**MACQUARIE UNIVERSITY**
**Faculty of Science and Engineering**
**Department of Computing**

# Assignment 2 (Report)

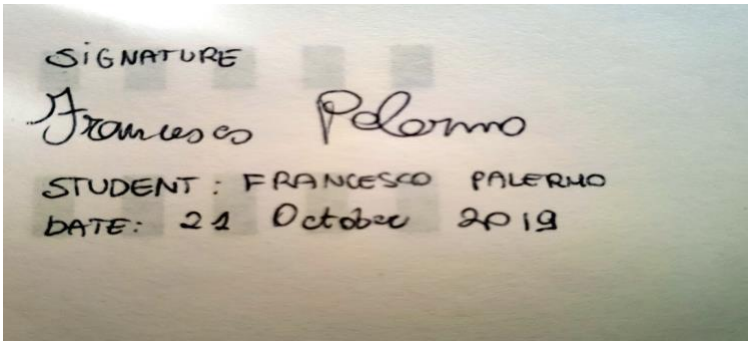
## Database Programming and Implementation (worth 15%)

**Student Name:** **Francesco Palermo**

**Student Number:** **45539669**

**Student Declaration:**

*I declare that the work reported here is my own. Any help received, from any person, through discussion or other means, has been acknowledged in the last section of this report.*



# 1. Initial State of the database

After running the two provided scripts in this order (create_DB.sql and populate_DB.sql) the screenshots of the tables that I have used throughout are shown below:

*SELECT **
*FROM invoice;*

| INVOICENO | CAMPAIGN_NO | DATEISSUED | DATEPAID | BALANCEOWING | STATUS |
|---|---|---|---|---|---|
| 1 | 1 | 2019-07-19 | 2019-07-31 | 0 | PAID |
| 2 | 2 | 2019-09-10 | NULL | 675 | UNPAID |

*SELECT **
*FROM campaign;*

| CAMPAIGN_NO | TITLE | CUSTOMER_ID | THEME | CAMPAIGNSTARTDATE | CAMPAIGNFINISHDATE | ESTIMATEDCOST | ACTUALCOST |
|---|---|---|---|---|---|---|---|
| 1 | Blue | 1 | Fall | 2018-01-01 | 2019-07-01 | 1500 | 1350 |
| 2 | Red | 3 | Spring | 2019-01-03 | NULL | 1000 | NULL |

*SELECT **
*FROM workson;*

| STAFFNO | CAMPAIGN_NO | WDATE | HOUR |
|---|---|---|---|
| 1 | 2 | 2019-02-01 | 8 |
| 1 | 2 | 2019-05-01 | 7 |
| 3 | 1 | 2018-05-01 | 5 |
| 5 | 1 | 2018-02-01 | 7 |
| 5 | 1 | 2019-06-01 | 8 |

*SELECT **
*FROM staffongrade;*

| STAFFNO | GRADE | STARTDATE | FINISHDATE |
|---|---|---|---|
| 1 | 1 | 2018-01-01 | 2019-01-04 |
| 1 | 2 | 2019-01-05 | NULL |
| 3 | 2 | 2018-01-05 | 2018-12-31 |
| 3 | 3 | 2019-01-01 | NULL |
| 5 | 4 | 2018-01-01 | 2019-06-01 |

*SELECT **
*FROM salarygrade;*

| GRADE | HOURLYRATE |
|---|---|
| 1 | 30 |
| 2 | 45 |
| 3 | 70 |
| 4 | 75 |
| 5 | 150 |

## 2. Stored Programs.

In this section all the procedures and functions will be displayed and briefly explainded.
In section 3, those will be tested with the provided *test_script.sql*.

The first program that has been developed is the trigger.

Below there is the actual code that was able to achieve the task 2.

```
delimiter //
CREATE TRIGGER tr_overdue
BEFORE UPDATE ON invoice
FOR EACH ROW


    BEGIN
    DECLARE msg VARCHAR(250);
    set msg = CONCAT('Invoice with number: ',NEW.INVOICENO,' is now overdue!');

    IF (NEW.STATUS ='OVERDUE') THEN
        -- INSERTING ROWS TO THE TABLE alert
        INSERT INTO alerts (message_date,origin,message) VALUES (now(),current_user(),msg);

    END IF;
    END//
DELIMITER ;
```

The trigger fires whenever the invoice table is updated. However, if the new *STATUS* is 'OVERDUE' before the update is achieved, the trigger inserts a new row in the table *alerts*.

The second procedure *sp_finish_campaign* is the more complex. In fact, in order to correctly develop this procedure other two functions have been implemented.

The first function *rate_on_date* is shown below:

```
drop function if exists rate_on_date;

delimiter //
CREATE FUNCTION rate_on_date(staff_id INT, given_date DATE)
RETURNS FLOAT
DETERMINISTIC
BEGIN
    -- declare a local variable that store the total hour_rate on a given date
     DECLARE hourly_rate FLOAT DEFAULT 0;

    -- return the total salary of a given staff (staff_id) on any particular date (given_date)

       SELECT sg.HOURLYRATE INTO hourly_rate
       FROM workson wo, staffongrade sog, salarygrade sg
       WHERE wo.STAFFNO = sog.STAFFNO AND sog.GRADE = sg.GRADE
              AND wo.STAFFNO = staff_id AND wo.WDATE = given_date AND given_date >= sog.STARTDATE and (given_date <= sog.FINISHDATE OR sog.FINISHDATE IS NULL);

     RETURN hourly_rate;

END //
DELIMITER ;
```

This function takes two parameters *staff*_id and *given_date* and return the hourly rate of that given staff in that particular date. In order to achieve that, three tables have been connected through an inner join.

Those tables are:

- *works on*
- *staffongrade*
- *salarygrade*

It is important to remind that a particular staff may receive two different rates (please check table *staffongrade* above, where *STAFFNO*=1 receives two different rates). This is the main reason why the WHERE has some conditions on dates.

The second function *cost_of_campaign* is shown below:

```
CREATE FUNCTION cost_of_campaign(camp_id INT)
RETURNS FLOAT
DETERMINISTIC
BEGIN

    -- declare a local variable that store the total cost of a campaign
    DECLARE v_finito FLOAT DEFAULT 0;
    DECLARE costo_totale FLOAT DEFAULT 0;
    DECLARE c_hour FLOAT DEFAULT 0;
    DECLARE c_rate FLOAT DEFAULT 0;

    -- declare a cursor
    DECLARE hour_rate CURSOR FOR
    SELECT  HOUR,rate_on_date(STAFFNO,WDATE)
    FROM workson
    WHERE CAMPAIGN_NO = camp_id;
    -- delare the handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_finito = 1;

    -- OPEN THE CURSOR AND LOOP TO GET THE TOTAL
    OPEN hour_rate;
    WHILE (v_finito=0) DO

            FETCH hour_rate INTO c_hour,c_rate;

            IF (v_finito=0) THEN
                SET costo_totale = costo_totale+(c_hour*c_rate);

            END IF;

    END WHILE;
    CLOSE hour_rate;

    RETURN costo_totale;


END//
delimiter ;
```

This function has one parameter *camp_id*, which is the uniquely identifier of a campaign, and returns the total cost (*costo_totale*) of the campaign.
A particular attention must be given to the cursor *hour_rate*. First of all, the cursor has been declared because we need to go through all the staff who worked for a particular campaign.
In addition to the cursor, an error handler have been declared (*v_finito*) in order to prevent 'gently' the error that arises when the cursor do not have any more rows to fetch.

In order to understand how this function works, a SQL statement will be shown and his results will be analysed.

```
SELECT  w.STAFFNO, w.CAMPAIGN_NO,c.TITLE,w.WDATE,w.HOUR,rate_on_date(w.STAFFNO,w.WDATE)
FROM workson w, campaign c
WHERE w.CAMPAIGN_NO = c.CAMPAIGN_NO ;
```

| STAFFNO | CAMPAIGN_NO | TITLE | WDATE | HOUR | rate_on_date(w.STAFFNO,w.WDATE) |
|---|---|---|---|---|---|
| 1 | 2 | Red | 2019-02-01 | 8 | 45 |
| 1 | 2 | Red | 2019-05-01 | 7 | 45 |
| 3 | 1 | Blue | 2018-05-01 | 5 | 45 |
| 5 | 1 | Blue | 2018-02-01 | 7 | 75 |
| 5 | 1 | Blue | 2019-06-01 | 8 | 75 |

In the table above, we have a clear picture of what is going on inside the cursor (note that the cursor loops over *HOUR* and *RATE_ON_DATE* function only, while here I preferred to give a wider view of the situation for a better interpretability).
The last column is obtained by calling the function *rate_on_date*. From the table we can see those people who worked for a particular campaign, the amount of hours that they worked and the rate received that particular date.

If the function works, we would expect these costs of campaign:

| ***CAMPAIGN_NO*** | ***TITLE*** | ***COST_OF_CAMPAIGN*** |
|---|---|---|
| 2 | Red | 675 |
| 1 | Blue | 1350 |

Finally, the procedure *sp_finish_campaign* can be implemented. After the two other functions have been run, this procedure became relatively easy to implement .

Below a screeshot of the procedure is given:

```
drop procedure if exists sp_finish_campaign;
delimiter //

CREATE PROCEDURE sp_finish_campaign (in c_title varchar(30))


BEGIN
    DECLARE v_campaign_count INT DEFAULT 0;
    DECLARE v_costof_camp FLOAT DEFAULT 0;

    -- count the number of campaign with the supplied c_title
    -- the count will be 1 if if the campaign exists
    SELECT COUNT(*) INTO v_campaign_count
    FROM campaign
    WHERE TITLE=c_title;

    -- if the campaign exists update CAMPAIGNFINISHDATE AND ACTUALCOST
    IF (v_campaign_count = 1) THEN
        -- UPDATE THE campaignfinishdate TO THE CURRENT DATE
        UPDATE campaign
        SET CAMPAIGNFINISHDATE = now()
        WHERE TITLE = c_title;
        -- UPDATE THE actual cost
        SELECT cost_of_campaign(CAMPAIGN_NO) INTO v_costof_camp
        FROM campaign
        WHERE TITLE=c_title;

        UPDATE campaign
        SET ACTUALCOST = v_costof_camp
        WHERE TITLE = c_title;
    ELSE
        SIGNAL SQLSTATE '45000'
        --  which means "unhandled user-defined exception."
        SET MESSAGE_TEXT = 'ERROR! Campaign title does not exist';

    END IF;


END//
delimiter ;
```

The procedure takes as input the title of the campaign (*c_title*) and as per definition does not really return anything back to the caller. However, if the campaign exists the procedure:

- Update the *CAMPAIGNFINISHDATE* to the current date
- Update the ACTUALCOST of the campaign

If the campaign title provided by the user does not exist, an error message is displayed. The results of this procedure obtained fom the provided sample data can be seen in the section 3.

The final procedure sync_invoice() does not have any input parameters and his goal is updating those invoice that are still recorded as UNPAID more than 30 days after the invoice was issued with the status 'OVERDUE'.
The code is shown below:

```
drop procedure if exists sync_invoice;
delimiter //
CREATE PROCEDURE sync_invoice()
BEGIN

-- declare a local variable
    DECLARE v_invoiceno INT DEFAULT 0;
    DECLARE date_diff INT DEFAULT 0;
    DECLARE dateissued DATE;
    DECLARE v_status VARCHAR(20);
    DECLARE v_finito FLOAT DEFAULT 0;

    -- declare a cursor
    DECLARE c_invoice CURSOR FOR
    SELECT  INVOICENO,DATEISSUED,STATUS,DATEDIFF(now(),DATEISSUED)
    FROM invoice
    WHERE (DATEDIFF(now(),DATEISSUED) >30 AND STATUS = 'UNPAID');
    -- delare the handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_finito = 1;

    -- OPEN THE CURSOR AND LOOP THROUGH THE ROWS
    OPEN c_invoice;
    WHILE (v_finito=0) DO

            FETCH c_invoice INTO v_invoiceno,dateissued,v_status,date_diff;

            IF (v_finito=0) THEN
                UPDATE invoice
                SET invoice.STATUS= 'OVERDUE'
                WHERE INVOICENO=v_invoiceno;

            END IF;

    END WHILE;
    CLOSE c_invoice;
END//
```

Although there is only one record in the sample data that match the condition specified, I decided to implement a cursor because in the future more invoices with the same properties can be identified and therefore a cursor can be helpful with this scenario.
The SQL-function *DATEDIFF* has been used to calculate the difference between two dates and only those invoices that that are still recorded as UNPAID more than 30 days after the invoice was issued will be saved in the cursor.

As I have done before, one SQL statement will be shown as a way of explaining how the procedure works.

```
select INVOICENO,DATEISSUED,STATUS,DATEDIFF(now(),DATEISSUED)
from invoice;
```

| INVOICENO | DATEISSUED | STATUS | DATEDIFF(now(),DATEISSUED) |
|---|---|---|---|
| 1 | 2019-07-19 | PAID | 103 |
| 2 | 2019-09-10 | UNPAID | 50 |

The invoice with *INVOICENO=2* is the candidate row that need to be updated. The results of this procedure obtained fom the provided sample data can be seen in the section 3.

## 3. Required Testing against Sample Database.

### 1) Trigger *tr_overdue*

Once the trigger has been created the first part of the *test_script.sql* was run. Below the two tables shows how the trigger works according to the instruction given.
Once this line of code have been run two tables were modified:

*update invoice set STATUS = 'OVERDUE' where INVOICENO = 2;*

The first table illustrates how the DML statement (update) actually updates the invoice with INVOICENO=2 with the status OVERDUE.

| INVOICENO | CAMPAIGN_NO | DATEISSUED | DATEPAID | BALANCEOWING | STATUS |
|---|---|---|---|---|---|
| 1 | 1 | 2019-07-19 | 2019-07-31 | 0 | PAID |
| 2 | 2 | 2019-09-10 | NULL | 675 | OVERDUE |

The second table shows how the trigger actually works. The trigger fires whenever the table is updated. However, if STATUS is updated with "OVERDUE" a new row is inserted into the *alerts* table with the required fields.

| message_no | message_date | origin | message |
|---|---|---|---|
| 2 | 2019-10-22 | 45539669… | Invoice with number: 2 is now overdue! |

## 2) Procedure *sp_finish_campaign*

In order to see the behavior of this procedure and see whether it is matching with our expectations, the following line of code have been executed first.

*call sp_finish_campaign('Red');*

To check what the procedure has produced, we need to see what is inside the table *campaign;*

| CAMPAIGN_NO | TITLE | CUSTOMER_ID | THEME | CAMPAIGNSTARTDATE | CAMPAIGNFINISHDATE | ESTIMATEDCOST | ACTUALCOST |
|---|---|---|---|---|---|---|---|
| 1 | Blue | 1 | Fall | 2018-01-01 | 2019-07-01 | 1500 | 1350 |
| 2 | Red | 3 | Spring | 2019-01-03 | 2019-10-30 | 1000 | 675 |

This result matches our expectation. In fact, according to the table above at pag 6 we would have expected that the total cost of the campaign with title *'Red'* was 675. In addition, the actual cost provided by the sample data for the campaign with title *'Blue'* is the same of what we previously predicted and this ensure the accuracy of the procedure.

To complete, the following call procedure is tested to ensure a signal message is shown in case a user accidentally insert a parameter with a title that does not exist in the *campaign* table.

*call sp_finish_campaign('GREEN');*


### 3) Procedure *sync_invoice()*

Let us see the behavior of this procedure by calling the procedure first and then checking the records of the *invoice* table:

*call sync_invoice();*

| INVOICENO | CAMPAIGN_NO | DATEISSUED | DATEPAID | BALANCEOWING | STATUS |
|---|---|---|---|---|---|
| 1 | 1 | 2019-07-19 | 2019-07-31 | 0 | PAID |
| 2 | 2 | 2019-09-10 | NULL | 675 | OVERDUE |

This result matches our expectation. In fact, we knew that the invoice with *INVOICENO=2* was the candidate invoice that need to be updated. We can clearly see how that record has been correctly updated with the *STATUS = 'OVERDUE'*.

## 4. More Extensive Testing.

The additional test I will show is about the performance of the implemented trigger and a new procedure that will implement a penalty rate for those campaign which estimate cost exceeds the estimated cost.

For the trigger, I am going to show the scenario where the user updates the trigger with a status that is different from OVERDUE.
For the sake of this scenario I created a new campaign (*CAMPAIGN_NO*=3) with the STATUS='UNPAID'.

| INVOICENO | CAMPAIGN_NO | DATEISSUED | DATEPAID | BALANCEOWING | STATUS |
|---|---|---|---|---|---|
| 1 | 1 | 2019-07-19 | 2019-07-31 | 0 | PAID |
| 2 | 2 | 2019-09-10 | NULL | 675 | UNPAID |
| 3 | 3 | 2019-09-19 | NULL | 0 | UNPAID |
| NULL | NULL | NULL | NULL | NULL | NULL |

Let us suppose that the user updates this last record by setting the status as *PAID.*

*update invoice set STATUS = 'PAID' where INVOICENO = 3;*

| INVOICENO | CAMPAIGN_NO | DATEISSUED | DATEPAID | BALANCEOWING | STATUS |
|---|---|---|---|---|---|
| 1 | 1 | 2019-07-19 | 2019-07-31 | 0 | PAID |
| 2 | 2 | 2019-09-10 | NULL | 675 | UNPAID |
| 3 | 3 | 2019-09-19 | NULL | 0 | PAID |
| NULL | NULL | NULL | NULL | NULL | NULL |

After having realized that the invoice table has been correctly modified, we need to make sure the table *alerts* has not been modified and no messages corcerns *INVOICENO=3.*

*SELECT **
*FROM alerts;;*

| message_no | message_date | origin | message |
|---|---|---|---|
| NULL | NULL | NULL | NULL |

Once again the actual output matches our expectations. No rows are shown in the *alert* table.

The next procedures is meant to select those campaigns which actual cost is higher than the estimated and therefore apply a rate penalty to the atual cost.
We then suppose that none of the campaigns paid the invoices yet.
In addition, more rows have been inserted in the database for this purpose.

```
-- extra procedure

INSERT INTO `staff` (`STAFFNO`, `STAFFNAME`) VALUES ('6', 'Frank');
INSERT INTO `staff` (`STAFFNO`, `STAFFNAME`) VALUES ('7', 'Diana');
INSERT INTO `staff` (`STAFFNO`, `STAFFNAME`) VALUES ('8', 'Luca');

INSERT INTO `workson` (`STAFFNO`, `CAMPAIGN_NO`, `WDATE`,  `HOUR`)
    VALUES ('6', '3', '2019-01-05',  '8');
INSERT INTO `workson` (`STAFFNO`, `CAMPAIGN_NO`, `WDATE`,  `HOUR`)
    VALUES ('7', '3', '2019-01-05',  '10');

INSERT INTO `workson` (`STAFFNO`, `CAMPAIGN_NO`, `WDATE`,  `HOUR`)
    VALUES ('8', '3', '2019-01-05',  '10');

INSERT INTO `staffongrade` (`STAFFNO`, `GRADE`, `STARTDATE`, `FINISHDATE`)
    VALUES ('6', '1', '2019-01-05','2019-01-06');
INSERT INTO `staffongrade` (`STAFFNO`, `GRADE`, `STARTDATE`, `FINISHDATE`)
    VALUES ('7', '2', '2019-01-05','2019-01-06');
INSERT INTO `staffongrade` (`STAFFNO`, `GRADE`, `STARTDATE`, `FINISHDATE`)
    VALUES ('8', '3', '2019-01-05','2019-01-06');

update invoice set STATUS = 'UNPAID' where INVOICENO = 1;
update invoice set STATUS = 'UNPAID' where INVOICENO = 2;
update invoice set STATUS = 'UNPAID' where INVOICENO = 3;
```

Next, we need to check the campaigns which actual cost is higher that the estimated. It is important to note that the *sp_finish_campaign* has been called for the new title campaign *Yellow*.

*SELECT **
*FROM campaign;*

| CAMPAIGN_NO | TITLE | CUSTOMER_ID | THEME | CAMPAIGNSTARTDATE | CAMPAIGNFINISHDATE | ESTIMATEDCOST | ACTUALCOST |
|---|---|---|---|---|---|---|---|
| 1 | Blue | 1 | Fall | 2018-01-01 | 2019-07-01 | 1500 | 1350 |
| 2 | Red | 3 | Spring | 2019-01-03 | 2019-10-31 | 1000 | 675 |
| 3 | Yellow | 3 | Winter | 2019-01-05 | 2019-10-31 | 1200 | 1390 |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

The only campaign that has the actual cost higher than the estimated is the new campaign inserted in the database (*CAMPAIGN_NO=3*).

The goal of the next procedure is to update the *ACTUAL COST* with a penalty decided by the user (*rate* is the input parameter).

```
drop procedure if exists penalty;
delimiter //

CREATE PROCEDURE penalty (in rate INT)


BEGIN
    DECLARE p_campaignno INT(11) DEFAULT 0;
    DECLARE p_actualcost FLOAT DEFAULT 0;

    SELECT CAMPAIGN_NO, ACTUALCOST INTO p_campaignno, p_actualcost
    FROM campaign
    WHERE ACTUALCOST > ESTIMATEDCOST;

    UPDATE campaign
    SET ACTUALCOST=ACTUALCOST + (ACTUALCOST*(rate/100))
    WHERE CAMPAIGN_NO = p_campaignno;


END//
delimiter ;
```

As we said before, we only identified one invoice which actual cost is higher than the estimated and it is that campaign with *title='YELLOW'*.
If we are applying a 10% penalty to that campaign in the actual cost we would expect:

ACTUAL COST = 1390 + (1390 *(10/100)) = 1529.

Lets call the procedure and verify that it has been correctly implemented.

*call penalty(10);*

*SELECT **
*FROM campaign;*

| CAMPAIGN_NO | TITLE | CUSTOMER_ID | THEME | CAMPAIGNSTARTDATE | CAMPAIGNFINISHDATE | ESTIMATEDCOST | ACTUALCOST |
|---|---|---|---|---|---|---|---|
| 1 | Blue | 1 | Fall | 2018-01-01 | 2019-07-01 | 1500 | 1350 |
| 2 | Red | 3 | Spring | 2019-01-03 | 2019-10-31 | 1000 | 675 |
| 3 | Yellow | 3 | Winter | 2019-01-05 | 2019-10-31 | 1200 | 1529 |

This result matches our expectation.