BIT2008A/ITEC2000A: Multimedia Data Management
Fall 2022

Lab 7

PostgreSQL

In this Lab session, we go through some exercises regarding JOIN and VIEW. Please follow the instructions step by step using pgAdmin.

Fruits.

1) Create a table named basket_a. It should have a column for fruit_id which is an INT and acts as the primary key. Its second column is fruit_a which is as VARCHAR holding the name of a fruit. We also have another column named num which is 0 by default and shows the number of a certain fruit that is available.
2) Create another table named basket_b similar to basket_a.
3) Insert some rows into both tables. The baskets should contain some items in common and some unique items that are available only in one of the baskets.
4) Write a JOIN query to show the sum of fruits that the baskets have in common. (Explore different options to find what type of JOIN you should use for that.)
5) Then, write another query to output all of the fruits that we have in both bags. The result table should have fruit_a, fruit_b, and num columns. If the item exists in both baskets, you should show the sum of the numbers. You may need to replace null values with 0 to make your calculations work. To do so, there is a function called COALESCE that you may find handy. Look up the documentation online to learn how to use that.
6) Create a VIEW to execute the query you wrote in Item 4 of these instructions. Assign common_fruits to its name. Checkout if SELECT works on that view as expected.
7) Write a Materialized VIEW named two_baskets to output the query that you executed in item 5. Check if SELECT works on it as expected.
8) By default, materialized VIEWs are initiated with the rows that exist when we are creating the view. There are other options, too. We can leave materialized VIEWs empty, then load them with data whenever we need. Drop two_baskets. Create two_baskets once more but this time add WITH NO DATA as its last line. Again, try to SELECT rows from your materialized VIEW.
9) You should REFRESH your materialized VIEW to load current data into it. Again, SELECT some rows to make sure the data is loaded.

Cities.

1) Create a table called city with a city_id as the primary key, city_name and country_name.
2) Enter 4 rows for Ottawa, Hamilton, Toronto, and London.
3) Create a VIEW called canada_city that returns only the rows that their country_name is Canada. Order the city names alphabetically.
4) Try to insert a city in England using the VIEW that you just created. What will happen to the VIEW that you just created and its corresponding table? This situation can cause a severe problem in real-world scenarios. We may create VIEWs to manage the access of different users to our tables. As a result, they should not be allowed to insert values beyond the scope of their VIEWs. In other words, they should not be able to insert a record into the table if it would not show up in their view. Rewrite your VIEW and add WITH CHECK OPTION as the last line to prevent such issues. Make sure that it works as expected. Try to insert New York into canada_city to see what will happen.
5) Create a materialized VIEW called city_backup. It simply takes a snapshot of our city table. When the tables get populated and heavy, it takes so long to REFRESH them with more recent data. While the refreshing is taking place, the materialized view is not in a stable situation, and queries on it will not proceed until loading new data is done. There is an option to refresh materialized VIEWs concurrently. To do so we need to define an INDEX on our VIEW.

6) Create a unique INDEX on the column city_id by finding the correct syntax from the documentation. Then enable concurrent data loading by executing, REFRESH MATERIALIZED VIEW CONCURRENTLY city_backup.