Where are we?

- Week -4: Data definition (Creation of the schema)
- Week -3: Data definition (Triggers)
- Week -2: More SQL queries
- Week -1: Transactions and concurrency in ORACLE.

But don’t forget to work on SQL queries!
Lab III. Part B

SQL: data definition

Database Laboratory
Objectives

• Present the data definition language in SQL
  – Present the syntax for creating tables
  – Present the syntax for modifying table definitions
  – Present the syntax for creating views
  – Present the syntax for granting authorisations
  – Present the syntax for the creation of activity rules (triggers)

• All this in Oracle
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).
Schema Definition (SQL)

\texttt{create schema [schema] [authorization user]}
\texttt{[list_of_schema_elements];}

A schema element can be any of the following:

\begin{itemize}
  \item Domain definition.
  \item Table definition.
  \item View definition.
  \item Constraint definition.
  \item Authorisation definition.
\end{itemize}

Removal of a relational schema definition:

\texttt{drop schema schema \{restrict | cascade\};}
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`

Not in ORACLE
A domain can be associated with a collection of constraints:

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

\[
\text{check } (\text{conditional_expression})
\]

\[
\text{[not] deferrable}
\]

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

- \text{deferrable} indicates that \text{(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.
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]
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.
Table Definition (SQL).

The clause for defining table constraints is the following one:

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

- Must be TRUE or UNDEFINED.
- Cannot include subqueries or references to other tables.
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 ) )
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) );
In Oracle

create table Provider
( id number primary key,
  name varchar(40) not null,
  address char(25),
  city char(30) );

create table Piece
( code number primary key,
  desc varchar(40),
  colour char(20),
  weight float,
  constraint r1 check (colour<>’red’ or weight>100) );

create table Supply
( id number,
  code number references Piece,
  price float,
  primary key (id, code),
  foreign key (id) references Provider(id);
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.

- **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.
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};
```
View

A view is an object which allows the SQL language to define external schemas:

- A view is a virtual table (it has no correspondence at the physical level).
- It can be handled as a basic table.
- A view is defined in terms of other tables or views.
- The updates can be transferred to the original tables (with certain limitations).
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.
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];
  ```
Views in SQL (Examples).

- Given the following database relation:

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

Define a view with only the French cooks:

\begin{verbatim}
CREATE VIEW French AS
  SELECT * FROM Cook WHERE country = "France"
WITH CHECK OPTION
\end{verbatim}

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

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

\begin{verbatim}
CREATE VIEW Report(country, avg_age) AS
  SELECT country, AVG(age) FROM Cook GROUP BY country
\end{verbatim}
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
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.
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.
Views in SQL (updatable views).

Example:

- Given the following relations:

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

  HOUSE(house\_code: \textit{code\_dom}, id: \textit{id\_dom}, addr: \textit{addr\_dom}, rooms: \textit{number})
  PK:\{house\_code\}  FK:\{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
create assertion constraint

check (conditional_expression)

[constraint_check];

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

Constraint R2:

\[ R2 \)  \text{Px: Piece, Sx: Supply } \forall \text{Px} : \text{Piece} (\exists Sx : \text{Supply}(Sx) ( Sx.\text{code}=\text{Px.\text{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.\text{code}=S.\text{code})));

Removal of a constraint

DROP ASSERTION constraint
Notion of trigger.

A trigger is a rule which is automatically activated by certain events and executes a particular action.
**Event-condition-action rules.**

Form of an activity rule:

\[ \text{event} - \text{condition} - \text{action} \]

*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
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]
Triggers in SQL.

Events:

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

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

\[
\text{event} ::= \\
\text{INSERT} \mid \text{DELETE} \mid \text{UPDATE} \ [\text{OF} \ \text{attribute\_name\_list}]
\]
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
## Triggers in SQL.

<table>
<thead>
<tr>
<th></th>
<th>FOR EACH STATEMENT</th>
<th>FOR EACH ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE</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>AFTER</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>
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
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
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 COMPIL
– 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 \) \; Px: \text{Piece}, \; Sx: \text{Supply} \; \forall Px : \; \text{Piece} \; (\exists Sx : \text{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?
Triggers in SQL (Example).

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.
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;
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;
Privilege Definition (SQL).

An user can only perform operations on an object (table or view) if the user has the corresponding privilege.

The operation we can grant privileges on are:

- update (the columns must be specified)
- insert (some columns can be specified)
- delete
- select
- create view: the user needs the privilege over the table expression that makes up the view (the SELECT instruction in the view).
Privilege Definition (SQL).

grant {all | select | insert [(column_commalist)] | delete | update [(column_commalist)]} on object to {user_commalist | public }
[with grant option]

Removing a privilege

revoke [grant option for] {all | select | insert [(column_commalist)] | delete | update [(column_commalist)]} on object to {user_commalist | public }
{restrict | cascade}

In ORACLE some details are different
EXERCISE

FIRST SESSION (everything except “total_loan”):

1. DESIGN (on a paper sheet) THE DATABASE:
   - Ascertain the tables which are required to express the library information.
   - Add the constraints (PK, FK, NNV) you think are needed.
   - Add the specific constraints expressed by the problem.

2. WRITE DOWN THE CORRESPONDING SQL INSTRUCTIONS TO CREATE THE SCHEMA IN WORDPAD (or other text editor):

3. START THE CREATION OF THE SCHEMA IN ORACLE.

4. CHECK THAT THE SCHEMA HAS BEEN CREATED

5. INSERT AND UPDATE SOME INFORMATION TO CHECK THE CONSTRAINTS
EXERCISE

SECOND SESSION:

1. THINK ABOUT HOW TO MAINTAIN THE ATTRIBUTE “Total_loans”.
2. WRITE DOWN THE EVENTS THAT AFFECT “Total_loans”
3. WRITE DOWN THE OPERATIONS THAT SHOULD BE DONE IN EACH EVENT.
4. WRITE DOWN THE CORRESPONDING TRIGGERS TO CREATE THE SCHEMA IN WORDPAD (or other text editor):
5. START THE CREATION OF THE TRIGGERS IN ORACLE.
   • If the trigger is created “con errores de compilación”, use the instruction “SHOW ERRORS” to see the errors, before going on.
6. CHECK THAT THE TRIGGERS HAVE BEEN CREATED
7. INSERT AND UPDATE SOME INFORMATION TO CHECK THAT THE TRIGGERS MAINTAIN THE ATTRIBUTE “Total_loans”.