# Advanced Database - TP 2

# Advanced SQL

## Exercice 1. Analytics Queries . Window Queries

If you don't remember about Analytical / Windows Queries (very important!), please re-read this chapter.

1. Gets the 2 persons per department, who have arrived the latest in the company.

2. Show your analytical Skill and Invents an interesting query using Windows Functions (i.e.: a SELECT query on EMP table): The query should include the usage of "ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING".

## Exercice 2. Index & Explain Plan

In this Exercise, we are going to use an Index to speed UP drastically a query.

We'll see it's difficult to take time into account for performance when analyzing a query. Because it depends on many factors: CPU, RAM, and most important if your results are already in CACHE or not (if you execute a query N time in a row, the first time it will be slow, then very quick because in CACHE).

1. Execute this script (create 3 tables and load random data inside):

```
----------------------------------------
------------ Create Employee Table -------
----------------------------------------
CREATE TABLE EMP_MEDIUM_TABLE
       (EMPNO INTEGER,
        MANAGER_ID INTEGER,
        DEPT_ID VARCHAR(10),
```

```
        GENDER VARCHAR(2) not null,
        NAME VARCHAR(1000));

INSERT INTO EMP_MEDIUM_TABLE
SELECT
    S AS empno,
    ROUND(random()*100) AS manager_id,
    ROUND(random()*10) AS dept_id,
    (ARRAY['M','F'])[round(random())+1] AS gender,
    gen_random_uuid() AS name_hashed
FROM generate_series(1,5000) as S;


-----------------------------------------
------------ Create Project Table --------
-----------------------------------------

CREATE TABLE PROJECT_MEDIUM_TABLE
       (PROJECTNO integer,
        NAME_hashed VARCHAR(100));

INSERT INTO PROJECT_MEDIUM_TABLE
SELECT     S                                                    PROJECTNO,
           gen_random_uuid() AS NAME_hashed
FROM generate_series(1,5000) as S;


-----------------------------------------
---- Create Join table: Project-Employee -
-----------------------------------------

CREATE TABLE PROJECT_EMP_MEDIUM_TABLE
       (PROJECTNO integer,
        EMPNO integer);

INSERT INTO PROJECT_EMP_MEDIUM_TABLE
SELECT     ROUND(random()*100) AS                               PROJECTNO,
           ROUND(random()*100) AS                               EMPNO
FROM generate_series(1,5000) as S;
```

2. The goal of all exercice will be to tune the following query:

```
SELECT gender, count(*) from EMP_MEDIUM_TABLE where MANAGER_ID = 7 group by gender;
```

Execute it and Note the response time.

Note: Response Time in second is not really a good metric to analyze queries. Because this number will depends on many external parameters than the query:

- Hardware of the machine: CPU, RAM, Disk Read Speed (SSD is faster than HDD), network…,
- Number of persons connected to the database, number of queries in parallel ...

That's why we'll find others more precise metrics: I/O, CPU time, 'Consistent Gets', etc...

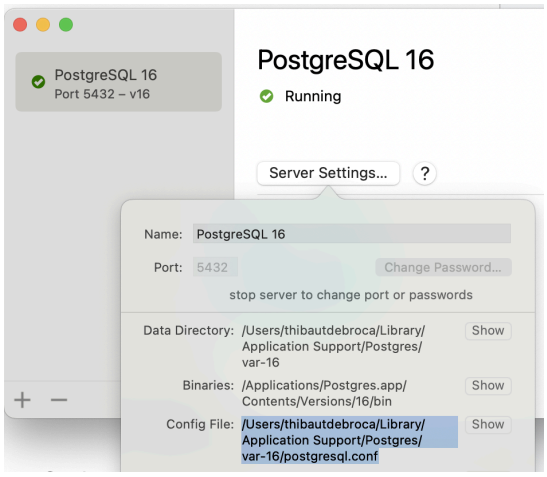
3.1. Use EXPLAIN plan to analyze the query:

```
EXPLAIN SELECT gender, count(*) from EMP_MEDIUM_TABLE where MANAGER_ID = 7 group by gender;
```

Keep the Execution Plan for this query

3.2. **Activate stats**

Find the conf file of your postgresql instance. You should go here and click "show":




Open postgresql.conf in a text editor.

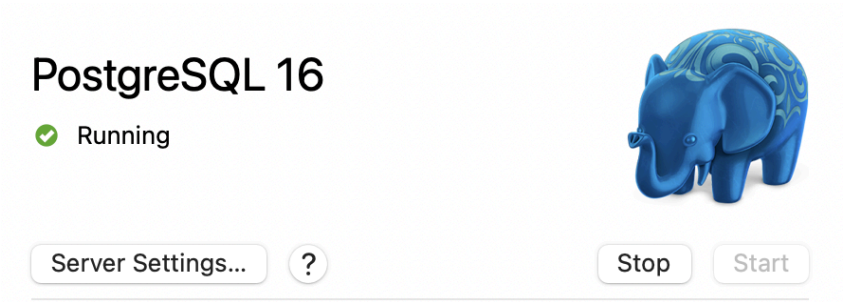
Find "shared_preload_libraries", remove trailing comment "#" and change value to "pg_stat_statements". This will gives this line (you can copy/paste):

```
shared_preload_libraries = 'pg_stat_statements'          # (change requires restart)
```

As a screenshot

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

Stop/start:




Run:

```
create extension pg_stat_statements;
```

Now you can run:

```
SELECT * FROM pg_stat_statements

        WHERE  query like '%group by gender' and query not like 'EXPLAIN%'
```

Notice the values of “calls” and “mean_exec_time” and explain them.

Re-run 10 times:

```
SELECT gender, count(*) from EMP_MEDIUM_TABLE where MANAGER_ID = 7 group by gender;
```

Keep the values of “calls” and “mean_exec_time”.

4. Add a **covering index** on both columns fetched:

```
create index MANAGER_ID_GENDER_INDEX ON EMP_MEDIUM_TABLE(MANAGER_ID, GENDER);
```

This should accelerate further queries with clause on the 2 columns.

5. Reset the stats:

```
select pg_stat_statements_reset();
```

If you run:

```
SELECT * FROM pg_stat_statements

        WHERE  query like '%group by gender' and query not like 'EXPLAIN%'
```

You should see nothing

6. Re-run query the same query 10 times:

```
SELECT gender, count(*) from EMP_MEDIUM_TABLE where MANAGER_ID = 7 group by gender;
```

Note the mean_exec_time and compare with the one before you added the INDEX.

7. Use the EXPLAIN plan again and compare the plan before the INDEX (you should see a difference and explain it).

## Exercice 3. Data Dictionary

*The data dictionary is a set of tables that store descriptions of database objects. A user can access the data dictionary through views (information_schema.tables, information_schema.views, pg_constraint, etc.).*

Use these views to create a table called MY_OBJECTS with 2 columns : Object (Name of your Object) / Type (Table, column, constraint …).

This will look like that:

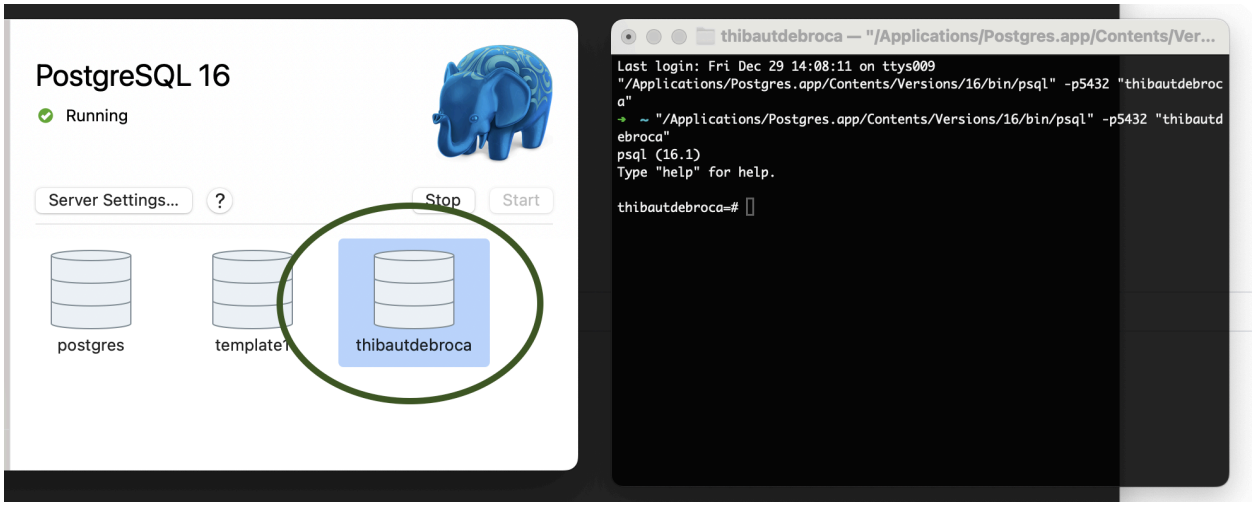
| Object | Type |
|---|---|
| EMP | Table |
| EMPNO | Column |
| PK_EMP | Constraint |

## Exercice 4. Use postgres via CLI

*Here you will see that you can manipulate the Database with a simple shell.*

*1. Launch CLI*

Double click on your database, this should open a CLI connected to your database.




2. On the CLI: Launch a

```
SELECT * FROM EMP;
```

## Exercice 5. Transaction Part 1 - Beginner

0. Open 2 sql clients on the same Database.

For this Exercise, you should have 2 clients either:

- Open 1 SQLDeveloper + 1 CLI Session.
- Open 2 CLI Sessions.

1. **Be sure to have autocommit disabled (default behaviour in SQL).**

**1.1 on the CLI, run:**

```
\set AUTOCOMMIT off
```

**1.2 On the UI, untick that option:**


2. **On the first client**: Launch:

```
UPDATE EMP SET SAL = 5000 WHERE EMPNO = 7369
```

3. **On the first client**, launch:

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

Can you see the UPDATE of the salary for the employee ?

4. **On the second client**, launch:

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

Can you see the UPDATE of the salary for the employee ? Why ? How could you make the update available for the second client ?

5. **On the second client**, launch:

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

What is happening and why ?

What is the name of this mechanism ? (hint: *ea*l**k)

6. **On the first client**, launch:

```
COMMIT
```

7. **On the second client**, launch:

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

What happened and why ?