Databases

Seminar 6

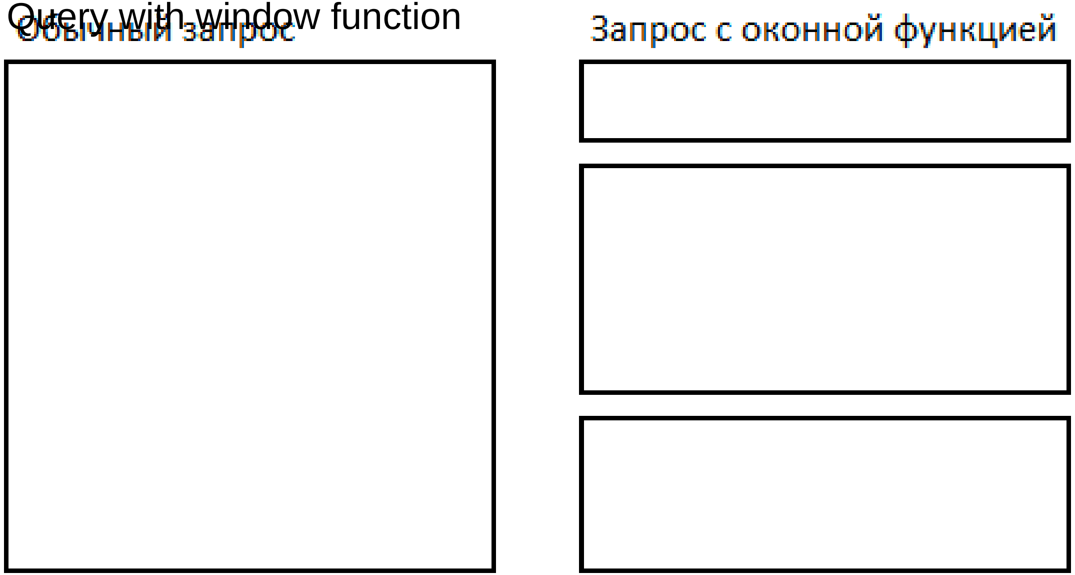
Analytical (window) functions

- Take an intermediate calculation result column as an argument and return a column.
- Can only be used in the ORDER BY and SELECT clauses, performing the final processing of the logical intermediate result.
- Act similar to aggregate functions, but do not reduce the level of detail.
- Aggregate data in portions, the quantity and size of which are regulated by a special syntax construct.

```
function_name(expression) OVER (
    [ <PARTITION BY clause> ]       -- window
    [ <ORDER BY clause> ]           -- sorting
    [ <ROWS or RANGE clause> ]      -- range of the window
) AS attr_name
```

In a normal query, the entire set of rows is processed as a single "whole piece", for which aggregates are considered.
And when using window functions, the query is divided into parts (windows) and its aggregates are already considered for each of the individual parts.

Normal query
Query with window function



An example of how to compare the salary of each employee with the average salary of their department:

- The first three columns are extracted directly from the emp salary table, and each row in the emp salary table corresponds to one row in the result table.
- The fourth column contains the average value calculated for all rows having the same depname value as the current row. (In fact, the average is calculated by the same ordinary, non-windowed avg function, but the OVER clause turns it into a windowed one, so that its action is limited to the window frames.)

```
SELECT
    depname,
    empno,
    salary,
    avg(salary) OVER
(PARTITION BY depname)
FROM
    empsalary;
```

```
depname   | empno | salary |         avg

-----------+-------+--------+----------------------
 develop   |    11 |   5200 | 5020.0000
 develop   |     7 |   4200 | 5020.0000
 develop   |     9 |   4500 | 5020.0000
 develop   |     8 |   6000 | 5020.0000
 develop   |    10 |   5200 | 5020.0000
 personnel |     5 |   3500 | 3700.0000
 personnel |     2 |   3900 | 3700.0000
 sales     |     3 |   4800 | 4866.6667
 sales     |     1 |   5000 | 4866.6667
 sales     |     4 |   4800 | 4866.6667
(10 rows)
```

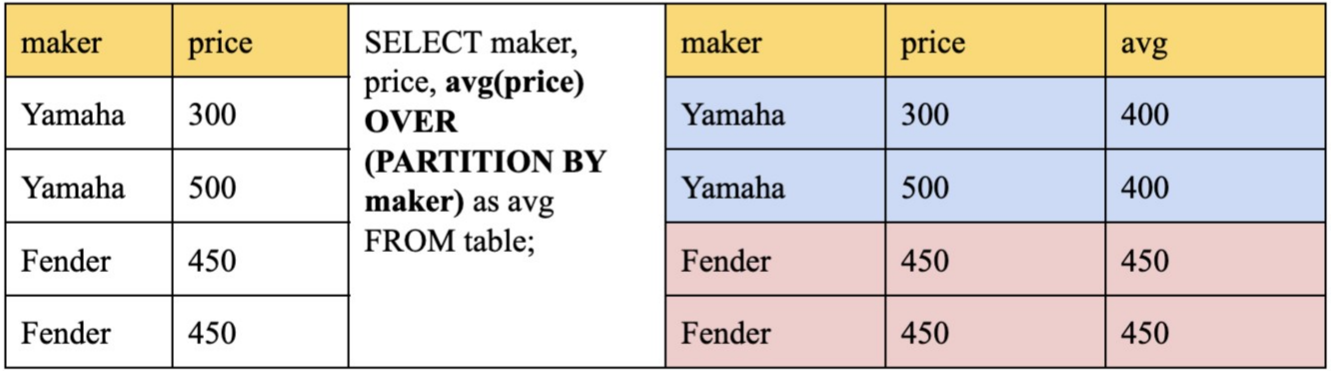
The use of OVER

- OVER defines the set of rows that the window function will use, including data sorting. (“window")
- In the expression that defines the window function, the OVER statement restricts sets of strings with the same values in the field that is being split.
- The OVER() statement itself is unlimited and contains all the rows from the result set.
- The OVER statement can be used multiple times in a single SELECT, each with its own division and sorting.

Rules of partitioning

Inside OVER, you must specify the table field on which the “window” will slide and the rule by which the rows will be partitioned:

1. **PARTITION BY**: responsible for the partitioning criterion
    - Logically divides the set into groups according to criteria.
    - Analytical functions are applied to groups independently.
    - If you do not specify the partitioning structure, the entire set is considered to be one group.



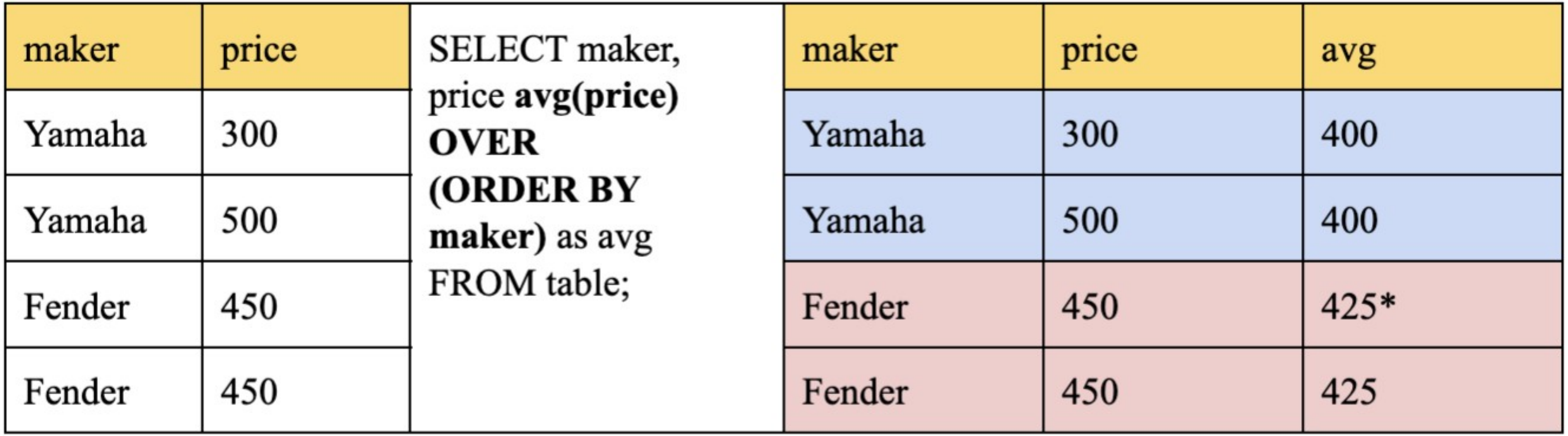
maker	price
Yamaha	300
Yamaha	500
Fender	450
Fender	450

```
SELECT maker,
price, avg(price)
OVER
(PARTITION BY
maker) as avg
FROM table;
```

maker	price	avg
Yamaha	300	400
Yamaha	500	400
Fender	450	450
Fender	450	450

2. **ORDER BY**: responsible for sorting

- Sets the sorting criteria within each group.
- Aggregate functions in the absence of the ORDER BY construction are calculated for all rows of the group, and the same value is given for each row, i.e. the function is used as a summary.
- If an aggregate function is used with the ORDER BY construct, then it is calculated from the current row and all rows before it, i.e. the function is used as a windowed one (the cumulative total is calculated).



maker	price
Yamaha	300
Yamaha	500
Fender	450
Fender	450

```
SELECT maker,
price avg(price)
OVER
(ORDER BY
maker) as avg
FROM table;
```

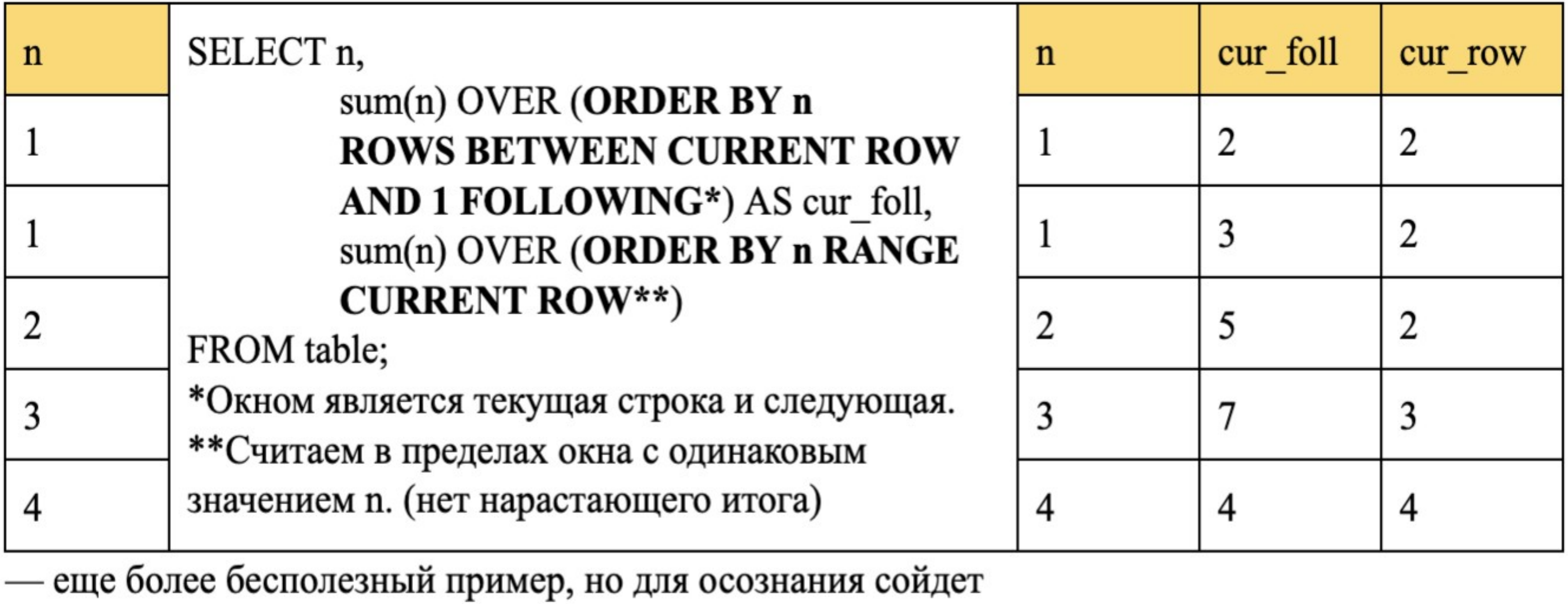
maker	price	avg
Yamaha	300	400
Yamaha	500	400
Fender	450	425*
Fender	450	425

[illegible] *425 = (300+500+450+450)/4;

3. **ROWS | RANGE**: additional restrictions on the range of window rows (the presence of ORDER BY is required):

- ROWS (by rows) — allows you to manually define the boundaries of the window for which the value is calculated; can work with PRECEDING/FOLLOWING.
- RANGE (based on the values from ORDER BY, a sub-window is formed)— close enough to the previous one, but still not the same (Alexander Faritovich's professional opinion: “I have no idea when this can be used”); But using ‘RANGE CURRENT ROW’ after ORDER BY allows you to get rid of the cumulative total; does not know how to work with PRECEDING/FOLLOWING.
- By default, it considers from UNBOUNDED PRECEDING to CURRENT ROW. (UNBOUNDED PRECEDING /FOLLOWING — we consider up to the end / beginning of the window.)
- We select the lines within the window, but if necessary, we can manually register the previous / subsequent lines so that it can go beyond the window.

Example for ROWS | RANGE



n
1
1
2
3
4

```
SELECT n,
        sum(n) OVER (ORDER BY n
        ROWS BETWEEN CURRENT ROW
        AND 1 FOLLOWING*) AS cur_foll,
        sum(n) OVER (ORDER BY n RANGE
        CURRENT ROW**)
FROM table;
```

*Окном является текущая строка и следующая.

**Считаем в пределах окна с одинаковым значением n. (нет нарастающего итога)

n	cur_foll	cur_row
1	2	2
1	3	2
2	5	2
3	7	3
4	4	4

— еще более бесполезный пример, но для осознания сойдет

Classification of windows functions

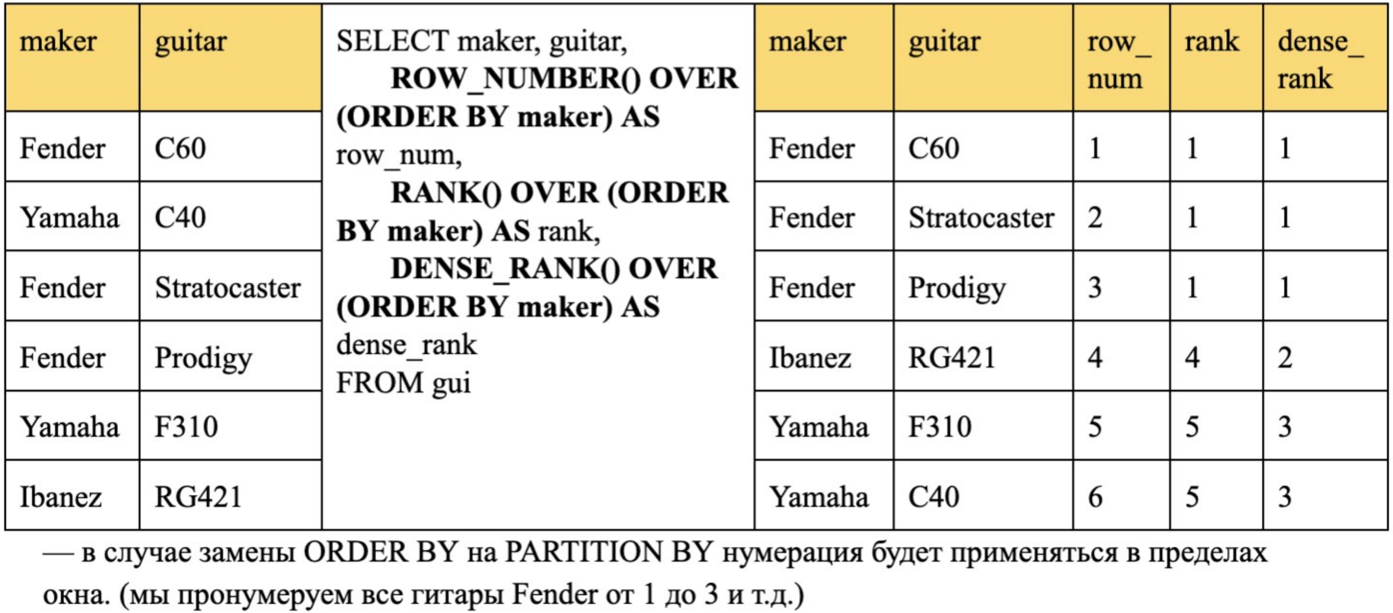
1. Aggregating (sum, avg, min, max, count)
2. Ranking (row_number, rank, dense_rank)
3. Value (lag, lead, first_value, last_value); value functions are used with field indication.

Ranking functions:

4. row_number() – the rows of the window are sequentially indexed in increments of 1.
5. rank() – rank each row of the window with a gap in the indexing when the values are equal.
6. dense_rank() – the rows of the window are indexed without gaps when the values are equal.

Ranking functions

1. row_number() – the rows of the window are sequentially indexed in increments of 1.
2. rank() – rank each row of the window with a gap in the indexing when the values are equal.
3. dense_rank() – the rows of the window are indexed without gaps when the values are equal.



maker	guitar
Fender	C60
Yamaha	C40
Fender	Stratocaster
Fender	Prodigy
Yamaha	F310
Ibanez	RG421

```
SELECT maker, guitar,
      ROW_NUMBER() OVER
(ORDER BY maker) AS
row_num,
      RANK() OVER (ORDER
BY maker) AS rank,
      DENSE_RANK() OVER
(ORDER BY maker) AS
dense_rank
FROM gui
```

maker	guitar	row_ num	rank	dense_ rank
Fender	C60	1	1	1
Fender	Stratocaster	2	1	1
Fender	Prodigy	3	1	1
Ibanez	RG421	4	4	2
Yamaha	F310	5	5	3
Yamaha	C40	6	5	3

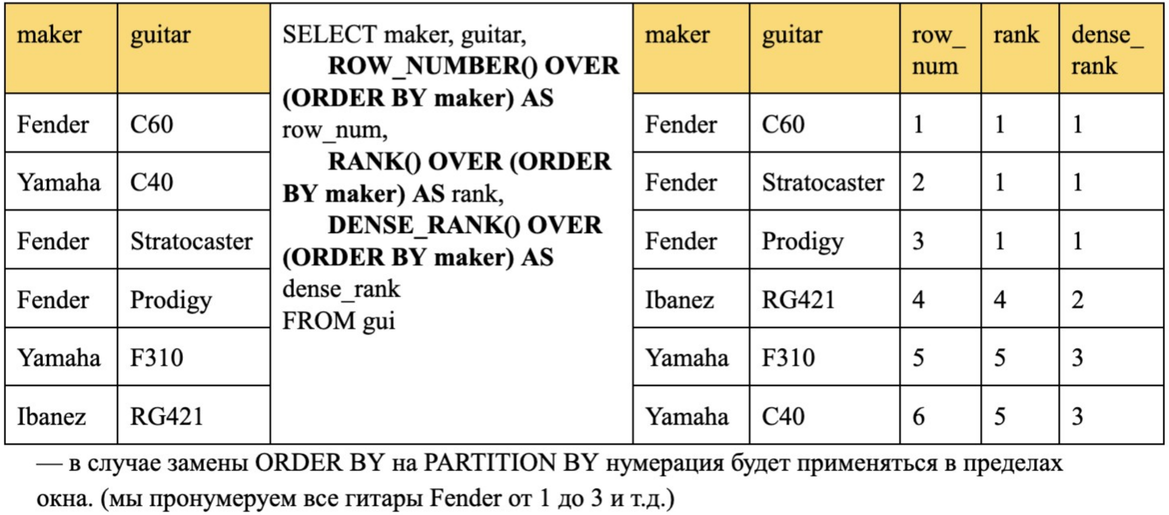
— в случае замены ORDER BY на PARTITION BY нумерация будет применяться в пределах окна. (мы пронумеруем все гитары Fender от 1 до 3 и т.д.)

Value functions

1. lag(attr, offset (offset), default_value(default value in case our line turns out to be the first)) – the previous value with a shift.
2. lead(attr, offset, default_value) – the next value with a shift.
3. first_value(attr) – the first value in the window from the first to the current line.
4. last_value(attr) – the last value in the window from the first to the current line.

Value functions

```
SELECT
    BusinessEntity,
    SalesYear,
    CurrentQuota,
    LAG(CurrentQuota, 1, 0) OVER
(ORDER BY SalesYear) AS
PrevQuota,
    LEAD(CurrentQuota, 1, 0) OVER
(ORDER BY SalesYear) AS
NextQuota
FROM
    SalesPersonQuotaHistory
WHERE
    BusinessEntityID = 275;
```



maker	guitar
Fender	C60
Yamaha	C40
Fender	Stratocaster
Fender	Prodigy
Yamaha	F310
Ibanez	RG421

```
SELECT maker, guitar,
    ROW_NUMBER() OVER
(ORDER BY maker) AS
row_num,
    RANK() OVER (ORDER
BY maker) AS rank,
    DENSE_RANK() OVER
(ORDER BY maker) AS
dense_rank
FROM gui
```

maker	guitar	row_ num	rank	dense_ rank
Fender	C60	1	1	1
Fender	Stratocaster	2	1	1
Fender	Prodigy	3	1	1
Ibanez	RG421	4	4	2
Yamaha	F310	5	5	3
Yamaha	C40	6	5	3

— в случае замены ORDER BY на PARTITION BY нумерация будет применяться в пределах окна. (мы пронумеруем все гитары Fender от 1 до 3 и т.д.)

Filtering based on the results of calculating the window function

- Window functions can used in a query only in the SELECT list and the ORDER BY clause.
- In all other cases, including GROUP BY, HAVING and WHERE, window functions cannot be used. This is because logically they are executed after these clauses, as well as after non-windowed aggregate functions, and therefore the aggregate function can be called in the arguments of the windowed one, but not vice versa.
- If you need to filter or group rows after calculating window functions, you can use a nested query.

Example: shows only rows with rank < 3

```
SELECT
    depname,
    empno,
    salary,
    enroll_date
FROM (
    SELECT
        depname,
        empno,
        salary,
        enroll_date,
        rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
    FROM
        empsalary
) AS ss
WHERE
    pos < 3;
```

Named windows

When multiple window functions are calculated in a query for similarly defined windows, it is possible to write a separate OVER clause for each of them. However, this approach would lead to duplication of code, which can inevitably result in errors. Therefore, it is better to define the window in a WINDOW clause and then reference it in the OVER clause.

Example of window functions

```
SELECT
    sum(salary) OVER w,
    avg(salary) OVER w
FROM
    empsalary
WINDOW
    w AS (PARTITION BY depname ORDER BY salary DESC);
```

Practice

1. Create a topic_6 schema
2. Create objects according to the script:

```
DROP SCHEMA IF EXISTS topic_6 CASCADE;
CREATE SCHEMA topic_6;

DROP TABLE IF EXISTS topic_6.participant;
CREATE TABLE topic_6.participant
(
    participant_id         INT PRIMARY KEY,
    participant_nm          VARCHAR(200),
    participant_birth_dt   DATE,
    participant_country_nm VARCHAR(200)
);
```

```
DROP TABLE IF EXISTS topic_6.competition;
CREATE TABLE topic_6.competition
(
    competition_id           INT PRIMARY KEY,
    competition_nm            VARCHAR(200),
    held_dt                  DATE,
    competition_country_nm   VARCHAR(100),
    result_sorting_type_code VARCHAR(10)
CHECK (result_sorting_type_code IN ('ASC',
'DESC'))
);
```

Practice

```
DROP TABLE IF EXISTS
topic_6.competition_result;
CREATE TABLE topic_6.competition_result
(
    competition_id         INT REFERENCES
topic_6.competition (competition_id),
    participant_id         INT REFERENCES
topic_6.participant (participant_id),
    participant_result_amt NUMERIC(20, 2)
);
```

3. Insert data into tables according to the script given professor. The result_sorting_type_code field determi which type of sorting should be used to rank the resu participants in the competition from worst to best

4. Get the top 1 result for each competition

5. For each competition, display the prizes

6. For each unused place, print the deviation from the result

7. For each competition, display all participants who a younger than the winner

8. For each competition, for each participant, display result of the participant, the result of the previous participant, the next participant, as well as the differen between them

9. For each competition, get the number of unique contestants without using GROUP BY

Practice

9. For each competition, get the number of unique contestants without using GROUP BY

10. Display statistics for each competition: specify which of the participants was the winner in the format "Took place X", "Did not get the prize"

11. Create a table with statistics for all competitions: display each participant, their date of birth, their result, the best result in the competition, the name of the participant with the best result, the deviation of the participant's result from 1st place, the average result in the competition, the deviation of the participant's result from the average, the minimum result in the competition, the name of the participant with the minimum result