

The image shows a close-up of a green printed circuit board (PCB) with intricate silver-colored circuit traces and numerous gold-plated pins. Overlaid on the circuit are two black silhouettes: a turtle on the left and a rabbit on the right, both facing right. The turtle is positioned as if it is moving slowly, while the rabbit is in a jumping or running pose. Below the circuit, there is a semi-transparent blue rounded rectangle containing the project title.

Project: Fabian Pascal

Instructions

- This project is an individual project.
 - Submit your answers by **Friday 17 September 2021, 17:00** to Luminus.
 - There is strictly no possibility of 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 the file “answers.sql” as instructed.
 - The SQL code that you submit should run without error in PostgreSQL version 13.
-

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.

```
1 CREATE TABLE employee (  
2   empid CHAR(9),  
3   lname CHAR(15),  
4   fname CHAR(12),  
5   address CHAR(20),  
6   city CHAR(20),  
7   state CHAR(2),  
8   zip CHAR(5));
```

```
1 CREATE TABLE payroll (  
2   empid CHAR(9),  
3   bonus INTEGER,  
4   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.

```
1 CREATE or REPLACE FUNCTION random_string(length INTEGER) RETURNS TEXT AS  
2 $$  
3 DECLARE  
4   chars TEXT[] := '{A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z}';  
5   result TEXT := '';  
6   i INTEGER := 0;  
7 BEGIN  
8   IF length < 0 then  
9     RAISE EXCEPTION 'Given length cannot be less than 0';  
10  END IF;  
11  FOR i IN 1..length  
12  LOOP  
13    result := result || chars[1+random()*(array_length(chars, 1)-1)];  
14  END LOOP;  
15  RETURN result;  
16 END;  
17 $$ LANGUAGE plpgsql;
```

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

```
1 INSERT INTO employee  
2   SELECT  
3     TO_CHAR(g, '09999') AS empid,  
4     random_string(15) AS lname,  
5     random_string(12) AS fname,  
6     '500 ORACLE PARKWAY' AS address,  
7     'REDWOOD SHORES' AS city,  
8     'CA' AS state,  
9     '94065' AS zip  
10  FROM  
11  generate_series(0, 9999) g;
```

```

1 INSERT INTO payroll(empid, bonus, salary)
2   SELECT
3     per.empid,
4     0 as bonus,
5     99170 + ROUND(random() * 1000)*100 AS salary
6   FROM
7     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.

```

1 CREATE OR REPLACE FUNCTION test (TEXT, INT) RETURNS TEXT AS
2 $$
3 DECLARE
4   r RECORD;
5   p TEXT;
6   e TEXT;
7   ap NUMERIC := 0;
8   ae NUMERIC := 0;
9 BEGIN
10  FOR i IN 1..$2
11  LOOP
12    FOR r in EXECUTE 'EXPLAIN ANALYZE ' || $1
13    LOOP
14      IF r::TEXT LIKE '%Planning%'
15      THEN
16        p := regexp_replace( r::TEXT, '.*Planning (?:T|t)ime: (.*?) ms.*', '\1');
17        END IF;
18      IF r::TEXT LIKE '%Execution%'
19      THEN
20        e := regexp_replace( r::TEXT, '.*Execution (?:T|t)ime: (.*?) ms.*', '\1');
21        END IF;
22      END LOOP;
23      ap := ap + (p::NUMERIC - ap) / i;
24      ae := ae + (e::NUMERIC - ae) / i;
25    END LOOP;
26    RETURN ROUND(ap, 2) || ' : ' || ROUND(ae, 2) ;
27 END;
28 $$ 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.

```

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

```

2. (3 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.

```

1 SELECT per.empid, per.lname
2 FROM employee per, payroll pay
3 WHERE per.empid = pay.empid AND pay.salary = 189170
4 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 1000 executions for each query as a comment in the space indicated in the file “`answers.sql`”.

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

- (a) (1 point) Complete the following query so that it produces the same result as the reference query in the same order. Only modify the “`ON`” and “`WHERE`” clauses (replace the occurrences of “`TRUE`” only).

```

1 SELECT per.empid, per.lname
2 FROM employee per RIGHT OUTER JOIN payroll pay
3     ON TRUE AND pay.salary = 189170
4 WHERE TRUE
5 ORDER BY per.empid, per.lname;

```

Solution:

```

1 SELECT per.empid, per.lname
2 FROM employee per RIGHT OUTER JOIN payroll pay
3     ON per.empid = pay.empid AND pay.salary = 189170
4 WHERE per.empid IS NOT NULL
5 ORDER BY per.empid, per.lname;

```

- (b) (1 point) 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).

```

1 SELECT per.empid, per.lname
2 FROM employee per, (SELECT TRUE) AS temp
3 WHERE TRUE
4 ORDER BY per.empid, per.lname;

```

Solution:

```

1 SELECT per.empid, per.lname
2 FROM employee per,
3     (SELECT * FROM payroll pay WHERE pay.salary = 189170) AS temp
4 WHERE per.empid = temp.empid
5 ORDER BY per.empid, per.lname;

```

- (c) (1 point) 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 occurrences of “TRUE” only).

```

1 SELECT per.empid, per.lname
2 FROM employee per
3 WHERE NOT EXISTS (
4     SELECT TRUE)
5 ORDER BY per.empid, per.lname;

```

Solution:

```

1 SELECT per.empid, per.lname
2 FROM employee per
3 WHERE NOT EXISTS (SELECT *
4     FROM payroll pay
5     WHERE pay.empid = per.empid AND pay.salary <> 189170)
6 ORDER BY per.empid, per.lname;

```

3. (4 points) The Long Way (just “Don’t repeat yourself”). 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 and 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 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.

Solution: See sample answers in the Luminus forum later.

– END OF PAPER –