FE 513: Final Exam

Rishikesh Yadav
CWID: 20007668

1. SQL

1. Import given bank data into PostgreSQL database.

-- Creating table “bank” and importing the data from the csv file in it.

QUERY:

```
DROP TABLE IF EXISTS bank ;
```

Data Output | Messages | Notifications

```
NOTICE:  table "bank" does not exist, skipping
DROP TABLE

Query returned successfully in 50 msec.
```

```
CREATE TABLE IF NOT EXISTS bank (
    id INTEGER NOT NULL,
    date DATE NOT NULL,
    asset INTEGER NOT NULL,
    liability INTEGER NOT NULL,
    idx INTEGER NOT NULL
) ;
```

Data Output | Messages | Notifications

```
CREATE TABLE

Query returned successfully in 61 msec.
```

```
COPY bank (id, date, asset, liability, idx) FROM 'C:\Users\psyad\Desktop\Stevens\Sem 4\FE 513\Final Exam\bank_data-1.csv' DELIMITER ',' CSV HEADER ;
```

Data Output | Messages | Notifications

```
COPY 37819

Query returned successfully in 139 msec.
```

```
SELECT * FROM bank ;
```

Data Output | Messages | Notifications

	id integer	date date	asset integer	liability integer	idx integer
1	23373	2002-09-30	95914	87304	1
2	23376	2002-12-31	95937	87453	2
3	23376	2002-03-31	83335	75939	3
4	23376	2002-06-30	84988	77125	4

Total rows: 1000 of 37819 | Query complete 00:00:00.221

2. Create a primary key for the import table.

   -- On importing the data into “bank” table, since the idx column has unique values, we make that the primary key of the table.

   QUERY:

```
ALTER TABLE bank ADD PRIMARY KEY (idx) ;
```

Data Output Messages Notifications

	id integer	date date	asset integer	liability integer	idx [PK] integer
1	23373	2002-09-30	95914	87304	1
2	23376	2002-12-31	95937	87453	2

3. Find the highest asset observation for each bank (i.e., There are 4 observations for each bank in a year. If there’re 100 unique banks in total, then the result contains 100 observations. Each observation may belong to different quarters.) Sort the resulting table according to asset value. Report the first 10 observations of output table.

   -- In order to find the highest asset observation, we first group the bank data by id and sort it by id ascending and asset descending, and then take the first row for each id. We will achieve this by having 2 sub queries in our query – first subquery being responsible for grouping by id and ordering the id ascendingly, while the second subquery will be responsible for grouping by id and sorting the asset in descending order. Then we will join these 2 sub queries on id and their first row will give us the highest asset observations for each bank.

   QUERY:

```
SELECT q1.id, q2.quarter, q2.date, q2.asset FROM
        (SELECT id, EXTRACT(quarter FROM date) AS quarter, date, asset,
         ROW_NUMBER() OVER (Partition by id ORDER BY id ASC) AS b
        from bank
        order by id asc) q1
INNER JOIN
        (SELECT id, extract(quarter from date) as quarter, date, asset,
         ROW_NUMBER() OVER (Partition by id ORDER BY asset DESC) AS b
        FROM bank
        ORDER BY id ASC) q2
ON q1.id = q2.id
AND q1.b = 1 AND q2.b = 1
ORDER BY q2.asset DESC
LIMIT 10;
```

Data Output Messages Notifications

	id integer	quarter numeric	date date	asset integer
1	628	4	2002-12-31	622000000
2	3510	3	2002-09-30	576000000
3	7213	4	2002-12-31	499000000
4	33869	4	2002-12-31	319000000
5	32633	1	2002-03-31	242000000
6	3618	4	2002-12-31	218000000
7	3511	4	2002-12-31	184000000
8	2558	4	2002-12-31	179000000
9	6548	4	2002-12-31	176000000
10	867	4	2002-12-31	115000000

Total rows: 10 of 10 Query complete 00:00:00.094

4. Show the query plan for question 1.3 using EXPLAIN tool.

-- EXPLAIN tool helps us understand the detailed step by step execution of any query

QUERY:

EXPLAIN ANALYSE

```
SELECT q1.id, q2.quarter, q2.date, q2.asset FROM
        (SELECT id, EXTRACT(quarter FROM date) AS quarter, date, asset,
         ROW_NUMBER() OVER (Partition by id ORDER BY id ASC) AS b
        from bank
        order by id asc) q1
INNER JOIN
        (SELECT id, extract(quarter from date) as quarter, date, asset,
         ROW_NUMBER() OVER (Partition by id ORDER BY asset DESC) AS b
        FROM bank
        ORDER BY id ASC) q2
ON q1.id = q2.id
AND q1.b = 1 AND q2.b = 1
ORDER BY q2.asset DESC
LIMIT 10 ;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Limit (cost=9543.73..9543.74 rows=4 width=44) (actual time=70.342..70.347 rows=10 loops=1)
2	-> Sort (cost=9543.73..9543.74 rows=4 width=44) (actual time=70.341..70.345 rows=10 loops=1)
3	Sort Key: q2.asset DESC
4	Sort Method: top-N heapsort Memory: 26kB

5	-> Merge Join (cost=6989.45..9543.69 rows=4 width=44) (actual time=13.909..68.981 rows=9614 loops=1)
6	Merge Cond: (q1.id = q2.id)
7	-> Subquery Scan on q1 (cost=3494.72..4723.84 rows=189 width=4) (actual time=4.721..38.577 rows=9614 loops=1)
8	Filter: (q1.b = 1)
9	-> WindowAgg (cost=3494.72..4251.10 rows=37819 width=52) (actual time=4.720..37.988 rows=9614 loops=1)
10	Run Condition: (row_number() OVER (?) <= 1)
11	-> Sort (cost=3494.72..3589.27 rows=37819 width=4) (actual time=4.707..5.958 rows=37819 loops=1)
12	Sort Key: bank.id
13	Sort Method: quicksort Memory: 1537kB
14	-> Seq Scan on bank (cost=0.00..619.19 rows=37819 width=4) (actual time=0.014..2.007 rows=37819 loops=1)
15	-> Materialize (cost=3494.72..4818.86 rows=189 width=44) (actual time=9.184..28.007 rows=9614 loops=1)
16	-> Subquery Scan on q2 (cost=3494.72..4818.39 rows=189 width=44) (actual time=9.183..26.110 rows=9614 loops=1)
17	Filter: (q2.b = 1)
18	-> WindowAgg (cost=3494.72..4345.65 rows=37819 width=52) (actual time=9.182..25.504 rows=9614 loops=1)
19	Run Condition: (row_number() OVER (?) <= 1)
20	-> Sort (cost=3494.72..3589.27 rows=37819 width=12) (actual time=9.163..10.585 rows=37819 loops=1)
21	Sort Key: bank_1.id, bank_1.asset DESC
22	Sort Method: quicksort Memory: 3298kB
23	-> Seq Scan on bank bank_1 (cost=0.00..619.19 rows=37819 width=12) (actual time=0.016..2.606 rows=37819 loops=1)
24	Planning Time: 0.172 ms
25	Execution Time: 70.765 ms
Total rows: 25 of 25	Query complete 00:00:00.119

5. Given the highest asset table from question 1.3, count how many observations are there for each quarter.

-- We simply achieve this by using the highest asset result table from question 1.3, grouping it by quarter and then taking a count of each quarter

QUERY:

```
SELECT obsv.quarter, count(*) FROM
    (SELECT q1.id, q2.quarter, q2.date, q2.asset FROM
        (SELECT id, EXTRACT(quarter FROM date) AS quarter, date, asset,
         ROW_NUMBER() OVER (Partition by id ORDER BY id ASC) AS b
        from bank
        order by id asc) q1
    INNER JOIN
        (SELECT id, extract(quarter from date) as quarter, date, asset,
         ROW_NUMBER() OVER (Partition by id ORDER BY asset DESC) AS b
        FROM bank
        ORDER BY id ASC) q2
    ON q1.id = q2.id
    AND q1.b = 1 AND q2.b = 1
    ORDER BY q2.asset DESC) obsv
GROUP BY obsv.quarter ;
```

Data Output Messages Notifications

	quarter numeric	count bigint
1	1	1192
2	2	757
3	3	1734
4	4	5931

Total rows: 4 of 4 Query complete 00:00:00.118

6. For the whole sample data, how many observations have asset value higher than 100,000 and liability value smaller than 100,000.

   -- This can be achieved with the help of a simple count query by applying the given conditions, i.e., asset value higher than 100,000 and liability value smaller than 100,000.
   QUERY:

```
SELECT COUNT(*) FROM bank WHERE asset > 100000 AND liability < 100000 ;
```

Data Output Messages Notifications

	count bigint
1	1411

Total rows: 1 of 1 Query complete 00:00:00.064

7. Each observation was given an 'idx' number. Find the average liability of observation with odd 'idx' number.

   -- This can again be achieved simply by taking the average of every observation that has an odd idx value.
   QUERY:

```
SELECT AVG( liability ) AS Odd_Average_Liability FROM bank WHERE idx % 2 = 1 ;
```

Data Output Messages Notifications

	odd_average_liability numeric
1	778839.890163934426

Total rows: 1 of 1 Query complete 00:00:00.064 Rows selected: 1

8. Find the average liability of observation with even 'idx' number. What's the difference between these two average numbers.

   -- Like 1.7, we will achieve this by taking the average of every observation that has an even idx value. Then, we will subtract the smaller average from the greater average to find their difference.

   QUERY:

```
SELECT AVG( liability ) AS Even_Average_Liability FROM bank WHERE idx % 2 = 0 ;
```

Data Output | Messages | Notifications

	even_average_liability numeric
1	787029.208260616638

Total rows: 1 of 1 | Query complete 00:00:00.100

```
SELECT even.Even_Average_Liability - odd.Odd_Average_Liability AS Liability_Difference
FROM
        (SELECT AVG( liability) AS Even_Average_Liability FROM bank WHERE idx % 2 = 0) even,
        (SELECT AVG( liability) AS Odd_Average_Liability FROM bank WHERE idx % 2 = 1) odd ;
```

Data Output | Messages | Notifications

	liability_difference numeric
1	8189.318096682212

Total rows: 1 of 1 | Query complete 00:00:00.107

9. For each bank find all records with increased asset. The record with increase asset means one record's asset value is larger than the one of previous quarter. (For instance, a bank (id: 123) has asset 30,000 in 3/31/02, asset 20,000 in 6/30/02 and asset 25,000 in 9/30/02. Then the record with bank id (123), asset value (25,000) and date (9/30/02) is recorded. Because its asset value is larger than asset value in 6/30/02.) Report the first 10 observations of output table.

   -- To achieve this, we will group the data by id and then order it ascendingly by quarter/ date. Then we will use lag function to find the asset value for previous quarter and compare it to the asset value for current quarter. From this we will only take those records that show an increase in the asset value from previous quarter for every bank.

   QUERY:

```
SELECT q1.id, q1.quarter, q1.date, q1.asset
FROM
        (SELECT id, EXTRACT(quarter FROM date) AS quarter, date, asset,
        LAG(asset) OVER w AS prev_asset
```

```
        FROM bank
        WINDOW w AS (PARTITION BY id ORDER BY EXTRACT(quarter FROM date) ASC)) q1
WHERE q1.asset > q1.prev_asset
LIMIT 10 ;
```

Data Output Messages Notifications

	id integer	quarter numeric	date date	asset integer
1	9	2	2002-06-30	361953
2	9	3	2002-09-30	383246
3	14	2	2002-06-30	73600000
4	14	4	2002-12-31	79600000
5	28	3	2002-09-30	12474
6	35	2	2002-06-30	492046
7	35	3	2002-09-30	503401
8	39	2	2002-06-30	203754
9	39	3	2002-09-30	205211
10	39	4	2002-12-31	206140

Total rows: 10 of 10 Query complete 00:00:00.297

2. User-defined Function in R

1. Download daily stock data using a given stock ticker for a given time period (set starting time, ending time and stock ticker as input variable).
2. Get the adjusted close price and consider this price data only for following tasks.
3. Perform a rolling window estimation on stock price vector to calculate the mean and standard deviation. For each time, you keep using a fixed size of data to calculate the mean and the standard deviation. The window size needs to be set as a input variable.
4. Store the statistical result of Q 2.3 into a dataframe. Plot this statistical dataframe using scatter plot. In the plot, x axis represents the index for each rolling window and y axis represents the statistical values. (Make sure you use different colors to differentiate between mean value and std value. Don't forget to include an legend.)
5. Return the statistical dataframe.

SCRIPT:

```
library(quantmod)
library(roll)
library(ggplot2)
library(data.table)

stock_data <- function(stock.ticker, start.date, end.date, rolling.size) {
```

```
# 2.1
# Here we download the daily stock data and convert it to a dataframe for further
# processing
stock.data <- as.data.frame(getSymbols
                    (stock.ticker, src = 'yahoo',
                              from = start.date,
                              to = end.date,
                              warnings = FALSE,
                              auto.assign = TRUE,env = NULL))

# 2.2
# Here we get the adjusted close price
adjusted.Close.Price <- stock.data[tail(names(stock.data), 1)]

# 2.3
# Here we calculate the the mean and standard deviation by performing a rolling
# window estimation on stock price vector.
mean <- rollapply
      (adjusted.Close.Price, rolling.size,
                        by = 1, FUN = mean, by.column = FALSE)
std.Dev <- rollapply
         (adjusted.Close.Price, rolling.size,
                           FUN = sd, fill=0,
                           align="r", by.column = FALSE)
std.Dev <- tail(std.Dev, -(rolling.size - 1))

# 2.4
# Here we store the statistical result of Q 2.3 into a dataframe
statistical <- do.call(rbind, Map(data.frame, A = mean, B = std.Dev))
colnames(statistical) <- c('Mean','Standard Deviation')

# Here we transform the dataframe for our plot
statistical.Transpose <- transpose(statistical)
colnames(statistical.Transpose) <- rownames(statistical)
rownames(statistical.Transpose) <- colnames(statistical)

# Here we pot statistical dataframe using scatter plot
statistical.plot <- ggplot() + geom_point(data = stack(statistical.Transpose[1,]),
                              aes(x = ind, y = values, color = "Mean")) +
              geom_point(data = stack(statistical.Transpose[2,]),
                     aes(x = ind, y = values, color = "Standard Deviation"))
              + labs(x = "Index", y = "Statistical Values",
                   title = "Statistical Result") +
              theme(axis.text.x = element_text(angle = 90,
                                  vjust = 0.5, hjust=1)) +
              scale_color_manual(name = "Statistical Values",
                        values = c("Mean" = "yellow",
                                 "Standard Deviation" = "green"))
```

```
  print(statistical.plot)

  # 2.5
  # Here we return the statistical dataframe result
  return(statistical)
}
```

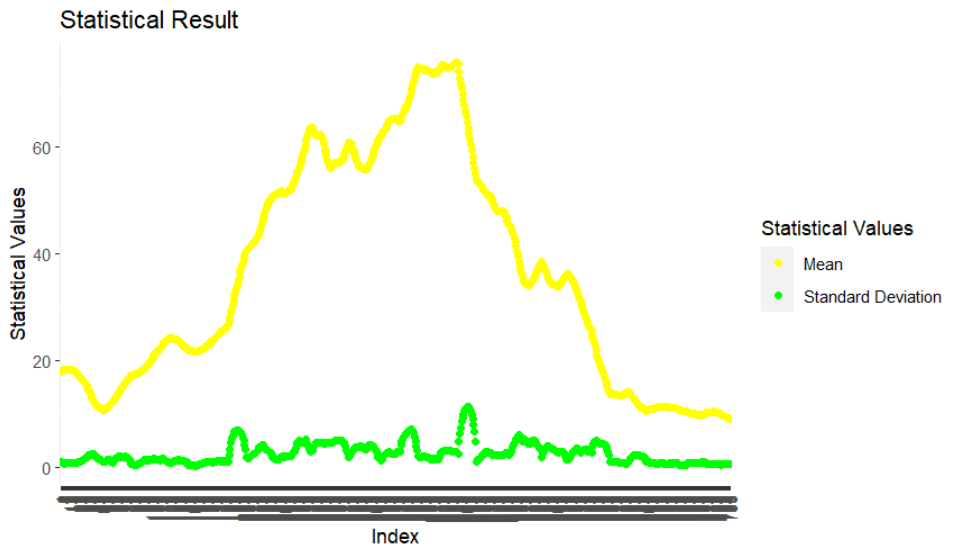
6. Test your function with suitable parameters.(1 or 2-year data and a rolling window size of 20). If you got a huge dataframe for the output, do NOT print the whole sample. Showing a part of it is enough

SCRIPT:

```
# 2.6
# Testing the above created function
stock.ticker <- 'SNAP'
start.date <- as.Date('2019-12-31')
end.date <- as.Date('2022-12-31')
rolling.size <- 20

statistical.stock.dataframe <- stock_data(stock.ticker, start.date, end.date, rolling.size)
print(statistical.stock.dataframe)
```

RESULTS:




```
> stock.ticker <- 'SNAP'
> start.date <- as.Date('2019-12-31')
> end.date <- as.Date('2022-12-31')
> rolling.size <- 20
>
> statistical.stock.dataframe <- stock_data(stock.ticker, start.date, end.date, rolling.size)
> print(statistical.stock.dataframe)
       Mean Standard Deviation
1   17.9900          1.0173131
2   18.1175          0.9562802
3   18.1975          0.9039960
4   18.2715          0.8373908
5   18.3815          0.7731905
6   18.3395          0.8675278
```

```
492 46.0820          2.5492298
493 45.6265          2.4707661
494 45.3850          2.4481041
495 45.1790          2.4913022
496 44.7775          2.8965690
497 44.4445          3.2653059
498 43.9650          3.7956708
499 43.4965          4.2834786
500 42.9410          4.6341200
 [ reached 'max' / getOption("max.print") -- omitted 238 rows ]
```

Rishikesh_Yadav_Final_Submission.r | statistical.stock.dataframe

Filter

	Mean	Standard Deviation
1	17.9900	1.0173131
2	18.1175	0.9562802
3	18.1975	0.9039960
4	18.2715	0.8373908
5	18.3815	0.7731905
6	18.3395	0.8675278
7	18.3560	0.8366937
8	18.3445	0.8525346
9	18.3540	0.8425736
10	18.3345	0.8556096
11	18.3050	0.8779910
12	18.2750	0.8922619
13	18.2405	0.9065055

Showing 1 to 14 of 738 entries, 2 total columns

3. PostgreSQL API in R

1. Make a connection to your local PostgreSQL database.

SCRIPT:

 # 3.1 - Here we make a connection to your local PostgreSQL database

```
library(RPostgreSQL)
db_name <- "FE_513"
username <- "postgres"
driver <- dbDriver("PostgreSQL")
conn <-dbConnect(driver, dbname = db_name, user = username, password = "root")
```

RESULT:

Rishikesh_Yadav_Final_API.r | conn

Show Attributes

Name	Type	Value
conn	S4 (RPostgreSQL::PostgreSQLConnection)	S4 object of class PostgreSQLConnection
Id	integer [2]	4500 0

Rishikesh_Yadav_Final_API.r × driver ×

Show Attributes

Name	Type	Value
driver	S4 (RPostgreSQL::PostgreSQLDriver)	S4 object of class PostgreSQLDriver
Id	integer [1]	4500

2. Query the PostgreSQL database via API to get the original bank data. (The bank data that you import to PostgreSQL database in Q 1.1) Store the data into a dataframe.

SCRIPT:

```
# 3.2 - Here we query the PostgreSQL database via API to get the original bank data
result <- dbGetQuery(conn, "SELECT * FROM bank;")
```

RESULT:

Rishikesh_Yadav_Final_API.r × result ×

Filter

	id	date	asset	liability	idx
1	23373	2002-09-30	95914	87304	1
2	23376	2002-12-31	95937	87453	2
3	23376	2002-03-31	83335	75939	3
4	23376	2002-06-30	84988	77125	4
5	23376	2002-09-30	90501	82248	5
6	234	2002-12-31	56866	49406	6
7	234	2002-03-31	55204	47914	7
8	234	2002-06-30	55180	47695	8
9	234	2002-09-30	56940	49249	9
10	23404	2002-12-31	78625	72580	10
11	23404	2002-03-31	72425	66709	11
12	23404	2002-06-30	73619	67798	12
13	23404	2002-09-30	73962	68002	13

Showing 1 to 14 of 37,819 entries, 5 total columns

3. Calculate asset growth rate for each quarter and each bank. (asset growth rate = (current quarter value - previous quarter value) / previous quarter value). The result start from second quarter, since we don't have all necessary data for first quarter calculation. Store the calculation result in a data frame.

SCRIPT:

```
# 3.3 - Here we calculate asset growth rate for each quarter and each bank with the
# given formula and store the result in a data frame.
library(dplyr)
result <- result %>% group_by(id) %>% arrange(id, date) %>% mutate(asset.growth.rate =
(asset - lag(asset, n = 1))/ lag(asset, n = 1))
```

RESULT:

Rishikesh_Yadav_Final_API.r × result ×

Filter

	id	date	asset	liability	idx	asset.growth.rate
1	9	2002-03-31	348727	321479	20912	NA
2	9	2002-06-30	361953	332900	20913	0.0379265156
3	9	2002-09-30	383246	352456	20914	0.0588280799
4	9	2002-12-31	371812	340365	20911	-0.0298346232
5	14	2002-03-31	68600000	64300000	27334	NA
6	14	2002-06-30	73600000	69200000	27335	0.0728862974
7	14	2002-09-30	72800000	68200000	27336	-0.0108695652
8	14	2002-12-31	79600000	74500000	27333	0.0934065934
9	28	2002-03-31	14340	7948	3937	NA
10	28	2002-06-30	12049	5354	3938	-0.1597629010
11	28	2002-09-30	12474	5543	3939	0.0352726367
12	35	2002-03-31	471056	438541	12623	NA
13	35	2002-06-30	492046	457116	12624	0.0445594579

Showing 1 to 14 of 37,819 entries, 6 total columns

4. Export the dataframe of Q 3.3 to the PostgreSQL database via API

SCRIPT:

#3.4 - Here we export the data frame of Q 3.3 to the PostgreSQL database via API

dbWriteTable(conn, " bank_data_from_api", result, row.names=FALSE, append=TRUE)

RESULT:

```
> dbWriteTable(conn, "bank_data_from_api", result, row.names=FALSE, append=TRUE)
[1] TRUE
>
```

Tables (7)

- bank
- bank_data_from_api

Query Query History

```
1 SELECT * FROM bank_data_from_api;
```

Data Output Messages Notifications

	id integer	date date	asset integer	liability integer	idx integer	asset.growth.rate double precision
1	9	2002-03-31	348727	321479	20912	[null]
2	9	2002-06-30	361953	332900	20913	0.0379265155838234
3	9	2002-09-30	383246	352456	20914	0.0588280798888253
4	9	2002-12-31	371812	340365	20911	-0.0298346231924143
5	14	2002-03-31	68600000	64300000	27334	[null]
6	14	2002-06-30	73600000	69200000	27335	0.0728862973760933
7	14	2002-09-30	72800000	68200000	27336	-0.0108695652173913
8	14	2002-12-31	79600000	74500000	27333	0.0934065934065934
9	28	2002-03-31	14340	7948	3937	[null]
10	28	2002-06-30	12049	5354	3938	-0.15976290097629
11	28	2002-09-30	12474	5543	3939	0.0352726367333389
12	35	2002-03-31	471056	438541	12623	[null]
13	35	2002-06-30	492046	457116	12624	0.0445594578988485
14	35	2002-09-30	503401	467080	12625	0.0230771106766441
15	35	2002-12-31	502674	465464	12622	-0.00144417671001845
16	39	2002-03-31	201681	181030	14696	[null]
17	39	2002-06-30	203754	182287	14697	0.0102786082972615

Total rows: 1000 of 37819 Query complete 00:00:00.132