

Database Benchmarking Project

Project Part I

Data Generation & System Selection

Data Generation script: DatabaseGeneration.java

System Selected: PostgreSQL Version 10.7

Data Generation:

Implemented a script in Java to automate the data generation process. The script has been given the connection parameters to the database which is running on our local Windows/Mac System. When the script is run, the connection to the database is established and using the established connection we are executing the CREATE TABLE statements and INSERT VALUES statement to populate the database.

The script when executed opens a Scanner instance through which we input the number of tables. Based on the number of tables, the script then takes in the name of each table and number of tuples in each table from user input. The scanner instance is then closed, and the data generation code is executed. In the data generation code, we first create the tables and populate them in a for loop. We are generating the data based on the *Attribute Specification of "Scalable" Wisconsin Benchmark Relations*. Data is generated in a for loop so one iteration of the for loop corresponds to one tuple in that table. The data generated for one tuple is then added to the batch of statements. Once all the tuples for that table have been generated and added to the batch, we execute the batch update statement to inject the data into the created database table. To establish connection to the PostgreSQL server from the script we have used JDBC API. The JDBC API uses JDBC drivers to connect with the database. (The dependency jar *postgresql-42.2.5* has been included with this file).

Reasons for choosing PostgreSQL:

- PostgreSQL is a strong relational database as it provides many good features when compared to other databases.
- One of the reasons we choose PostgreSQL is because its syntax is easier and more elegant. For example, for the case sensitive string comparison with MySQL or Oracle, the syntax would be like "SELECT * FROM SOME_TABLE WHERE UPPER (SOME_FIELD) LIKE UPPER ('SEARCH_THING')". whereas in PostgreSQL if you want to do case sensitive string comparison than it would be like this "SELECT * FROM SOME_TABLE WHERE SOME_FIELD LIKE 'SEARCH_THING'".
- PostgreSQL also supports many useful data types like Enumerated types, Network address types, Geometric/Spatial types, XML and JSON types, Boolean. One of the best data types that PostgreSQL adds to the mix is the array type, which lets you have arrays of any other type of data in a single field.
- PostgreSQL also features a pretty robust set of operators and functions for testing, comparing, manipulating, and converting arrays.

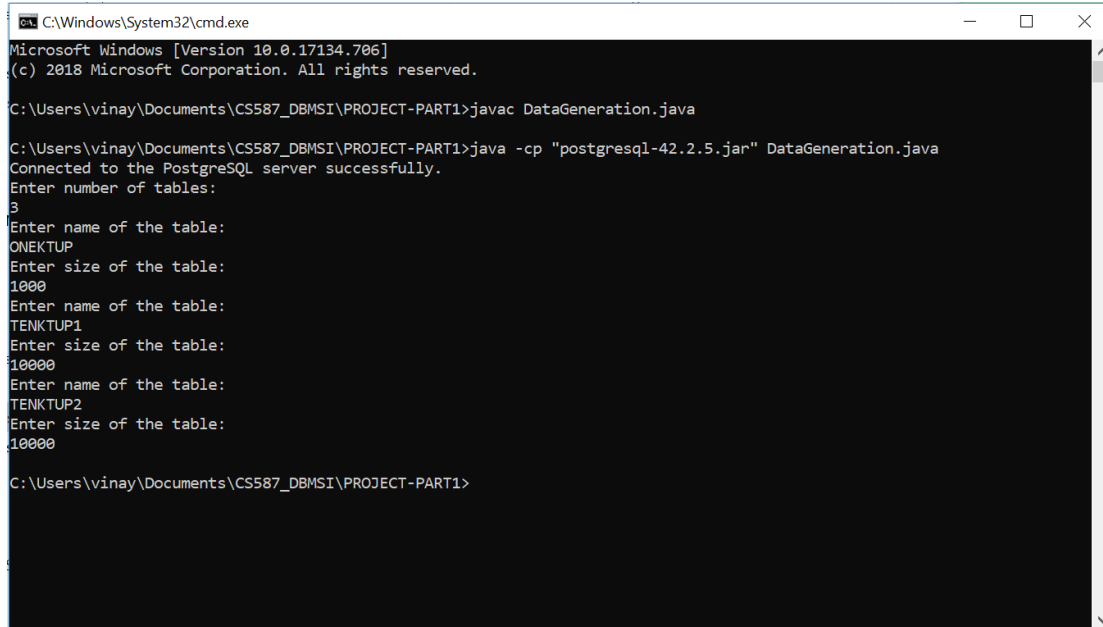
- PostgreSQL offers support for a variety of popular languages; it supports Python, Perl, Tcl, and PL/pgSQL4, but there are optional modules for Java, R, PHP, Ruby, Scheme, and Unix shell. This means that you can construct your procedural logic in a syntax that you are comfortable with, or that best lends itself to your task.
- PostgreSQL has a nice GUI tool like Pgadmin. It is very easy to write and executing queries in Pgadmin.
- Also, PostgreSQL is built for truly massive databases. In fact, there's no maximum database size in PostgreSQL, and individual tables can be up to 32 terabytes a piece with upwards of 250 columns. Even individual fields can contain up to a gigabyte of data.

We populated 3 tables namely:

ONEKTUP with 1000 tuples

TENKTUP1 with 10000 tuples

TENKTUP2 with 10000 tuples



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\vinay\Documents\CS587_DBMSI\PROJECT-PART1>javac DataGeneration.java

C:\Users\vinay\Documents\CS587_DBMSI\PROJECT-PART1>java -cp "postgresql-42.2.5.jar" DataGeneration.java
Connected to the PostgreSQL server successfully.
Enter number of tables:
3
Enter name of the table:
ONEKTUP
Enter size of the table:
1000
Enter name of the table:
TENKTUP1
Enter size of the table:
10000
Enter name of the table:
TENKTUP2
Enter size of the table:
10000

C:\Users\vinay\Documents\CS587_DBMSI\PROJECT-PART1>
```

Attached below is ONEKTUP table screenshot.

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the database structure, with the 'onektup' table selected under the 'public' schema. The main window shows the 'Data Output' tab for the 'onektup' table. The table contains 10 rows of data, each representing a different integer type and its corresponding value. The columns are: unique1 integer, unique2 [PK] integer, unique3 integer, two integer, four integer, ten integer, twenty integer, onepercent integer, tenpercent integer, twentypercent integer, fiftypercent integer, and evenor integer.

	unique1 integer	unique2 [PK] integer	unique3 integer	two integer	four integer	ten integer	twenty integer	onepercent integer	tenpercent integer	twentypercent integer	fiftypercent integer	evenor integer
1	589	0	589	1	1	9	9	89	9	4	1	
2	239	1	239	1	3	9	19	39	9	4	1	
3	948	2	948	0	0	8	8	48	8	3	0	
4	596	3	596	0	0	6	16	96	6	1	0	
5	186	4	186	0	2	6	6	86	6	1	0	
6	84	5	84	0	0	4	4	84	4	4	0	
7	656	6	656	0	0	6	16	56	6	1	0	
8	579	7	579	1	3	9	19	79	9	4	1	
9	969	8	969	1	1	9	9	69	9	4	1	
10	350	9	350	0	2	0	10	50	0	0	0	

One of the difficulties we faced while doing the first part of the project was in data generation especially string generation. At first, the string generation was confusing but with revisions of the code we were able to get clarity. Another difficulty we faced was installing Postgres 10.7.2 on Mac System but after one day's struggle we were able to successfully install it. One of the lessons we learned while implementing Part 1 of the project was the importance of understanding the data completely before populating the database. Understanding the data gives us a better hold in query execution. Testing the script generating the data for all possible edge cases before actually injecting data into the database is very important. Many of the data anomalies are easier to figure out with the script than after they are inserted into the database.