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.

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

ORACLE
- **create view**: defines a view or derived relation in the relational schema.
- **create assertion**: defines general integrity constraints.

ORACLE
- **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**};

2.5.1.2.- Domain Definition (SQL)

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:

[**constraint** *constraint*]

check (*conditional_expression*)

[**not**] **deferrable**

- *conditional_expression* can express any condition that must meet every value in the domain (<u>must be TRUE or UNDEFINED</u>)
- **deferrable** indicates that (*if set to deferred and not to immediate*) the system must check the constraint at the end of the current transaction.
- **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).

CREATE TABLE *table*

column_definition_list

[*table_constraint_definition_list*];

The definition of a table column is done as follows:

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.

2.5.1.3.- Table Definition (SQL).

The clause for defining table constraints is the following one:

[**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
Default value: the operation is not allowed
[**on delete** [**cascade** |
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))

2.5.1.3.- Example: Provider-Pieces-Supply (SQL)

```
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) );
```

⇐ R1

and R2?

2.5.1.3.- Table Definition (SQL). MATCH

R(*FK*) →S(*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 *UK* as the values in the attributes of *FK*.

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

ORACLE

- weak (the clause **match** is not included): if in a tuple of *R* all the values for the attributes of *FK* have a non-null value, then there must exist a tuple in *S* taking the same values for the attributes of *UK* as the values in the attributes of *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**} };

With ORACLE some things are different

To remove a table from the relational schema:

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

In ORACLE it is CASCADE CONSTRAINTS

2.5.1.4.- Constraint definition (SQL)

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 ∀Px : Piece (∃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

SELECT

- Allows information retrieval from the database

- Syntax:

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

1 **from** R1 [AS] R1X, R2 [AS] R2X, ... , Rn [AS] RnX

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.

2.5.2.1- The SELECT command.

RENAME

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

Player AS Competitor → Renames the *Player* relation

Player AS T(na, ag, co) → Renames the *Player* relation and all its attributes.

2.5.2.1- The SELECT command.

RENAME *(Cont)*

- The reserved word **AS** is optional.
- In case a query refers two or more times to the same table, renaming is indispensable.

Example:

Player(id: *number*, name: *varchar*, age: *number*, country: *varchar*)

PK:{id}

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

```
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

1 **from** R1 [AS] R1X, R2 [AS] R2X, ... , Rn [AS] RnX

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.

2.5.2.1- The SELECT command: Logical approach.

FORMALISATION (SYNTAX):

FORMULAS IN THE CLAUSE 'WHERE'.

A condition is an expression that can be:

- IS NULL (RX.Ai)
- RX.Ai α SX.Aj
- RX.Ai α a

where:

- α is a comparison operator (<, >, ≤, ≥, =, <>).
- 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*).

2.5.2.1- The SELECT command: Logical approach.

Formulas are constructed by applying the following rules:

- Every condition is a formula.
- If ***F*** is a formula, the ***(F)*** and *NOT* ***F*** are formulas.
- If ***F*** and ***G*** are formulas, then ***F OR G***, ***F 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

1 **from** R1 [AS] R1X, R2 [AS] R2X, ... , Rn [AS] RnX

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.Ai α SX.Aj 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.Ai α 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.Ai) 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.

G	H	F =G AND H	F = G OR H
false	false	false	False
undefined	false	false	undefined
true	false	false	true
false	undefined	false	undefined
undefined	undefined	undefined	undefined
true	undefined	undefined	true
false	true	false	true
undefined	true	undefined	true
true	true	true	true

G	F = NOT G
false	true
undefined	undefined
true	false

2.5.2.1- The SELECT command: Logical approach.

4) If *F* is of the form:

EXISTS(select *

from R1 [AS] R1X, R2 [AS] R2X, ... , Rn [AS] RnX

[**where** *G(R1X, R2X, ..., RnX)*])

- Then F is evaluated to true if there exist some values for the variables R1X, ..., RnX in the extensions of R1, ..., 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) ^ CROSSES(x, ‘r1’)

Tuple variables: PX:PROVINCE | ∃PPX:CROSSES (PPX.pcode = PX.pcode ^ PPX.rcode = ‘r1’)

SELECT clause:

```
SELECT PX.pcode, PX.name
FROM PROVINCE PX, CROSSES PPX
WHERE PPX.pcode = PX.pcode AND PPX.rcode = ‘r1’
```

```
SELECT *
FROM PROVINCE PX
WHERE EXISTS(SELECT *
             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) ^ ¬∃z CROSSES(x, z)

Tuple variables: PX:PROVINCE | ¬∃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:

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

- is equivalent to the formula $\exists R1X{:}R1(\exists R2X{:}R2 \ldots(\exists RnX{:}Rn\ (F(R1X, R2X, \ldots, RnX))\ldots)$

- In SQL there is no universal quantifier; we must use the existential quantifier in its place through the conversion:

$$\forall x\ F(x) \equiv \neg\exists x\ (\neg\ F(x))$$

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.

UNION

- 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;

2.5.2.1- The SELECT command: Algebraic approach.

DIFFERENCE

- 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

2.5.2.1- The SELECT command: Algebraic approach.

INTERSECTION

- 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: *varchar*, age: *number*, country: *varchar*)

Team2(name: *varchar*, age: *number*, 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.

JOIN

- 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:

  table_reference [**natural**] [**inner**] **join** *table_reference*

  [**on** *conditional_expression* | **using** (*column_list*)]

- OUTER JOIN:

  table_reference [**natural**]

  {**left** [**outer**] | **right** [**outer**] | **full** [**outer**]} **JOIN** *table_reference*

  [**on** *conditional_expression* | **using** (*column_list*)]

2.5.2.1- The SELECT command: Algebraic approach.

JOIN *(Cntd.)*

Examples: INNER JOIN.

table_reference [**natural**] [**inner**] **join** *table_reference*

[**on** *conditional_expression* | **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 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.

JOIN *(Cntd.)*

Examples: OUTER JOIN.

table_reference [natural]

{left [outer] | right [outer] | full [outer]} JOIN *table_reference*

[on *conditional_expression* | 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 ⟹ All the houses appear
2. PERSON natural left join HOUSE ⟹ All the owners appear
3. PERSON natural full join HOUSE ⟹ All the houses and owners appear

2.5.2.1- The SELECT command: Algebraic approach.

JOIN *(Cntd.)*

- JOIN UNION

... FROM T1 UNION JOIN T2 ≡ ... 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.

SELECTION

- The expression of the Relational Algebra:

  R **WHERE** F(Ai, Aj, Ak, ...)

  is equivalent to the expression in SQL:

  SELECT * FROM R **WHERE** F(R.Ai, R.Aj, R.Ak, ...)

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

  SELECT * FROM R1, R2, ..., Rn **WHERE** F(R1.Ai, ..., Rn.Zk)

  Its equivalent in Relational Algebra would be:

  R1 × R2 × ... × Rn **WHERE** F (R1.Ai, ..., Rn.Zk)

2.5.2.1- The SELECT command: Algebraic approach.

Operator	RELATIONAL ALGEBRA	SQL
SELECTION	R WHERE F	SELECT ... FROM R WHERE F
PROJECTION	R $[A_{i\,,}\, A_{j\,...,}\, A_k]$	SELECT $A_{i\,,}\, A_{j\,...,}\, A_k$ FROM R
CARTESIAN PRODUCT	$R_1 \times R_2, \ldots \times R_n$	SELECT ... FROM R_1, R_2, ..., $R_{n,}$ o SELECT...FROM R_1 CROSS JOIN R_2, ..., CROSS JOIN R_n
JOIN	$R_1 \bowtie R_2$	SELECT... FROM R_1 NATURAL JOIN R_2
UNION	$R_1 \cup R_2$	SELECT * FROM R_1 UNION SELECT * FROM R_2
DIFFERENCE	$R_1 - R_2$	SELECT * FROM R_1 EXCEPT SELECT * FROM R_2
INTERSECTION	$R_1 \cap R_2$	SELECT * FROM R_1 INTERSECT SELECT * FROM R_2

D1

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.

INSERT

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

- The syntax is:

  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*)

name	age	country
. . .	. . .	. . .

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*)

name	age	country
. . .	. . .	. . .
Carmelo Cotón	27	France
. . .	. . .	. . .

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*)

name	age	country
. . .	. . .	. . .

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*)

name	**age**	**country**
. . .	. . .	. . .
Carmelo Cotón	27	?
. . .	. . .	. . .

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*)

name	age	country
. . .	. . .	. . .

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

name	age
Paco	22
Antonio	19
Soledad	26

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

name	**age**	**country**
. . .	. . .	. . .
Paco	22	?
Soledad	26	?
. . .	. . .	. . .

INSERT INTO Cook(name, age)

SELECT name, age

FROM PERSON

WHERE age > 20;

2.5.2.2- SQL as a data manipulation language: modification.

UPDATE

- 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.

DELETE

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

  DELETE FROM *table* [**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.

DELETE FROM Cook **WHERE** 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:

  CREATE | REPLACE VIEW *view* [(*column_list*)]

  AS *table_expression* [**with check option**]

where:

- *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:
  - **DROP VIEW** *view* [**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:

Check Option ensures that cooks who are not French cannot be added to the view

```
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:

```
CREATE VIEW Report(country, avg_age) AS
    SELECT country, AVG(age) FROM Cook GROUP BY country
```

2.6.3.- Views in SQL (updatable views).

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: *id_dom*, name: *name_dom*, age: *age_dom*)
  PK:{id}

  HOUSE(house_code: *code_dom*, id: *id_dom*, addr: *addr_dom*, rooms: *number*)
  PK:{house_code} FK:{id} → 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

2.7.1.- Notion of trigger.

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.

2.7.3.- Applications of triggers.

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:

{**BEFORE** | **AFTER** | **INSTEAD OF**} *event* [*events_disjunction*]
ON {*relation_name* | *view_name*}

events_disjunction ::= **OR** *event* [*events_disjunction*]

event ::=
INSERT | **DELETE** | **UPDATE** [**OF** *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.

	FOR EACH STATEMENT	FOR EACH ROW
BEFORE	The rule is executed once before the execution of the update operation	The rule is executed once before the update of each tuple which is affected by the update operation
AFTER	The rule is executed once after the execution of the update operation	The rule is executed once after the update of each tuple which is affected by the update operation

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]

2.7.4.- Triggers in SQL (Example).

The constraint R2 such as this

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

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?

2.7.4.- Triggers in SQL (Example).

We must detect the events which might affect the I.C. :

table,	***operation,***	***attribute***
Supply,	Deletion,	-
Supply,	Update,	code
Piece,	Insertion,	-

Then we must define triggers to control these events.

2.7.4.- Triggers in SQL (Example).

```
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;
```

2.7.4.- Triggers in SQL (Example).

```
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;
```

2.8.- 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.