Unit 5:

SQL:
Data Definition Language (DDL)

Bases de Datos y Sistemas de información
Departamento de Sistemas Informáticos y Computación / Universidad Politécnica de Valencia

V. 16.3

Unit 5. SQL: Data Definition Language (DDL)

SQL

DDL (Data Definition Language): Creation, modification, and deletion of the components of the relational DB schema.

DML (Data Manipulation Language): Queries and database updates.

Control Language: Dynamically changes the database properties

Unit 5. SQL: 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 authorizations for the operations over the DB objects.

All these commands have the opposite operation (**DROP** / **REVOKE**) and modification (**ALTER**).

Schema definition

CREATE SCHEMA [*schema_name*] [**AUTHORIZATION** *user*]
[*list_of_schema_elements*]

A schema element can be any of the following:

- Domain definition.
- Table definition.
- View definition.
- Constraint definition.
- Authorization definition.

Cascade: automatically drops objects (tables, functions, etc.) that are contained in the schema.

Removal of a relational schema definition:

DROP SCHEMA *schema_name* { **RESTRICT** | **CASCADE** };

Create table

CREATE TABLE *table_name*
(*column_definition_list* [*table_constraint_definition_list*] **);**

Where a ***column_definition*** is done as follows:

column_name { *datatype* | *domain* }
[**DEFAULT** { *literal* | *system_function* | **NULL**}]
[*column_construct_definition_list*]

Constraints over a column

The constraints that can be defined over a **column** are:

- NOT NULL: not null value constraint.
- Constraint definition for single column *PK*, *Uni*, *FK*.
- General constraint definition with the *check* clause.

[**CONSTRAINT** *constraint_name*]
{ **NOT NULL**
| **PRIMARY KEY**
| **UNIQUE**
| **REFERENCES** *table_name* [(*column_name*)]
[**ON DELETE**
{ **CASCADE** | **SET NULL** | **SET DEFAULT** | **NO ACTION** }]
[**ON UPDATE**
{ **CASCADE** | **SET NULL** | **SET DEFAULT** | **NO ACTION** }]
| **CHECK** (*conditional_expression*) }
[*When_check_constraint*]

The operation is not allowed if the PK is violated (default option)

Example

```
CREATE TABLE puerto
(nompuerto VARCHAR2(35) CONSTRAINT PK_puerto PRIMARY KEY,
 altura NUMBER(4),
 categoria CHAR(1),
 pendiente NUMBER(3,2),
 netapa NUMBER(2) NOT NULL
        CONSTRAINT FK_puerto_eta REFERENCES etapa (netapa),
 dorsal NUMBER(3)
        CONSTRAINT FK_puerto_cicli REFERENCES ciclista (dorsal)
);
```

Constraints over a table

The constraints that can be defined over a **table** are:

- Constraint definition for single column PK, Uni, FK.
- General constraint definition with the check clause.

```
[ CONSTRAINT constraint_name ]
    {     UNIQUE ( list_of_column_names )
        | PRIMARY KEY ( list_of_column_names )
        | FOREIGN KEY ( list_of_column_names )
            REFERENCES table_name [( list_of_column_names )]
              [ ON DELETE
                  { CASCADE | SET NULL | SET DEFAULT | NO ACTION }]
              [ ON UPDATE
                  { CASCADE | SET NULL | SET DEFAULT | NO ACTION }]
        | CHECK ( conditional_expression )   }
        [ When_to_check_constraints ]
```


Example

```
CREATE TABLE llevar
        (dorsal NUMBER(3) NOT NULL CONSTRAINT FK_llevar_cicli
                REFERENCES ciclista (dorsal),
        netapa NUMBER(2)
            CONSTRAINT FK_llevar_etapa REFERENCES etapa (netapa),
        codigo CHAR(3)
            CONSTRAINT FK_llevar_mai REFERENCES maillot (codigo),
        CONSTRAINT PK_lle PRIMARY KEY ( netapa, codigo )
);
```

Types of referential integrity

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.
- **Weak** (**default value**. 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 FK. This is the only type supported by Oracle

When to check the constraints

[[**NOT**] **DEFERRABLE**]
[**INITIALLY** { **IMMEDIATE** | **DEFERRED** }]

- **deferrable** indicates that the constraint state can be modified between **deferred** (evaluated at the end of the transaction) and **immediate** (evaluated after every update of the database).
- **not deferrable** (default option) indicates that the constraint state cannot be modified and then it is assumed to be immediate forever and for every transaction.

For the deferrable option, we can specify how it starts for every transaction:

- INITIALLY IMMEDIATE
- INITIALLY DEFERRED

To change the mode of one or more constraints:

SET CONSTRAINT { *list_of_constraint_names* | **ALL** }
{ **IMMEDIATE** | **DEFERRED** }

4. Table modification

```
ALTER TABLE table_name
  {    ADD ( column_definition)
     | MODIFY [ COLUMN ] ( column_name )
          { DROP DEFAULT |
            SET DEFAULT { string | system_function | NULL} |
            ADD constraint_definition |
            DROP constraint_name }
     | DROP [ COLUMN ]  column_name
     { RESTRICT | CASCADE }
  }
```

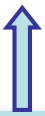
Example:

ALTER TABLE ciclista ADD (estatura NUMBER(3))

5. Table deletion

The SQL instruction to delete one table is

DROP TABLE *table_name* { **CASCADE CONSTRAINTS** }



To delete all the Foreign Keys referencing this table

6. View definition

- A view is a virtual table which is derived from other tables (base or virtual).
- Can be queried like any other base table.

CREATE VIEW *view_name*
[(*list_of_column_names*)]
AS SELECT_statement
[**WITH CHECK OPTION**]

If not specified, name coincides with the ones returned by the SELECT_statement

No update or insertion will be allowed if it violates the view definition

Example (Cycling race schema)

We are going to write many queries using the stages with mountain passes.

```
CREATE VIEW Etapas_con_puertos AS
        SELECT *
        FROM Etapa
        WHERE netapa IN (SELECT netapa FROM Puerto);
```

Now, we can write queries using this view:

List the km of the longest stage including at least one maintain pass

```
        SELECT MAX(km)
        FROM Etapas_con_puertos;
```


Updating views

Updates are transferred to the original tables with some limitations.

A view is not updatable if:

- It contains set operators (UNION, INTERSECT,...).
- It contains the DISTINCT operator
- It contains aggregated functions (SUM, AVG, ..)
- It contains the clause GROUP BY

If the view uses two or more tables, only will be allowed the modifications affecting the table containing the primary key what could be primary key of the view

7. View deletion

DROP VIEW *view_name* { CASCADE CONSTRAINTS }

To delete all the Foreign Keys referencing this view

Trigger

A **trigger** defines an action that the database should take when some event occurs.

Many DBMS do not support the creation of general constraints using assertions. These general constraints can be created using triggers.

A trigger can be also used to enforce complex enterprise constraints or to audit changes to data.

Sometimes, we need an autonomous behavior of the database which can be implemented using triggers. For example, this autonomous behavior can be used to maintain the value of a derived column.

Trigger components

A trigger has **3 components**:

- Event
- Condition (optional)
- Action

When the **event** happens, the DBMS will execute the **action** of the trigger if, and only if, the **condition** is true.

The event can be an INSERT, UPDATE, or DELETE statement, a CREATE, ALTER, or DROP statement.

The action contains SQL statements and code to be executed.

Trigger syntax

```
CREATE [ OR REPLACE ] TRIGGER trigger_name
        { BEFORE | AFTER | INSTEAD OF }
        { INSERT | DELETE | UPDATE OF [ list_of_columns ] }
                        ON table_name
        [ FOR EACH ROW ]
        [ DECLARE list_of_variables]
        [ WHEN ( condition) ]
        BEGIN
                { SQL_statement  |
                  SQL/PSM block |
                  { CALL } SQL_procedure }
        END ;
```

Row-level vs statement-level

	Without FOR EACH ROW	FOR EACH ROW
BEFORE	The trigger is executed once before the execution of the event	The trigger is executed before updating each row
AFTER	The trigger is executed once after the execution of the event	The triggers is executed after updating each row
INSTEAD OF		The trigger is executed for each row instead of the event

Trigger parameters

Parameters: old and new

- Only available in statement-level triggers (FOR EACH ROW)
- Can be used in the *condition* and in the *action* to refer to the tuples affected by the event
  - event INSERT: *new*
  - event DELETE: *old*
  - event UPDATE: *old* and *new*

WHEN (condition)

- Logical expression using the SQL syntax (as used in the WHERE of an SQL query)

Action

- Program written in the PL/SQL language
- Includes assignment, loops, and selection instructions.
- The trigger parameters can be used with ":" (*:old* and *:new*)

Example

Consider the schema *R (A : dom_A, B : dom_B)*

When a new row is inserted in the table *R*, insert a copy in the table *R2*

```
CREATE TRIGGER T1
   AFTER INSERT ON R
   FOR EACH ROW
      BEGIN
            INSERT INTO R2 VALUES (:NEW.A, :NEW.B) ;
      END;
```

Example

Register in the table *control_R* the date and user who update the table R

```
CREATE TRIGGER T2
      AFTER INSERT OR UPDATE OR DELETE  ON  R
      BEGIN
          INSERT INTO control_R VALUES (user, sysdate)
      END;
```

Access Control

- Each object that is created in SQL has an owner.

  The owner of one database schema is the owner of all its components.

- The owner is the only person who can perform any operation on the object.

- To give other users access to the object, the owner must explicitly grant them the necessary privileges using the GRANT statement.

GRANT

GRANT
{ **ALL** |
SELECT |
INSERT [(*list_of_columns*] |
DELETE |
UPDATE [(*list_of_columns*)] }
ON *table_name* **TO** { *list_of_users* | **PUBLIC** }
[**WITH GRANT OPTION**]

To all the users

Allows the user to pass the privileges to other users

The **REVOKE** statement is used to take away privileges that were granted with the GRANT statement. It has the same syntax than GRANT.

Exercise 1

Consider the following schema:

Actor (**act_code**: char(9), **name**: char(40), **age**: integer)
PK: {act_code} NNV: {name}
Panel_member (**mem_code**: char(9), **name**: char(40), **speciality**: char(15))
PK: {mem_code} NNV: {name}
Role (**role_code**: char(2), **description**: real, **duration**: real, **mem_code** : char(9))
PK: {role_code} NNV: {description, duration, mem_code}
FK: {mem_code} → Panel_member RESTRICTED DELETION, RESTRICTED CASCADE
Performance (**role_code**: char(2), **act_code**: char(9), **per_date**: date)
PK: {role_code, act_code} NNV: {per_date}
FK: {role_code} -> Role ON DELETE CASCADE, RESTRICTED UPDATE
FK: {act_code} -> Actor RESTRICTED DELETION, RESTRICTED UPDATE

Exercise 1

Consider that the following view is created

```
CREATE VIEW Young_Actor
        SELECT A.act_code, A.name, A.age
        FROM Actor A
        WHERE A.age <20;
```

And the following DML instruction:

```
INSERT INTO Young_Actor (act_code, name, age)
        VALUES (18, 'Pepe', 25);
```

Indicate the state of the database after the execution of the instruction above. Assume that before this instruction all tables were empty.

Exercise 2

Given the following DDL command in SQL

```
CREATE TABLE Performance
 ( role_code CHAR(2) PRIMARY KEY
        REFERENCES Role(role_code) ON DELETE CASCADE,
 act_code CHAR(9) PRIMARY KEY,
per_date DATE NOT NULL,
FOREIGN KEY act_code REFERENCES Actor(act_code));
```

Indicate whether the definition of the Performance relation is correct. In case it is not, spot out the errors and write the command again in a correct way

Exercise 3

CS_PAIS (cod_pais:char(5),nombre:char(20))
CP:{cod_pais} VNN:{nombre}
CS_ACTOR (cod_act:char(5),nombre:char(70),fecha_nac:date, cod_pais:char(5))
CP:{cod_act} VNN:{nombre,fecha_nac,cod_pais}
CAj:{cod_pais} → CS_Pais(cod_pais)
CS_LIBRO (cod_lib:char(5),titulo:char(70),anyo:number,autor:char(80))
CP:{cod_lib} VNN:{titulo,autor}
CS_PELICULA (cod_peli:char(5),titulo:char(70),anyo:number, duracion:number,
cod_lib:char(5),director:char(70))
CP:{cod_peli} VNN:{titulo,duracion}
CAj:{cod_lib} → CS_Libro(cod_lib)
CS_GENERO (cod_gen:char(5),nombre:char(30))
CP:{cod_gen}
CS_ACTUA (cod_act:char(5),cod_peli:char(5),papel:char(10))
CP:{cod_act,cod_peli} VNN:{papel}
CAj:{cod_peli} → CS_Pelicula(cod_peli)
CAj:{cod_act} → CS_Actor(cod_act)
CS_CLASIFICACION (cod_gen:char(5),cod_peli:char(5))
CP:{cod_gen,cod_peli}
CAj:{cod_peli} → CS_Pelicula(cod_peli)
CAj:{cod_gen} → CS_Genero(cod_gen)

Exercise 3

We want to add a new attribute called “principales” to the “Película” table to store the number of actors appearing in each movie with the “principal” role. We are going to implement this using a trigger.

a) Analyze the schema to determine the events that can activate this trigger.

b) Write a trigger to modify the database when an update of ”papel” in “Actua” is performed