2.5.- The standard language SQL

- SQL (Structured Query Language) is a standard language for defining and manipulating (and selecting) a relational database.

- SQL includes:
  - Features from Relational Algebra (Algebraic Approach).
  - Features from Tuple Relational Calculus (Logical Approach).

- The most extended version nowadays is SQL2 (also called SQL’92).
  - Almost all RDBMSs are SQL2 compliant.
  - Some features from SQL3 (and some of the upcoming SQL4) are being included in many RDBMSs.
2.5.1.- SQL as a data definition language (DDL)

SQL commands for defining relational schemas:

- **create schema**: gives name to a relational schema and declares the user who is the owner of the schema.

- **create domain**: defines a new data domain.

- **create table**: defines a table, its schema and its associated constraints.

- **create view**: defines a view or derived relation in the relational schema.

- **create assertion**: defines general integrity constraints.

- **grant**: defines user authorisations for the operations over the DB objects.

All these commands have the opposite operation (DROP / REVOKE) and modification (ALTER).
2.5.1.1.- Schema Definition (SQL)

```
create schema [schema] [authorization user]
[list_of_schema_elements];
```

A schema element can be any of the following:

- Domain definition.
- Table definition.
- View definition.
- Constraint definition.
- Authorisation definition.

Removal of a relational schema definition:

```
drop schema schema {restrict | cascade};
```
create domain domain [as] datatype

[default {literal | system_function | null } ]

[domain_constraint_definition];

System functions:

- user
- current_user
- session_user
- current_date
- current_time
- current_timestamp
2.5.1.2.- Domain Definition (SQL)

A domain can be associated with a collection of constraints:

\[
\text{[constraint } \text{constraint]}
\]

\text{check (conditional_expression)}

[not] \text{deferrable}

- \text{conditional_expression} can express any condition that must meet every value in the domain (must be TRUE or UNDEFINED)

- \text{deferrable} indicates that (if set to deferred and not to immediate) the system must check the constraint at the end of the current transaction.

- \text{not deferrable} indicates that the system must check the constraint after each atomic update instruction on the database.
2.5.1.2.- Domain Definition (SQL).

Example

CREATE DOMAIN angle AS FLOAT

    DEFAULT 0

    CHECK (VALUE >= 0 AND VALUE < 360)

    NOT DEFERRABLE;

Removal of a domain:

    drop domain domain [restrict | cascade]
2.5.1.3.- Table Definition (SQL).

```sql
CREATE TABLE table
    column_definition_list
 [table_constraint_definition_list];
```

The definition of a table column is done as follows:

```sql
column {datatype | domain}
 [default {literal | system_function | null }]
 [column_construct_definition_list]
```

The constraints that can be defined over the columns are the following:

- **not null**: not null value constraint.
- Constraint definition for single column PK, Uni, FK.
- General constraint definition with the **check** clause.
The clause for defining table constraints is the following one:

```sql
[constraint constraint]
{ primary key (column_list)
  | unique (column_list)
  | foreign key (column_list)
    references table[(column_list)]
  [match {full | partial}] * NOT IN ORACLE
  [on update [cascade | * NOT IN ORACLE
    set null | set default | no action ]]* NOT IN ORACLE
  [on delete [cascade | * NOT IN ORACLE
    set null | set default | no action ]]* NOT IN ORACLE
  | check conditional_expression }
[constraint_check]
```

- Must be TRUE or UNDEFINED.
- Cannot include subqueries or references to other tables.
2.5.1.3.- Example: Provider-Piece-Supply

piece_code_d: string(4)
id_d: integer (positive)

Provider(id: id_d, name: string(40), address: string(25), city: string(30))
    PK: {id}
    NNV: {name}

Piece(code: piece_code_d, desc: string(40), colour: string(20), weight: real)
    PK: {code}

Supply (id: id_d, code: piece_code_d, price: real)
    PK: {id, code}
    FK: {id} → Provider
    FK: {code} → Piece

Integrity constraints:

R1) Px: Piece        ∀Px: Piece (Px.colour='red' → Px.weight>100 )

R2) Px: Piece,    Sx: Supply ∀Px: Piece (∃Sx: Supply (Sx.code=Px.code ) )
create schema Store
  authorization Joe
create domain piece_code_d as char(4)
create domain id_d as integer check value>0
create table Provider
  ( id id_d primary key,
    name varchar(40) not null,
    address char(25),
    city char(30) )
create table Piece
  ( code piece_code_d primary key,
    desc varchar(40),
    colour char(20),
    weight float,
    constraint r1 check (colour<>’red’ or weight>100) )
create table Supply
  ( id id_d, code piece_code_d references Piece,
    price float,
    primary key (id, code),
    foreign key (id) references Provider(id) );
2.5.1.3.- Table Definition (SQL). MATCH

R(\text{FK}) \rightarrow S(\text{UK})

- complete (match full): in a tuple of \( R \) all the values must have a null value or none of them. In the latter case, there must exist a tuple in \( S \) taking the same values for the attributes in \( \text{UK} \) as the values in the attributes of \( \text{FK} \).

- partial (match partial): if in a tuple of \( R \) one or more attributes of \( \text{FK} \) do not have a non-null value, then there must exist a tuple in \( S \) taking the same values for the attributes of \( \text{UK} \) as the values in the non-null attributes of \( \text{FK} \).

- weak (the clause match is not included): if in a tuple of \( R \) all the values for the attributes of \( \text{FK} \) have a non-null value, then there must exist a tuple in \( S \) taking the same values for the attributes of \( \text{UK} \) as the values in the attributes of \( \text{FK} \).
2.5.1.3.- Table Definition Modification (SQL).

In order to modify the definition of a table:

```
alter table base_table

{add [column] column_definition
 | alter [column] column
     {set default {literal | system_function | null }
      | drop default}
 | drop [column] column {restrict | cascade}
};
```

To remove a table from the relational schema:

```
drop table base_table {restrict | cascade};
```

With ORACLE some things are different

In ORACLE it is CASCADE CONSTRAINTS
create assertion *constraint*

check (*conditional_expression*)

[constraint_check];

The condition must be TRUE.
2.5.1.4.- Example: Provider-Pieces-Supply (SQL)

*Constraint R2 :*

\[ R2 \) Px: Piece, Sx: Supply \forall Px : Piece (\exists Sx : Supply(Sx) ( Sx.code=Px.code ) ) \]

*is defined through a general constraint:*

create assertion R2 check

not exists(select * from Piece P

where not exists(select *

from Supply S

where P.code=S.code));

Removal of a constraint

DROP ASSERTION *constraint*
2.5.2.- SQL as a data manipulation language

• SQL, as a data manipulation language, incorporates:

  ➢ The SELECT query command: integrates the logical and algebraic approaches.

  ➢ The commands for data modification: INSERT, DELETE and UPDATE.
2.5.2.1- The SELECT command

- Allows information retrieval from the database

- Syntax:

```sql
5 select [all | distinct] selected_item_list | *
1 from table
2 [where conditional_expression]
3 [group by column_list]
4 [having conditional_expression]
6 [order by column_reference_list]
```
2.5.2.1- The SELECT command

3 select R1X.A, R2X.B, ...... , RnX.AA
2 [where F(R1X, R2X, ..., RnX)]

being:

• R1, R2, ..., Rn are relations.
• A, B, ..., AA are attributes from the previous relations.
• R1X, R2X, ..., RnX are alternative names (alias).
• F(R1X, R2X, ..., RnX) is a condition.

The result is a relation which is composed of the attributes A, B, ..., AA of the tuples in the relations R1, R2, ..., Rn for which F is true.
In order to rename tables and attributes in SQL we use the reserved word **AS**.

It allows the renaming of a relation as well as of all its attributes (the original relation definition does not change, it only changes for the SELECT command)

Examples:

Player\texttt{(name: varchar, age: number, country: varchar)}

Player AS Competitor \quad \rightarrow \quad \text{Renames the } Player \text{ relation}

Player AS T(na, ag, co) \quad \rightarrow \quad \text{Renames the } Player \text{ relation and all its attributes.}
The reserved word `AS` is optional.

In case a query refers two or more times to the same table, renaming is indispensable.

Example:

```sql
Player(id: number, name: varchar, age: number, country: varchar)
PK: {id}
```  

List the pairs of players’ names who are from the same country:

```sql
Select J1.name, J2.name
from Player AS J1, Player AS J2
where J1.country = J2.country and J1.id < J2.id;
```
2.5.2.1- The SELECT command: Logical approach.

3  select R1X.A, R2X.B, ...... , RnX.AA
2  [where  F(R1X, R2X, ..., RnX)]

where:

• In the SELECT clause, we indicate the attributes we want to retrieve.
• In the FROM part, we declare the tuple variables.
• WHERE is a logical formula in which the only free variables are those declared in the FROM part.
• The formula in the WHERE clause is constructed by using a syntax which is very close to a first order logic language.
FORMALISATION (SYNTAX):
FORMULAS IN THE CLAUSE ‘WHERE’.

A condition is an expression that can be:

- IS NULL (RX.Ai)
- RX.Ai \( \alpha \) SX.Aj
- RX.Ai \( \alpha \) a

where:

- \( \alpha \) is a comparison operator (<, >, \( \leq \), \( \geq \), =, \( <> \)).
- \( A_i \) and \( A_j \) are attribute names of the relations over which we have defined variables RX and SX.
- \( a \) is a value in the domain associated with the attribute RX.Ai (except null).
Formulas are constructed by applying the following rules:

- Every condition is a formula.
- If $F$ is a formula, the $(F)$ and $\text{NOT } F$ are formulas.
- If $F$ and $G$ are formulas, then $F \text{ OR } G$, $F \text{ AND } G$ are also formulas.
- If $S$ is a SELECT command, then EXISTS$(S)$ is a formula.
- Nothing more is a formula.
2.5.2.1- The SELECT command: Logical approach.

3 select R1X.A, R2X.B, ...... , RnX.AA
2 [where $F(R1X, R2X, ..., RnX)$]

The SELECT command returns a relation in which each tuple in the relation is formed by the attribute values R1X.A, R2X.B, ...... , RnX.AA such that:

- These values appear in the variables R1X, R2X, ..., RnX.
- Hence, these values appear in the extensions of the relations R1, R2, ..., Rn.
- These values make the formula $F(R1X, R2X, ..., RnX)$ true.
2.5.2.1- The SELECT command: Logical approach.

FORMULA EVALUATION (SEMANTICS).

Truth value for a condition:

- If $F$ is of the form $RX.A_i \alpha SX.A_j$ then $F$ is evaluated to undefined if at least an attribute $Ai$ or $Aj$ has null value in the tuple which is assigned to $RX$ or to $SX$, otherwise it is evaluated to the truth value of the condition.

- If $F$ is of the form $RX.A_i \alpha a$ then $F$ is evaluated to undefined if $Ai$ has null value in the tuple which is assigned to $RX$, otherwise it is evaluated to the truth value of the comparison.

- If $F$ is of the form $IS\ NULL(RX.A_i)$ then $F$ is evaluated to true if $Ai$ has null value for the tuple which is assigned to $RX$, otherwise it is evaluated to false.
2.5.2.1- The SELECT command: Logical approach.

Truth value of a formula:

1) Let $F$ be a condition, then its truth value is the truth value of the condition.

2) If $F$ is of the form $(G)$, then $F$ is evaluated to the truth value of $G$.

3) If $F$ is any of the following forms NOT $G$, $G$ AND $H$ or $G$ OR $H$ where $G$ and $H$ are formulas, then $F$ is evaluated according to the following truth tables:
### 2.5.2.1- The SELECT command: Logical approach.

<table>
<thead>
<tr>
<th>G</th>
<th>H</th>
<th>F = G AND H</th>
<th>F = G OR H</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>False</td>
</tr>
<tr>
<td>undefined</td>
<td>false</td>
<td>false</td>
<td>undefined</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>undefined</td>
<td>false</td>
<td>undefined</td>
</tr>
<tr>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>true</td>
<td>undefined</td>
<td>undefined</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>undefined</td>
<td>true</td>
<td>undefined</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
<th>F = NOT G</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
4) If $F$ is of the form:

$$\text{EXISTS}( \text{select} \ast \text{ from } R1 \text{ [AS] } R1X, R2 \text{ [AS] } R2X, \ldots, Rn \text{ [AS] } RnX \text{ [where } G(R1X, R2X, \ldots, RnX) ] )$$

- Then $F$ is evaluated to true if there exist some values for the variables $R1X$, $\ldots$, $RnX$ in the extensions of $R1$, $\ldots$, $Rn$ for which $G$ is evaluated to true.

- Otherwise it is evaluated to false.
2.5.2.1- The SELECT command: Logical approach.

Example:

RIVER(rcode:rcode_dom, name:name_dom)
PROVINCE(pcode:pcode_dom, name:name_dom)
CROSSES(pcode:pcode_dom, rcode:rcode_dom)

Query1: “Provinces that are crossed by the river with code r1”.

First Order Logic:

PROVINCE(x,y) \land CROSSES(x, 'r1')

Tuple variables: PX:PROVINCE | \exists PPX:CROSSES (PPX.pcode = PX.pcode \land PPX.rcode = 'r1')

SELECT clause:

SELECT PX.pcode, PX.name
FROM PROVINCE PX, CROSSES PPX
WHERE PPX.pcode = PX.pcode AND PPX.rcode = 'r1'
FROM CROSSES PPX
WHERE PPX.pcode = PX.pcode AND PPX.rcode = 'r1'
2.5.2.1- The SELECT command: Logical approach.

Example:

RIVER(rcode:rcode_dom, name:name_dom)
PROVINCE(pcode:pcode_dom, name:name_dom)
CROSSES(pcode:pcode_dom, rcode:rcode_dom)

Query2: “Provinces which are crossed by no river”.

First Order Logic: PROVINCE(x,y) \land \neg \exists z \text{ CROSSES}(x, z)

Tuple variables: PX:PROVINCE | \neg \exists PPX:CROSSES (PPX.pcode = PX.pcode)

SELECT clause:

SELECT *
FROM PROVINCE PX
WHERE NOT EXISTS(SELECT *
FROM CROSSES PPX
WHERE PPX.pcode = PX.pcode)
2.5.2.1- The SELECT command: Logical approach.

Syntax of the existential quantifier in SQL:

```sql
EXISTS(SELECT *
FROM R1 R1X, R2 R2X, ..., Rn RnX
WHERE F(R1X, R2X, ..., RnX))
```

- is equivalent to the formula
  ```latex
  \exists R1X:R1(\exists R2X:R2 ...(\exists RnX:Rn (F(R1X, R2X, ..., RnX)))...)
  ```

- In SQL there is no universal quantifier; we must use the existential quantifier in its place through the conversion:
  ```latex
  \forall x F(x) \equiv \neg \exists x (\neg F(x))
  ```
2.5.2.1- The SELECT command: Logical approach.

Example:

RIVER(rcode:rcode_dom, name:name_dom)
PROVINCE(pcode:pcode_dom, name:name_dom)
CROSSES(pcode:pcode_dom, rcode:rcode_dom)

Query3: “List the rivers which cross all the provinces”.

Tuple variables:  
RX:RIVER|∀PX:PROVINCE (∃PPX:CROSSES (PPX.pcode=PX.pcode ^
PPX.rcode=RX.rcode))
RX:RIVER|¬∃PX:PROVINCE (¬∃PPX:CROSSES (PPX.pcode=PX.pcode ^
PPX.rcode=RX.rcode))

SELECT clause:

SELECT * FROM RIVER RX
WHERE NOT EXISTS (SELECT * FROM PROVINCE PX
WHERE NOT EXISTS(SELECT * FROM CROSSES PPX
WHERE PPX.pcode = PX.pcode AND
PPX.rcode = RX.rcode))
2.5.2.1- The SELECT command: Algebraic approach.

- Connects the content of two relations (or two query results) in a single table.
- In order to execute the UNION operator correctly, we require that both relations are compatible.

Example:

Cook(name: varchar, age: number, country: varchar)
Waiter(name: varchar, age: number, country: varchar)

List the adult workers in the restaurant:

```
Select name from Cook where age >= 18
UNION
Select name from Waiter where age >= 18;
```
The reserved word in SQL to perform a difference between relations is `EXCEPT`.

In order to execute the `EXCEPT` operator correctly, we require that both relations are compatible.

Example:

- **Cook**(name: `varchar`, age: `number`, country: `varchar`)
- **Waiter**(name: `varchar`, age: `number`, country: `varchar`)

List the workers who work only as cooks in the restaurant:

1. `Select * from (Cook except Waiter)`
2. `Cook except Waiter`
The reserved word in SQL to perform an intersection between relations is INTERSECT.

In order to execute the INTERSECT operator correctly, we require that both relations are compatible.

Example:

Cook(name: varchar, age: number, country: varchar)
Waiter(name: varchar, age: number, country: varchar)

List the workers who work as cooks and waiters in the restaurant:

1. Select * from (Cook intersect Waiter)
2. Cook intersect Waiter
2.5.2.1- The SELECT command: Algebraic approach.

**CARTESIAN PRODUCT**

- In order to execute the Cartesian product correctly, we require that both relations have different attribute names.
- In SQL the Cartesian product is just computed by adding both relations, separated by commas, in the FROM clause.

Example:

Team1\((name: \text{varchar}, \text{age: number}, \text{country: varchar})\)

Team2\((name: \text{varchar}, \text{age: number}, \text{country: varchar})\)

List all the possible combinations from players of Team 1 and players from Team 2:

Select * from Team1, Team2

Select * from Team 1 CROSS JOIN Team2

List pairs of players from Team1 who are from the same country:

Select * from Team1 e1, Team1 e2 where e1.country = e2.country and e1.age < e2.age
2.5.2.1- The SELECT command: Algebraic approach.

**PROJECTION**

- In order to project several attributes we just write the name of the attributes we want to retrieve after the SELECT clause, separated by commas.
- The attributes can be renamed using the AS clause.

Example:

Cook(name: varchar, age: number, country: varchar)

List the name of the cooks in the restaurant:

Select name from Cook
2.5.2.1- The SELECT command: Algebraic approach.

- There are several variants corresponding to the JOIN operator of the Relational Algebra.
- There are two main kinds of JOIN in SQL: inner and outer.
- INNER JOIN:

  \[ \text{table_reference [natural] [inner] join table_reference} \]
  \[ \text{on conditional_expression | using (column_list)} \]

- OUTER JOIN:

  \[ \text{table_reference [natural]} \]
  \[ \{ \text{left [outer] | right [outer] | full [outer]} \} \text{ JOIN table_reference} \]
  \[ \text{on conditional_expression | using (column_list)} \]
2.5.2.1- The SELECT command: Algebraic approach.

**JOIN (Cntd.)**

Examples: INNER JOIN.

\[ \text{tablereference} \ [\text{natural}] \ [\text{inner}] \ \text{join} \ \text{tablereference} \]

\[ \text{on} \ \text{conditional_expression} \ | \ \text{using} \ (\text{columnlist}) \]

PERSON(id: id\_dom, name: name\_dom, age: age\_dom)

HOUSE(house\_code: code\_dom, owner: id\_dom, addr: addr\_dom, rooms: number)

- Obtain a list with the houses and associated with its owner:
  1. PERSON inner join HOUSE on PERSON.id = HOUSE.owner
  2. PERSON natural inner join HOUSE AS V(cv, id, addr, nh)
  3. SELECT * FROM PERSON, HOUSE WHERE id = owner
2.5.2.1- The SELECT command: Algebraic approach.

Examples: OUTER JOIN.

\[
\text{table reference} \ [\text{natural}]
\]
\[
\{\text{left [outer]} \mid \text{right [outer]} \mid \text{full [outer]}\} \text{ JOIN table reference}
\]
\[
[\text{on } \text{conditional_expression} \mid \text{using (column_list) } ]
\]

PERSON(id: id\_dom, name: name\_dom, age: age\_dom)

HOUSE(house\_code: code\_dom, owner: id\_dom, addr: addr\_dom, rooms: number)

- Obtain a list with every house and, in case it has an owner, associated with its owner. Get also a list with every person. Another with every house and owner:

1. PERSON natural right join HOUSE \[\rightarrow\] All the houses appear
2. PERSON natural left join HOUSE \[\rightarrow\] All the owners appear
3. PERSON natural full join HOUSE \[\rightarrow\] All the houses and owners appear
2.5.2.1- The SELECT command: Algebraic approach.

- JOIN UNION

... FROM T1 UNION JOIN T2 \equiv ... FROM

select t1.*, null, null, ..., null from t1

union all

select null, null, ..., null, t2.* from t2
2.5.2.1- The SELECT command: Algebraic approach.

- The expression of the Relational Algebra:
  
  \[ R \text{ WHERE } F(A_i, A_j, A_k, \ldots) \]

  is equivalent to the expression in SQL:

  \[ \text{SELECT } \ast \text{ FROM } R \text{ WHERE } F(R.A_i, R.A_j, R.A_k, \ldots) \]

- In case we include several relations in the FROM clause in a SELECT:

  \[ \text{SELECT } \ast \text{ FROM } R_1, R_2, \ldots, R_n \text{ WHERE } F(R_1.A_i, \ldots, R_n.Z_k) \]

  Its equivalent in Relational Algebra would be:

  \[ R_1 \times R_2 \times \ldots \times R_n \text{ WHERE } F \left( R_1.A_i, \ldots, R_n.Z_k \right) \]
# 2.5.2.1- The SELECT command: Algebraic approach.

<table>
<thead>
<tr>
<th>Operator</th>
<th>RELATIONAL ALGEBRA</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
<td>R WHERE F</td>
<td>SELECT ... FROM R WHERE F</td>
</tr>
<tr>
<td>PROJECTION</td>
<td>R [A₁, A₂, ..., Aₖ]</td>
<td>SELECT A₁, A₂, ..., Aₖ FROM R</td>
</tr>
</tbody>
</table>
| CARTESIAN PRODUCT | R₁ × R₂, ..., × Rₙ                | SELECT ... FROM R₁, R₂, ..., Rₙ, o  
                                    |                         | SELECT...FROM R₁ CROSS JOIN R₂, ..., CROSS JOIN Rₙ |
| JOIN           | R₁ ⋈ R₂                             | SELECT... FROM R₁ NATURAL JOIN R₂ |
| UNION          | R₁ ∪ R₂                             | SELECT * FROM R₁ UNION SELECT * FROM R₂ |
| DIFFERENCE     | R₁ − R₂                             | SELECT * FROM R₁ EXCEPT SELECT * FROM R₂ |
| INTERSECTION   | R₁ ∩ R₂                             | SELECT * FROM R₁ INTERSECT SELECT * FROM R₂ |
2.5.2.1- The SELECT command: Algebraic approach.

Example:

RIVER(rcode: rcode_dom, name: name_dom)
PROVINCE(pcode: pcode_dom, name: name_dom)
CROSSES(pcode: pcode_dom, rcode: rcode_dom)

Query2: “Provinces which are crossed by no river”.
Relational Algebra:

PROVINCE[pcode, name] – (PROVINCE ※ CROSSES)[pcode, name]

SQL:
SELECT pcode, name FROM PROVINCE
EXCEPT
SELECT pcode, name FROM PROVINCE NATURAL JOIN CROSSES
Diapositiva 43

D1  DSIC; 19/04/2005
2.5.2.2- SQL as a data manipulation language: modification.

- DML can also insert one or more tuples in a relation.

- The syntax is:

  ```sql
  insert into table [(column_list)]
  { default values | values (atom_list) | table_expression}
  ```

- If we do not include the column list the complete rows will be inserted into `table`. 
2.5.2.2- SQL as a data manipulation language: modification.

Cook(name: varchar, age: number, country: varchar)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

INSERT INTO Cook
VALUES (“Carmelo Cotón”, 27, “France”);
2.5.2.2- SQL as a data manipulation language: modification.

Cook(name: varchar, age: number, country: varchar)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carmelo Cotón</td>
<td>27</td>
<td>France</td>
</tr>
</tbody>
</table>

INSERT INTO Cook
VALUES ("Carmelo Cotón", 27, "France");
2.5.2.2- SQL as a data manipulation language: modification.

**Cook** (name: `varchar`, age: `number`, country: `varchar`)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
INSERT INTO Cook(age, name)
VALUES (27, “Carmelo Cotón”);
```
### 2.5.2.2- SQL as a data manipulation language: modification.

**Cook** (*name*: `varchar`, *age*: `number`, *country*: `varchar`)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carmelo Cotón</td>
<td>27</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**INSERT INTO** `Cook`(*age, name*)

**VALUES** (27, “Carmelo Cotón”);
2.5.2.2- SQL as a data manipulation language: modification.

```
insert into table [(column_list)]
{ default values | values (atom_list) | table_expression}
```

- If we do not include the column list the complete rows will be inserted into `table`.
- If we include the `default values` option a single row will be inserted with the values by default which are appropriate for each column (according to the definition of `table`).
- In the option `values(atom_list)`, the atoms are given by scalar expressions.
- In the option `table_expression`, we insert the resulting rows of the execution of the expression ( SELECT ).
2.5.2.2- SQL as a data manipulation language: modification.

**COOK** (name: varchar, age: number, country: varchar)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PERSON** (name: varchar, age: number)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paco</td>
<td>22</td>
</tr>
<tr>
<td>Antonio</td>
<td>19</td>
</tr>
<tr>
<td>Soledad</td>
<td>26</td>
</tr>
</tbody>
</table>

**INSERT INTO** COOK(name, age)

```
SELECT name, age
FROM PERSON
WHERE age > 20;
```
**2.5.2.2- SQL as a data manipulation language: modification.**

Cook
\( \text{name: varchar, age: number, country: varchar} \)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paco</td>
<td>22</td>
<td>?</td>
</tr>
<tr>
<td>Soledad</td>
<td>26</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**INSERT INTO** Cook(name, age)

**SELECT** name, age

**FROM** PERSON

**WHERE** age > 20;
2.5.2.2- SQL as a data manipulation language: modification.

- Can modify the values of the attributes of one or more selected tuples.

- The syntax is:

```
update table
set assignment_list
[where conditional_expression]
```

Where an assignment is of the form:

```
column = {default | null | scalar_expression}
```
2.5.2.2- SQL as a data manipulation language: modification.

- If we include the clause ‘where’ the modification will only be applied to the rows which make the condition true.

Example: Decrement by 1 the age of the French cooks.

```
UPDATE Cook SET age = age - 1
WHERE country = "France" ;
```
2.5.2.2- SQL as a data manipulation language: modification.

- Removes one or more tuples from a relation.
- The syntax is:

  \[
  \text{DELETE FROM} \ table \ [\text{WHERE} \ conditional\_expression]
  \]

- If we include the clause ‘where’ the rows which make the condition true will be removed.

Example: Remove all the cooks who are younger than 18.

  \[
  \text{DELETE FROM} \ Cook \ \text{WHERE} \ \text{age} < 18;
  \]
2.6.1.- Notion of view.

• A view is a virtual table which is derived from other tables (base or virtual).

• Features of a view:
  – It is considered part of the external schema.
  – A view is a virtual table (it doesn’t have any correspondence at the physical level).
  – Can be queried like any other base table.
  – Updates are transferred to the original tables (with some limitations).
2.6.2.- Applications of views.

- To specify tables with information which is accessed frequently but which does not have a physical correspondence:
  - Derived information from several tables.
  - Derived information from the aggregation of tuples (group by), such as statistics.
  - In general: derived information obtained by complex queries which are accessed frequently.

- As a privacy mechanism: definition of views only with the table attributes the author can have access to.

- To create external schemas.
2.6.3.- Views in SQL.

- The syntax for the definition of views in SQL is as follows:

```
CREATE | REPLACE VIEW view [(column_list)]
    AS table_expression [with check option]
```

where:

- CREATE VIEW is the command.

- view is the name of the virtual table which is being defined.

- (column_list) are the names of the table attributes (it is optional):
  
  - If not specified, name coincides with the names of the attributes which return the table_expression.

  - It is compulsory if some attribute in table_expression is the result of an aggregation function or an arithmetic expression.
2.6.3.- Views in SQL.

- The syntax for the creation of views in SQL is as follows:

\[
\text{CREATE | REPLACE VIEW } \text{view} \ [(\text{column\_list})] \\
\quad \text{AS } \text{table\_expression} \ [\text{with check option}]
\]

where:

- \text{table\_expression} is a SQL query whose result will include the content of the view.
- WITH CHECK OPTION is optional and must be included if the view is to be updated in an appropriate way.
- To remove a view we use the command:

\[
\text{DROP VIEW } \text{view} \ [\text{restrict | cascade}];
\]
2.6.3.- Views in SQL (Examples).

• Given the following database relation:

  **Cook**(name: `varchar`, age: `number`, country: `varchar`)

Define a view with only the French cooks:

  ```sql
  CREATE VIEW French AS
  SELECT * FROM Cook WHERE country = "France"
  WITH CHECK OPTION
  ```

Define a view with the average age of the cooks grouped by country:

  ```sql
  CREATE VIEW Report(country, avg_age) AS
  SELECT country, AVG(age) FROM Cook GROUP BY country
  ```
Reasons why a view is NOT updatable:

- It contains set operators (UNION, INTERSECT, …).
- It contains the DISTINCT operator
- It contains aggregated functions (SUM, AVG, ..)
- It contains the clause GROUP BY
2.6.3.- Views in SQL (updatable views).

View over a base table:

- The system will translate the update over the view to the corresponding action to the base relation.
  - Provided that no integrity constraint defined on the relation is violated.
2.6.3.- Views in SQL (updatable views).

View over a join of two relations:

- The update can only modify one of the two base tables.

- The update will modify the base relation which complies with the property of key preservation (the table whose primary key could also be the primary key of the view).

  - Provided that no integrity constraint defined on the affected relation is violated.
2.6.3.- Views in SQL (updatable views).

Example:

• Given the following relations:

  PERSON(id: \textit{id} \textit{dom}, name: \textit{name} \textit{dom}, age: \textit{age} \textit{dom})
  \text{PK}\{\textit{id}\}

  HOUSE(house\_code: \textit{code} \textit{dom}, id: \textit{id} \textit{dom}, addr: \textit{addr} \textit{dom}, rooms: \textit{number})
  \text{PK}\{\textit{house} \textit{code}\}  \text{FK}\{\textit{id}\} \rightarrow \text{PERSON}

• Given the following view which is defined over these relations:

  CREATE VIEW ALL\_HOUSE AS
  SELECT * FROM PERSON NATURAL JOIN HOUSE

Can we modify the address of a house in ALL\_HOUSE?
Yes, the PK in HOUSE could work as the PK in ALL\_HOUSE

Can we modify the name of the HOUSE owner?
No, the update is ambiguous
A trigger is a rule which is automatically activated by certain events and executes a particular action.
2.7.2.- Event-condition-action rules.

Form of an activity rule:

event - condition - action

*action* which the system executes as a response of the happening of an *event* when a certain *condition* is met:

- *event*: update operation
- *condition*: logical expression in SQL. The action will only be executed if this condition is true. If the condition is not specified, the condition is assumed to be true.
- *action*: a procedure written in a programming language which include manipulation instructions to the DB.
Define the active behaviour of a database system:

- Check of general integrity constraints
- Restoration of consistency
- Definition of operational rules in the organisation
- Maintenance of derived information
2.7.4.- Triggers in SQL.

Rule definition::=

\{CREATE \| REPLACE\} TRIGGER rule_name
\{BEFORE \| AFTER \| INSTEAD OF\} event [events_disjunction]
ON \{relation_name \| view_name\}
[ [REFERENCING OLD AS reference_name
    [NEW AS reference_name] ]
[FOR EACH \{ROW \| STATEMENT\} [WHEN ( condition ) ] ] ]
PL/SQL block

events_disjunction ::= OR event [events_disjunction]

event ::= INSERT \| DELETE \| UPDATE [OF attribute_name_list]
2.7.4.- Triggers in SQL.

Events:

\{\text{BEFORE} \mid \text{AFTER} \mid \text{INSTEAD OF}\} \text{ event } [\text{events\_disjunction}] \\
\text{ON } \{\text{relation\_name} \mid \text{view\_name}\}

\text{events\_disjunction} ::= \text{OR } \text{event } [\text{events\_disjunction}]

\text{event} ::= \\
\text{INSERT} \mid \text{DELETE} \mid \text{UPDATE} [\text{OF } \text{attribute\_name\_list}]

2.7.4.- Triggers in SQL.

Events:

Event parameterisation:
- The events in the rules defined with FOR EACH ROW are parameterised
- Implicit parameterisation:
  - event INSERT or DELETE: $n$ ($n$ being the degree of the relation)
  - event UPDATE: $2^n$
- Name of the parameters:
  - event INSERT: $NEW$
  - event DELETE: $OLD$
  - event UPDATE: $OLD$ and $NEW$
- They can be used in the condition of the rule
- They can be used in the PL/SQL block
## 2.7.4. Triggers in SQL.

<table>
<thead>
<tr>
<th></th>
<th>FOR EACH STATEMENT</th>
<th>FOR EACH ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BEFORE</strong></td>
<td>The rule is executed once before the execution of the update operation</td>
<td>The rule is executed once before the update of each tuple which is affected by the update operation</td>
</tr>
<tr>
<td><strong>AFTER</strong></td>
<td>The rule is executed once after the execution of the update operation</td>
<td>The rule is executed once after the update of each tuple which is affected by the update operation</td>
</tr>
</tbody>
</table>
2.7.4.- Triggers in SQL.

CONDITIONS

WHEN (condition)

– Logical expression with a similar syntax as the condition of the ‘WHERE’ clause of the SELECT instruction
– It cannot contain queries or aggregated functions
– It can only refer to the parameters in the event
2.7.4.- Triggers in SQL.

**ACTIONS**

PL/SQL block

- block written in the programming language Oracle PL/SQL
- Manipulation statements over the DB: INSERT, DELETE, UPDATE, SELECT ... INTO ...
- Program statements: assignment, selection, iteration
- Error handling statements
- Input/output statements
2.7.4.- Triggers in SQL.

Rule language:

– Definition: CREATE TRIGGER rule_name ...
– Removal: DROP TRIGGER rule_names
– Modification: REPLACE TRIGGER rule_name ...
– Recompilation: ALTER TRIGGER rule_name COMPILE
– Disable/enable rule: ALTER TRIGGER rule_name [ENABLE | DISABLE]
– Disable/enable all the rules defined over a relation:
  ALTER TABLE relation_name [{ENABLE | DISABLE} ALL TRIGGERS]
The constraint R2 such as this

\[ R2 \begin{align*} \text{Px: Piece, Sx: Supply} & \forall \text{Px : Piece} ( \exists Sx : \text{Supply} ( Sx.\text{code}=\text{Px.code} ) ) \end{align*} \]

can be defined through the following assertion:

create assertion R2 check
not exists (select * from Piece P
    where not exists (select *
        from Supply S
        where P.code=S.code));

How can this constraint be controlled through triggers?
We must detect the events which might affect the I.C.:

<table>
<thead>
<tr>
<th>table,</th>
<th>operation,</th>
<th>attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply,</td>
<td>Deletion,</td>
<td>-</td>
</tr>
<tr>
<td>Supply,</td>
<td>Update,</td>
<td>code</td>
</tr>
<tr>
<td>Piece,</td>
<td>Insertion,</td>
<td>-</td>
</tr>
</tbody>
</table>

Then we must define triggers to control these events.
CREATE TRIGGER T1
AFTER DELETE ON Supply OR UPDATE OF code ON Supply
FOR EACH ROW
DECLARE
  N: NUMBER;
BEGIN
  SELECT COUNT(*) INTO N
  FROM Supply S
  WHERE :old.code = S.code;
  IF N=0 THEN
    RAISE_APPLICATION_ERROR(-20000, 'We can’t delete this supply, otherwise the piece would remain without supplies.‘);
  END IF;
END;
CREATE TRIGGER T2
AFTER INSERT ON Piece
FOR EACH ROW
DECLARE N: NUMBER;
BEGIN
    SELECT COUNT(*) INTO N
    FROM Supply S WHERE :new.code = S.code;
    IF N=0 THEN
        RAISE_APPLICATION_ERROR(-20000, 'We cannot insert a new piece, because this piece has no supplies. Insert the two tuples (piece and supply) inside a transaction by disabling this trigger first.);
    END IF;
END;

2.7.4.- Triggers in SQL (Example).
2.8.- Limitations of the relational model.

- The traditional data model (relational, hierarchical and network) has had great success in traditional business and transactional applications.

- Traditional models present deficiencies in six applications:
  - Design and manufacturing in engineering (CAD/CAM/CIM),
  - Scientific experiments,
  - Telecommunications,
  - Geographical Information Systems,
  - Multimedia, and
  - Strategic data warehouses.
2.8.- Limitations of the relational model.

- Requirements and characteristics for the new applications:
  - More complex structures for the objects in the database,
  - Longer transactions,
  - New datatypes needed to store images or big text/binary blocks, and
  - Need for defining specific (non standard) operations for the applications.

- Evolution of relational databases:
  - Deductive databases,
  - Active databases,
  - Object-oriented databases
  - Object-relational databases (SQL3)
  - Multidimensional databases.