Advanced SQL minicourse from Kaggle (https://www.kaggle.com/learn/advanced-sql)

I. Inner join and Union

a. Inner join

ID	Symbol
A	1
B	2

ID	Name
A	Hello
B	I
C	am
D	Vinh

------->

ID	Symbol	Name
A	1	Hello
B	2	I

b. Union

ID	Symbol
A	1
B	2
C	3

ID	Symbol
D	4
E	3

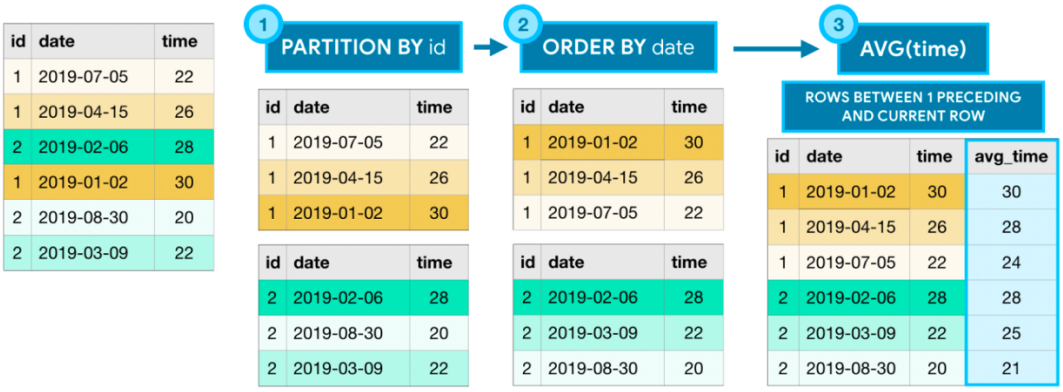
-------->

ID	Symbol
A	1
B	2
C	3
D	4
E	3

II. Analytic function

Syntax:

```
query = """
        SELECT *,
            AVG(time) OVER(
                                PARTITION BY id
                                ORDER BY date
                                ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
                                ) as avg_time
        FROM `bigquery-public-data.runners.train_time`
        """
```



id	date	time
1	2019-07-05	22
1	2019-04-15	26
2	2019-02-06	28
1	2019-01-02	30
2	2019-08-30	20
2	2019-03-09	22

id	date	time
1	2019-07-05	22
1	2019-04-15	26
1	2019-01-02	30

id	date	time
2	2019-02-06	28
2	2019-08-30	20
2	2019-03-09	22

id	date	time
1	2019-01-02	30
1	2019-04-15	26
1	2019-07-05	22

id	date	time
2	2019-02-06	28
2	2019-03-09	22
2	2019-08-30	20

id	date	time	avg_time
1	2019-01-02	30	30
1	2019-04-15	26	28
1	2019-07-05	22	24
2	2019-02-06	28	28
2	2019-03-09	22	25
2	2019-08-30	20	21

Window frame clauses

There are many ways to write window frame clauses:

- *ROWS BETWEEN 1 PRECEDING AND CURRENT ROW - the previous row and the current row.*
- *ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING - the 3 previous rows, the current row, and the following row.*
- *ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING - all rows in the partition.*

Of course, this is not an exhaustive list, and you can imagine that there are many more options!

Three types of analytic functions

The example above uses only one of many analytic functions. BigQuery supports a wide variety of analytic functions, and we'll explore a few here. For a complete listing, you can take a look at the documentation.

1) Analytic aggregate functions

*As you might recall, **AVG()** (from the example above) is an aggregate function. The **OVER** clause is what ensures that it's treated as an analytic (aggregate) function. **Aggregate functions** take all of the values within the window as input and return a single value.*

- ***MIN()** (or **MAX()**) - Returns the minimum (or maximum) of input values*
- ***AVG()** (or **SUM()**) - Returns the average (or sum) of input values*
- ***COUNT()** - Returns the number of rows in the input*

2) Analytic navigation functions

***Navigation functions** assign a value based on the value in a (usually) different row than the current row.*

- ***FIRST_VALUE()** (or **LAST_VALUE()**) - Returns the first (or last) value in the input*

- ***LEAD()*** *(and* ***LAG()****) - Returns the value on a subsequent (or preceding) row*

3) Analytic numbering functions

Numbering functions *assign integer values to each row based on the ordering.*

- ***ROW_NUMBER()*** *- Returns the order in which rows appear in the input (starting with 1)*
- ***RANK()*** *- All rows with the same value in the ordering column receive the same rank value, where the next row receives a rank value which increments by the number of rows with the previous rank value.*

III. Nested and repeated data

a. Nested:

- *Nested columns have type* ***STRUCT*** *(or type* ***RECORD****). The column Toy is nested data.*

`pets` table

ID	Name	Age	Animal
1	Moon	9	Dog
2	Ripley	7	Cat
3	Napoleon	1	Fish

`toys` table

ID	Name	Type	Pet_ID
1	McFly	Frisbee	1
2	Fluffy	Feather	2
3	Eddy	Castle	3

vs.

`pets_and_toys` table

ID	Name	Age	Animal	Toy
1	Moon	9	Dog	{**Name**: McFly, **Type**: Frisbee}
2	Ripley	7	Cat	{**Name**: Fluffy, **Type**: Feather}
3	Napoleon	1	Fish	{**Name**: Eddy, **Type**: Castle}

The schema:

`pets_and_toys` table schema

```
SchemaField('ID', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
SchemaField('Age', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Animal', 'STRING', 'NULLABLE', None, ()),
SchemaField('Toy', 'RECORD', 'NULLABLE', None, (
   SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
   SchemaField('Type', 'STRING', 'NULLABLE', None, ())
))
```

Query: access with "." operator.

```
query = """
        SELECT Name AS Pet_Name,
               Toy.Name AS Toy_Name,
               Toy.Type AS Toy_Type
        FROM `bigquery-public-data.pet_records.pets_and_toys`
        """
```

Pet_Name	Toy_Name	Toy_Type
Moon	McFly	Frisbee
Ripley	Fluffy	Feather
Napoleon	Eddy	Castle

b. *Repeated:*

Toys have the same ID is group.

`pets` table

ID	Name	Age	Animal
1	Moon	9	Dog
2	Ripley	7	Cat
3	Napoleon	1	Fish

`toys_type` table

ID	Type	Pet_ID
1	Frisbee	1
2	Bone	1
3	Rope	1
4	Feather	2
5	Ball	2
6	Castle	3

vs.

`pets_and_toys_type` table

ID	Name	Age	Animal	Toys
1	Moon	9	Dog	[Frisbee, Bone, Rope]
2	Ripley	7	Cat	[Feather, Ball]
3	Napoleon	1	Fish	[Castle]

The schema:

`pets_and_toys_type` table schema

```
SchemaField('ID', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
SchemaField('Age', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Animal', 'STRING', 'NULLABLE', None, ()),
SchemaField('Toys', 'STRING', 'REPEATED', None, ())
```

Use UNNEST function to access:

```
query = """
        SELECT Name AS Pet_Name,
               Toy_Type
        FROM `bigquery-public-data.pet_records.pets_and_toys_type`,
            UNNEST(Toys) AS Toy_Type
        """
```

Pet_Name	Toy_Type
Moon	Frisbee
Moon	Bone
Moon	Rope
Ripley	Feather
Ripley	Ball
Napoleon	Castle

c. *Nested and repeated data*

The column Toys of more_pets_and_toys have nested and repeated data:

`pets` table

ID	Name	Age	Animal
1	Moon	9	Dog
2	Ripley	7	Cat
3	Napoleon	1	Fish

`more_toys` table

ID	Name	Type	Pet_ID
1	McFly	Frisbee	1
2	Scully	Bone	1
3	Pusheen	Rope	1
4	Fluffy	Feather	2
5	Robert	Ball	2
6	Eddy	Castle	3

vs.

`more_pets_and_toys` table

ID	Name	Age	Animal	Toys
1	Moon	9	Dog	[{**Name**: McFly, **Type**: Frisbee}, {**Name**: Scully, **Type**: Bone}, {**Name**: Pusheen, **Type**: Rope}]
2	Ripley	7	Cat	[{**Name**: Fluffy, **Type**: Feather}, {**Name**: Robert, **Type**: Ball}]
3	Napoleon	1	Fish	[{**Name**: Eddy, **Type**: Castle}]

The schema:

`more_pets_and_toys` table schema

```
SchemaField('ID', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
SchemaField('Age', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Animal', 'STRING', 'NULLABLE', None, ()),
SchemaField('Toys', 'RECORD', 'REPEATED', None, (
   SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
   SchemaField('Type', 'STRING', 'NULLABLE', None, ())
))
```

Query:

```
query = """
        SELECT Name AS Pet_Name,
               t.Name AS Toy_Name,
               t.Type AS Toy_Type
        FROM `bigquery-public-data.pet_records.more_pets_and_toys`,
            UNNEST(Toys) AS t
        """
```

Pet_Name	Toy_Name	Toy_Type
Moon	McFly	Frisbee
Moon	Scully	Bone
Moon	Pusheen	Rope
Ripley	Fluffy	Feather
Ripley	Robert	Ball
Napoleon	Eddy	Castle

IV. Writing efficient queries

Some useful functions

We will use two functions to compare the efficiency of different queries:

- *show_amount_of_data_scanned() shows the amount of data the query uses.*
- *show_time_to_run() prints how long it takes for the query to execute.*

```
star_query = "SELECT * FROM `bigquery-public-data.github_repos.contents`"
show_amount_of_data_scanned(star_query)

basic_query = "SELECT size, binary FROM `bigquery-public-data.github_repos.contents
`"
show_amount_of_data_scanned(basic_query)
```

```
Data processed: 2504.697 GB
Data processed: 2.396 GB
```

Strategies:

1. *Only select the column you want.*
2. *Read less data. Query from small table to big table.*
3. *Avoid N:N join*