ADV. DB & BIG DATA TP2 REPORT

Exercise 1

1. We rank the employees ordering by hiredate and we partition then by deptno, then we take the 2 first of each.

```
SELECT *
FROM (
    SELECT empno, ename, efirst, deptno, hiredate,
    RANK() OVER (PARTITION BY deptno ORDER BY hiredate DESC) AS rank
    FROM EMP
) where rank < 3;
```

2.

```
SELECT empno, ename, sal,
       SUM(sal) OVER (
           ORDER BY empno
           ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
       ) AS cumulative_sal
FROM EMP;
```

Exercise 2

1. Table created, now let's begin with the questions.
2. After running the query, we have this runtime:

```
Total query runtime: 156 msec.
```

3.

3.1

As said at the lab, the msec is not really a good metric to analyze queries. We can use the EXPLAIN option before SELECT so we get a more detailed description of the query.

	QUERY PLAN text
1	HashAggregate (cost=114.75..114.77 rows=2 width=10)
2	Group Key: gender
3	-> Seq Scan on emp_medium_table (cost=0.00..114.50 rows=49 widt...
4	Filter: (manager_id = 7)

3.2

Now, we can run `SHOW data_directory;` to see where the postgre files are.

	data_directory text
1	C:/Program Files/PostgreSQL/17/data

That directory in my case is where the postgresql.conf is located. I opened it and edited the variable requested:

```
#session_preload_libraries = ''
shared_preload_libraries = 'pg_stat_statements'     # (change requires restart)
#jit_provider = 'llvmjit'       # JIT library to use
```

Now, after running create extension pg_stat_statements and then re-runing the query 10 times we get a mean of 0.3

4. (5 and 6) After adding the index we get a mean of 0.05 it seems now the mean is way smaller than without the index, as expected.

7.

	QUERY PLAN text
1	GroupAggregate (cost=0.28..5.40 rows=2 width=10)
2	Group Key: gender
3	-> Index Only Scan using manager_id_gender_index on emp_medium_table (cost=0.28..5.14 rows=49 widt...
4	Index Cond: (manager_id = 7)

It seems he cost is less now, before was 144, now was 0.28

Exercice 3

First create the table:

```
CREATE TABLE IF NOT EXISTS MY_OBJECTS (
    Object VARCHAR(255),
    Type VARCHAR(50)
);
```

Then, insert the views that contain the objects, taking the columns that contain the name and adding the type.

```
INSERT INTO MY_OBJECTS (Object, Type)
SELECT table_name AS Object, 'Table' AS Type
FROM information_schema.tables
WHERE table_schema = 'public'
UNION
SELECT column_name AS Object, 'Column' AS Type
FROM information_schema.columns
WHERE table_schema = 'public'
UNION
SELECT conname AS Object, 'Constraint' AS Type
FROM pg_constraint
WHERE connamespace = (SELECT oid FROM pg_namespace WHERE nspname = 'public');
```

And last, we view the table:

```
SELECT * FROM MY_OBJECTS
ORDER BY type
```

And here I show some rows of the table:

	object character varying (25)	type character varying (15)
79	mgr	Column
80	min_exec_time	Column
81	temp_blk_write_time	Column
82	pk_project_emp	Constraint
83	pk_emp	Constraint
84	pk_dependent	Constraint
85	ck_sal	Constraint
86	fk_dependent_emp	Constraint
87	mobtel_check	Constraint
88	pk_dept	Constraint
89	fk_project	Constraint
90	pk_project	Constraint
91	fk_emp	Constraint
92	fk_emp_dept	Constraint
93	efirst_ename_tel_unique	Constraint
94	emp_medium_table	Table
95	project	Table
96	sales_staff	Table
97	dept	Table
98	dependents	Table
99	emp	Table

Exercice 4

To access from windows, we need the command psql from cmd like this, and then, type the password:

```
psql -U postgres
```

Then, write:

```
postgres-# \c TP_1
```

And we are inside the database. We simply run a query to make sure all works:

```
TP_1=# SELECT * FROM emp;
 empno | ename  | efirst |    job    | mgr  |  hiredate  | sal  | comm |    tel     | deptno | mobtel
-------+--------+--------+-----------+------+------------+------+------+------------+--------+--------
  7369 | SMITH  | JOHN   | CLERK     | 7902 | 1980-12-17 |  800 |      | 0149545243 |     20 |
  7499 | ALLEN  | BOB    | SALESMAN  | 7698 | 1981-02-20 | 1600 |  300 | 0149547243 |     30 |
  7521 | WARD   | PETER  | SALESMAN  | 7698 | 1981-02-22 | 1250 |  500 | 0149545247 |     30 |
  7566 | JONES  | JOHN   | MANAGER   | 7839 | 1981-04-02 | 2975 |      | 0149545456 |     20 |
  7654 | MARTIN | JOE    | SALESMAN  | 7698 | 1981-09-28 | 1250 | 1400 | 0149545784 |     30 |
  7698 | BLAKE  | BOB    | MANAGER   | 7839 | 1981-05-01 | 2850 |      | 0149545254 |     30 |
  7782 | CLARK  | JOHN   | MANAGER   | 7839 | 1981-06-09 | 2450 |      | 0149545245 |     10 |
  7788 | SCOTT  | GUY    | ANALYST   | 7566 | 1982-12-09 | 3000 |      | 0149545249 |     20 |
  7844 | TURNER | PETER  | SALESMAN  | 7698 | 1981-09-08 | 1500 |    0 | 0149548243 |     30 |
```

Exercice 5

On the first terminal:

```
TP_1=# UPDATE EMP SET SAL = 5000 WHERE EMPNO = 7369;
UPDATE 1
TP_1=*# UPDATE EMP SET SAL = 7000 WHERE EMPNO = 7369;
UPDATE 1
TP_1=*#
```

If then we try to see the update, we see it's updated:

```
TP_1=# UPDATE EMP SET SAL = 7000 WHERE EMPNO = 7369;
UPDATE 1
TP_1=*# SELECT sal FROM EMP WHERE EMPNO = 7369
TP_1-*# ;
 sal
------
 7000
(1 fila)
```

On the second terminal, we won't see it until terminal 1 does the command COMMIT; then, the second one will be able to see it.

We will try to do an update on the second terminal:

```
TP_1=# UPDATE EMP SET SAL = 7000 WHERE EMPNO = 7369;
```

As we see, terminal 2 is locked (It doesn't let you type after it did the UPDATE query). This happens to avoid inconsistency, since the autocommit is off, terminal 2 isn't able to see the changes of terminal 1, so it could lead to various inconsistencies if we try to do transactions from both, so this database is configured so if a client is doing a transaction, other clients can't do one until the active one is committed or aborted.

The name of this mechanism is lock.

So, we do as said before:

```
TP_1=*# COMMIT;
COMMIT
TP_1=#
```

And then terminal 2 unlocks and also will be able to see the update that terminal 1 did.

```
TP_1=# UPDATE EMP SET SAL = 7000 WHERE EMPNO = 7369;
UPDATE 1
TP_1=*#
```