BS Degree in Applied Mathematics & Computer Science Academic year: 2024/25 - 3rd year, 2nd term Subject: File Structures and Databases Second Assignment's Report: DB development and Querying	uc3m | Universidad Carlos III de Madrid

Lecturer:	Francisco Javier Calle Gómez		
Group:	308	Lab User	fsdb308
Student:	Siro Brotón Gutiérrez	NIA:	100496683
Student:	Yan Chen Zhou	NIA:	100496757
Student:	Ivan Ocheretianyi	NIA:	100487317

1 Introduction

After our last job with foundicu remaking their database into something useful, we are now asked to create some of the necessary utilities for working with that new database "we" created (we will obviously use the given design instead of our previous work). We embark now on another endeavor to try and prove that everything is possible with enough queries.

We are asked to implement 4 different kinds of code:
- Isolated queries to filter and analyze the huge amount of data in the db.
- A package of functions useful to operate in the database as an employee of Foundicu.
- User interfaces (views) to allow users the access to their information, while imposing some reasonable constraints.
- Triggers to maintain some of the properties and utility of the original design after some use.

We will develop each part in a file called **'foundicu_TYPE.sql'** and include the testing separately on **'foundicu_tests.sql'**.

In this assignment we hope to get a deeper understanding of sql and how to work with it.

2 Queries

We implemented the required queries on the file **'foundicu_queries.sql'**

2.1 BoreBooks

The first query consists of 2 parts:

$$\pi_{\text{Title,Author,count}}\ \sigma_{\text{count}>2}\ \mathcal{G}_{\text{Title}}(\text{Editions})$$

to get books with more than 2 languages and:

$$\pi_{\text{Title,Author}}(\text{Editions} \rhd_{\text{ISBN}}(\text{Copies} \ltimes \text{Loans}))$$

to get books with no loans. To get our final query we just join the previous two:

$$(\pi_{\text{Title,Author,count}}\ \sigma_{\text{count}>2}\ Ģ_{\text{title}}(\text{Editions})) * (\pi_{\text{Title,Author}}(\text{Editions} \triangleright_{\text{ISBN}}(\text{Copies} \ltimes \text{Loans})))$$

The code is similar in structure but with sql syntax:

```
Unset
WITH
    B3 AS (
        SELECT COUNT(DISTINCT LANGUAGE), TITLE, AUTHOR
        FROM EDITIONS
        GROUP BY TITLE, AUTHOR HAVING COUNT(DISTINCT LANGUAGE) > 2
    ),
    LB AS (
        SELECT DISTINCT TITLE, AUTHOR
        FROM EDITIONS
        WHERE ISBN IN (
            SELECT DISTINCT ISBN
            FROM COPIES C
            JOIN LOANS L
            ON C.SIGNATURE = L.SIGNATURE
        )                                     -- editions that were loaned at
least once
    )                                         -- books that were laoned at
least once
SELECT DISTINCT B3.TITLE, B3.AUTHOR
    FROM B3
    LEFT JOIN LB
    ON LB.TITLE = B3.TITLE AND LB.AUTHOR = B3.AUTHOR
    WHERE LB.TITLE IS NULL AND LB.AUTHOR IS NULL
;
```

For testing we executed the query and checked that some of those results had, in fact, 3 languages and no loans

2.2 Reports on employees

For this query we need a few different subqueries to calculate the different fields. We will use current_date as the date the request is done, all functions will start with lowercase and attributes with initial uppercase. Also, the first 2 subqueries are not strictly necessary since they can be done directly in the final query, but because it will be too long we have decided to make it more readable by doing them separately.

- The age is obtained like:

$$A \equiv \pi_{\text{Passport, current\_date - Birthdate AS Age}}(\text{Drivers})$$

- For contracted years we do:

  if cont_end is not null: $B \equiv \pi_{\text{Passport, floor((Cont\_end - Cont\_start) / 365.25) AS Contracted\_years}}(\text{Drivers})$

  else: $B \equiv \pi_{\text{Passport, floor((current\_date - Cont\_start) / 365.25) AS Contracted\_years}}(\text{Drivers})$

- For active years we do something similar, but without using the end of the contract:

$$C \equiv \pi_{\text{Driver AS Passport, floor((current\_date - min(Taskdate)) / 365.25) AS Active\_years}}(\text{Services} \bowtie_{\text{FOREIGN KEY}} \text{Loans})$$

- We can calculate the number of stops per year using previous field Active_years like:

$$D \equiv \pi_{\text{Driver AS Passport, count / Active\_years AS Stops\_year}}\ \text{Ģ}_{\text{Driver}}(\text{Services})$$

- The number of loans per year can be calculated doing:

$$E \equiv \pi_{\text{Driver AS Passport, count / Active\_years AS Loans\_year}}\ \text{Ģ}_{\text{Driver}}(\text{Services} \bowtie_{\text{FOREIGN KEY}} \text{Loans})$$

- Finally, the percentage of unreturned loans is obtained in the following way:

$$F \equiv \pi_{\text{Driver AS Passport, count / (Active\_years * Loans\_year) AS Unreturned\_loans}}\text{Ģ}_{\text{Driver}}(\text{Services} \bowtie_{\text{FOREIGN KEY}} \sigma_{\text{Return = Null}}(\text{Loans}))$$

Then we put everything together (joining by passport) and get The query:

$$\pi_{\text{Drivers.fullname, A.Age, B.Contracted\_years, C.Active\_years, D.Stops\_year, E.Loans\_year, F.Unreturned\_loans}}(\text{Drivers}\bowtie A\bowtie B\bowtie C\bowtie D\bowtie E\bowtie F)$$

When implementing the query some of the attributes will use others that have been computed in other subqueries, so the real sql implementation will have to deal with that. We deal with it noticing that there are 2 big groups: the first ones depending on what we have in Drivers (Driver_Data) and the latter, formulated using Services and Loans (Driver_Stats). We can compute first the simpler subqueries to use them for later subqueries.

Unset

```
WITH
    DRIVER_STATS AS (
        SELECT
            CASE
                WHEN SUM(N_LOANS) IS NULL THEN 0
                ELSE SUM(N_LOANS)
            END AS UNRETURNED_LOANS,
            CASE
                WHEN SUM(N_UNRETURNED_LOANS) IS NULL THEN 0
                ELSE SUM(N_UNRETURNED_LOANS)
            END AS TOTAL_LOANS,
            COUNT(*) AS TOTAL_STOPS,
            --  AS FIRST_STOPDATE,
            FLOOR((SYSDATE - MIN(TASKDATE))/365.25) AS ACTIVE_YEARS,
            SERVICES.PASSPORT
        FROM SERVICES
        LEFT JOIN (
            SELECT
                COUNT(RETURN) AS N_LOANS,
                COUNT(CASE WHEN RETURN < TRUNC(SYSDATE) THEN 1 END) AS
N_UNRETURNED_LOANS,
                TOWN, PROVINCE, STOPDATE
            FROM LOANS
            GROUP BY TOWN, PROVINCE, STOPDATE
            -- ORDER BY COUNT(*) DESC
        ) LOANS_BY_SERVICE
        ON SERVICES.TOWN = LOANS_BY_SERVICE.TOWN
            AND SERVICES.PROVINCE = LOANS_BY_SERVICE.PROVINCE
            AND SERVICES.TASKDATE = LOANS_BY_SERVICE.STOPDATE
        GROUP BY SERVICES.PASSPORT
        -- ORDER BY TOTAL_LOANS DESC
    ),
    DRIVER_DATA AS (
        SELECT
            D.PASSPORT,
            D.FULLNAME,
            FLOOR(
                MONTHS_BETWEEN(TRUNC(sysdate), D.BIRTHDATE)/12
            ) AS AGE,
            CASE
                WHEN D.CONT_END IS NULL THEN FLOOR((TRUNC(SYSDATE) -
D.CONT_START)/365.25)
                ELSE FLOOR((D.CONT_END - D.CONT_START)/365.25)
            END AS CONTRACTED_YEARS
        FROM DRIVERS D
    )
SELECT
    D.FULLNAME,
    D.AGE,
    D.CONTRACTED_YEARS,
    S.ACTIVE_YEARS,
    CASE
        WHEN S.ACTIVE_YEARS = 0 THEN S.TOTAL_STOPS
        ELSE S.TOTAL_STOPS/S.ACTIVE_YEARS
    END AS STOPS_PER_ACTIVE_YEAR,
    CASE
        WHEN S.ACTIVE_YEARS = 0 THEN S.TOTAL_LOANS
        ELSE S.TOTAL_LOANS/S.ACTIVE_YEARS
    END AS LOANS_PER_ACTIVE_YEAR,
    CASE
        WHEN S.TOTAL_LOANS = 0 THEN 0
        ELSE S.UNRETURNED_LOANS/S.TOTAL_LOANS
    END AS RATE
FROM DRIVER_DATA D
JOIN DRIVER_STATS S
ON D.PASSPORT = S.PASSPORT
ORDER BY FULLNAME DESC
;
```

For the testing we tried a few cases, from which the following is the most instructive: We introduce a new driver with a finished contract of more than 2 years and we insert a few loans to a service we assign to him, some of which will be unreturned. Since we have total control over the history of the new driver we can calculate by hand the expected results and compare them with what our query shows.

```
Unset
insert into drivers values('123', '123@hotmail.com', 'Sujeto de la Prueba
Estandar', '29-FEB-00', 123456789, 'Casa 1 a la derecha',
'01-MAR-00','02-MAR-24');
insert into assign_drv values('123', '1-NOV-20', 'AN-02');
insert into assign_bus values('BUS-029', '1-NOV-20', 'AN-02');
insert into services values('Villaverde', 'Madrid', 'BUS-029', '1-NOV-20',
'123');
  insert into loans values('CH068', 1546522482, '1-NOV-20', 'Villaverde',
'Madrid', 'L', 100, NULL);
insert into loans values('YB164', 1546522482, '1-NOV-20', 'Villaverde',
'Madrid', 'L', 100, SYSDATE+365);
```

3 Package

We implemented the required package procedures on the file **'foundicu_package.sql'**

In sql, as in many other languages, you have to define things before using them. We will need to define the package, all 3 functions we are asked to implement and the operations related to the user session: set_current_user and get_current_user

```
Unset
CREATE OR REPLACE PACKAGE foundicu AS
    current_user CHAR(10);
    FUNCTION get_current_user RETURN current_user%TYPE;
    PROCEDURE set_current_user(new_user IN current_user%TYPE);
    PROCEDURE insert_loan(copy_signature IN loans.signature%TYPE);
    PROCEDURE insert_reservation(isbn IN editions.isbn%TYPE, reservation_date
in date);
    PROCEDURE record_books_returning(copy_signature IN loans.signature%TYPE);
END foundicu;
```

Then we start the actual package. All procedures have a structure similar to a definition block followed by a code block.

3.0 Definition of current user

We have implemented 2 functions related to the user since all of the procedures use data related to the person doing the petition. They are simply updating or returning the value of the global variable *current_user*, but because it is protected we cannot simply access it normally.

3.1 Insert Loan

The function insert loan is used by the user to make a request for a book. It will have to check a lot of conditions to maintain the consistency of the database, the rules for loaning and the rules of physics. The main concerns, and the ones being checked are:

- The user exists.
- If the user is still banned we have to stop the loan creation process, even though there should be no reservations for banned users.
- If we already have a reservation for that copy we make it a loan, but if we don't have reservations we try to create a loan for that copy for the next 14 days. If the copy is available we check the loan quota and if everything's okay then we find the next service in the area and try to make a reservation for that service.

We have decided on the implicit assumption "No user will make a reservation on a book they already have reserved, they would instead extend the reservation" because the logic is complex enough and due to time constraints we were already unable to implement some additional improvements.

The code for this procedure is long and looks like:

```
Unset
    PROCEDURE insert_loan(copy_signature IN loans.signature%TYPE) IS
        --reservated NUMBER;
        ban_date users.BAN_UP2%TYPE;
        count_users NUMBER;
        count_reservations NUMBER;
        v_taskdate SERVICES.TASKDATE%TYPE;
        v_user_type USERS.TYPE%TYPE;
        v_town USERS.TOWN%TYPE;
        v_province USERS.PROVINCE%TYPE;
        v_population MUNICIPALITIES.POPULATION%TYPE;
    BEGIN
         -- go through users and check by primary key if there is such user as
current
        SELECT MAX(BAN_UP2), COUNT (*) INTO ban_date, count_users FROM USERS
            WHERE USER_ID = current_user;
        IF count_users = 0
                         THEN  RAISE_APPLICATION_ERROR(-20001,  'Current  user
('||current_user||') doesnt exist');
        ELSE
            dbms_output.put_line('Current user exists');
        END IF;

         IF SYSDATE < ban_date -- even if user has reservation, we do not allow
to make them loans
                         THEN  RAISE_APPLICATION_ERROR(-20002,  'Current  user
('||current_user||') is banned up to '||ban_date);
```

```
        END IF;
          -- check reservations and if there is a reservation for the current
user
        -- if there is a reservation, update the loan type to 'L'
        SELECT COUNT(*) INTO count_reservations FROM LOANS
            WHERE USER_ID = current_user
            AND SIGNATURE = copy_signature
            AND TYPE = 'R';

        IF count_reservations <> 0
            -- if there is a reservation, update the loan type to 'L'
              --THEN UPDATE LOANS l SET TYPE='L' WHERE l.user_id=current_user
AND l.signature=signature;
            THEN UPDATE LOANS SET TYPE = 'L'
                WHERE USER_ID = current_user
                AND SIGNATURE = copy_signature;
            dbms_output.put_line('Loan updated from reservation to loan');
        ELSE
              -- to insert new loan, we need to insert a new route and assign
drv
               -- INSERT INTO LOANS VALUES(signature, user_id, SYSDATE, town,
province, type, time, return);
            dbms_output.put_line('No reservation found for this user');
              -- if there is no reservation, check reservations with this book
comparing time and stop dates to check if the copy is available for two weeks
            SELECT COUNT(*) INTO count_reservations FROM LOANS
                WHERE SIGNATURE = copy_signature
                AND RETURN > SYSDATE + 14;

            -- get some data for simplicity
            SELECT TYPE, TOWN, PROVINCE
                INTO v_user_type, v_town, v_province
                FROM USERS
                WHERE USER_ID = current_user;

            IF count_reservations = 0 THEN
             -- if the copy is available, check if the user has not reached the
upper limit for loans by counting his loans
                SELECT COUNT(*) INTO count_reservations FROM LOANS
                    WHERE USER_ID = current_user
                    AND TYPE = 'L'
                       AND RETURN > SYSDATE; -- meaning the book loaned and not
returned yet

                IF v_user_type = 'L' THEN
                    SELECT POPULATION
                    INTO v_population
                    FROM MUNICIPALITIES
                    WHERE TOWN = v_town AND PROVINCE = v_province;
                END IF;

                IF (v_user_type = 'P' AND count_reservations < 2)
                          OR (v_user_type = 'L' AND count_reservations < 2 *
v_population) THEN
                       -- if the user is not sanctioned(and it is not as at the
very beginning the error would be raised), insert a new loan row
                    SELECT MIN(TASKDATE) INTO v_taskdate FROM SERVICES
                        WHERE TASKDATE > SYSDATE
                        AND TOWN = v_town
                        AND PROVINCE = v_province;

                        INSERT INTO LOANS (SIGNATURE, USER_ID, STOPDATE, TOWN,
PROVINCE, TYPE, "TIME", RETURN)  -- in quotes as TIME is a reserved word
                        VALUES (copy_signature, current_user, v_taskdate,
                         v_town, v_province, 'R', DEFAULT, v_taskdate + 14); --
to have return date two weeks from now
                    dbms_output.put_line('New loan inserted');
                ELSE
                        -- if the user has reached the upper limit for loans,
raise error
                      dbms_output.put_line('Error. Current user has reached the
upper limit for loans');
                    RETURN;
                END IF;
            -- if the copy is not available, raise error
            ELSE
                    dbms_output.put_line('Error. Copy is not available for two
weeks');
                RETURN;
            END IF;
        END IF;
        COMMIT;
    END insert_loan;
```

For testing purposes we chose a user and modified their loans in different ways. Then we use our procedure and see if the expected result matches reality. One of the tests was:

```
Unset
  -- this user has 6 loans, so comfortable to test everything
  SELECT SIGNATURE, USER_ID, TYPE FROM LOANS WHERE USER_ID = 9994309731;

  -- modify one to make it R
  UPDATE LOANS
      SET TYPE = 'R'
      WHERE SIGNATURE = 'YC183'
```

```
    AND USER_ID = 9994309731;

-- set current user
EXEC foundicu.set_current_user(9994309731);

-- should modify R to L
EXEC foundicu.insert_loan('YC183');
/*
Current user exists
Loan updated from reservation to loan

PL/SQL procedure successfully completed.
*/
```

3.2 Insert Reservation

We take as an implicit assumption that Foundicu is a company that disregards workers rights, that is, we can assign a reservation whenever we want: on weekends, on International Workers' Day or even Christmas.

The first thing we have to do is check:

- The user exists
- They haven't reached their loaning quota (depending on the type of user we will need the population of the town)
- The user is not banned

We then search for a copy of the wanted book available for at least 14 days. Finally, we try to find a service for that date in that place: if we find it we insert the loan as 'R' and , if not we say that there is no service and don't insert anything.

The implementation is as follows:

```
Unset
             PROCEDURE   insert_reservation(v_isbn   IN   editions.isbn%TYPE,
reservation_date in DATE) IS
        -- receives an ISBN and a date;
        copy_signature copies.signature%TYPE;
        user_data users%ROWTYPE;

        loan_count NUMBER;
        date_of_service services.taskdate%TYPE;
        count_reservations NUMBER;
        v_population MUNICIPALITIES.POPULATION%TYPE;
```

```
    BEGIN
        BEGIN
        -- checks that the current USER exists
        SELECT * INTO user_data FROM users WHERE users.user_id = current_user;
        EXCEPTION
            WHEN NO_DATA_FOUND
                      THEN dbms_output.put_line('Error. Current user does not
exist.');
            WHEN TOO_MANY_ROWS
                 THEN dbms_output.put_line('Error. Multiple current users found
with the same primary key.');
        END;
          -- and has quota for reserving (has not reached the upper borrowing
limit
        SELECT COUNT(*) INTO loan_count FROM LOANS l
            WHERE l.user_id = current_user
                AND l.return > SYSDATE
                AND l.type = 'L';

        IF user_data.type = 'L' THEN
            SELECT POPULATION
            INTO v_population
            FROM MUNICIPALITIES
            WHERE TOWN = user_data.town AND PROVINCE = user_data.province;
        END IF;

         IF (user_data.type = 'P' AND loan_count >= 2) or (user_data.type = 'L'
AND loan_count >= 2*v_population)
                THEN dbms_output.put_line('Error. Current user has reached the
upper limit for loans');
            RETURN;
        END IF;

        -- and they are not sanctioned);
        IF SYSDATE < user_data.ban_up2
            THEN dbms_output.put_line('Error. Current user is banned.');
            RETURN;
        END IF;

         -- checks the availability of a copy of that edition for two weeks (14
days) from the date provided,
        SELECT min(c.signature) INTO copy_signature FROM COPIES c
            WHERE c.isbn = v_isbn -- find all copies of the book
                 AND c.signature NOT IN ( -- exclude the copies that are not
available; could be done by LEFT JOIN
                SELECT l.signature
                FROM LOANS l
                WHERE l.signature IN (
                    SELECT signature FROM COPIES WHERE isbn = v_isbn
                )
                AND l.stopdate - 14 <= reservation_date
                AND l.return + 14 >= reservation_date
            );
        IF copy_signature IS NULL
                   THEN dbms_output.put_line('Error. No available copies for
loaning.');
            RETURN;
        END IF;
        -- and then places the hold (else, reports the hinder).

        SELECT COUNT(*) INTO count_reservations FROM SERVICES s
            JOIN STOPS st ON s.town = st.town AND s.province = st.province
            WHERE s.town = user_data.town
                AND s.province = user_data.province
                AND s.taskdate = reservation_date;
        IF count_reservations = 0
                THEN dbms_output.put_line('Error. No service available for the
following dates.');
            RETURN;
        END IF;
          -- if found, it is only one as TOWN, PROVINCE AND TASKDATE are the
primary key that is unique
         INSERT INTO LOANS (SIGNATURE, USER_ID, STOPDATE, TOWN, PROVINCE, TYPE,
"TIME", RETURN)
                      VALUES (copy_signature, current_user, reservation_date,
user_data.town, user_data.province, 'R', DEFAULT, reservation_date + 14); --
to have return date two weeks from reservation

    END insert_reservation;
```

For testing we create a new scenario to allow for new reservations:

```
Unset
EXEC foundicu.set_current_user(9994309731);

  -- to create service that will be in future
  -- first create driver assignment
  INSERT INTO ASSIGN_DRV (PASSPORT, TASKDATE)
      VALUES ('ESP>>101010101111',  TO_DATE('06-APR-25', 'DD-MON-YY'));

  -- creat bus assignment
  INSERT INTO ASSIGN_BUS (PLATE, TASKDATE, ROUTE_ID)
      VALUES ('BUS-017', TO_DATE('06-APR-25', 'DD-MON-YY'), 'MU-02');

  -- now create service
  INSERT INTO SERVICES (TOWN, PROVINCE, BUS, TASKDATE, PASSPORT)
      VALUES ('Paramo de los Sequillos', 'Cuenca', 'BUS-017',
      TO_DATE('06-APR-25', 'DD-MON-YY'), 'ESP>>101010101111');

  -- will produce error as there is no ride at that day
      EXEC   foundicu.insert_reservation('84-218-2589-5',   TO_DATE('07-APR-25',
'DD-MON-YY'));

   -- execute procedure. It will create new reservation as date matches the bus
drive
      EXEC   foundicu.insert_reservation('84-218-2589-5',   TO_DATE('06-APR-25',
'DD-MON-YY'));
```

3.3 Returned Books

The final procedure of the package is simpler. We need to first check that the user did borrow the book. After that, we can update the return date of the loaned book.
Moreover, we need to check that the user has returned the book within the time period (14 days after the initial loan date). If the user has returned the book late, they will be penalised and a ban will be imposed equivalent to the number of days that they were late on returning the book.

```
Unset
PROCEDURE record_books_returning(copy_signature IN loans.signature%TYPE) IS
        count_users NUMBER;
        loan_date loans.stopdate%type;
        ban_days NUMBER;
    BEGIN
        SELECT COUNT(*) INTO count_users FROM USERS
            WHERE USER_ID = current_user;

        IF count_users = 0
                          THEN  RAISE_APPLICATION_ERROR(-20001,  'Current  user
('||current_user||') doesnt exist');
        END IF;
        -- CHECK IF THE BOOK IS BEING LOANED BY CURRENT USER
        SELECT stopdate INTO loan_date FROM loans l
            WHERE l.signature = copy_signature
                AND l.user_id = current_user
                AND l.return IS NULL;

        -- UPDATE RETURN OF LOAN
        UPDATE loans SET return = SYSDATE
            WHERE signature = copy_signature AND user_id = current_user;
        dbms_output.put_line('Copy of book '||copy_signature||' loaned to user
with id '||current_user||' successfully returned.');

        -- BANNING USER IF RETURNED LATE
        ban_days := SYSDATE-loan_date-14;
        IF ban_days > 0 THEN
                   UPDATE  users  SET  ban_up2=SYSDATE+ban_days  WHERE  USER_ID  =
current_user;
              dbms_output.put_line('Copy of book '||copy_signature||' returned
late, banning current user '||current_user||' to '||(SYSDATE+ban_days));
        END IF;

        COMMIT;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
                 dbms_output.put_line('Error. No unreturned loan of this copy
('||copy_signature||') by current user ('||current_user||') has been found.');
        WHEN TOO_MANY_ROWS THEN
               dbms_output.put_line('Error. Found multiple unreturned loans of
the same copy ('||copy_signature||') by current user ('||current_user||').');
    END record_books_returning;
```

To test the validity of this last procedure we again choose a user and, modifying the conditions of one of their loans, execute the code to see if it works as expected:

```
Unset
begin
    foundicu.set_current_user(1546522482);
    UPDATE loans SET RETURN=NULL WHERE USER_ID='1546522482' AND
signature='IJ548';
    foundicu.record_books_returning('IJ548');
    foundicu.record_books_returning('IJ548'); -- ERROR
end;
/

begin
    foundicu.set_current_user(69);
    foundicu.record_books_returning('IJ548'); -- ERROR
end;
/
```

4 External Design

We implemented the required external design on the file **'foundicu_views.sql'**

4.1 Get user data

This view is a read only table extracting the data from a user. The relational algebra is almost trivial. Just project everything except name and surnames, that we have to concatenate to get "fullname", and get the needed user:

$$\pi_{\text{userID, ID\_card, concat(Name, surn1, surn2) AS fullname, birth, town, prov…}}(\sigma_{\text{user\_id = session\_user}}(\text{Users}))$$

The sql implementation is the same thing:

```
Unset
CREATE OR REPLACE VIEW my_data AS
SELECT
       USER_ID,
       ID_CARD,
       NAME || ' ' || SURNAME1 || ' ' || SURNAME2 AS FULLNAME,
       BIRTHDATE,
       TOWN,
       PROVINCE,
       ADDRESS,
       EMAIL,
       PHONE,
       TYPE,
       BAN_UP2
FROM users u
WHERE u.user_id = foundicu.get_current_user()
WITH READ ONLY;
```

To ensure it works and that it is read only we execute the following test

```
Unset
EXEC foundicu.set_current_user(9994309824);
SELECT * FROM my_data;
-- CANNOT INSERT THIS: READ ONLY
INSERT INTO my_data VALUES('1', '1111', NULL, SYSDATE, '', '', '', NULL,
    111111111111, 'P', NULL);
```

4.2 Get user loans

For this view we need the loans that are actualized loans, not just reservations. For this we will need another query, this time operable but only regarding the text in the post.The relational algebra will be still fairly simple:

$$\sigma_{\text{user\_id = session\_user, type = 'L'}}(\text{Loans}) \bowtie \text{Posts}$$

```
Unset
    CREATE OR REPLACE VIEW my_loans AS
    SELECT
        l.signature,
        l.STOPDATE,
        l.TOWN,
        l.PROVINCE,
        l.TYPE,
        l.TIME,
        l.RETURN,
        p.POST_DATE,
        p.TEXT,
        p.likes,
        p.dislikes
    FROM loans l
    LEFT JOIN posts p ON l.signature = p.signature AND l.user_id = p.user_id
AND l.stopdate = p.stopdate
    WHERE l.user_id = foundicu.get_current_user() AND l.type = 'L' AND
l.return < SYSDATE
    WITH CHECK OPTION;
```

Now the problem is the modifiable part. We will need a way to make sure nothing is deleted or inserted and only the text is updated, and when it is updated the date will have to change. This is done with 2 different triggers: one in charge of managing the post creation and modification and another one for avoiding the insertion or deletion of loans:

```
Unset
CREATE OR REPLACE TRIGGER my_loans_trigger
    INSTEAD OF UPDATE
    ON my_loans
    FOR EACH ROW
BEGIN
    IF :OLD.text IS NULL THEN
        -- IF INSERTING THEN
            INSERT  INTO  POSTS  (SIGNATURE,  USER_ID,  STOPDATE,  TEXT,  POST_DATE,
LIKES, DISLIKES)
                           VALUES  (:NEW.SIGNATURE,  foundicu.get_current_user(),
:NEW.STOPDATE, :NEW.TEXT, SYSDATE, 0, 0);
    ELSE
        -- IF UPDATING THEN
        UPDATE POSTS
            SET POST_DATE = SYSDATE,
                TEXT = :NEW.text
            WHERE SIGNATURE = :NEW.SIGNATURE
```

```
                AND USER_ID = foundicu.get_current_user();
    END IF;
END my_loans_trigger;
/

CREATE OR REPLACE TRIGGER guard_my_loans
    INSTEAD OF INSERT OR DELETE ON my_loans
    FOR EACH ROW
BEGIN
    IF INSERTING THEN
         RAISE_APPLICATION_ERROR(-20042, 'Insertion is not allowed on my_loans
view.');
    ELSIF DELETING THEN
          RAISE_APPLICATION_ERROR(-20042, 'Deletion is not allowed on my_loans
view.');
    END IF;
END guard_my_loans;
/
```

To test that the posts updates correctly and that insertions and deletions are indeed disallowed, we run the following tests:

```
Unset
  EXEC foundicu.set_current_user(5005122262);
  UPDATE my_loans
      SET TEXT = 'THIS IS TEST'
      WHERE SIGNATURE='UC856';
  SELECT * FROM my_loans;
  SELECT * FROM POSTS WHERE USER_ID = 5005122262;

  -- to test case of insertion
  EXEC foundicu.set_current_user(8612169569);
  UPDATE my_loans
      SET TEXT = 'HIHIHI'
      WHERE SIGNATURE = 'JL729';
  SELECT SIGNATURE, TEXT FROM POSTS WHERE USER_ID = 8612169569;
```

4.3 Get user reservations

The last view is a way for the user to manage their upcoming reservations. There is no need for relational algebra since it is extremely simple, but here it is just in case:

$$\sigma_{\text{user\_id = session\_user, type = 'R'}}(\text{Loans})$$

```
Unset
    CREATE OR REPLACE VIEW my_reservations AS
    SELECT
        SIGNATURE,
        STOPDATE,
        TOWN,
        PROVINCE,
        TYPE,
        TIME,
        RETURN
    FROM loans l
    WHERE l.user_id = foundicu.get_current_user() AND l.type = 'R'
    WITH CHECK OPTION;
```

The big part of this view is the trigger, since we have to control that what the user wants to change doesn't break the consistency of our database. It is a trigger executed for each row instead of updating or inserting:

- When inserting we simply insert in the real Loans table.
- For updating we check what they are trying to change and if it is not permitted we raise an exception

Deletion is not a concern since sqlplus does that part for us.
The sql implementation is the following:

```
Unset
CREATE OR REPLACE TRIGGER my_reservations_trigger
        INSTEAD OF INSERT OR UPDATE ON my_reservations
        FOR EACH ROW
    BEGIN
        IF INSERTING THEN
            INSERT INTO LOANS (SIGNATURE, USER_ID, STOPDATE, TOWN, PROVINCE,
TYPE, TIME, RETURN)
                VALUES (:NEW.SIGNATURE, foundicu.get_current_user(),
:NEW.STOPDATE, :NEW.TOWN, :NEW.PROVINCE, :NEW.TYPE, :NEW.TIME, :NEW.RETURN);
        ELSIF UPDATING THEN
            IF :OLD.SIGNATURE!=:NEW.SIGNATURE OR :OLD.TYPE!=:NEW.TYPE
                THEN RAISE_APPLICATION_ERROR(-20043, 'Error. Only date and
time from my_reservations are allowed to be changed');
            END IF;

            UPDATE loans SET
                STOPDATE = :NEW.STOPDATE,
                TOWN = :NEW.TOWN,
                PROVINCE = :NEW.PROVINCE,
                TIME = :NEW.TIME
                WHERE SIGNATURE = :OLD.SIGNATURE
                    AND USER_ID = foundicu.get_current_user();
        END IF;
    END my_reservations_trigger;
    /
```

Our test case is simply trying to insert, update and then delete a reservation for a user:

```
Unset
EXEC FOUNDICU.SET_CURRENT_USER(1546522482);

-- INSERT A RESERVATION (ALLOWED)
INSERT  INTO  my_reservations  VALUES('NE000',  '16-NOV-24',  'Sotolemures',
'Barcelona', 'R', 750, NULL);

-- CHECK IF OUTPUT IS THE SAME
  SELECT * FROM my_reservations;
  SELECT * FROM loans WHERE user_id=foundicu.get_current_user() AND type='R';

-- UPDATE (FIRST RETURNS ERROR, SECOND IS VALID)
  UPDATE my_reservations SET signature='IJ548' WHERE signature='NE000';
  UPDATE my_reservations SET TIME=2000 WHERE signature='NE000';

-- DELETE AN ENTRY (ALLOWED)
 DELETE FROM my_reservations WHERE SIGNATURE='NE000' AND STOPDATE='16-NOV-24';
```

5 Explicitly Required Triggers

We implemented the required trigger on the file **'foundicu_triggers.sql'**

5.1 Prevent posts from municipal libraries (1.4.A)

The first trigger we have decided to implement is the one preventing libraries from having posts associated with them. It is executed before insert or before update for every row. To ensure data consistency, we first remove all existing posts associated with municipal libraries in the database: a total of 4867 rows.

Now we can properly start the trigger. The trigger is simple in concept: we check the user type before inserting or updating a post and if it is a library we raise an error stopping the insertion. While we are at it, we can also make sure the user exists. The implementation is really the same in the inserting and the updating case:

```
Unset
-- Delete posts from libraries: 4867 rows
    DELETE FROM POSTS
        WHERE USER_ID IN (
            SELECT DISTINCT USER_ID
                FROM USERS
                WHERE type='L'
        );

    -- Trigger for restricting library posts
    CREATE OR REPLACE TRIGGER restrict_library_posts
        BEFORE INSERT OR UPDATE OF USER_ID ON posts
        FOR EACH ROW
    DECLARE
        user_type users.type%TYPE;
    BEGIN
        IF UPDATING THEN
            SELECT type INTO user_type FROM users
                WHERE user_id=:OLD.user_id;
            IF user_type='L' THEN
                RAISE_APPLICATION_ERROR(-20051, 'Error. Library with id '||
:NEW.user_id ||' cannot write posts');
                RETURN;
            END IF;
        ELSIF INSERTING THEN
            SELECT type INTO user_type FROM users
                WHERE user_id=:NEW.user_id;
            IF user_type='L' THEN
                RAISE_APPLICATION_ERROR(-20051, 'Error. Library with id '||
:NEW.user_id ||' cannot write posts');
                RETURN;
            END IF;
        END IF;
    EXCEPTION
        WHEN NO_DATA_FOUND
            THEN dbms_output.put_line('Data Integrity Error. User not found');
    END restrict_library_posts;
    /
```

To test the trigger, we simply insert a post using a user with type 'L' (library):

```
Unset
-- INSERT LIBRARY POST: ERROR
INSERT         INTO         posts         VALUES('AA957',         '9994309856',
TO_DATE('19-11-2024','DD-MM-YYYY'), SYSDATE, 'text', 0, 0);

-- INSERT USER POST: GOOD
```

```
INSERT          INTO          POSTS          VALUES('JG545',          '9266310304',
TO_DATE('22-11-2024','DD-MM-YYYY'), SYSDATE, 'text', 1, 1);
```

5.2 Automatic update of deregistration date (1.4.B)

The second trigger we will implement is the one controlling the deregistration date of copies. It is a trigger executed after insert or update on every row (we added a *before* part for checks).

When we mark a book as deregistered we take the system time and insert it into the *deregistered* attribute of that copy.

Once again, we can take advantage of the already existing structure of the trigger to impose some other constraints, like "deregistered / destroyed books cannot be reintroduced into circulation". The sql implementation follows the structure:

```
Unset
CREATE OR REPLACE TRIGGER copy_deregistration
        FOR INSERT OR UPDATE OF CONDITION ON copies
    COMPOUND TRIGGER
        BEFORE EACH ROW IS
        BEGIN
            IF UPDATING THEN
                IF :OLD.CONDITION='D' AND :NEW.CONDITION<>'D' THEN
                    RAISE_APPLICATION_ERROR(-20052, 'Cannot change copy
condition from deregistered to another value (they are already physically
destroyed)!');
                END IF;
            END IF;

            IF :NEW.CONDITION='D' THEN
                :NEW.DEREGISTERED := SYSDATE;
            END IF;
        END BEFORE EACH ROW;
    END copy_deregistration;
    /
```

The test in this case is simply deregistering a copy and checking that the date is today's. We can also try putting it back in circulation just to check that the error really is raised:

Unset
```
  UPDATE copies SET condition='D' WHERE copies.signature='AA070';
  SELECT deregsitered FROM copies WHERE copies.signature='AA070';
  UPDATE copies SET condition='N' WHERE copies.signature='AA070'; -- ERROR
```

5.3 Implementing Reads column for Books table (1.4.D)

The last chosen trigger is the one regarding the number of reads of a certain copy. The trigger is executed after insert, update or delete (again, the *before* part is to perform checks); and can be implemented in such a way that it is run for every row adding or subtracting 1, or once per statement with count but it is slower. We decided on "per row" granularity.

We have made a few implicit semantics:

- Libraries shouldn't increase the number of reads a copy has when loaning it, since multiple readers may read that copy without us knowing.
- Nobody can change from a loan to a reservation, they already got the book.
- We count loans as a read, but not reservations.

The first thing we do is create the new column *reads* and initialize the read to however many loans there were before the implementation of the trigger (just in case):

Unset
```
-- Add new constraint to loans to avoid exploits
    ALTER TABLE loans ADD CONSTRAINT ck_type CHECK (type in ('L', 'R'));

    -- Create new column 'reads' in table 'books'
    ALTER TABLE books ADD reads INTEGER DEFAULT 0;
```

To maintain consistency we can also count the loans already stored in the database:

Unset
```
BEGIN
        FOR book_reads IN (
            SELECT e.title, e.author, COUNT(*) as new_reads
            FROM editions e
            JOIN copies c ON e.isbn = c.isbn
            JOIN loans l ON l.signature = c.signature
            GROUP BY e.title, e.author
        ) LOOP
            UPDATE books
            SET reads = book_reads.new_reads
            WHERE title = book_reads.title AND author = book_reads.author;
        END LOOP;
    END;
    /
```

Now we can start with the actual trigger. We check the constraints we set at the beginning, and go to the 3 different cases:

- If the trigger is activated while deleting we get the book data and remove 1 from its counter if it was already loaned.
- If we are updating from 'R' to 'L' or inserting a new 'L' we add 1.

In the end we have to get something similar to:

```
Unset
    --- Create trigger that updates read counter upon new loan insertion,
update or deletion.
    CREATE OR REPLACE TRIGGER update_book_read
        FOR INSERT OR UPDATE OF type OR DELETE ON loans
    COMPOUND TRIGGER
        loan_title editions.title%TYPE;
        loan_author editions.author%TYPE;

        BEFORE EACH ROW IS
        BEGIN
            IF UPDATING THEN
                IF :NEW.type = 'R' AND :OLD.type = 'L' THEN
                    RAISE_APPLICATION_ERROR(-20053, 'Cannot change from a loan
to a reservation!');
                END IF;
            END IF;
        END BEFORE EACH ROW;

        AFTER EACH ROW IS
        BEGIN
            IF DELETING THEN
                -- Obtain loan's book title and author
                SELECT e.title, e.author INTO loan_title, loan_author FROM
editions e
                    JOIN copies c ON e.isbn=c.isbn
                    WHERE c.signature=:OLD.signature;

                -- Decrease reads count
                UPDATE books
                    SET reads=reads-1
                    WHERE title=loan_title AND author=loan_author;
            ELSIF UPDATING THEN
                IF :OLD.TYPE != 'L' AND :NEW.TYPE='L' THEN
                    -- Obtain loan's book title and author
                    SELECT e.title, e.author INTO loan_title, loan_author FROM
editions e
                        JOIN copies c ON e.isbn=c.isbn
                        WHERE c.signature=:NEW.signature;

                    -- Increase reads count
                    UPDATE books
                        SET reads=reads+1
                        WHERE title=loan_title AND author=loan_author;
                    END IF;
            ELSIF INSERTING THEN
                IF :NEW.TYPE = 'L' THEN
                    -- Obtain loan's book title and author
                    SELECT e.title, e.author INTO loan_title, loan_author FROM
editions e
                        JOIN copies c ON e.isbn=c.isbn
                        WHERE c.signature=:NEW.signature;

                    -- Increase reads count
                    UPDATE books
                        SET reads=reads+1
                        WHERE title=loan_title AND author=loan_author;
                END IF;
            END IF;
        END AFTER EACH ROW;
    END;
    /
```

Testing is as simple as inserting and deleting loans and seeing the number change:

```
Unset
-- 0 reads
SELECT READS FROM BOOKS WHERE title='Pablo Picasso' AND author='Picasso,
Pablo, ( 1881-1973)';
insert into loans values('IG254', 1546522482, '16-NOV-24', 'Villatrigos',
'Valencia', 'L', 100, NULL);

-- 1 reads
SELECT READS FROM BOOKS WHERE title='Pablo Picasso' AND author='Picasso,
Pablo, ( 1881-1973)';
delete loans where signature='IG254' and user_id='1546522482' and
stopdate='16-NOV-24';
```

```
-- 0 reads
SELECT READS FROM BOOKS WHERE title='Pablo Picasso' AND author='Picasso,
Pablo, ( 1881-1973)';
```

6 Custom Errors

We have defined a series of custom application errors to ensure the correct functionality of our implemented code:

Section	Code	Message
package	-20001	'Error. Current user does not exist.'
package	-20002	'Error. Current user is banned.'
4.2	-20042	'Insertion/deletion is not allowed on my_loans view.'
4.3	-20043	'Error. Only date and time from my_reservations are allowed to be changed'
5.1	-20051	'Error. Library with id (user_id) cannot write posts'
5.2	-20052	'Cannot change copy condition from deregistered to another value (they are already physically destroyed)!'
5.3	-20053	'Cannot change from a loan to a reservation!'

7 Concluding Remarks

The final result is a good approximation to what the specifications given might refer to. Evidently, in a real case we would do intensive testing, multiple iterations of every function to get the opinion of the client and work exclusively on this project. We have thought of multiple ways for each function and have reached a balance between usability, robustness and efficiency.

Most of the documentation comes from https://docs.oracle.com/database/121/ and some old comments in https://stackoverflow.com/.

We have spent an average of 20 to 30 hours each with varying degrees of success. The project was challenging but reasonable within a difficulty range appropriate to what should be known about sql for a first approach. The main difficulty was the amount of work for the period of time it was supposed to be done in (a lot of exams in these past 2 weeks) and the obscurity of some of the tests needed to check that everything worked: checking that a view works is so much easier than testing all conditions for a trigger.