DATABASES

GS WEB APPLICATION DEVELOPMENT

Teacher: Vanesa Maldonado Guerrero
Curs 2025/26

Introduction to the DDL language

Data Definition Language (DDL) is the part of the SQL language that performs the data definition function for the DBMS. Essentially, it handles the creation, modification, and deletion of database objects (i.e., metadata). Of course, it is also responsible for creating the tables.

The elements of the database, called objects—tables, views, columns, indexes, etc.—are stored in the data dictionary.

Objects are manipulated and created by users. In principle, only **administrators** and **owner users** can access each object, unless the object's privileges are modified to allow access to other users.

It's important to note that no DDL instruction can be undone by a **ROLLBACK** instruction, so extreme caution must be exercised when using them.

In other words, **DDL instructions generate actions that cannot be undone unless a backup or other recovery method is available**.

database creation

Creating the database involves specifying the files and locations that will be used.

Logically, it is only possible to create a database if you have **DBA (Database Administrator) privileges**.

The SQL command to create a database is **CREATE DATABASE**. This command creates a database with the specified name.

Example:

```
CREATE DATABASE prueba;
```

In many systems that is enough to create the database.

In the case of Oracle, we need to specify many more parameters.

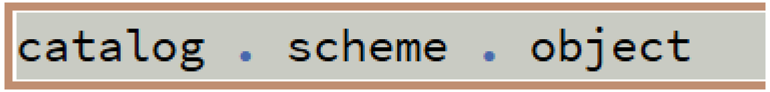
Example:

```
CREATE DATABASE prueba
        LOGFILE test.log
        MAXLOGFILES 25
        MAXINSTANCES 10
        ARCHIVELOG
        CHARACTER SET WIN1214
        NATIONAL CHARACTER SET AL32UTF8
        DATAFILE test1.dbf AUTOEXTEND ON MAXSIZE 500MB;
```

database objects

According to current standards, a database is a set of **objects** designed to manage data. These objects are contained in **schemas** , which are typically associated with a particular user profile.

In standard SQL, there is the concept of a **catalog**, which is used to store schemas, and these are used to store objects. Thus, the fully qualified name of an object would be given by:



```
catalog . scheme . object
```

In other words, **objects** belong to **schemes** and schemes belong to **catalogs**.

Oracle does not have catalogs. However, it does have schemas. Oracle schemas are associated with users: each user has a schema (with the same name as the user) designed to store their objects.

Table creation: table names

They must comply with the following rules (the rules are from Oracle, they may differ in other DBMSs):

- They must begin with a letter
- They must not be longer than 30 characters.
- Only letters of the (English) alphabet, numbers, or the underscore sign are allowed (the $ and # signs are also allowed , but they are used in a special way, so they are not recommended).
- There cannot be two tables with the same name within the same schema (the names can coincide if they are in different schemas)
- It cannot match the name of an SQL reserved word (for example, you cannot call a table SELECT).
- If the name contains spaces or national characters (allowed only in some databases), it is usually enclosed in double quotes. The SQL 99 standard (adopted by Oracle) allows the use of double quotes when naming a table to make it case-sensitive (differentiating between “FACTURAS” and “Facturas”).

These are also the properties that any object name in a database must fulfill (view names, columns, constraints, etc.)

Table creation: Create Table command

This is the SQL command that allows you to create a table. By default, it will be stored in the space and schema of the user who creates the table.

Syntax:

```
CREATE TABLE  [ schema . ] tableName (
        columnName1 dataType [ DEFAULT value ]
        [ restrictions ] [ , ... ]
);
```

Example:

```
CREATE TABLE suppliers ( name VARCHAR( 25 ));
```

Create a table with a single field of type VARCHAR.

The table can only be created if the user has the necessary permissions. If the table belongs to another schema (assuming the user has permission to save tables in that other schema), the schema name is prefixed to the table name.

```
CREATE TABLE otherUser.providers (
  name VARCHAR( 25 )
);
```

Table creation: Create Table command

A default value can be specified for the attribute using the DEFAULT clause .

Example:

```
CREATE TABLE Suppliers (
                          name VARCHAR( 25 ),
                          location VARCHAR(30) DEFAULT 'Palencia'
);
```

Thus, if we add a supplier and do not specify a location, Palencia will be taken as the location of said supplier.

We can use DEFAULT and use system functions.

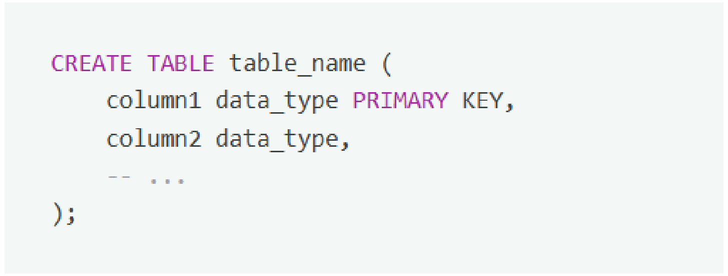
For example:

```
CREATE TABLE Prestamos(
             loan_id NUMBER (8),
             loan_date DATE  DEFAULT SYSDATE
);
```

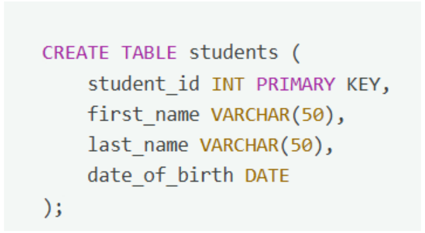
Create Table With a Primary Key

A primary key is essential in SQL databases to uniquely identify each row in a table. It ensures data integrity by containing only unique and non-NULL values.

A table can have only one primary key, which can consist of a single column or multiple columns, called a composite key.



```sql
CREATE TABLE table_name (
    column1 data_type PRIMARY KEY,
    column2 data_type,
    -- ...
);
```



```sql
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    date_of_birth DATE
);
```

Table creation: data types

When creating tables, you must specify the data type for each field. Therefore, we need to know the different data types.

These are:

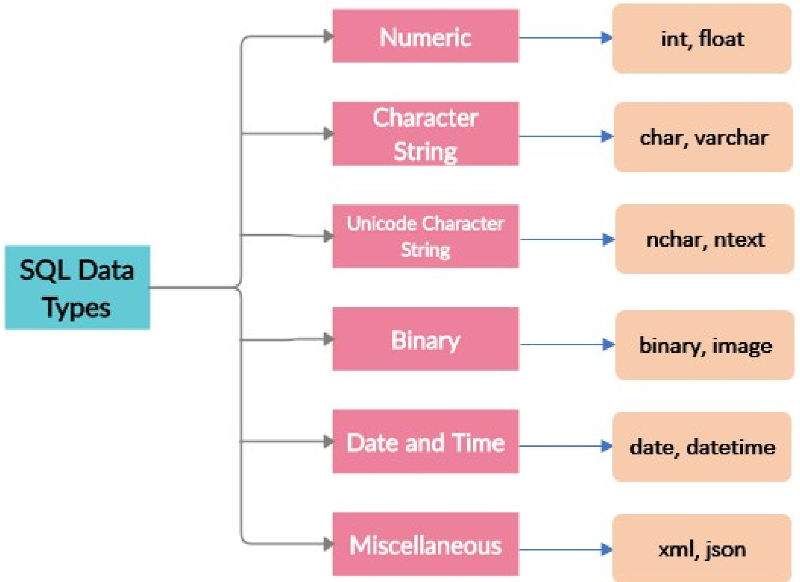


Table creation: Domains

In standard SQL, we have the ability to create domains. The statement that performs this task is CREATE DOMAIN.

Syntax:

```
CREATE DOMAIN name [AS] data_type
[ DEFAULT expression ]
[restrictions [...]]
```

Example:

```
CREATE DOMAIN Tdireccion AS VARCHAR(3);
```

Thanks to that instruction we can create the following table:

```
CREATE TABLE personal(
cod_pers SMALLINT,
nombre VARCHAR(30),
  direccion Tdireccion
)
```

delete tables

The **DROP TABLE** command followed by the name of a table allows you to delete the table in question.

When deleting a table:

- All the data disappears
- Any views and synonyms referring to the table will still exist, but will no longer work (it is advisable to remove them).
- Pending transactions are accepted (COMMIT) in those databases that have the possibility of using transactions.

Logically, only tables for which we have deletion permission can be deleted.

Syntax:

```
DROP TABLE personas
```

Normally, deleting a table is irreversible, and there is no confirmation prompt, so it is advisable to be very careful with this operation.

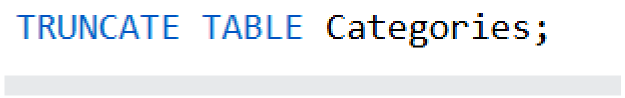
delete table contents

Oracle provides a non-standard command to permanently delete data from a table

It is the **TRUNCATE** command . This command is followed by the name of the table to be deleted.

TRUNCATE deletes the contents of the table, but not the structure itself. It even removes the space occupied by the table from the data file.

Although it only deletes data, it is a DDL instruction (therefore it is not revocable) and it is unconditional: that is, it does not allow deleting only part of a table.



```
TRUNCATE TABLE Categories;
```

Modify tables: rename a table

The standard way (standard SQL) is done:

```
ALTER TABLE oldName RENAME TO newName ;
```

In Oracle, in addition to the previous command, this can be done using the **RENAME** command (which allows renaming any object).
Syntax:

```
RENAME oldName TO newName ;
```

But for consistency it is better to do it the first way (the standard way).

Modify tables: add columns

Syntax:

```
ALTER TABLE tableName ADD( columnName DataType [
Properties ] [ , nextColumnDataType [ properties ] ...)
```

Allows you to add new columns to the table. You must specify their data type and properties if necessary (similar to CREATE TABLE).

The new columns are added at the end; you cannot specify another position (remember that the order of the columns does not matter).

Example:

```
ALTER TABLE facturas ADD ( fecha DATE);
```

Many databases (but not Oracle) require writing the word COLUMN after the word ADD .

Modify tables: delete columns

Syntax:

```
ALTER TABLE tableName DROP( column [ , nextColumn,...] );
```

Remove the indicated column irreversibly.

You cannot delete a column if it is the only column remaining in the table. In that case, you will have to use the **DROP TABLE** command .

Example of column deletion:

```
ALTER TABLE invoices DROP ( date );
```

As in the previous case, in standard SQL you can write the text COLUMN after the word DROP .

Modify tables: modify columns

Allows you to change the data type and properties of a specific column. Syntax:

```
ALTER TABLE tableName MODIFY( column type [ properties ]
       [ columnNext  type [ properties ] ... ]
```

The changes allowed in the columns are (in Oracle):

- Increase precision or width of data types
- The width can only be reduced if the maximum width of a field is exceeded if that column has null values in all records, or if all existing values are smaller than or equal to the new width.
- You can convert from CHAR to VARCHAR2 and vice versa (if the width is not modified)
- You can convert from DATE to TIMESTAMP and vice versa
- Any other changes are only possible if the table is empty.
- If, through this command, we modify the value of the DEFAULT property of a table, this change will only have an effect on the insertion of new rows

```
ALTER TABLE facturas MODIFY( fecha TIMESTAMP);
```

In the case of standard SQL, ALTER is used instead of MODIFY (which can also optionally be followed by COLUMN).

```
ALTER TABLE invoices ALTER COLUMN date TIMESTAMP;
```

Modify tables: rename column

This allows you to rename a column. Syntax:

```
ALTER TABLE tableName
RENAME COLUMN oldName TO newName
```

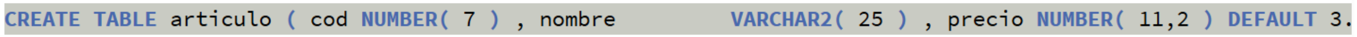
Example:

```
ALTER TABLE facturas RENAME COLUMN fecha TO fechaYhora ;
```

default value

Each column can be assigned a default value during its creation using the **DEFAULT** property . This property can be set during table creation or modification by adding the word **DEFAULT** after the field's data type and then specifying the desired default value.

Example:



```
CREATE TABLE articulo ( cod NUMBER( 7 ) , nombre        VARCHAR2( 25 ) , precio NUMBER( 11,2 ) DEFAULT 3.
```

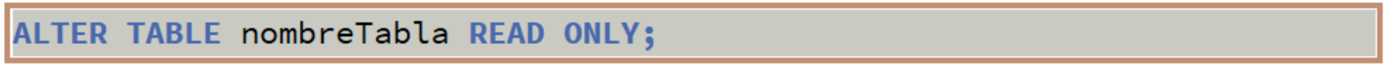
The word DEFAULT can be added during table creation or modification (using the ALTER TABLE command). The value specified by DEFAULT is applied when adding rows to a table, leaving the column value empty. Instead of NULL, the column will be assigned the specified default value.

set tables to read-only mode

This is an option within the **ALTER TABLE** command that restricts a table to only allow read operations (such as the SELECT statement). In other words, it prevents the modification and deletion of its data.

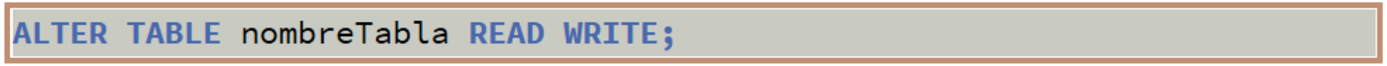
In other words, it does not allow any DML instruction (including the DROP TABLE instruction) to be performed on it.

Syntax:



```
ALTER TABLE nombreTabla READ ONLY;
```

To return the table to its normal state, use:

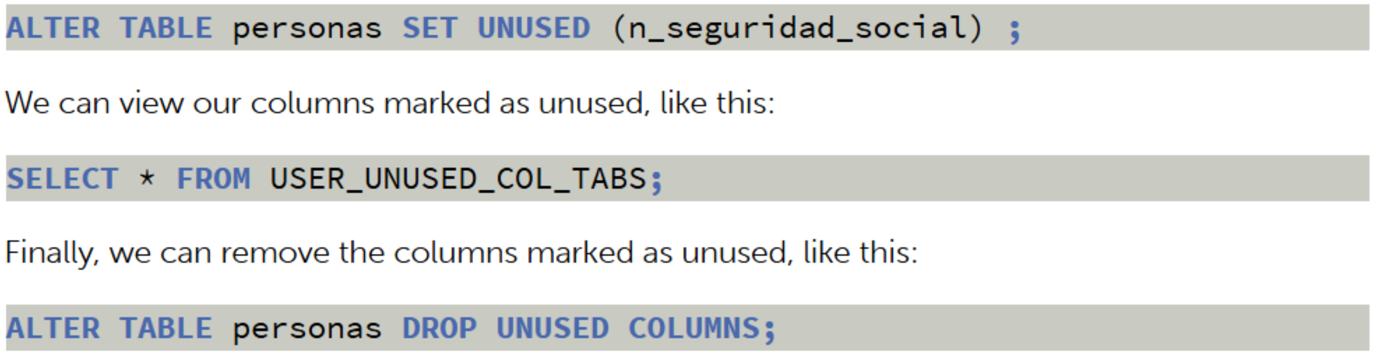


```
ALTER TABLE nombreTabla READ WRITE;
```

modification SET UNUSED

This is a custom Oracle table modification that marks one or more table columns with a " **UNUSED** " marker . You can mark columns that you've detected aren't used frequently.

Example:



```sql
ALTER TABLE personas SET UNUSED (n_seguridad_social) ;
```

We can view our columns marked as unused, like this:

```sql
SELECT * FROM USER_UNUSED_COL_TABS;
```

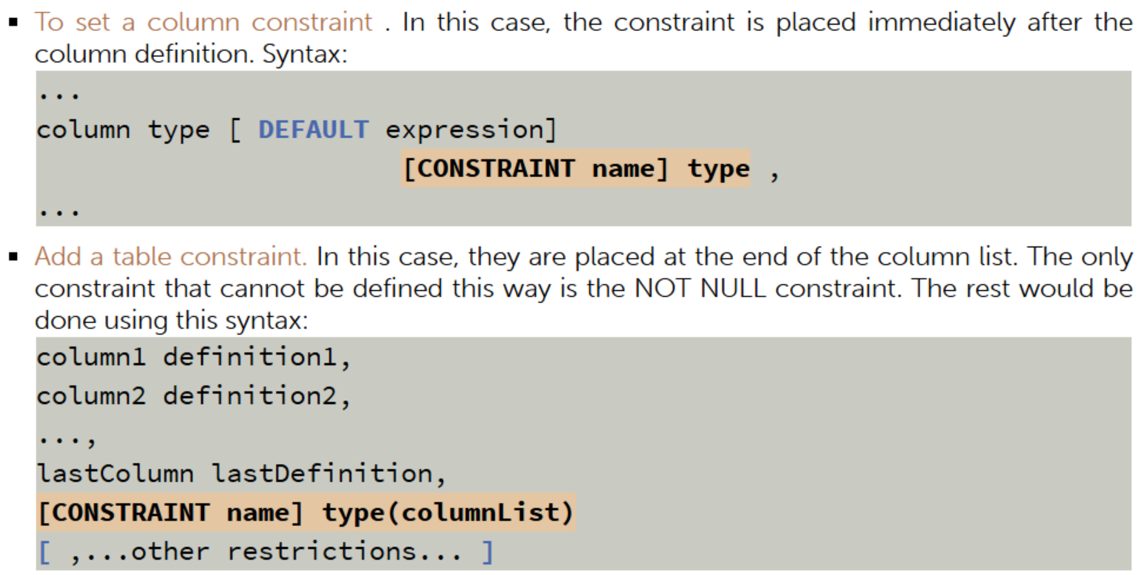
Finally, we can remove the columns marked as unused, like this:

```sql
ALTER TABLE personas DROP UNUSED COLUMNS;
```

establishment of restrictions

A restriction is a mandatory condition for one or more columns of table.

Constraints can be applied when creating (**CREATE**) or modifying (**ALTER**) a table. There are actually two ways to apply constraints:



- To set a column constraint . In this case, the constraint is placed immediately after the column definition. Syntax:

```
...
column type [ DEFAULT expression]
                    [CONSTRAINT name] type ,
...
```

- Add a table constraint. In this case, they are placed at the end of the column list. The only constraint that cannot be defined this way is the NOT NULL constraint. The rest would be done using this syntax:

```
column1 definition1,
column2 definition2,
...,
lastColumn lastDefinition,
[CONSTRAINT name] type(columnList)
[ ,...other restrictions... ]
```

name of the restrictions

It's very good practice to assign a name to each constraint you implement. Otherwise, the database itself will assign a name to the constraint.

Restriction names cannot be repeated for the same scheme; we must use unique names.

A common method is to include the table name, the fields involved, and the type of constraint in the table name. For example, `**piece_id_pk**` could indicate that the `**id**` field in the ` **piece**` table has a primary key.

For example, to make the primary key of the **Students** table the **student code**, the constraint name could be **alu_cod_pk**.

prohibit null

The **NOT NULL** constraint allows you to prohibit null values in a given table. This forces the column to have a value in order for the record to be stored.

It can be **placed** during the creation (or modification) of the field by adding the word NOT NULL after the type:

```
CREATE TABLE cliente(
      dni VARCHAR2(9) CONSTRAINT clientes_nn1 NOT NULL
);
```

The **NOT NULL** constraint is unique in that it can only be placed immediately after the name of the column to which it applies. This is because **NOT NULL** can only be applied to one column at a time.

unique values

UNIQUE constraints ensure that the content of one or more columns cannot have repeated values in different rows.

Example:

```
CREATE TABLE cliente (
      dni VARCHAR2( 9 ) CONSTRAINT clientes_nn1 UNIQUE
);
```

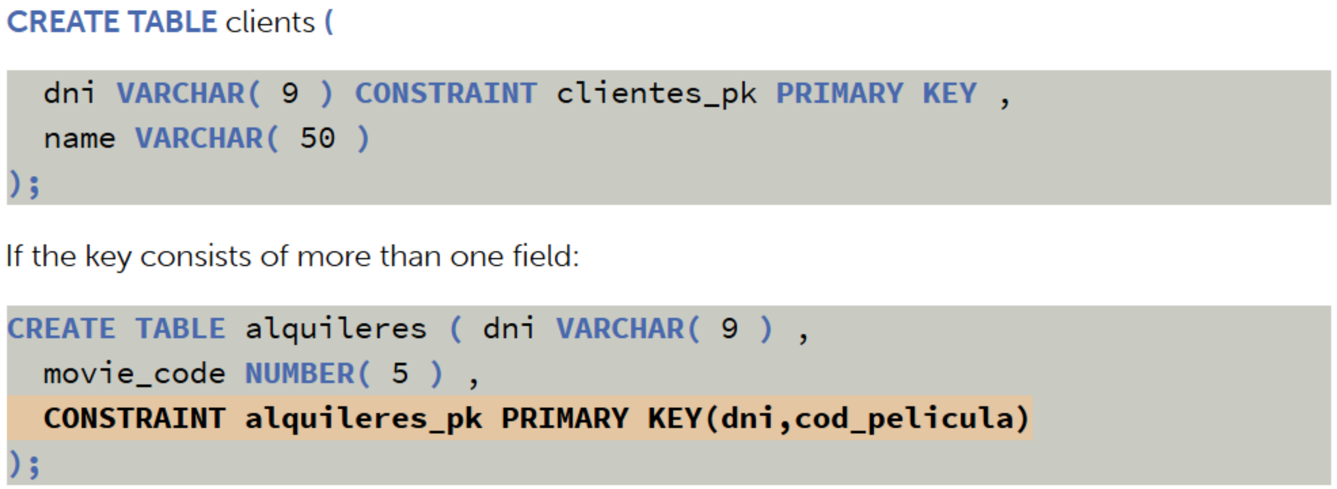
This method allows you to name the constraint. If the repetition of values refers to multiple fields, the method would be:

```
CREATE TABLE alquiler(
  dni VARCHAR2(9),
  cod_pelicula NUMBER(5),
  CONSTRAINT alquiler_uk UNIQUE(dni,cod_pelicula)
);
```

primary key

The **primary key** of a table consists of the columns that identify each record. The primary key ensures that the fields it contains cannot be empty or have duplicate values. Furthermore, these fields become part of the table's main index, which is used for faster access to the data. The primary key also ensures that the fields are NOT NULL (cannot be empty) and that the values are UNIQUE (cannot be repeated).

If the key consists of a single field, it is sufficient to:



```
CREATE TABLE clients (
  dni VARCHAR( 9 ) CONSTRAINT clientes_pk PRIMARY KEY ,
  name VARCHAR( 50 )
);
```

If the key consists of more than one field:

```
CREATE TABLE alquileres ( dni VARCHAR( 9 ) ,
  movie_code NUMBER( 5 ) ,
  CONSTRAINT alquileres_pk PRIMARY KEY(dni,cod_pelicula)
);
```

secondary or foreign key

A secondary or **foreign key** is used to indicate that one or more fields in a table are related to the primary key (or even a candidate key) of another table and, therefore, cannot contain values that are not related in the other table.

This is an example of a foreign key indication:

```
CREATE TABLE alquileres(
  dni VARCHAR2(9)
    CONSTRAINT alquileres_fk1 REFERENCES clientes(dni),
  cod_pelicula NUMBER(5)
    CONSTRAINT alquileres_fk2 REFERENCES peliculas(cod),
  CONSTRAINT alquileres_pk PRIMARY KEY(dni,cod_pelicula)
);
```

When a relationship consists of more than one column, it must (as is always the case with multi-column constraints) be placed after the list of table columns. However, any constraint (whether single-column or multiple) can also be indicated at the end. Example:

```
CREATE TABLE stocks(
  type CHAR2 (9),
  model NUMBER (3),
  n_almacen NUMBER (1)
  quantity NUMBER (7),
  CONSTRAINT stocks_fk1 FOREIGN KEY(type,model)
              REFERENCES pieces ,
  CONSTRAINT stocks_fk2 FOREIGN KEY(store_number)
              REFERENCES warehouses ,
  CONSTRAINT stocks_pk
      PRIMARY KEY (type,model,store_number)
);
```

secondary or foreign key

The downside is that referential integrity causes several problems, due to its side effects .

For example, suppose we link room rentals in a rentals table to the ID number of the person renting. The ID number is the key in the customers table. Well, we can't delete a person from the customers table who has rentals. Nor can we modify their ID number for the same reason.

In response to this, we have the option of applying **special policies**. These policies are keywords that are placed after the REFERENCES clause when adding a FOREIGN KEY constraint.
Thus, the policies that dictate what to do when primary data related to secondary keys is deleted are:

- **ON DELETE SET NULL**. Places null values in all related secondary keys.
- **ON DELETE CASCADE.** Deletes all rows related to the one we deleted.
- **ON DELETE SET DEFAULT**. This sets the default value of the column in the related rows.
- **ON DELETE NOTHING**. It does nothing.

These same commands can be applied when modifying primary keys. These include ON UPDATE DO NOTHING , ON UPDATE CASCADE , ON UPDATE SET NULL , and ON UPDATE SET DEFAULT .

secondary or foreign key

```
CREATE TABLE alquileres(
  dni VARCHAR(9),
  cod_pelicula NUMBER(5),
  CONSTRAINT alquileres_pk PRIMARY KEY(dni,cod_pelicula),
  CONSTRAINT alquileres_fk1 FOREIGN KEY (dni)
        REFERENCES clientes(dni) ON DELETE SET NULL,
  CONSTRAINT alquileres_fk2 FOREIGN KEY (cod_pelicula)
        REFERENCES peliculas(cod) ON DELETE CASCADE
);
```

restricciones de validación

These are constraints that dictate a condition that the contents of a column must meet. A single column can have multiple **CHECKS** in its definition (several CONSTRAINTs would be written consecutively, without commas).
Example :

```
CREATE TABLE income (
  cod NUMBER(5) PRIMARY KEY,
  concepto VARCHAR2(40) NOT NULL,
  amount NUMBER( 11,2 ) CONSTRAINT income_ck1
                              CHECK (importe>0)
                        CONSTRAINT income_ck2
                              CHECK (imports < 8000)
);
```

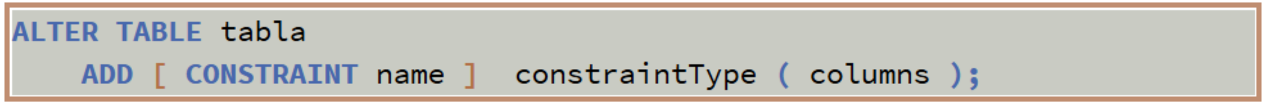
In this case, the CHECK restrictions prohibit adding data whose amount is not between 0 and 8000.

Although it would be more convenient this way:

```
CREATE TABLE income (
  cod NUMBER(5) PRIMARY KEY,
  concepto VARCHAR2(40) NOT NULL,
  amount NUMBER( 11,2 ) CONSTRAINT income_ck1
                       CHECK (importe>0 AND importe<8000)
);
```

add restrictions to a table

You may want to add constraints after creating the table. In that case, use the following syntax:



```
ALTER TABLE tabla
    ADD [ CONSTRAINT name ]  constraintType ( columns );
```

RestrictionType is the text CHECK , PRIMARY KEY , UNIQUE or FOREIGN KEY .
If we want to add a NOT NULL constraint, we do so using ALTER TABLE .. MODIFY and then specifying the constraint we want to add.

remove restrictions

```
ALTER TABLE tabla
        DROP{PRIMARY KEY |UNIQUE(listaColumnas) |
                CONSTRAINT  constraintName }  [ CASCADE ]
```

The PRIMARY KEY option removes a primary key. UNIQUE removes the uniqueness constraint applied to the specified list of columns.

More versatile, the CONSTRAINT option removes the restriction whose name is indicated.

The CASCADE option causes the removal of integrity constraints that depend on the removed constraint to be cascaded, and which would otherwise prevent the removal of that constraint.

In other words, we cannot delete a primary key that has related secondary keys. However, if we specify CASCADE when deleting the primary key, all related FOREIGN KEY constraints will also be deleted.

For example in:

```
CREATE TABLE curso(
  cod_curso CHAR( 7 ) PRIMARY KEY ,
  start_date DATE ,
  end_date DATE ,
  title VARCHAR2( 60 ) ,
  cod_siguientecurso CHAR( 7 ) ,
  CONSTRAINT courses_ck1 CHECK( end_date>start_date ) ,
  CONSTRAINT cursos_fk1 FOREIGN KEY(cod_siguientecurso)
              REFERENCES curso ON DELETE SET NULL);
```

activation and deactivation of restrictions

deactivate restrictions:

Sometimes it's useful to temporarily disable a restriction to bypass the rules it imposes. The CASCADE option also disables the restrictions that depend on the one that was disabled.

The syntax is (in Oracle):

```
ALTER TABLE tabla DISABLE CONSTRAINT nombre [ CASCADE ] ;
```

activate restrictions:

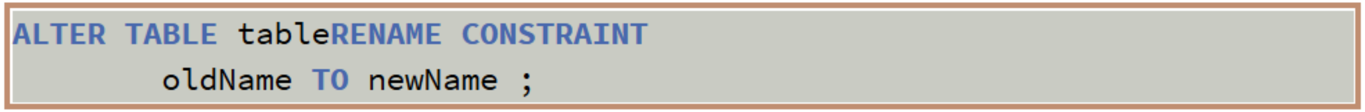
Reactivation is only allowed if the table values meet the activated constraint. If cascading deactivation occurred, each constraint must be activated individually.

Cancel the deactivation:

```
ALTER TABLE tabla ENABLE CONSTRAINT number ;
```

rename restrictions

To do this, use this command:



```
ALTER TABLE tableRENAME CONSTRAINT
        oldName TO newName ;
```