COL362 : Assignment 1

Aditi Singla (2014CS50277)
Vaibhav Bhagee (2014CS50297)
Praveen Kulkarni (2014CS50599)

29th January 2018

1 Database design

1.1 Details of tables

The tables which have been created are as follows:

- student(student_id, name)
- teacher(teacher_id, name)
- course(course_id, name)
- registers(student_id, course_id)
- teaches(teacher_id, course_id)
- section(section_number, course_id)

The foreign key constraints have been applied to the *teaches*, *registers* and *section* tables with both $DELETE\ CASCADE$ (to ensure that if an entry in the parent table is deleted, then the entries in children tables are deleted as well) and $UPDATE\ CASCADE$ (to ensure that if an entry in the parent table is updated, then the entries in the children tables are also updated).

1.2 Data generation and testing details

The data has been generated using a python script *generate.py* inside the $data_generator$ directory. This script takes in the number of rows as input (let's say n) and then generates SQL $INSERT$ commands for inserting n random entries in *student*, *teacher* and *course* tables. After that, we generate the $INSERT$ statements for the *registers* table (cross product of entries in *student* and *course* tables), the *teaches* (cross product of entries in *teacher* and *course* tables) and the *section* table (cross product of entries in *course* and $[A, B, C, D]$).

For testing purposes, first, $INSERT$ queries are generated with $n = 10$ using the above mechanism. Then, we generate segments of tests to test the various constraints enforced. Alongside the segments of tests, the constraints which they cover has been mentioned in the comments. The tests cover the $DELETE$ and $UPDATE$ cascades, $PRIMARY\ KEY$ constraints, etc.

For, testing various data loading methods in the following question, the queries are generated into the $big_queries.sql$ file.

For, bulk loading, we first $COPY$ the tables into the csv files and store theme in */tmp/* folder by default. After that, we empty the tables and then load them using $COPY$ from the csv files and note the time.

For JDBC, the same $big_queries.sql$ file is read line by line and executed programmatically, to note the times.

The whole data and table generation procedure has been automated through the makefile and the instructions have been outlined in the enclosed README file.

2 Comparing various data loading methods

Various methods of loading data into the tables have been studied. The times have been obtained for inserting 499849 records in the *registers* table. The results obtained with possible reasoning have been discussed below:

2.1 Using Bulk Load

We use the postgress for bulk loading of the tables from csv files. The generation of the csv files is described as above.

Observation: We observe that this method is the fastest of the three methods compared. This is due to the fact that bulk loads are heavily optimised for loading large volumes of data all at once.

They optimise over and above the basic system optimisations, like delay in checking of constraints until after populating the tables, batching of tuples while checking constraints in other tables etc.

2.2 Using Insert statements for each Tuple

We use a single $INSERT$ statement to load all the tuples into the database. The query for loading has been generated as discussed above.

Observation: We observe that this method is significantly fast as compared to loading the dataset using one $INSERT$ command per tuple. Also, this is slower as compared to *bulkloading* but faster as compared to loading data programmatically using $JDBC$.

By default, SQL commits the results after every $INSERT$ command, to the underlying file system based implementation of the database. Thus, one reason why loading all the tuples using a single insert statement is faster is that there is a single commit at the end of loading all the rows instead of commit after loading each row. Thus, a single insert statement enjoys the liberty and advantages of batching filesystem operations (efficiency due to use of buffer caches, reordering operations for optimisation etc.)

Even though this method is almost similar to bulk load (both try to load data at once), this is still slower. This is due to the fact that bulk loads are heavily optimised for large datasets, as discussed above.

2.3 Using JDBC

JDBC is the core API of Java that provides a standard interface to SQL-compliant databases like PostgreSQL, i.e. we use Java JDBC driver to interact with the PostgreSQL databases.

Installation: We need to setup PostgreSQL JDBC along with Java on our system. We need

to download the latest version of postgresql jar file to setup the PostgreSQL JDBC driver (Link can be found in the references). This can then be used along with -classpath option.

Now, we first connect to the existing database, the name for which can be taken as a command line argument. The userid and password for accessing the database have been assumed to be 'postgres' and 'postgres' respectively. Once the connection is established, tables have been created and populated using single insert command for each table. The times have then been recorded for populating all the tables separately.

Observation: We observe that this is slower than both the above methods of bulk loading and using $INSERT$ commands for each tuple. The possible reason behind this behaviour could be that JDBC provides an interface to connect to the database backend. While using the $INSERT$ commands involve direct transmission of query string to database backend, for further parsing, retrieval of data and transmission of results to the client, and hence is optimised and fast, on the other hand, using JDBC involves running a JAVA program, establishing a connection with the database and then sending over the query. This in itself takes more time and hence, is slower than the other two methods.

3 References

- https://stackoverflow.com/questions/46715354/how-does-copy-work-and-why-is-it-so-much-faster-than-insert
- https://www.tutorialspoint.com/postgresql/postgresql_java.htm
- https://askubuntu.com/questions/413585/postgres-password-authentication-fails
- https://jdbc.postgresql.org/download.html