Database Management Systems M149 – Fall 2019 – Project 1

Stefou Theodoros
CS2190002
theo.stefou@gmail.com

The work for the project was done on a linux machine, using Postgresql version 9.5.
The web application was developed using the Django framework. The code can be accessed through this github repository https://github.com/TheoStefou/M149-Project1.

Inside the tar of my report, the following can be found:

snapshots.tar.gz	- The snapshots of the web app and database
schema.pdf	- The schema of the database
stored_functions.txt	- The create function statements that I used to create the stored functions

Notes:

-Regarding the schema:
I created a table which holds the general information about the logs(log_id, timestamp, type, source_ip) called “log”, where type can be one of: access, receiving, received, served, replicate, remove. Then, for these types of logs we have available, I created some tables to hold their arguments using one to one relationships. More specifically, one table to hold the arguments of access logs, one table to hold the arguments of log types that can be found in dataxceiver (receiving, received, served) and then 2 more tables to hold block_ids and destination_ips for logs of types replicate/remove with one to many relationships.
I chose to have one-column primary keys across all tables to keep it simple. Therefore, in the last 2 tables mentioned, I added an extra column as a primary key which is basically a line counter (because one log may have multiple blocks I could not have log_id as primary key but also didn’t want to have log_id, block_id as composite key).
I created an extra table for user addresses, because I used django’s user model and wanted to extend it in an easy way. The manual states that one to one relationships is the way to go when extending users with simple fields.
For user history, the entries are in the form username, action, timestamp where username is unique for every user so I used username instead of id for readability. Action is just a string stating the function that was executed with what arguments or just the log id if the action was a new log insertion.

-Regarding the parsing of the logs and the db population:
In the github repository, you can find the python scripts I wrote using the python csv module that handled the parsing of the files and the creation of csv files that corresponded one to one with the tables of the schema. I created 3 scripts that cleaned the initial log files and kept just the useful information and one more that compiled the cleaned files to csv files-tables. After running these scripts, all I did was import the csv files into the tables using pg admin. Because I generated the primary keys of the data on my own in the scripts, after some insertions I also had to move the next value of the primary key sequence to the max of the current, because this is not done automatically.

I ignored logs that had exceptions or not valid http methods (1 of the 9 that are stated here in capital letters: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)

For more details about what logs I ignored, please take a look at the scripts as they are very straightforward (in the root of the github directory). (Mostly ones that had arguments that could not be converted to the format/type they are expected to be in.)

-Regarding the stored functions:

In stored_functions.txt you can see the create function statements I used to create the functions that are required.

Please notice that because there was a slight confusion in piazza regarding function 4, I have provided two solutions: 1 that just compares the number of logs in the range and 1 (bottom of file called 2nd_approach) that finds the total number of logs PER DAY for each block, then averages those numbers for each block. Even though the second approach is “more correct” and therefore the only one that should have been submitted, I decided to submit both for reference. Thus, my “final answer” is the one labeled 2nd approach.

-Regarding the indices that were added:

Query(args)	No index (sec)	With index* (on column)
1 ('2014-10-10 00:00:00', '2017-11-11 13:13:13')	1.080	0.590 (timestamp)
2('receiving', '2014-10-10 00:00:00', '2017-11-11 13:13:13')	1.758	0.796 (timestamp, type)
3 ('2018-11-08')	3.161	X (timestamp)
4_2nd_approach ('2016-11-08', '2018-11-11')	8.253	X (timestamp)
5 ()	30.572	15.642 (referer)
6 ()	1.173	Not worth
7 (200)	1.705	Not worth
8 ()	2.212	1.180 (type)
9 ()	2.285	1.221 (type)
10 (‘5.0’)	1.204	Not worth
11 ('GET', '2016-10-10 00:00:00', '2018-10-10 10:10:10')	3.421	X (timestamp)
12 ('GET', 'HEAD', '2016-10-10 00:00:00', '2018-10-10 10:10:10')	6.430	X (timestamp, type)
13 ('2016-10-10 00:00:00', '2018-10-10 10:10:10')	8.495	X (timestamp)

*X if no difference in execution time (the columns I tried indexing to check if there is a difference in time)

Conclusion: I decided to keep indices on timestamp, type and referer.
Postgres has created indices on primary and foreign keys by default, which ensures that the joins of most queries are optimized.

Most queries were pretty fast, so there was no need to waste space to decrease their execution time (and most likely the query optimizer would ignore the indices for queries that fast).