BACHELOR'S DEGREE IN INFORMATICS ENGINEERING
Academic course: 2018/2019 -- 2nd Year, 2nd term
Subject: Files and Databases
Lab work Report 2: Relational Dynamics and Triggers

uc3m | Universidad Carlos III de Madrid

Lecturer:	PEDRO MIGUEL SUJA GOFFIN		
Group:	88	**Lab User**	FSDB284
Student:	María del Carmen Díaz de Mera Gómez-Limón	**NIA:**	100383384
Student:	Marina Torelli Postigo	**NIA:**	100383479

1. Introduction

For this laboratory assignment we were tasked with a series of queries, views, triggers and designs to be done on the relational design from the previous assignments. The goal was to better understand the advanced elements and relational dynamics of a design.

The other files submitted together with the report are:

- queries_script.sql: SQL code for the requested queries
- views_script.sql: SQL code for the requested views as well as the views done for the external designs
- triggers_script.sql: Implementation in SQL of the triggers

Each section of the report deals with a different type of element, explaining their design and presenting their code in SQL.

- Queries
- Views
- External Design
- Triggers

2. Queries

1. Not by myself: soloists who do not interpret their songs.

- Relational Algebra:

First we use a query to look for those artists that are soloists

$\rho_{soloist}(\pi_{artist.name}\ \sigma_{members.group\_name \text{ IS NULL}}$ (ARTIST]* MEMBERS))

Then we look for the artists that have interpreted their songs on the radio

$\rho_{playedartist}(\pi_{writer}$ (PLAYBACKS*TRACKS))

The result is the difference between the two previous subqueries

π_{name} (soloist) - π_{name} (playedartists)

- SQL Code:

```
WITH
soloist AS (SELECT artists.name name
                FROM    ARTISTS    LEFT    OUTER    JOIN    MEMBERS    ON
            artists.name=members.group_name
                WHERE members.group_name IS NULL),
playedartists AS (SELECT DISTINCT writer name FROM PLAYBACKS NATURAL JOIN
TRACKS)
SELECT name FROM soloist MINUS SELECT name FROM playedartists;
```

- Tests

We run the different subqueries independently and they run as expected

2. Still standing: Last date and time each group was played on the radio.

- Relational Algebra:

This query organises in a descendant way the dates each artist was played on the radio. For that we will use the function row_number() over a partition by artist ordering by playdatetime.

$\rho_{\text{dates}}(\pi_{\text{artist, playdatetime, row\_number() num}}$ (PLAYBACKS*DISCS))

From this query we will take the values where the row_number()=1 so that we can determined the last time each artist was played on the radio.

$\pi_{\text{artist, last\_date}}\ \sigma_{\text{num=1}}$ (dates))

- SQL code:

```
WITH
dates AS (SELECT artist,
            playdatetime,
            row_number() OVER (PARTITION BY artist ORDER BY playdatetime
            DESC) num
            FROM PLAYBACKS JOIN DISCS USING(isvn))
SELECT artist, TO_CHAR(playdatetime, 'DD-MM-YY HH:MM') last_date
            FROM dates
            WHERE num=1;
```

- Tests

When we run the query we get 692 rows, while there are 695 artists in total. Running the query *SELECT name FROM ARTISTS MINUS SELECT artist FROM PLAYBACKS JOIN DISCS USING(isvn)* we get 3 rows selected which are the groups that haven't been played in the radio so 3+692=695 which are the total artists.

Then we have taken some rows from the result, for example the artist ‘Zete’ which was last played 10-11-18. We run the query *SELECT max(playdatetime) FROM PLAYBACKS JOIN DISCS USING(isvn) WHERE artist='Zete'* and effectively get the date 10-11-18.

3. Revival Channels: broadcaster (s) that a) plays the oldest themes (highest average age of played songs); and b) the one playing more often old themes (over 30 yo).

- Relational Algebra:

A) First we get the average date grouping all the playbacks by radio and ordering them descending.

$\rho_{broadcasters} (\pi_{station,\ avg(rel\_date)}\ \gamma_{station}\ \tau_{avg\_date}\ (PLAYBACKS*DISCS))$

Then we just have to choose the one at the first row, which has the oldest average of the publication date of the albums playes

$\pi_{station,\ avg\_date}\ \sigma_{rownum=1}\ (broadcasters)$

B) Similar to the previous query but using the function count(‘X’) and adding the condition that they must be older than 30 years.

$\rho_{broadcasters}(\pi_{station,\ count('x')\ old\_plays}\ \sigma_{sysdate-playdatetime>=30}\ \gamma_{station}$

$\tau_{old\_plays}\ (PLAYBACKS))$

$\pi_{station,\ old\_plays}\ \sigma_{rownum=1}\ (broadcasters)$

- SQL:

```
-- A) plays the oldest themes (highest average age of played songs)
WITH broadcasters AS (SELECT station,
                -- We convert the dates to julian format and then to
          number to calculate the average
                to_date(round(avg(to_number(to_char(rel_date,
          'J')))),'J') avg_date
                FROM PLAYBACKS JOIN DISCS USING (isvn)
                GROUP BY station
                ORDER BY avg_date ASC)
SELECT station, avg_date FROM broadcasters WHERE ROWNUM = 1;

-- B) the one playing more often old themes (over 30 yo).
WITH
broadcasters AS (SELECT station, count('X') old_plays
                FROM PLAYBACKS JOIN DISCS USING (isvn)
                -- Compare the publication date with the current date to
take the ones over 30 years old
```

```
          WHERE  to_number(to_char(sysdate,'YYYY'))  -  to_number(
     to_char(rel_date, 'YYYY'))>=30
          GROUP BY station
          ORDER BY old_plays DESC)
SELECT station, old_plays FROM broadcasters WHERE ROWNUM = 1;
```

- Tests

When we run the two queries we get the same radio station, Radio IP with an average publication date of their plays of 28/02/84 and 2340 plays of old songs. If we use only the subqueries broadcasters to get the whole list we see that it's the one with the oldest average date and with the most plays, so the result of the query is correct

4. Trending catchy: singles most listened to during yesterday.

- Relational Algebra:

We join the tables playbacks and discs and group and count the discs that were played on the radio yesterday (sysdate-1), whose format is 'S' representing Singles.

$\pi_{\text{album, isvn, count('x') nplays}}\ \sigma_{\text{playdatetim= sysdate-1 AND format ='S'}}\ G_{\text{album, isvn}}$

τ^{nplays} (PLAYBACKS*DISCS)

- SQL Code:

```
SELECT album, isvn, count('x') nplays
FROM PLAYBACKS JOIN DISCS USING (ISVN)
-- We only consider playbacks done yesterday
WHERE playdatetime >= sysdate-1 AND playdatetime < sysdate
-- The album format has to be a single
AND format='S'
GROUP BY album, isvn
ORDER BY nplays DESC;
```

- Tests

When we run the query we get no rows selected. Using the query *SELECT DISTINCT playdatetime FROM PLAYBACKS ORDER BY playdatetime DESC* we can see that it's because there are no plays after 31/12/18. To check the validity of the query we changed the date in the condition to another one, for exaple *WHERE playdatetime >= '25/10/18' AND playdatetime < '26/10/18'* and effectively get a list of singles and their number of plays from that day.

5. Ruling stone: longest-lived group among those in the base.

- Relational Algebra:

The first two subqueries will get all the members that have been part of a group and get the earliest start time from all of them and the latest end time. For that we use the function row_number() over a partition by group_name ordering by the start or end time.

$\rho_{\text{beginning}} (\pi_{\text{group\_name, start\_g, row\_number() num\_beg}} (\text{MEMBERS}))$

$\rho_{\text{ending}} (\pi_{\text{group\_name, end\_g, row\_number() num\_end}} (\text{MEMBERS}))$

From those subqueries we will take the values from the members that have row_number()=1 group in both start and end and represent the bounds for the existence of the group. With them we calculate the difference to get the time that has existed the group they belong to. This table will be ordered by that time so that we get the top of all artists.

$\rho_{\text{top\_time}}(\pi_{\text{group\_name, (end\_g-start\_g) livetime}}\ \sigma_{\text{num\_beg=1 AND num\_end=1}}$

$\top_{\text{livetime}}$ (beginning*ending))

From the previous subquery, we select the top row to get the longest lived group.

$\pi_{\text{group\_name, livetime}}\ \sigma_{\text{ROWNUM=1}}(\text{top\_time})$

- SQL Code:

```
WITH
beginning AS (SELECT group_name, start_g, row_number() OVER (PARTITION BY
group_name ORDER BY start_g ASC) num_beg FROM MEMBERS),
ending AS (SELECT group_name,
      -- If the member is still part of the group (end_g is null) we take
the current date to calculate the time the group has existed.
      (CASE
            WHEN end_g IS NOT NULL THEN end_g
            ELSE sysdate
      END) end_g,
      row_number()  OVER  (PARTITION  BY  group_name  ORDER  BY  end_g  DESC)
num_end
      FROM MEMBERS),
top_time AS (SELECT group_name, (end_g-start_g) livetime
            FROM beginning JOIN ending USING (group_name)
            -- We  take  the  lowest  starting  point  from  all  the  members
      that have taken part in the group and the highest ending point)
            WHERE num_beg=1 AND num_end=1
            ORDER BY livetime DESC)
-- From the previous subquery we only take the top group
SELECT group_name, livetime FROM top_time WHERE ROWNUM=1;
```

- Tests

When we run the query we get the group 'La Ciudadela', which has existed for 16209 days. Running the query without the condition *WHERE ROWNUM=1* we get the full list where we can check that it is indeed the group which has existed for the longest.

3. Explicitly required views

- **Top-sales of banned songs:** three best-seller singles that were not played on radio in the previous month (up to 30 days before).

  - Relational Algebra

  For this query we will first get the songs that haven't been played in the radio in the last 30 days. We will label them as 'banned'.

  ρ_{banned} ($\pi_{\text{title\_s, trackN, side, isvn}}$ $\sigma_{\text{(trackN, side, isvn) NOT IN (}\pi\text{ trackN,side,isvn } \sigma\text{sysdate-playdatetime <31 (Playbacks)))}}$ (Tracks))

  Then we will get another query labelled as 'best-sellers'. where we will order all tracks by their sales, counting how many times their identifier appears in the natural join of the tables tracks, discs and sale_line, which show every sale of all albums.

  $\rho_{\text{best\_sellers}}$ ($\pi_{\text{title\_s, trackN, side, isvn, count('X') sales}}\sigma_{\text{discs.format='S'}}$ $\varsigma_{\text{title\_s,trackN,side,isvn}}$ τ_{sales}(Tracks*Discs*Sale_Line)

  With those two queries we can obtain the full top of sales of songs that haven't been played in the radio in the last 30 days by means of the intersection

  $\rho_{\text{top\_sales}}$(best_sellers $\cap$ banned)

  Once we have all rows, ordered by sales, the last query consists on selecting the first three rows from top_sales join with the sales from the query best_sellers

  $\pi_{\text{title\_s, trackN, side, isvn, sales}}\sigma_{\text{rownum<=3}}$(top_sales*best_sellers)

  - SQL Code

```
CREATE OR REPLACE VIEW top_sales AS
WITH
banned AS (SELECT DISTINCT title_s, trackN, side, isvn
      FROM TRACKS
      -- They are the ones not played in the radio in the last 30 days
      WHERE  (trackN,  side,  isvn)  NOT  IN  (SELECT  trackN,  side,  isvn  FROM
PLAYBACKS WHERE ((sysdate-playdatetime) < 31))),
best_sellers AS (SELECT title_s, trackN, side, isvn, count('X') sales
      FROM TRACKS NATURAL JOIN DISCS NATURAL JOIN SALE_LINE
      -- We only have to consider singles
      WHERE discs.format = 'S'
      -- We have to count the number of appearances of each distinct single
      GROUP BY title_s, trackN, side, isvn
      -- The best-sellers appear on top
      ORDER BY sales DESC),
top_sales AS (SELECT title_s, trackN, side, isvn FROM best_sellers INTERSECT
SELECT title_s, trackN, side, isvn FROM banned)
-- We get the top three from top sales with their number of sales from the
query best-sellers
SELECT  title_s,  trackN,  side,  isvn,  sales  FROM  top_sales  JOIN  best_sellers
USING(title_s, trackN, side, isvn) WHERE ROWNUM <= 3;
```

- Tests

  - ➔ Retrieval: With *SELECT * FROM top_sales* we effectively get a table with three rows with a song and its number of plays ordered descending
  - ➔ Deletion: Not allowed in this view
  - ➔ Insertion: Not allowed
  - ➔ Update: Not allowed

This query is designed for the retrieval of data and doesn't allow other operations because it contains information from several tables.

- **Top five week-peak**: the five artists (either soloists or groups) that have been the most listened to (most played on radio) in the last seven days.

  - Relational Algebra

    First we will get the full top of artists, we join the tables with the plays with the tracks and their albums, which contain the name of the artist. We only consider plays that are in the range of 7 days from the current date and group by artist to count their appearances as the number of plays.

    $\rho_{top\_artists}(\pi_{artist,count('X')}\sigma_{(sysdate-datetime)<7}\ \mathcal{G}_{artist}\ \tau_{count('X')}$ (Playbacks*Tracks*Discs))

    The final query takes the first five rows from top_artists

    $\pi_{artist,nPlays}\sigma_{rownum<=5}(top\_artists)$

- SQL Code

```
CREATE OR REPLACE VIEW week_peak AS
WITH top_artists AS (SELECT artist, count('X') nPlays
      FROM PLAYBACKS NATURAL JOIN TRACKS NATURAL JOIN DISCS
      -- We consider only the plays of last week
      WHERE (sysdate-playdatetime)<7
      GROUP BY artist
      -- We want the artists with the most plays at the top
      ORDER BY nPlays DESC)
SELECT artist, nPlays FROM top_artists WHERE ROWNUM <=5;
```

- Tests

  - ➔ Retrieval: With *SELECT * FROM week_peak* we get no rows selected, but with a quick check we get that it's because there are no playbacks registered with a date from the previous week. If we take out the condition, we get five artists and their number of plays ordered descending.
  - ➔ Deletion: Not allowed in this view
  - ➔ Insertion: Not allowed
  - ➔ Update: Not allowed

As the previous view, it is only designed for the retrieval of data

- **SoundBoss**: manager whose vinyls are the most listened to (for each month).

  - Relational Algebra

First we will design a query to get all managers with their listens for each month. We will get the information with a join from the playbacks table with the discs table, grouping by the manager´s name and surname, to count the number of plays they have.

$\rho_{\text{sales\_managers}}$ ($\pi_{\text{mng\_name, mng\_surn1, month, count('X')}}$ $\mathcal{G}_{\text{mng\_name, mng\_surn1, month}}$ $\top_{\text{playdatetime}}$ (Playbacks*Discs))

Once we have the managers and their plays we do another query to obtain the position of each manager in the top with a partition by month ordering by plays

$\rho_{\text{top\_months}}$($\pi_{\text{mng\_name,mng\_surn1,month,nPlays, row\_number()}}$ (sales_managers)

Once we have the full top, with the final query we just have to select those managers whose row_number() is one over the partition and have therefore the most sales of that month

$\pi_{mng\_name,mng\_surn1,month,nPlays}\sigma_{row\_number()=1}(top\_months)$

- SQL Code

```
CREATE OR REPLACE VIEW soundboss AS
WITH
sales_managers AS (SELECT mng_name, mng_surn1,
      to_char(playdatetime, 'MM-YYYY') month,
      count(*) nPlays
      FROM PLAYBACKS JOIN DISCS USING(isvn)
      -- We want to count the appearance of the managers in each month
      GROUP BY mng_name, mng_surn1, to_char(playdatetime, 'MM-YYYY')
      ORDER BY month ASC),
top_months  AS  (SELECT mng_name,  mng_surn1,  month,  nPlays,  row_number()  OVER
(PARTITION BY month ORDER BY nPlays DESC) top FROM sales_managers)
SELECT mng_name, mng_surn1, month, nPlays FROM top_months WHERE top=1;
```

- Tests
  - ➔ Retrieval: With *SELECT * FROM soundboss* we get 24 rows selected, one for each month in 2017 and 2018 (the years with data recorded) with the manager and the number of plays they got.
  - ➔ Deletion: Not allowed in this view
  - ➔ Insertion: Not allowed
  - ➔ Update: Not allowed

- **Wreck-hit:** least listened to single(s) (for each month).

  - Relational Algebra

We do a similar thing to the previous view, first we get the plays of each single for each month from the natural join of the table singles and playbacks

$\rho_{top\_singles}(\pi_{isvn,month,count('X')}\,\mathcal{G}_{isvn,month}\,\top_{month}(Singles*Playbacks))$

Then we use the function row_number() over a partition by month ordering by the number of plays to get the position of each single in a subquery

$\rho_{top\_month}(\pi_{isvn,\ month,nPlays,\ row\_number()}(top\_singles)$

With the final query, we select all the singles from the previous subquery which are at the position 1 of each month, (row_number() returns 1)

$\pi_{isvn,\ month,nPlays}\sigma_{row\_number()=1}(top\_month)$

- SQL Code

```
CREATE OR REPLACE VIEW wreck_hit AS
WITH
top_singles AS (SELECT isvn, to_char(playdatetime, 'MM-YYYY') month, count('X') nPlays
      FROM SINGLES NATURAL JOIN PLAYBACKS
      GROUP BY isvn, to_char(playdatetime, 'MM-YYYY')
      ORDER BY month ASC),
top_month AS (SELECT isvn, month, nPlays, row_number() OVER (PARTITION BY month ORDER BY nPlays ASC) top FROM top_singles)
SELECT isvn, month, nPlays FROM top_month WHERE top=1;
```

- Tests

  - ➔ Retrieval: With *SELECT * FROM week_peak*, as in the previous view, we get 24 rows selected for each month with the isvn representing the single and their number of plays.
  - ➔ Deletion: Not allowed in this view
  - ➔ Insertion: Not allowed
  - ➔ Update: Not allowed

4. External design

- **Client Usage:**

For this functionality we will create a new role called client, which will have access to two newly created views, one to access their personal information (personal_data) and the other to see their personalized recommendations.

- Personal information

Clients should have access to the personal information as well as their orders, they should also be able to change and update it. To identify the client we will use the username as dni, which we will get using the query *(SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL).*

For this functionality we will create two views, one for each table the client needs access to, so that they are able to insert and update through those views

The relational algebra design and code for the two views is:

$\pi_{\text{e\_mail, name, surn1, surn2, birthdate, phone, address}}\sigma_{\text{dni=username}}(\text{Clients})$

```
CREATE OR REPLACE VIEW client_data AS
SELECT e_mail, name, surn1, surn2, birthdate, phone, address FROM
CLIENTS
```

```
        WHERE  dni=(SELECT  SYS_CONTEXT  ('USERENV',  'SESSION_USER')  FROM
DUAL);
```

$\pi_{isvn,\ e\_mail,\ order\_s,\ qtty,\ delivery}\sigma_{dni=username}$(Sale_Line* Clients)

```
        CREATE OR REPLACE VIEW client_orders AS
        SELECT ISVN, e_mail, order_s, qtty, delivery
        FROM SALE_LINE NATURAL JOIN CLIENTS
        WHERE  dni=(SELECT  SYS_CONTEXT  ('USERENV',  'SESSION_USER')  FROM
DUAL);
```

- Recommendations

For this views first we will get the orders of the client, with their publication date and ordered by the date of order to get the most recent one purchased, as in the previous view, we will use the username as dni of the client with the same query mentioned.

ρ_{orders} ($\pi_{album,\ isvn,\ order\_s,\ rel\_date}\sigma_{dni=username}\ \top_{order\_s,\ rel\_date}$(Discs * Sale_Line * Clients))

We will take the information from the natural join of the tables discs, sale_line are clients, where we can obtain the information of the albums the specific client has purchased.

From this query the first row corresponds to the album we will base our recommendations in.

Next we will obtain all the recommendations. We will look for albums which the client hasn't purchased, so their isvn won't be in the selection of the isvn of the previous query (*isvn NOT IN SELECT isvn FROM orders*).

To look for the ones closest to the publication date of the last purchase we will compare the dates with <= for simplicity, only considering albums published before and not after. However, if the client hasn't purchased anything yet, we will consider the current date (sysdate). For this we will use a case selection. If the previous query returns rows (exists), we will use the value from the first row (the last album purchased), else we will consider sysdate.

We will order by rel_date so the closest in publication appear first.

ρ_{recs} ($\pi_{album,\ artist,\ isvn,\ rel\_date}\sigma_{(rel\_date<=sysdate\ ||\ \pi_{order\_s}\sigma_{rownum=1}(orders))}$ && (isvn NOT IN (π_{isvn}(orders))) $\top_{rel\_date}$(Discs))

The final query consists in selecting the first five rows from the recommendations query.

$\pi_{album,\ artist,\ isvn}\ \sigma_{rownum\ <=\ 5}$ (recs)

```
CREATE OR REPLACE VIEW recommendations AS
WITH
-- All albums purchased by the client
```

```
orders AS (SELECT album, isvn, order_s, rel_date
      FROM DISCS NATURAL JOIN SALE_LINE NATURAL JOIN CLIENTS
      -- The username is the dni representing the client
      WHERE  dni=(SELECT  SYS_CONTEXT  ('USERENV',  'SESSION_USER')  FROM
DUAL)
      -- We consider first the last albums purchased and published
      ORDER BY order_s, rel_date DESC),
-- All recommended albums
recs AS (SELECT album, artist, isvn, rel_date FROM DISCS
      WHERE rel_date <= (CASE
            --  We  consider  the  last  album  purchased  (first  row  from
orders)
            WHEN EXISTS (SELECT order_s FROM orders WHERE ROWNUM=1) THEN
                  (SELECT order_s FROM orders WHERE ROWNUM=1)
            --  If  the  client  doesn't  have  any  orders  we  consider  the
current date
            ELSE sysdate
      END)
      -- The album mustn't have been already purchased by the clie
      AND isvn NOT IN (SELECT isvn FROM orders)
      -- The closest albums to the selected date go first
      ORDER BY rel_date DESC)
-- We select the first five recommendations
SELECT album, artist, isvn FROM recs WHERE ROWNUM<=5;
```

Once we have created the two views we should create the role as the database administrator and grant them access to this two views.

```
CREATE ROLE client NOT IDENTIFIED;
```

The client will be able to read and write on their personal information.

```
GRANT UPDATE, INSERT, SELECT ON client_data TO client;
GRANT UPDATE, INSERT, SELECT ON client_orders TO client;
```

However, they will only be able to see their recommendations, not alter them.

```
GRANT SELECT ON recommendations TO client;
```

- **Warehouse Usage**

As in the previous design, we will create a new role with access to some new views. To store the information of which worker has prepared which order we will add a new column to the sale_line table, the column worker which will be a 8-digit number identifying the worker that has prepared the order.

```
ALTER TABLE SALE_LINE ADD worker NUMBER(8);
```

- Worker orders

This view will allow the worker to see the orders that hasn't been delivered yet and therefore he has to prepare. As with clients, we will use the same query to obtain the username as the worker identifier. The view gives access to all the columns from sale_line corresponding to the worker, and with a delivery date either null or bigger that the current date (and therefore not delivered)

$\sigma_{\text{worker=username AND (delivery IS NULL OR delivery > sysdate)}}(\text{Sale\_Line})$

```
CREATE VIEW worker_orders AS
SELECT * FROM SALE_LINE
WHERE delivery IS NULL OR delivery > sysdate
AND worker = (SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL);
```

- Productivity

We will group the orders of the worker by date and count them to check each day's productivity. We will only show the days from the previous month (31 days prior).

$\pi_{\text{order\_s, count('X')}}\sigma_{\text{worker=username AND (sysdate-order\_s)<=31}}G_{\text{order\_s}}(\text{Sale\_Line })$

```
CREATE VIEW productivity AS
SELECT to_char(order_s, 'DD-MM-YYYY') day, count('X') nOrders
FROM SALE_LINE
WHERE  worker  =  (SELECT  SYS_CONTEXT  ('USERENV',  'SESSION_USER')  FROM
DUAL)
-- We will consider the 31 days prior to the current date
AND (sysdate-order_s) <=  31
GROUP BY to_char(order_s, 'DD-MM-YYYY');
```

- Employee of the month

We will first consider a query that gives as the number of orders each employee has done for each month, with the count function grouping by worker and month as the number of orders.

$\rho_{\text{orders}}(\pi_{\text{worker, order\_s, count('X')}}G_{\text{worker, order\_s}}(\text{Sale\_Line }))$

From that query we will add the row number over a partition by month and ordering by the number of orders done from each worker that month. That way we will get the top chart for each month. We will only consider the months from last year, so the difference between the sysdate and the date has to be 365.

$\rho_{\text{top\_months}}(\pi_{\text{worker, month, nOrders, row\_number()}}\sigma_{\text{sysdate-month <= 365}}(\text{Orders }))$

Finally, for the last query we just have to take the rows from the query top_months where row_number() has taken the value one, and is therefore the worker with the most orders for that month.

$\pi_{\text{worker, month, nOrders}}\ \sigma_{\text{row\_number()=1}}\ (\text{top\_months})$

```
CREATE VIEW employee_month AS
```

```
      WITH
      orders AS (SELECT worker, to_char(order_s, 'MM-YYYY') month,
                  count('X') nOrders
                  FROM SALE_LINE
                  GROUP BY worker, to_char(order_s, 'MM-YYYY')),
      top_months AS (SELECT worker, month, nOrders,
                  -- We partition by month and order by number of orders to
get the position of each worker in the top of the month
                  row_number() OVER (PARTITION BY month ORDER BY nOrders DESC)
      top
                  FROM orders
                  -- We only consider the previous year
                  WHERE ((sysdate-to_date(month, 'MM-YorYYY'))<=365))
      -- We select all the workers that were at the top for their month
      SELECT worker, month, nOrders FROM top_months WHERE top=1;
```

The last thing we have to do is create the role and grant the permissions to the views

```
      GRANT UPDATE, INSERT, SELECT ON worker_orders TO worker;
      GRANT SELECT ON productivity TO worker;
      GRANT SELECT ON employee_month TO worker;
```

The only one they will be able to alter is their orders, the other ones are view only.

5. Explicitly required Trigger

For this section we have designed 4 triggers (from which only 3 were compulsory) and implemented all of them in SQL, when only one was required.

- **Format**: control redundancy due to 'format' attribute (every single or album corresponds to a disc with the correspondent format, either 'S' or 'L').

For this trigger we considered a functionality that checked the format of a disc when it was updated or inserted, so that it only accepts the values 'S' and 'L', if any other value is stated, the trigger raises an exception and prints a message is printed on the screen stating the problem and therefore prevent the wrong insertion or update.

- Table: Discs, column format
- Event: Insert or update
- Temporality: before
- Granularity: For each row
- Action:

  Declare exception bad_format

  Check if the inserted or update format (:NEW) equals either 'S' or 'L'. If not, raise bad_format exception.

  In the case of exception, print a message warning about the wrong format

```
CREATE OR REPLACE TRIGGER control_format
BEFORE INSERT OR UPDATE OF format ON DISCS
```

```
FOR EACH ROW
DECLARE
      bad_format EXCEPTION;
BEGIN
      -- Check if the format is correct
      IF (:NEW.format != 'S' OR :NEW.format != 'L') THEN RAISE bad_format;
      END IF;
EXCEPTION
      WHEN bad_format THEN DBMS_OUTPUT.PUT_LINE('Wrong format');
END;
```

- **Golondrinajes**: implement *on delete set default* set on the design (and not implemented).

We want to implement the on delete set default functionality on the playbacks table, so that when we delete a row from the table tracks,, the row that references the track isn't deleted, but instead set to the default values, so that it now references the song 'Moments' by 'Golondrinajes'.
For that we will implement a trigger in the table tracks so that when something is deleted in that table, its references in playbacks are updated (trackN, side and isvn being the foreign key)

- Table: Tracks
- Event: Deletion
- Temporality: After
- Granularity: For each row
- Action:
  Declare variables to store default variables for the foreign keys we will update.
  Store the default values in the variables.
  Update all rows in the playbacks table referencing the deleted values with the default ones.

```
CREATE OR REPLACE TRIGGER golondrinajes
AFTER DELETE ON TRACKS
FOR EACH ROW
DECLARE
      dfttrack NUMBER(2);
      dftside VARCHAR2(1);
      dftalb NUMBER(8);
BEGIN
      -- First query to get the foreign key that reference the default values
      SELECT trackN, side, ISVN
      INTO dfttrack, dftside, dftalb
      FROM TRACKS NATURAL JOIN DISCS WHERE title_s='Moments' AND
artist='Golondrinajes';
      -- Second query to update the rows from playbacks that reference the
deleted row (represented with :OLD) with the default values
      UPDATE PLAYBACKS
      SET trackN = dfttrack, side=dftside, isvn=dftalb
      WHERE trackN= :OLD.trackN AND side= :OLD.side AND isvn = :OLD.isvn;
END;
```

```
/
```

- **NO refund:** prevent deletion of already delivered orders.

This trigger is implemented on the sale_line table, where orders and their delivery dates are stored. To prevent deletion, we have to do a check before every deletion, and if the order is delivered it will return an exception to prevent the deletion.

- Table: Sale_line
- Event: Deletion
- Temporality: Before
- Granularity: For each row
- Action:

  Declare the exception no_refunds

  Check if the delivery date of the deleted row (:OLD) is smaller than the current date (sysdate), and therefore in the past. If it is, raise the exception no_refunds, if not, do nothing.

  In the case of exception, print a message in the screen telling the user that the order was already delivered

```
CREATE OR REPLACE TRIGGER no_refund
BEFORE DELETE ON SALE_LINE
FOR EACH ROW
DECLARE
      no_refunds EXCEPTION;
BEGIN
      IF (:OLD.delivery <= sysdate) THEN RAISE no_refunds;
      END IF;
EXCEPTION
      WHEN no_refunds THEN DBMS_OUTPUT.PUT_LINE('Cannot delete a delivered
order');
END;
/
```

- **Empty-vinyl**: remove a disc when it does not have songs on one of its sides.

To implement this functionality we have to check for the deletion of songs from the tracks table. When a song is deleted we check if there are still other songs in the same side. If there aren't we delete the disc that track belonged to.

- Table: Sale_line
- Event: Deletion
- Temporality: After
- Granularity: For each row
- Action:

  Delete the disc that contained the deleted track (represented by the isvn) when the query that returns the list of tracks in the same side and album doesn't return any row (not exists).

```
CREATE OR REPLACE TRIGGER empty_vinyl
```

```
AFTER DELETE ON TRACKS
FOR EACH ROW
BEGIN
      DELETE FROM DISCS
      WHERE isvn=:OLD.isvn
      AND NOT EXISTS (SELECT * FROM TRACKS
                  WHERE isvn=:OLD.isvn AND side=:OLD.side);
END;
/
```

6. Concluding Remarks

The start of this lab work was slow, while we were figuring the correct syntax and the right way to do somo of the most complicated operations for the queries. We had a little trouble understanding some of the required functionalities, specially for some of the triggers. The triggers that weren't designed was because we couldn't figure out what was required. For example, the two last ones regarding the periods, we didn't know and couldn't find anywhere in the design the membership periods, so we couldn't design the trigger.

We have dedicated most of the time doing this lab to the research of how to implement the most difficult functionalities. The most useful thing that we found was the function row_number(), which we can use to order things with a partitioning.

In general we have learned a lot about complex queries and have noticed how with each one we implemented we found the next one easier as we were learning and figuring things out.