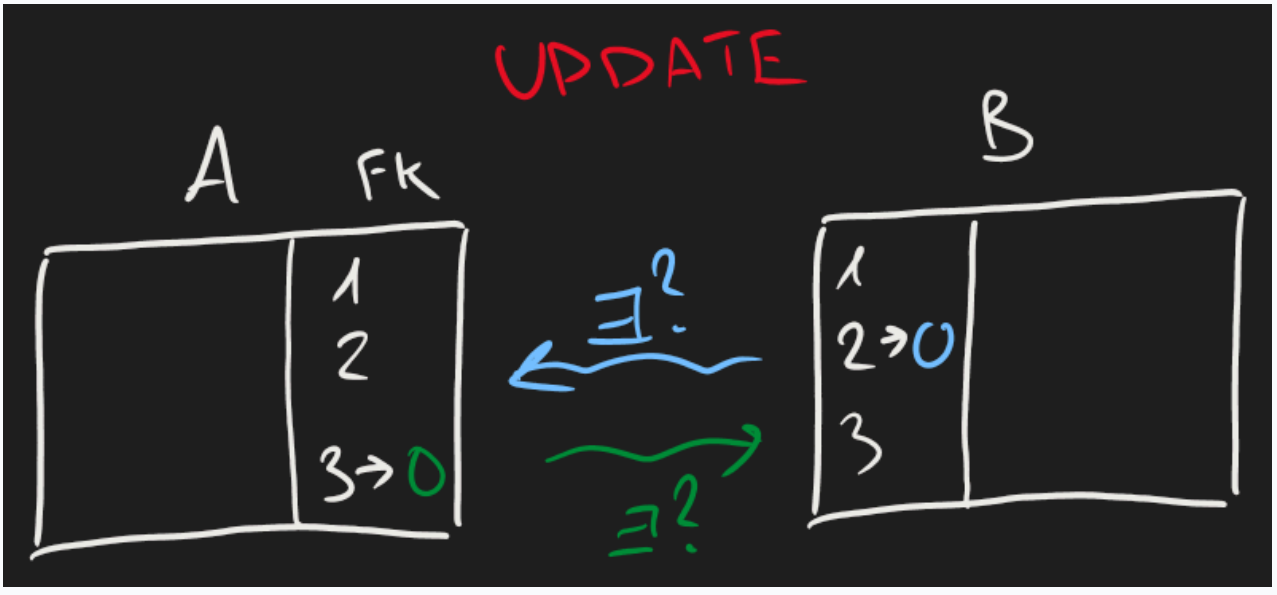


2.3. Update

Observation. In SQL, an *update* is always *deleting old rows* and *adding new rows* combined

Let A, B be tables, where A holds a foreign key pointing to B. Considering the previous row, we simply have the *two referential integrity checks* combined.

- However, here `CASCADE` behavior is defined differently: instead of deleting, we replace.



2.4. SQL Implementation

Implementing a `CASCADE` or `SET` behavior is simple. We use the `ON` keyword.

```
ALTER TABLE T
    ON UPDATE CASCADE

ALTER TABLE T
    ON UPDATE SET <...> (NULL/DEFAULT)

ALTER TABLE T
    ON UPDATE|DELETE <...>
```

X

---

3. Terminology

Definition. When we attempt to make some *modifications on a database*, affecting one table or more, the chain of modifications is said to be a *transaction*.

- Usually *referential integrity* is done *at the end* of a transaction

Definition. *Rollback* is when we *abort* a transaction. Usually it is done when *violations of a restrictions* cannot be solved by *cascades*.

X

Stored Procedures and Functions in SQL

Stored Procedures and Functions in SQL

X

Persistent Stored Modules in SQL. Main idea: having persistent storage modules. Syntax for stored procedure and function in SQL. Statements useful for PSMs: return values, declare and assign variables, define code blocks and basic control flows in SQL.

X

---

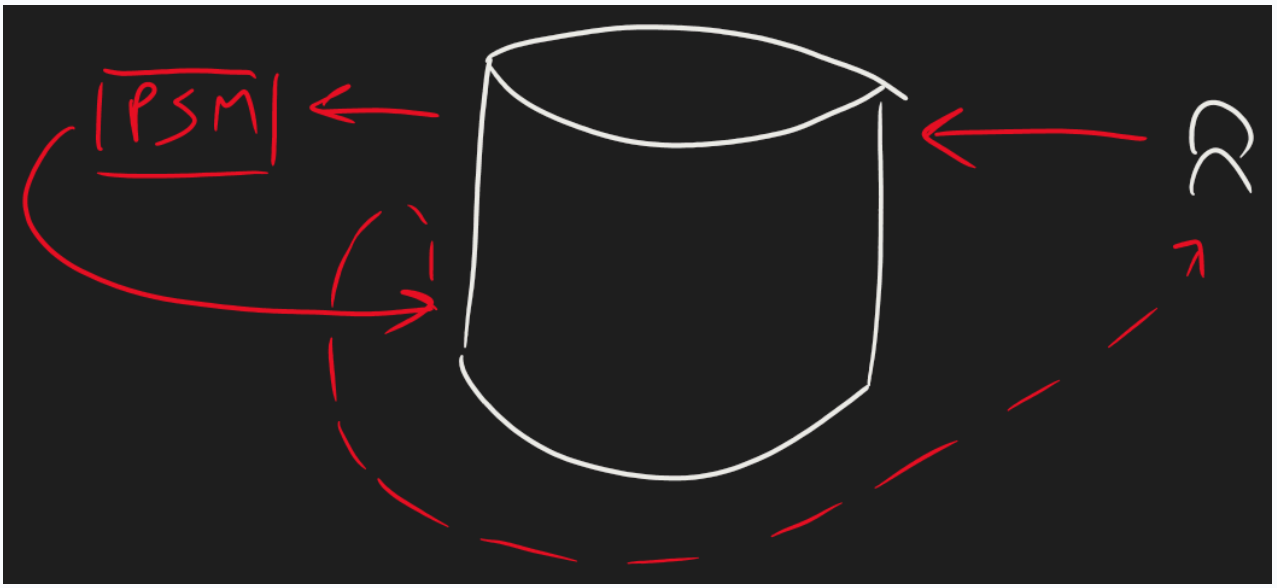
0. Voci correlate

- Introduction to SQL

1. Idea of Persistent Store Modules

IDEA. We want *code* stored in the databases implementing *DML SQL* statements, such as queries, database modifications...
We want it to be able to do either of the following:

- Give us an *output*: maybe we want to know something about it
- Modify the *database*: sometimes basic database design is not sufficient enough to force certain dependencies (in particular *functional dependencies*)
- Accept an *input* of parameters: we might want to seek for specific items in our database



X

2. Procedures and Functions

In SQL, we mainly have *three types* of *Persistent Stored Modules*. For now, we will see the simple ones, which are *stored procedures* and *functions*.

```
CREATE PROCEDURE <my_procedure>.<my_schema> (<parameters>)
    @input_1 <dtype> = defvalue_1,
    ...
AS
    <local_variables>;
    <instructions>;
GO;

CREATE FUNCTION <my_function> (<parameters>) RETURNS <output_dtype>
    @input_1 <dtype> = defvalue_1,
    ...
AS
    <local_variables>;
    <instructions>;
GO;
```

- Note that with functions, we can have that `RETURNS` datatype can be also a *table* (so not just a scalar value!). To do it, you just have to specify the name of an existing table.
- `<my_schema>` is optional
- To delete or modify functions, use the statements `DELETE`, `ALTER`

    - We can combine definition and modification by using `CREATE OR ALTER`. Particularly useful when we are developing the trigger, as we do not have to manually delete the PSM.
- We can call a procedure by using the `EXEC` (or `EXECUTE`) statement, accompanied with a `GO`.
- We can omit the parameters of the procedure/function
- We can omit default values
- When we call a procedure/function with parameters, we have to ways:
    - Implicit parametrization: `EXECUTE my_procedure var1, var2, ...;`
    - Explicit parametrization: `EXECUTE my_procedure @input_1 = var_1, ...`

X

---

3. Imperative SQL

When implementing *PSMs* in SQL, it might be useful to know about *imperative statements* in SQL (i.e. statements which are common in *imperative paradigm languages*)

- `RETURN <expression>` assigns the *value returned* by a function and ends the function run
- `DECLARE <name> <dtype>` is used to *declare local variables*
    - Note that imperative SQL is strongly typed
    - Moreover every variable must start with the `@` symbol
- `SET <variable> = <espression>` is used to *assign declared local variables*
- `BEGIN ... END` is used to define code blocks
    - Equivalent to curly brackets `{, }` in C
- `IF <expr> <...> ELSE <...>` defines an *if-else* flow
- `WHILE <expr>` defines a *while loop* on `expr`
    - `BREAK` allows us to immediately exit from the innermost `WHILE` loop
    - `CONTINUE` allows us to restart the loop

Observation. Every variable and code is stored in the *database*, not in any sort of *virtual memory*. This is what makes *SQL* very efficient for databases.

X

Triggers in SQL

Triggers in SQL

X

Triggers in SQL. Three types of triggers in SQL Server. Tipologies of DML triggers: "instead of" and "after" triggers. Graphical idea. Observation: stored procedures in triggers. Recursive triggers in SQL.

X

---

0. Voci correlate

- Stored Procedures and Functions in SQL

1. Definition of Triggers

Triggers are a *special type of storage procedure* (PSM) in SQL. A *trigger* is an *instruction executed automatically as a result of an event*, e.g. a modification to the database. An *"architecture"* of a trigger must contain the following:

- Conditions under which the trigger is *"fired"*
- Actions which the trigger does

In *SQL Server* (SMSS) there exist 3 types of triggers: DML, DDL and Logon. We will only see the *DML triggers* in this course.

DML Triggers are fired automatically as a response to the following *DML* events (Database Modifications in SQL):

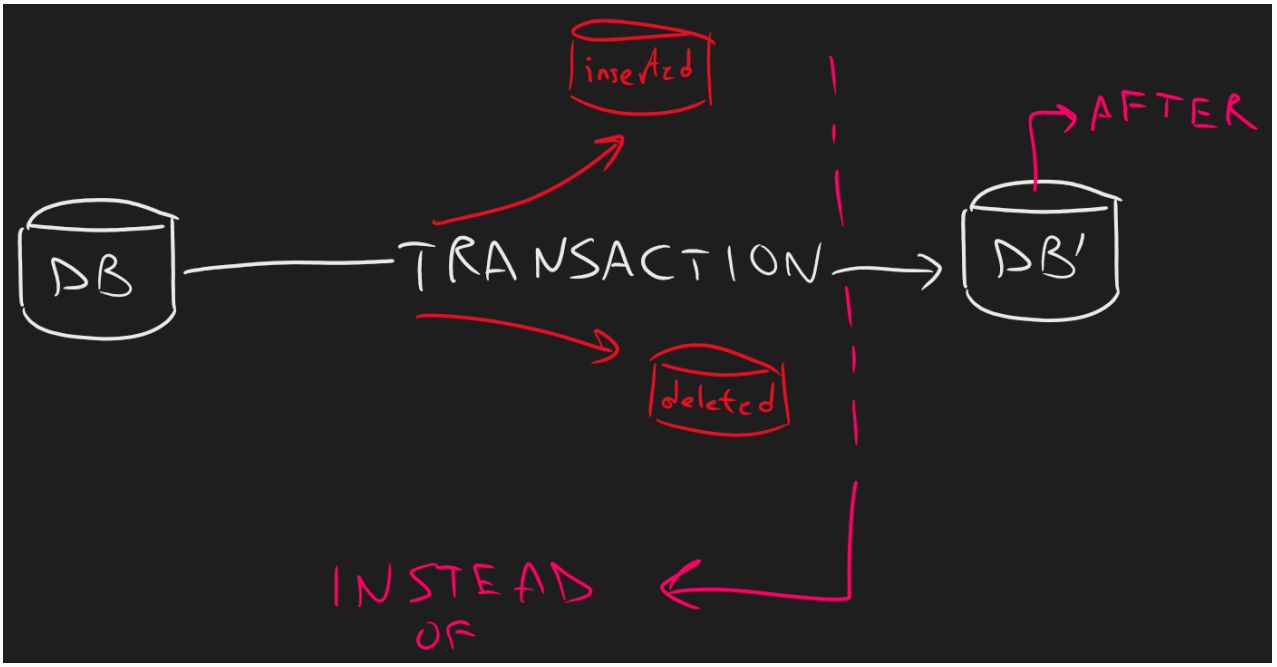
- Data insertion
- Data update
- Data deletion

Moreover, we can classify each *DML trigger* into two subtypes:

- *"After"* triggers occur after the *DML event*: in other words, the *"damage"* has already been done.
- *"Instead of"* triggers substitute the *DML event* itself (so it occurs before the event)
  - Note that this implies we cannot forget to *"re-implement" the intent of the modification*!
  - **Corollary**: we can make *"immune tables"* by creating empty triggers

In any case, each time we run a *DML* trigger we get two additional records:

- *Inserted*: keeps track of inserted records (or new values in case of update)
- *Deleted*: keeps track of deleted records (or old values in case of update)



X

2. Syntax of a DML Trigger

Let us present the skeleton of a DML trigger.

```
CREATE OR ALTER TRIGGER <my_trigger>
    <INSTEAD OF/AFTER> <operation>
    ON <my_table>
AS
    <instructions>
GO;
```

Observation. We can use *stored procedures* on triggers. Useful for certain functional dependencies, such as generating repeating occurrences of an appointment.

X

3. Recursive Triggers

Observation. Just like we have *recursive functions* with imperative programming, we have *recursive triggers* with SQL. In particular, INSTEAD OF-type triggers are predefined to be recursive in T-SQL; whereas AFTER-type triggers need to have its recursivity manually enabled.

```
ALTER DATABASE <my_database>
SET RECURSIVE_TRIGGERS on;
GO
```

Example. Implement a trigger on the relation $R(A, B)$ where $A, B \in \mathbb{Z}$, such that the following functional dependency is true:

$$(a, b) \in R \land a * b > 10 \implies (a - 1, b + 1) \in R$$

This can be implemented by the following recursive trigger:

```
CREATE TRIGGER rec_tr
    ON R
    AFTER INSERT
AS
    BEGIN
        INSERT INTO R
        SELECT a-1, b+1
        FROM inserted i
        WHERE a*b>10;
    END
GO;
```

X

Miscellaneous SQL Tips

Miscellaneous SQL Tips

X

Miscellaneous tips for programming in SQL. Using temporary tables, `SELECT INTO` statement and adding IDs to tables.

X

0. Voci correlate

- Introduction to SQL

1. Temporary Tables in SQL

In *SQL*, it is possible to use *temporary tables*. Basically, they are tables which are automatically deleted at the end of the execution of a transaction (or trigger, stored procedure, et cetera...).
To create one, it is just necessary to place the prefix #.

Example.

```
CREATE TABLE #tmp_table
(
    var_1 varchar(100),
    ...
);
```

X

2. Creating Tables from Queries

It is also possible to *create tables* from queries; to do it we can use the `SELECT INTO` statement.

Example.

```
SELECT *
    INTO #tmp_table
    FROM my_table
```

X

3. Adding IDs

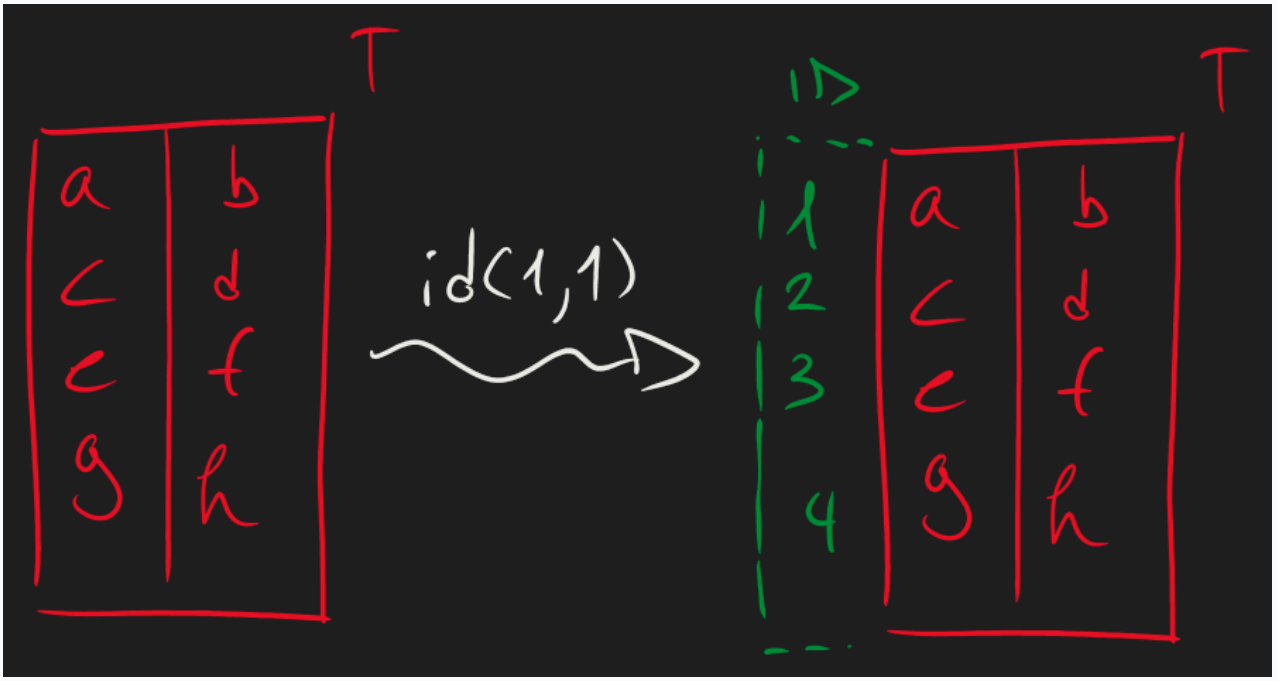
If you want to compare two tables whose primary keys are altered, you can do it by adding IDs to each table. To do it, there is the predefined function `identity(n, m)` where n is the starting number and m is

the increment.

Example.

```
SELECT * INTO #tmp_i FROM INSERTED
SELECT * INTO #tmp_d FROM DELETED

ALTER TABLE #tmp_i ADD id INT IDENTITY(1,1);
ALTER TABLE #tmp_d ADD id INT IDENTITY(1,1);
```



X

Views in SQL

Views in SQL

X

Views in SQL: definition of a view, types of a SQL view. Syntax for creating views. The `WITH` clause. Examples.

X

0. Voci correlate

- Introduction to SQL
- Database Definition in SQL
- Indexes in SQL

1. Definition and Syntax of SQL View

Definition. A *view* is a *relationship* defined in terms of base tables and (eventually) other views. In other words, it is a *virtual table* or a *logic table* stored in the memory. You can treat a view as if it were a table, such as issuing queries on it.

Definition. A view is *virtual* if it not stored in the database; it is *materialized* if they are actually stored in the database.

T-SQL supports *virtual* views by default; to create a *materialized view*, it is necessary to use indexes (Indexes in SQL).

The syntax to create a view is as follows:

```
CREATE [materialized] VIEW <my_view> AS  <query>;
```

---

2. Local Views

You can also make *local views* in SQL, i.e. views to be defined locally to a query (and they disappear in the following queries)

Example.

```
WITH loc_view(v_1, v_2, ...) AS <my_query>
<another_query>
```

- `my_query` must return variables `v_1, v_2, ...`

---

3. Relations between Tables and Views

It is immediate that there is a "$\implies$" type of relationship between tables and views; the state of a table implies the state of the view. If we update the original table, then its view will get updated.

However, what happens to the "$\impliedby$" situation? What happens if I attempt to *update a view*? What would happen to the table?

Proposition. A view is *updatable* (meaning it can be updated without any problems) if and only if:

- In the `FROM` clause there is *only one relation*
- The `SELECT` clause contains *only relation attributes*; so we do not have any kind of aggregations or no uniqueness
- Any unspecified attribute from the table can be `NULL`
- The query has no `GROUP BY`

The reason that these type of views are *updatable* is that their update has only one consequence on one table. In the other cases, we could modify the table in a lot of manners!

Q. Suppose a *view* is *non-updatable*. Can we still make it updatable?
A. Yes, with *INSTEAD OF-type triggers*. In this way, we can arbitrarily define a consequence on the base tables.

Example. Suppose that we have a view, from a books database, which gives us information about books' prices on discount (minimum, maximum). Precisely, given as

```
VIEW v AS
    SELECT b.gategory, min(b.price), max(b.price)
    FROM BOOKS b
    WHERE b.promoted = 1
    GROUP BY b.category
```

If we want to delete an entry from a view, we have to set its `promoted` value to 0 from its original database. So we would implement the following trigger:

```
TRIGGER t
    as INSTEAD OF DELETE on v
AS
    BEGIN
        UPDATE BOOKS SET PROMOTED=0
            FROM deleted d
            WHERE d.category = v.category
    END
END;
```

X

Concurrence Problems in Databases

Concurrence Problems in Databases

X

Concurrence problems in databases, by levels: attribute-level, tuple-level and table-level.

X

0. Voci correlate

- Database Definition in SQL
- Database Modifications in SQL
- Concorrenza e Parallelismo
- Segnali

1. Introduction to the problem

Up until now, we have used a *database* as a *singular user*. However, reality is way different: multiple users access the same data concurrently. Meaning that modifications, queries and such operations can be done at the same time.

This reveals to be problematic in every *"layer"*, as we will see with the following examples.

1.1. Attribute-Level Concurrency

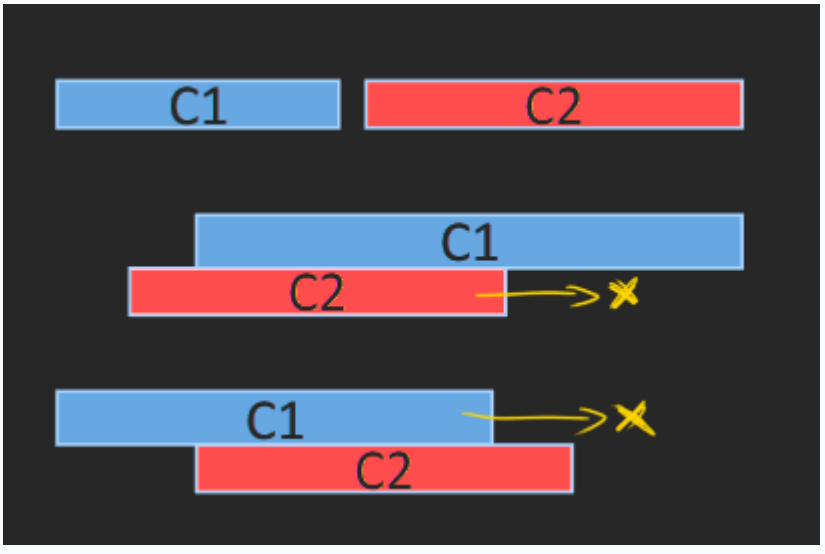
Assume the following situation where we have *two clients* accessing the same attribute from the same record:

```sql
-- Client 1: Add balance
UPDATE ACCOUNT SET BALANCE = BALANCE + 3000
WHERE ACCOUNT_NUMBER = 1

--  Client 2: Remove balance
UPDATE ACCOUNT SET BALANCE = BALANCE - 1500
WHERE ACCOUNT_NUMBER = 1
```

Assuming that our initial balance was 10.000€, what is the final result? The thing is, it depends on which "commands" were processed first!

- Ideally, they would be executed one after another, giving us a final result of 11.500€
- *However*, some transactions might be *"lost"* as one command can be ran while the other was still running, causing one of the commands to be *"ignored"*!
  - So, maybe it could be 13.000€ or 8.500€
  The crucial problem is similar to the case of *lost increment* (Segnali > ^45417b), problematic topic in operating systems with multithreaded programming.



1.2. Tuple-Level Concurrency

The case is similar as above, but instead of dealing with *single entries*, we're dealing with an entire row.
Assume that a row with id 1 assumes the following entries: `STREET -> NULL, CITY -> Lisboa`
If we have the following situation with the two clients:

```sql
-- C1
UPDATE CLIENT SET STREET = 'Rua da República'
WHERE ID = 1

-- C2
UPDATE CLIENT SET CITY = 'Porto'
WHERE ID = 1
```

Then we can have the following outcomes:

- `STREET -> Rua da República` and `CITY -> Porto` (Both changed)
- `STREET -> Rua da República` and `CITY -> Lisboa` (City did not change)
- `STREET -> NULL` and `CITY -> Porto` (Street did not change)

1.3. Table-Level Concurrency

Assume we have the following case, with loans and accounts:

```sql
-- C1
UPDATE LOAN SET AMOUNT = AMOUNT + BALANCE
FROM ACCOUNT
WHERE LOAN_NUMBER = ACCOUNT_NUMBER AND BALANCE < 0

-- C2
UPDATE ACCOUNT SET BALANCE = BALANCE + 1000
WHERE ACCOUNT_NUMBER = 10
```

In this case, person with account number 10 could have a negative balance prior to the update and positive after C1. This fact has an impact in C1, or viceversa...
This case is even more problematic, as *queries* can have entirely different outcomes.

X

---

2. Possible Solutions

We have understood that *concurrence* is a problem with SQL servers. There are many ways to do it:

- Use *syncronization constructs* already used with Operating Systems (examples: Semaphores, Mutex, et cetera...).
  - Examples: https://www.sqlservercentral.com/articles/mutexes-in-sql, https://www.sqlservercentral.com/articles/implementing-a-t-sql-semaphore
- Use *transactions* (Transactions in SQL)

X

Transactions in SQL

Transactions in SQL

Motivations to transactions. Definition and properties (ACID) of transactions. Types of transactions. T-SQL syntax for transactions.

---

0. Voci correlate

- Concurrence Problems in Databases

1. Motivations for Transactions

In a SQL Database we might have the following problems:

- Concurrency (Concurrence Problems in Databases)
- Risk of Failure; what if the systems crashed midway through the execution? What if something went wrong in a query?
  Transactions are made to *solve* these problems.

---

2. Definition and Properties of Transaction

DEFINITION. (*Transaction*)
A *transaction* is a *sequence of* ≥ 1 *SQL statements* that are treated as if it were just a single operation.

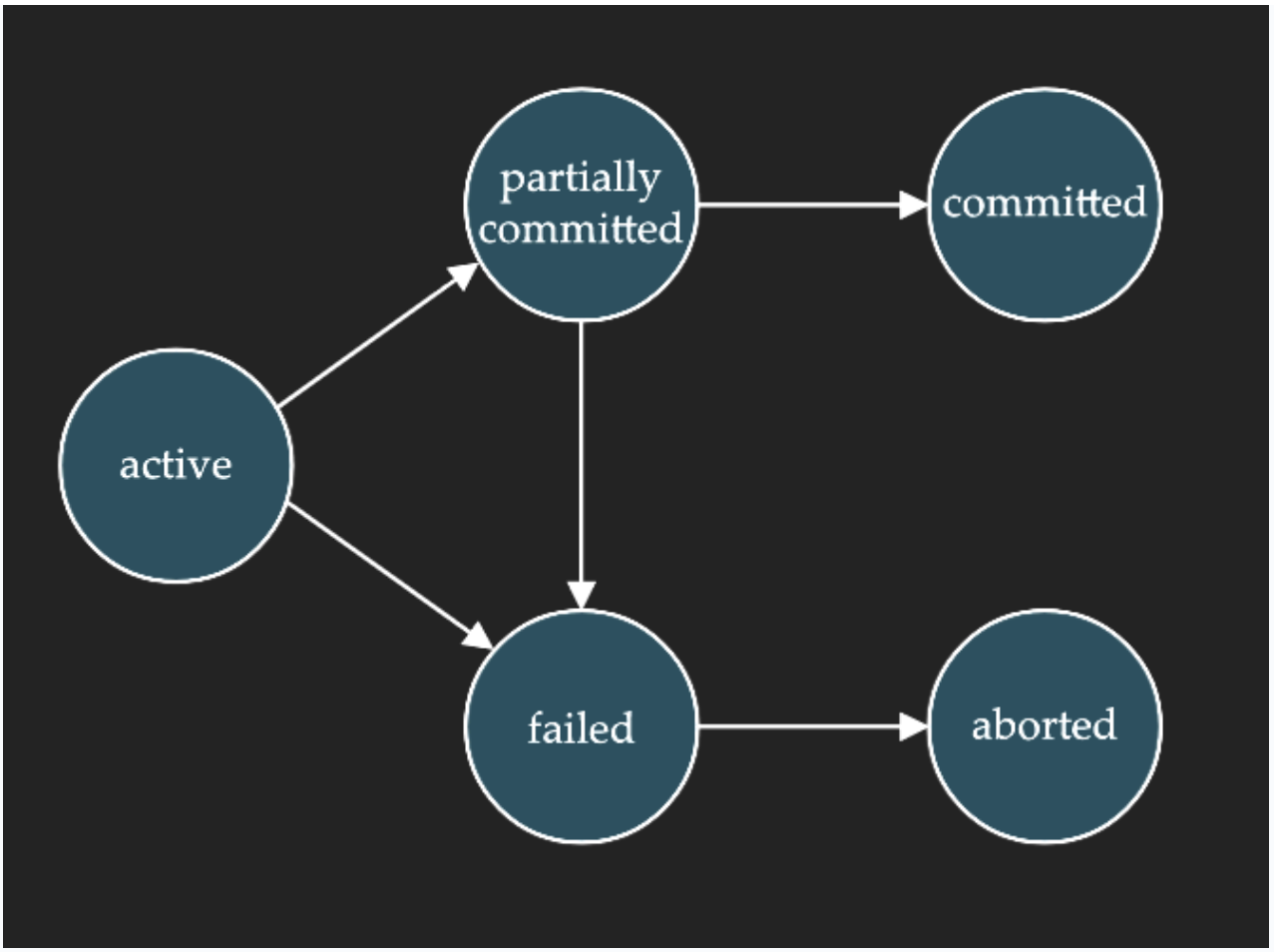
Transactions' properties can vary between database systems, but the following is a common structure for transactions

DEFINITION. (*ACID Database System*)
A database system is said to be *ACID* iff its transactions have the following properties:

- *Atomicity*: every transaction is either executed completely or not at all
- *Consistency*: DB restrictions must be maintained
- *Isolation*: transactions should be executed as if they were isolated, i.e., in a non-competitive environment; so the intermediate results of each transaction are "hidden" from other concurrently executing transactions
- *Durability*: the effect of a transaction is permanent, after it is completely executed

A transaction can have *many states*:

- **Active**: initial state, remains like this during execution
- **Partially committed**: after the last statement has been executed
- **Failed**: transaction cannot continue due to an error. It can either restart or get aborted
- **Aborted**: after *Failed*, the DB returns to its initial state before the transaction had begun
- **Committed**: after *Partially committed*, transaction ends successfully



X

3. Types of Transactions

DEFINITION. A certain item of information in a database is said to be *dirty* iff it was written by a transaction and has *not yet been committed*.

Therefore, a *read*-type statement is said to be *dirty* if its associated read value is *dirty*.

A transaction can have *different types*, depending on how a *dirty read* is handled.

3.1. Read Uncommitted

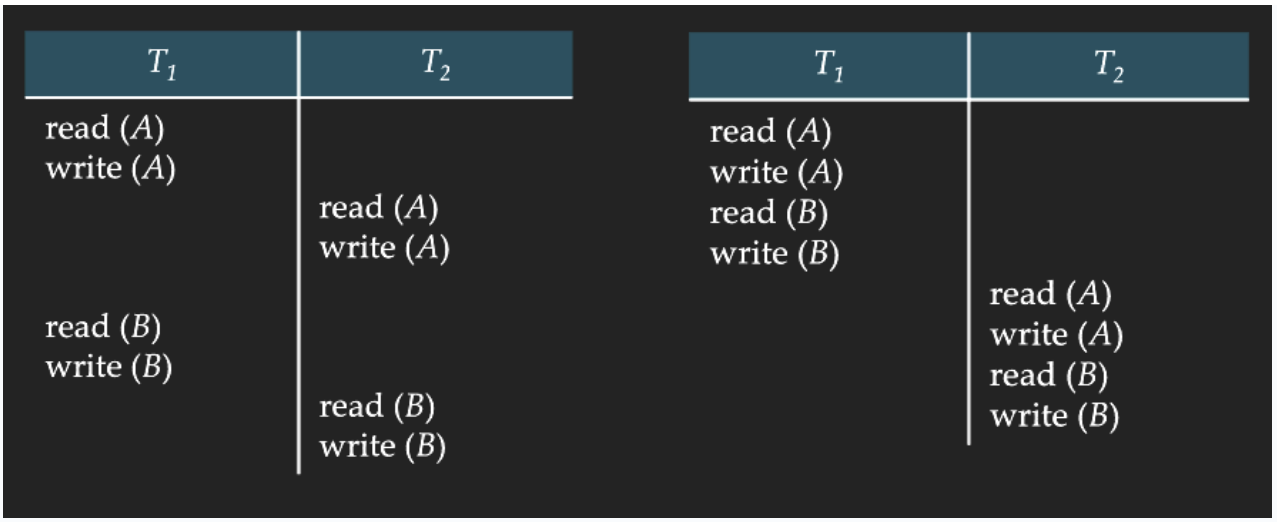
A transaction is said to be *"read uncommitted"* if it allows *dirty reads* to be done.

Q. Is this not bad design, as it ignores the benefits of transactions?
A. Yes, however waiting for commits can be slow and expensive. In a way, we are sacrificing isolation for higher performance

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
<...>
```

3.2. Serializable Scheduling

In some cases, we can reorganize the actions done in transactions to be done *sequentially*, without any conflicts. In this case, a scheduling is said to be *serializable in terms of conflicts*



3.3. Read Committed

A transaction is said to be *"Read Committed"* if it does not allow dirty reads.

This gives us a *higher isolation level* than *read uncommitted*; however, it still does not guarantee *serializability*, as we could still have concurrency conflicts.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

3.3. Repeatable Read

A transaction is said to be *"Repeatable Read"* if it *does not allow dirty reads* and *does not allow changing values with multiple reads*.

This gives us an even higher *isolation level*, but still does not guarantee serializability.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

---

4. T-SQL Syntax for Transactions

Let us see the syntax for *transactions* in *T-SQL*.

1. **Begin and end transaction**

```
-- Commit transaction
BEGIN TRANSACTION <my_transaction>
...
COMMIT TRANSACTION <my_transaction>

-- Rollback transaction
BEGIN TRANSACTION <my_transaction>
...
ROLLBACK TRANSACTION <my_transaction>
```

Observation: In Microsoft's SQL Server Manager Studio, transactions with errors are automatically rolled back. Therefore, there is usually no need to do exception handling.

2. **Error Handling**

```
BEGIN TRY
    <...>
END TRY
BEGIN CATCH
    <...>
END CATCH
```

Obeservation: We can do *interlaced transactions*, meaning we can put a transaction inside another. This will be useful for the next construct

3. **Savepoints** can be used to rollback only a part of transaction

```
BEGIN TRANSACTION <my_transaction>
...
SAVEPOINT TRANSACTION <my_savepoint>
    <...>
ROLLBACK TRANSACTION <my_savepoint> -- Rollbacks only things done after the
savepoint!
...
```

X

Indexes in SQL

Indexes in SQL

X

Indexes in SQL. Main idea, motivations. Implementation of indexes. Advantages and disadvantages.

X

0. Voci correlate

- Introduction to SQL
- Binary Tree (TO DO!) #TODO
- Views in SQL

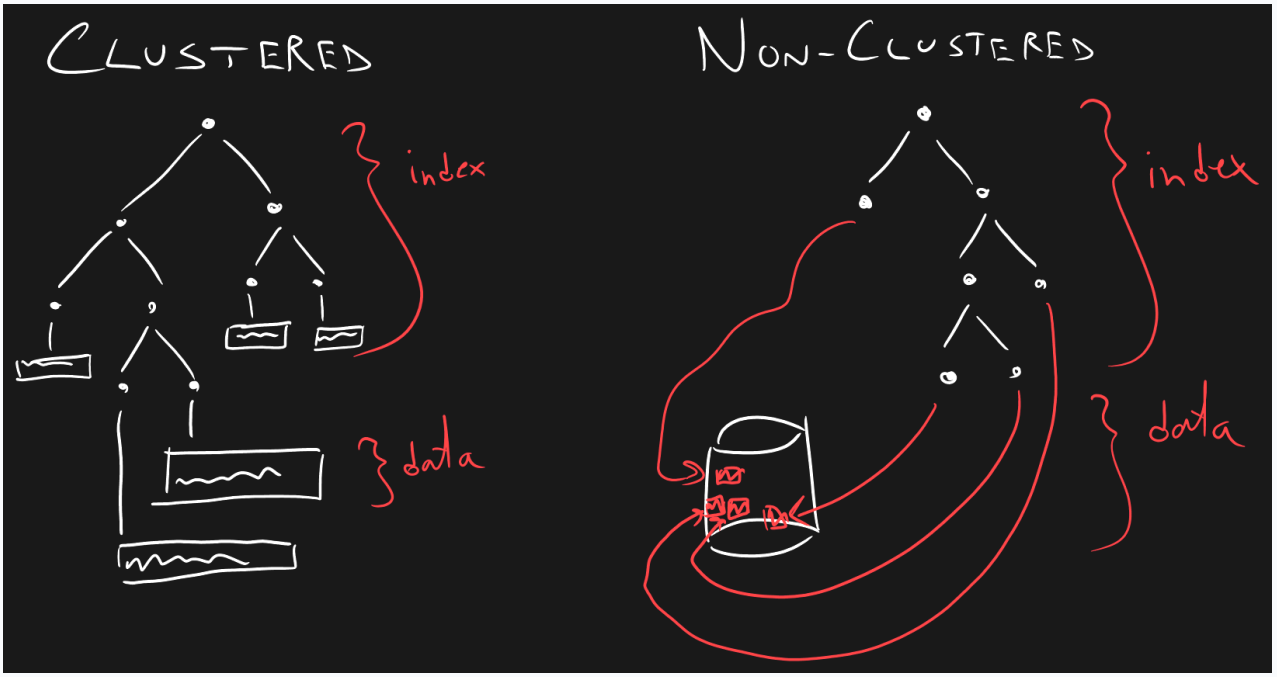
1. Definition of Index

IDEA. The idea of an *database index* is the same as a phone book; it gives us a sort of ordered structure where we can search items and find them in an efficient manner, reducing the search time greatly.

Definition. (*Index*)
An Index is an *on-disk* (or *in-memory*) structure to speed data retrieval from a table or a view. If it is stored on-disk, it is represented as a *binary tree*.

There are mainly *two types* of indexes:
Clustered indexes sort and store information based on its key values

- There can be *only one clustered index per table*, as records can only be sorted by a single key
- At the *lowest level* (sheet) we have all the data
- It contains the records
- With clustered indexes we have an *indexed view of the database*, which is a *materialized view* (Views in SQL).
  Non-Clustered indexes have *indexes* separated from the data
- It contains values of the keys, and each key has a *pointer* to the record that contains the value corresponding to the key



2. Syntax for Indexes

1. **Non-Clustered Indexes**

```
CREATE INDEX <my_index> ON <my_table>(<a1>, ..., <an>)

CREATE UNIQUE INDEX <my_index> ON <my_table>(<a1>, ..., <an>)

DROP INDEX <my_index>
```

2. **Clustered Indexes OR Materialized Views**

```
CREATE VIEW <my_indexed_view>
    WITH SCHEMABINDING AS
    SELECT <...>
    ...

CREATE UNIQUE CLUSTERED INDEX <my_clustered_index>
    ON <my_indexed_view>(<...>)

DROP VIEW <my_clustered index>
```

X

3. Advantages and Disadvantages of Indexes

Advantages

Using an index means finding a tuple *almost immediately* instead of analyzing the entire table

- Meaning we can pass a time complexity of $O(n)$ to $O(f(n) < n)$!
- In the case of binary trees, we have $O(n)$ becomes $O(\log n)$.

Disadvantages

However, there is no such thing as free lunch. In fact, we are trading better time performance for additional space, as they have to be stored somewhere.

- Moreover, time complexity is improved *only* on search; for initialization and maintenance the time complexity can be increased as they are slow operations and might have to be done in real time.

X

Introduction to Relational Algebra (OPTIONAL)

Introduction to Relational Algebra

X

Introduction to Relational Algebra (Optional): Operations and its corresponding SQL statements

X

0. Voci Correlate

- Basic Queries in SQL
- Joins in SQL
- Bibliography: GeekforGeeks

1. Definition of Relational Algebra

"Relational Algebra is a procedural query language. Relational algebra mainly provides a theoretical foundation for relational databases and SQL." (see Bibliography, first paragraph) - in other words, Relational Algebra provides us the mathematical foundation for SQL operations. In this chapter we will see a brief overview of relational algebra operations and its corresponding SQL statements.

2. Selection and Projection

#Definizione

> Definizione (selection).
>
> Let $R(V_1, \dots, V_n)$ be a relation, let φ be a logical formula on the variables $v_1, \dots, v_n$. Then the *selection operator* $\sigma_\varphi(R) = R_\varphi \subseteq R$ is defined as
>
> $$\sigma(\varphi) = R_\varphi := \{r \in R : r \models \varphi\}$$
>
> In other words, we are obtaining a subset of rows which satisfies the condition φ.

#Definizione

> Definizione (projection).
>
> Let $R(V_1, \dots, V_n)$ be a relation, $V \subset (V_1, \dots, V_n)$. Then the *projection* operator $\pi_V(R)$ returns $R(V)$.
>
> More simply, we are selecting certain rows of a relation.

#Esempio

Suppose we have a relation $R(A, B)$ and we run the following query:

```SQL
SELECT B
FROM R
WHERE A>10 OR B<-10
```

Then this query can be translated as

$$\pi_B \left(\sigma_{A>10 \vee B<-10}(R)\right)$$

3. Set Operators

#Definizione

Definizione (union, intersect and exception).

Let $R(V)$ and $S(V)$ be relations with the same columns. Then we define the *set operators* on R, S as the corresponding *union*, *intersection* and *difference* definitions for set theory.

4. Joins

#Definizione

Definizione (cross product).

Let $R(V)$, $S(W)$ be two relations. Then we define their *cross product* as the combination of every row of R with every row of S , producing all the possible combinations.

$$R(V) \times S(W) := T(V, W) = \{v, w \mid v \in V, w \in W\}$$

#Definizione

Definizione (natural join).

Let $R(V_1, \dots, V_n, C)$ and $S(W_1, \dots, W_n, C)$ with the common columns C. Then its *natural join* is defined as the relation with columns $V_1, \dots, V_n, C, W_1, \dots, W_n$ such that each value of C is the same. It is denoted as

$$R \bowtie S$$

#Definizione

Definizione (theta-join).

Let $R(V_1, \dots, V_n)$ and $S(W_1, \dots, W_n)$ be relations. Let θ be a logical formula on the values of the variables for R, S. We define the *theta-join* of R, S as the relation with columns $V_1, \dots, V_n, W_1, \dots, W_n$ such that each row satisfies the formula W

$$R \underset{\theta}{\bowtie} S$$

#Esempio

Let us take the following SQL query:

```SQL
SELECT R.a
FROM
    R JOIN S
        ON R.a > S.b
WHERE
    R.a < 10
```

This can be converted into a relational algebra formula as

$$\pi_{R.A}\left(\sigma_{R.a<10}\left(R \underset{R.a>S.b}{\bowtie} S\right)\right)$$