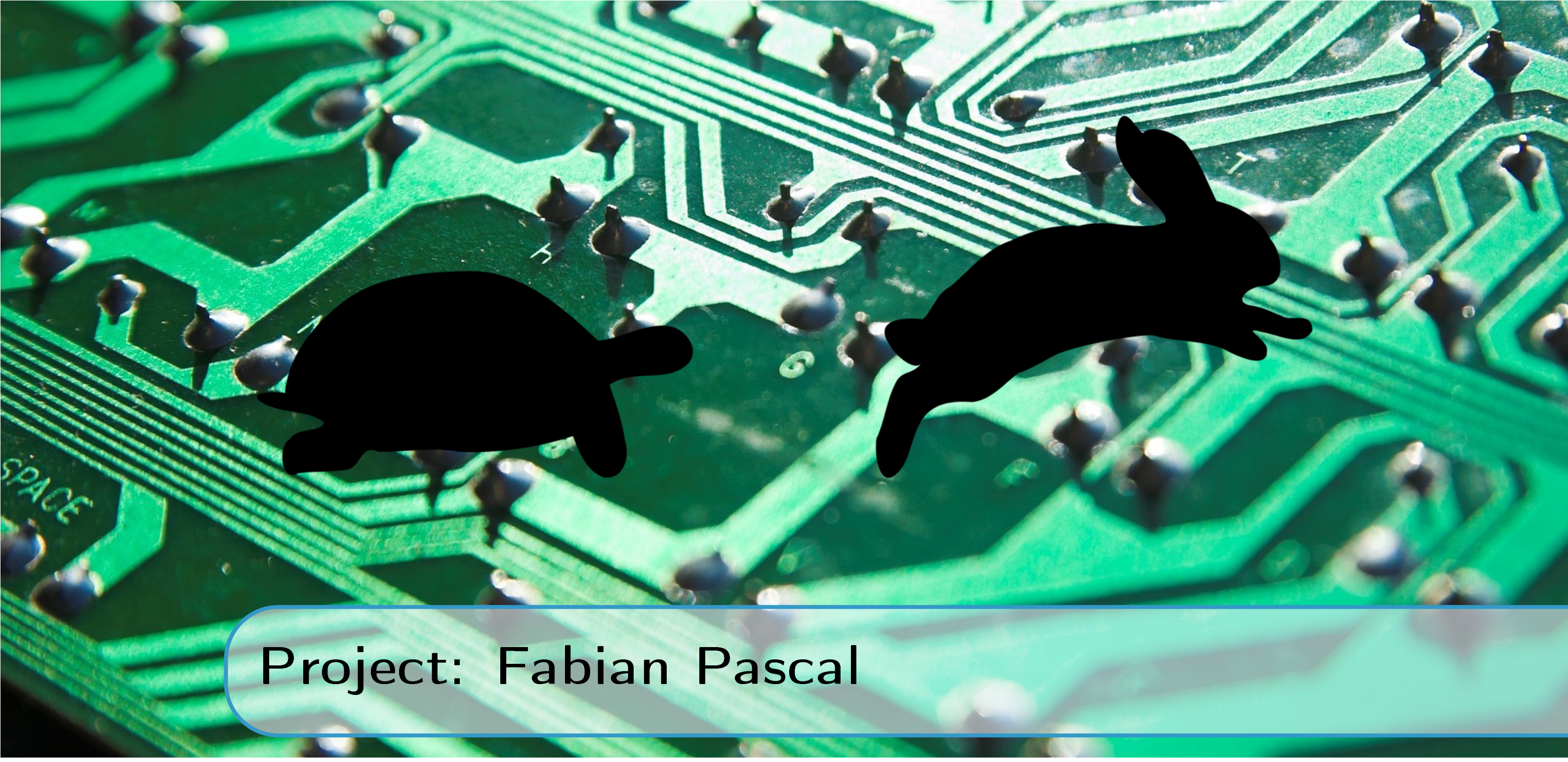

Project: Fabian Pascal

Instructions

- This project is an individual project.
- The maximum mark for this project is 10 points for 10% of the final mark.
- Submit your answers by **Friday 11 February 2022, 17:00** to Luminus.
- There is strictly no late submission.
- Download the SQL files
  - "`FPSchema.sql`",
  - "`FPPopulate.sql`",
  - and "`FPTest.sql`".

  from Luminus

  "`Files > Cases  > Fabian Pascal`"
- Download the template answer file "`answers.sql`" from Luminus
  "`Files > Projects > Fabian Pascal`".
- Write your answers in the indicated sections of the file "`answers.sql`" .
- Submit the completed file "`answers.sql`" to Luminus
  "`Files > Projects > Fabian Pascal > Submissions`".
- Do not submit other files.
- Follow the naming requirements, include your student number and write your answers in the corresponding sections of the file "`answers.sql`" as instructed.
- The SQL code that you submit should run without error in PostgreSQL version 14.

In 1988, Fabian Pascal, a database designer and programmer and prolific blogger on database issues, (see `http://www.dbdebunk.com`) published the article "SQL Redundancy and DBMS Performance" in the journal Database Programming & Design. 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.

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. (0 points) 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.

   (a) 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.

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

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

   (b) Populate the database. PL/pgSQL is a procedural language to write code that can be executed by the PostgreSQL server directly. Use the "`FPPopulate.sql`" SQL script to generate a random instance of the database.

   Use the PL/pgSQL "`random_string()`" function to generate random strings of upper case alphabetical characters of a fixed length.

   ```sql
   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;
   ```

   Use the following SQL DML code to call the function and insert data into the two tables.

   ```sql
   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;
```

(c) 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`.

The code of the function is given below and is available in the "`FPTest.sql`" SQL script.

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

(d) Run the PL/pgSQL function above with the Query Tool in pgAdmin 4. You can measure the planning and execution times of a query by running the following SQL query with the Query Tool in pgAdmin 4. The following is an example.

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

2. (5 points) "There's more than one way to do it" (Perl motto). Fabian Pascal proposed to evaluate the performance of different equivalent queries finding the identifier and the last name of the employees earning a salary of $189170. Let us consider the following reference SQL query.

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

We are looking for different but equivalent SQL queries that produce the same result in the same order for database instances generated with the given scripts.

Write each query in the space indicated in the file "`answers.sql`".

Tabulate the average planning and execution times over 100 executions for each query as a comment in the space indicated below each query in the file "`answers.sql`".

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

(a) Complete the following query so that it produces the same result as the reference query in the same order. Only modify the "`WHERE`" clauses (replace the occurrence of "`TRUE`" only).

```
SELECT per.empid, per.lname
FROM employee per FULL OUTER JOIN payroll pay
    ON per.empid = pay.empid AND pay.salary = 189170
WHERE TRUE
ORDER BY per.empid, per.lname;
```

Solution:

```
SELECT per.empid, per.lname
FROM employee per FULL OUTER JOIN payroll pay
    ON per.empid = pay.empid AND pay.salary = 189170
WHERE per.empid IS NOT NULL AND pay.empid IS NOT NULL
ORDER BY per.empid, per.lname;
```

(b) Complete the following query so that it produces the same result as the reference query in the same order. Only modify the "`FROM`" and "`WHERE`" clauses (replace the occurrences of "`TRUE`" only).

```
SELECT per.empid, per.lname
FROM employee per, (SELECT TRUE) AS temp
WHERE TRUE
ORDER BY per.empid, per.lname;
```

Solution:

```
SELECT per.empid, per.lname
FROM employee per,
  (SELECT * FROM payroll pay WHERE pay.salary = 189170) AS temp
  WHERE per.empid = temp.empid
ORDER BY per.empid, per.lname;
```

(c) Complete the following query so that it produces the same result as the reference query in the same order. Only modify the the "`WHERE`" clause of the subquery (replace the occurrence of "`'TRUE'`" only).

```
SELECT per.empid, per.lname
FROM employee per
WHERE per.empid NOT IN (SELECT 'TRUE')
ORDER BY per.empid, per.lname;
```

Solution:

```
SELECT per.empid, per.lname
FROM employee per
WHERE per.empid NOT IN (SELECT pay.empid
  FROM payroll pay
  WHERE pay.salary <> 189170)
ORDER BY per.empid, per.lname;
```

It is recommended that you use `EXPLAIN (F7)` of pgAdmin 4 and look at the graphical representation and analysis of the query plans for your own learning. It is also recommended that you try other possible equivalent queries. It is recommended that you use the `test()` stored function to find the average planning time and the average execution time of 1000 executions of each query.

3. (5 points) The Long Way (just "Don't repeat yourself" - or others -). We investigate constructions that may prevent PostgreSQL from optimizing a query.

   (a) Propose a new query that produces the same result as the reference query in the same order for database instances generated with the given scripts and that is as slow as possible.

   Do not modify the schema or the data.

   Avoid joining the same table multiple times, using a sleep function, creating temporary tables, or 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 20 executions (use the `test()` stored function, the marking team may ask you for a demonstration).

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 receive more than 1 mark. Note that identical (even coincidentally so) 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 20 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.

Solution: See sample answers in the Luminus forum later.