**Window function**

Window functions are the advanced feature used in any SQL Database and are handy to use. They give you an access to some of the data by performing aggregations on a set of rows which is specified on the current query row.

**How they are defined?**

Window functions are defined using the over() clause. Window function could be used with or without the other conditional queries, but they can’t be defined without the over() clause. Also, they should be called immediately after the window function.

**Why they are used?**

* Windows functions are helpful because in this we do not have to write sub select queries to get the entire information.
* **Unlike with Aggregate functions,** the rows remain separate.

In this given task, I have used some specialized window functions:

* **row\_number()** - The row number function is used to number every row within its partition. These rows can be ordered by using ORDER BY and can be numbered as per that.
* **Rank() - T**he window function rank is used to mention the row number within its partition but if the data is same when arranged i.e. two rows having rank as 3. The next row i.e. 5 will be ranked as 5. It means that it skips a rank 4.
* Dense\_rank() - Dense\_rank does not skip a rank although it gives the same rank to peer groups**.**
* lag(value, offset) - The lag value gives us the previous value within a partition. The offset decided whether the value would be just previous or may be 2 values behind. The default value of offset is 1 which gives us the recent previous value.
* lead(value, offset) - The lead window function gives us the next value within a partition. Value can be any as per the offset definition.

**Other than window functions, I have also used aggregate functions(SUM, COUNT) and some additional conditional queries like PARTITION BY for grouping mechanism for aggregations, ORDER BY for arranging the results in an order either ASC (Ascending) or DESC (Descending) and CASE statement for conditional expressions.**

* We used PARTITION BY condition to make groups as per the user id. In out task, we partitioned using the id\_user i.e. 1,2,3,4,5
* In those partitions we have arranged the rows using ORDER BY function on variable called topup\_date
* The CASE statement is similar to if/else statement which is defined in many programming languages.

**The Problem:** This problem is based on a topups table which is in the format called .tsv file. This table consists of attributes called:

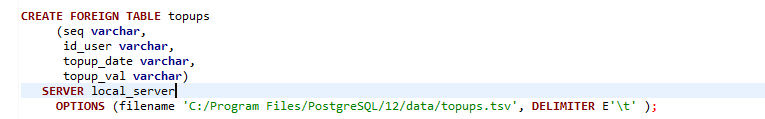
* seq
* id\_user
* topup\_date
* topup\_value

**Task 1: Load Dataset into PostgreSQL**

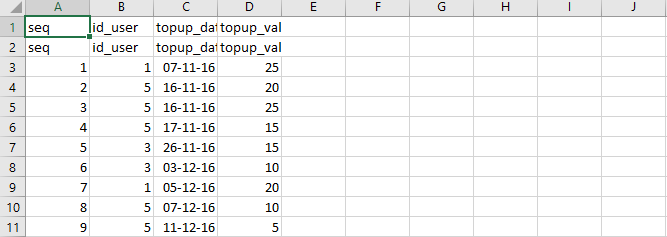
The file is loaded in the database using the foreign table feature of Postgre SQL. But to use it, we first need to fetch data from external source. In this case, this feature can only be used by first using Foreign Data Wrapper and creating an extension called file\_fdw.

Later we create the table and imported it from the path where it is stored.

Query:



Solution:

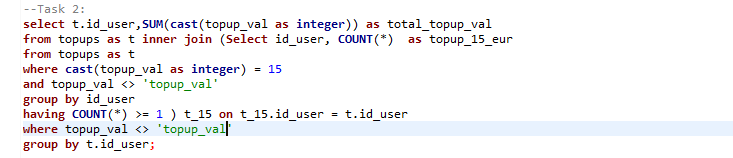


Just to keep the data as is we didn’t change the data type and neither we dropped the first row while importing the table

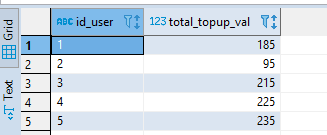
**Task 2: Print out the list of user IDs and the total of all top-ups done by them but ONLY for the users that had at least one top-up ever done by the amount of €15 exactly.**

* We first tried finding the users who have done the recharge of 15 euro and its count (how many time) for each user.
* Then we computed the total\_topup\_val for the above users by using the function called SUM() on variable called topup\_val

Query:



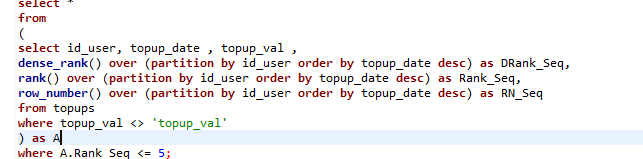
Solution:



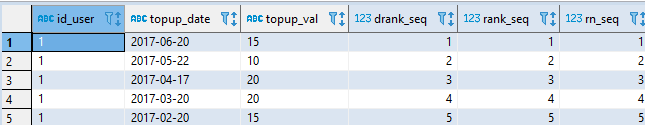
**Task 3: Show the 5 (but not more) rows containing most recent top-ups per user. In case of more top-ups done within a day, print those with higher amounts first.**

In this query we found out the 5 recent top-ups per user by using different window functions for calculating rank. Firstly, we found out the rank of each row in a partition where arranged in descending on the basis of topup\_date. Then using where clause on rank\_seq, we calculated the top 5 recent users.

Query:



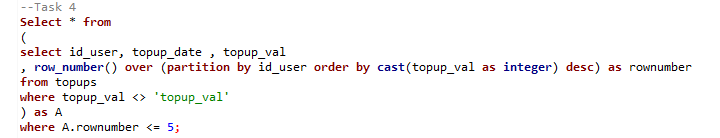
Solution:



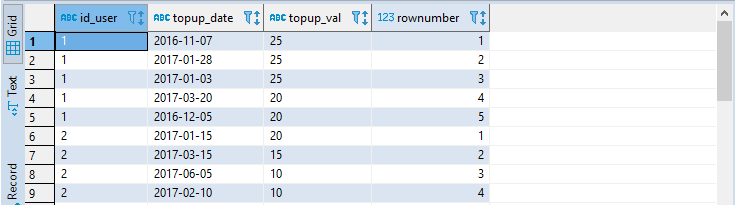
**Task 4: Show the 5 largest top ups done per user.**

This query computes the 5 largest top ups per user where each partition is arranged as per the top up value and containing rows at max 5.

Query:



Solution:



**Task 5: Using the window functions enrich the original set (create new, derived table) to include extra columns as per description below:**

* **prv\_topup\_dt - previous topup date of the same user**
* **days\_since - number of days since previous topup by user**
* **promo\_ind - Y/N flag. Put Y for top-ups of €20 or more, otherwise N.**
* **previous\_qual\_topup\_dt - the date of previous topup of €20 or more done by the same user**
* **to\_1st\_ratio - (bonus) Y/X fraction value where Y is the current topup value and X is the amount of the first ever topup done by the user.**

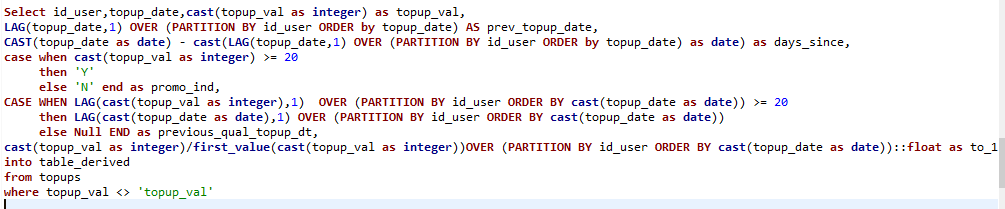
We created a new table called **table\_derived** to store the new variables. Each variable has a different goal. For **prv\_topup\_dt** we used the LAG() function to get the previous date. For the date difference we simply calculated the different between the present and previous top-up date and stored in variable **days\_since**.

On the basis of top up value, we assigned ‘Y’ or ‘N’ using CASE Statement. If value was greater than or equal to 20 it stores ‘Y’ else ‘N’in variable called **promo\_ind**.

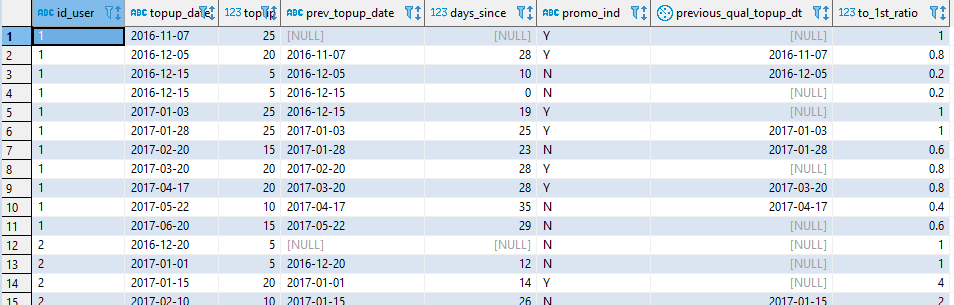
Now for variable **previous\_qual\_topup\_dt** we check the previous top-up value using lag() window function where offset is 1. If the value is greater than or equal to 20, the top-up date is set as previous date else it is set to Null.

For the bonus task, we used a window function called first\_value() . This functions takes the first value of each partition and is used to calculate the **to\_1st\_ratio.**

Query:



Solution:

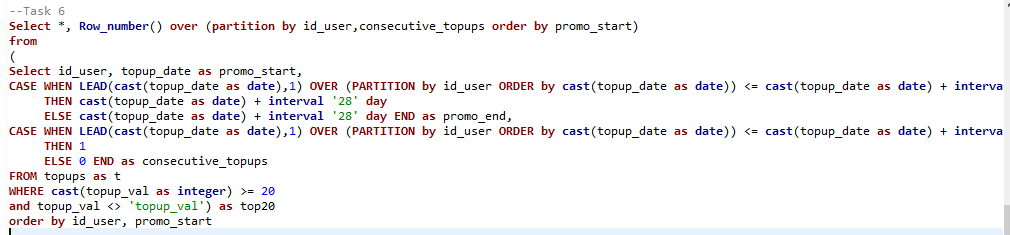


**Task 6: (Incomplete)**

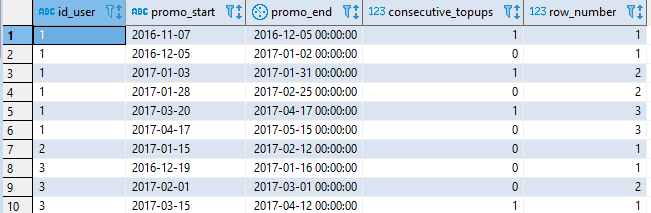
**The mobile operator runs a promotion. If you top up by at least €20 you get a free credit for 28 calendar days immediately following the day of that top up. Print out the list of consolidated periods when users were eligible to make free calls. Include initial eligibility date and the date when the free credit effectively ends. Consider that with regular top-ups the free credit period may be effectively prolonged multiple times. For instance for id\_user=4 the first promotion period started on 2016-12-20 and with the four qualifying topups being done on time it was effectively extended up until 2017-04-12:**

When the value is greater that or equal to 20 we add the free credit for 28 days and stored the end date in **promo\_end** variable.Using the lead window function we compared the promo end dates with next topup date and assigned the consecutive\_topups variable a value either 1 or 0.

Query:



Solution:



The Files are stored in the csv files for every task. Please refer it if required.

