# Project 1 - Fabian Pascal's Experiment

## INSTRUCTIONS

1. This project is an individual project.
2. Submit the file "`answers.sql`" and the file "`plans.pdf`" by **Friday 5 February 2021, 18:30** to Luminus:

   "`Files > Projects > Fabian Pascal > Submissions`.
3. Do not submit other files.
4. The penalty for late submission is 1 mark per started period of 6 hours after the submission deadline.
5. Follow the naming requirements, include your student number and write your answers the file "`answers.sql`" as instructed.

---

In 1988, Fabian Pascal, a database designer and programmer (and prolific blogger on database issues, see [1]) published the article "SQL Redundancy and DBMS Performance" in the journal Database Programming & Design [2]. He compared and discussed the plan and performance of seven equivalent SQL queries with different database management systems. For the experiment he proposed a schema and a synthetic instance on which the seven queries are executed.

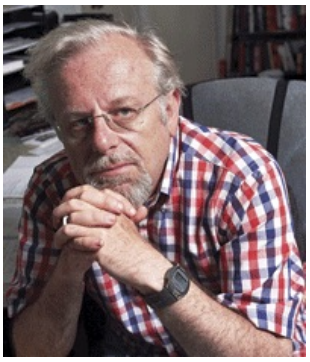

Figure 1: Fabian Pascal

At the time, the different systems could or could not execute all the queries and the performances significantly differed among and within individual systems while one would expect the DBMS optimiser to choose the same optimal execution plan for these queries.

In this project, we propose to replay Fabian Pascal's experiment with PostgreSQL current version.

[1] Database Debunkings. `http://www.dbdebunk.com`. Visited on 22 January 2021.
[2] F. Pascal. "SQL Redundancy and DBMS Performance". In: Database Programming & Design 1.12

Fabian Pascal proposed a very simple schema with two tables: `employee` and `payroll`. The table `employee` records information about employees of a fictitious company. Employees have an employee identifier, a first name and a last name, an address recorded as a street address, a city, a state and a zip code. The table `payroll` records, for each employee, her bonus and salary.

Download the following files from Luminus "`Files > Projects > Fabian Pascal > Code`"

`answers.sql`,

`FPSchema.sql`,

`FPPopulate.sql`,

and `FPTest.sql`.

Create a database in PostgreSQL. Use the "`FPSchema.sql`" SQL script to create the tables `employee` and `payroll` with the domains suggested in Fabian Pascal's original article. The excerpt of the script is given below.

```
CREATE TABLE employee (
        empid CHAR(9),
        lname CHAR(15),
        fname CHAR(12),
        address CHAR(20),
        city CHAR(20),
        state CHAR(2),
        zip CHAR(5)
);

CREATE TABLE payroll (
        empid CHAR(9),
        bonus INTEGER,
        salary INTEGER
);
```

Populate the database. PL/pgSQL is a procedural language to write code that can be executed by the PostgreSQL server directly. You may use or modify the following PL/pgSQL function to generate random strings of upper case alphabetical characters of a fixed length.

Use the "`FPPopulate.sql`" SQL script to generate random sample instance of the database.

```
CREATE or REPLACE FUNCTION random_string(length INTEGER) RETURNS TEXT AS
$$
DECLARE
        chars TEXT[] := '{A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z}';
        result TEXT := '';
        i INTEGER := 0;
BEGIN
        IF length < 0 then
                RAISE EXCEPTION 'Given_length_cannot_be_less_than_0';
        END IF;
        FOR i IN 1..length
        LOOP
                result := result || chars[1+random()*(array_length(chars, 1)-1)];
        END LOOP;
        RETURN result;
END;
$$ LANGUAGE plpgsql;
```

You may use or modify the following SQL DML code to insert data into the two tables.

```
INSERT INTO employee
SELECT
        TO_CHAR(g, '09999') AS empid,
        random_string(15) AS lname,
        random_string(12) AS fname,
        '500_ORACLE_PARKWAY' AS address,
        'REDWOOD_SHORES' AS city,
        'CA' AS state,
        '94065' AS zip
FROM
        generate_series(0, 9999) g;
```

```
INSERT INTO payroll(empid, bonus, salary)
SELECT
        per.empid,
        0 as bonus,
        99170 + ROUND(random() * 1000)*100 AS salary
FROM
        employee per;
```

Fabian Pascal proposed to find the identifier and the last name of the employees earning a salary of $189170. One possible SQL query to answer this question is as follows.

```
SELECT per.empid, per.lname
FROM employee per, payroll pay
WHERE per.empid = pay.empid AND pay.salary = 189170;
```

Use the "`FPTest.sql`" SQL script to create a function that measure the average execution time of 1000 evaluations of query. The code of the function is given below.

To measure the planning and execution times of a query, we create a PL/pgSQL function called `test` that takes an SQL query `Q` and a number $N$ as its parameters and returns the **average planning and execution times**, as reported by `EXPLAIN ANALYZE Q` over $N$ executions of the query `Q`.

```
CREATE OR REPLACE FUNCTION test (TEXT, INT) RETURNS TEXT AS
$$
DECLARE
r RECORD;
p TEXT;
e TEXT;
ap NUMERIC := 0;
ae NUMERIC := 0;
BEGIN
FOR i IN 1..$2
LOOP
FOR r in EXECUTE 'EXPLAIN␣ANALYZE␣' || $1
LOOP
IF r::TEXT LIKE '%Planning%'
THEN
p := regexp_replace( r::TEXT, '.*Planning␣(?:T|t)ime:␣(.*)␣ms.*', '\1');
END IF;
IF r::TEXT LIKE '%Execution%'
THEN
e := regexp_replace( r::TEXT, '.*Execution␣(?:T|t)ime:␣(.*)␣ms.*', '\1');
END IF;
END LOOP;
ap := ap + (p::NUMERIC − ap) / i;
ae := ae + (e::NUMERIC − ae) / i;
END LOOP;
RETURN ROUND(ap, 2) || '␣:␣' || ROUND(ae, 2) ;
END;
$$ LANGUAGE plpgsql;
```

Run the PL/pgSQL function above with the Query Tool in pgAdmin 4.

For example, you can measure the planning and execution times of a query above by running the following SQL query with the Query Tool in pgAdmin 4.

```
SELECT test('
        SELECT per.empid, per.lname
        FROM employee per, payroll pay
        WHERE per.empid = pay.empid AND pay.salary = 189170;
', 1000);
```

Question 1 (5 points): "There's more than one way to do it" (Perl motto).

We investigate different but equivalent SQL queries that answer the same question and meet the requirements indicated, respectively.

(a) Write a simple query with OUTER JOIN and only IS NULL or IS NOT NULL conditions in the WHERE clause.

(b) Write a nested query with a correlated subquery in the WHERE clause.

(c) Write a nested query with an uncorrelated subquery in the WHERE clause.

(d) Write a nested query with an uncorrelated subquery in the FROM clause.

(e) Write a double-negative single nested query with a correlated subquery in the WHERE clause.

Queries that do not execute with PostgreSQL, do not strictly and minimally follow the requirements or do not produce the correct answer may receive 0 mark or be penalised.

Write each query in the space indicated in the file "`answers.sql`". Tabulate the average planning and execution times over 1000 executions for each query as a comment in the space indicated in the file "`answers.sql`" .

Question 2 (3 points): The Long Way (just "Don't repeat yourself").

We investigate the constructions that may prevent PostgreSQL from optimizing a query.

(a) Propose a new query, different from the above queries, that answers the same question and is non-trivially as slow as possible. Do not modify the schema and the data.

Avoid joining the same table multiple times, using the `SLEEP` function, creating temporary tables, and other devices that may be arbitrated as unnecessary at the discretion of the marking team.

The execution of the query must terminate on your machine and you should be able to measure and indicate its average execution time over 1000 executions.

This question is marked competitively, first on the speed and then on the interest and originality of the answer. Only the slowest, most interesting and most original queries will receive more than 1 or 0 mark. Note that identical answers are correspondingly less original.

Write the query in the space indicated in the file "`answers.sql`" . Tabulate the average planning and execution times over 1000 executions for the query as a comment in the space indicated in the file "`answers.sql`" . The marking team will rerun your query in a separate server to ensure that the comparative evaluation is fair.

Queries that do not execute with PostgreSQL or do not produce the correct answer shall receive 0 mark.

Question 3 (2 points): The Better Way.

We investigate the addition of constraints or indexes to improve the performance of the queries in Question 1.

(a) Add constraints and indexes. Only add those that are strictly necessary to improve the performance of (some of) the queries in Question 1. Write the SQL DDL statements that create the constraints and indexes in the space indicated in the file "`answers.sql`" . Tabulate the average planning and execution time over 1000 executions for each of the five queries as a comment in the space indicated in the file "`answers.sql`" .

(b) For each of the five queries of Question 1, save a picture of the query execution plan before and after adding the constraints or indexes. Submit the ten pictures of the ten plans labelled with the query and the indication"before" and"after", accordingly, in a single PDF file called "`plans.pdf`" . Write your student number as a header of the series of pictures.

– END OF PAPER –