Grouping the aggregated rows and using sub queries

- The learning objectives for this week are:
  - Knowing how the `GROUP BY` clause operates and how it is related to the aggregate functions
  - Knowing how to use the aggregate functions in filtering with the `HAVING` clause
  - Knowing what are subqueries and how they can be used in a `SELECT` statement
  - Knowing different kind of use-cases for subqueries, including row filter criteria, correlated subqueries, in the `SELECT` list, in the `HAVING` clause, and in the `FROM` clause

Grouping the aggregated rows

- So, an aggregate function performs a calculation for multiple rows so that the end result is a single value
- If the result table always contains just a single row, how can we write a query such as, *"what's the average grade from each course?"*
- To achieve this, we need to *group* the rows based on a specific column and perform the aggregate function for each group separately
- This can be done using the `GROUP BY` clause

The GROUP BY clause

```
GROUP BY column_list [ HAVING group_filtering_condition ]
```

- The `GROUP BY` clause uses a column or a group of columns in a `SELECT` statement to form groups of rows which the aggregate function operators on:

```
-- what's the average grade from each course?
SELECT course_code, AVG(grade) as average_grade FROM CourseGrade
-- form the groups based on the course_code
GROUP BY course_code
```

The GROUP BY clause

- The result table will have a row for *each distinct column value* of the `GROUP BY` column
- Each row has the corresponding aggregate function result for that group
- In the example's case the result table would contain the average grade for each distinct `course_code`:

course_code	average_grade
a290	2.5
a450	3.0
…	…

The GROUP BY clause

- As mentioned, the `GROUP BY` clause can have multiple columns
- In this case the result table will have a row for *each distinct combination of column values* of the `GROUP BY` columns

```
-- what's the average grade from each course instance?
SELECT course_code, instance_number, AVG(grade) as average_grade FROM CourseGrade
-- form the groups based on the course_code and instance_number
GROUP BY course_code, instance_number
```

The GROUP BY clause

- In the example's case the result table would contain the average grade for each distinct combination of `course_code` and `instance_number`:

```
SELECT course_code, instance_number, AVG(grade) as average_grade FROM CourseGrade
GROUP BY course_code, instance_number
```

course_code	instance_number	average_grade
a290	1	4.5
a290	2	3.0
a450	1	2.9
…	…	…

The GROUP BY clause

- It is worth noting that in the `SELECT` statement we can only select columns that are either aggregate functions or columns used in the `GROUP BY` clause:

```
-- ❌ student_number is not an aggreagate function, nor it is in the GROUP BY clause.
-- This will lead into an error
SELECT course_code, student_number, AVG(grade) as average_grade FROM CourseGrade
GROUP BY course_code
```

- This causes the following error:

> *"Column 'CourseGrade.student_number' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause"*

Combining with the WHERE clause

- We can use the `WHERE` clause to apply filtering **before the grouping is done** by the `GROUP BY` clause:

```
-- how many employees whose salary is above 10000 there are in each department?
SELECT deptno, COUNT(*) AS number_of_employees
FROM Employee
WHERE salary > 10000 -- The WHERE clause is applied before grouping is done
GROUP BY deptno
ORDER BY deptno
```

Using aggregate functions in filtering

- The `WHERE` clause can't use aggregate functions because it is **applied before** the `GROUP BY` clause while the query is executed
- Instead, we can use the `HAVING` clause to filter based on aggregate functions:

```
-- which departments have more than 10 employees?
-- ❌ can't use aggregate functions with the WHERE clause, this won't work
SELECT deptno, COUNT(*) AS number_of_employees
FROM Employee
WHERE COUNT(*) > 10
GROUP BY deptno

-- ✅ we should use the HAVING clause instead
SELECT deptno, COUNT(*) AS number_of_employees
FROM Employee
GROUP BY deptno
HAVING COUNT(*) > 10
```

Examples of GROUP BY clause

- What do we get from the following queries?

```sql
-- ❓ what do we get from this query?
SELECT campus_code, COUNT(campus_code) AS number_of_teachers
FROM Teacher
GROUP BY campus_code

-- ❓ what do we get from this query?
SELECT course_code, AVG(grade) AS average_grade
FROM CourseGrade
WHERE grade_date BETWEEN '2024-01-01' AND '2024-12-31'
GROUP BY course_code
HAVING COUNT(grade) > 10
```

SQL logical query processing order

- The following diagram repsents **the logical query processing order** in SQL
- We can only use data from **the previous phases**, meaning that, for example a `WHERE` statement can't use data from a `GROUP BY` statement, but `ORDER BY` statement can

```sql
SELECT deptno, COUNT(*) AS number_of_employees
FROM Employee
-- ❌ Not ok, WHERE statement is processed before GROUP BY statement
WHERE COUNT(*) > 10
GROUP BY deptno
-- ✅ Ok, GROUP BY statement is processed before ORDER BY statement
ORDER BY number_of_employees
```

Subqueries

- A *subquery* is a query within another query, which is used to retrieve data that will be processed by the outer query
- The most common use case for a subquery is to use subquery result in a filtering condintion in a `WHERE` clause
- Subqueries can also contain another subquery
- Most of our examples will cover usage of subqueries with the `SELECT` statement, but they can be used with e.g. `INSERT INTO` and `UPDATE` statements as well

```
--- who are the teachers with a above average salary?
SELECT first_name, surname, salary
FROM Teacher
WHERE salary > (
  -- subquery for calculating the average salary
  SELECT AVG(salary) FROM Teacher
)
```

Subqueries in a SELECT statement

- The subquery is placed inside brackets `()` and its result will be passed to the outer query
- In a `SELECT` statement, we can nest a subquery within a:
  - `WHERE` clause as a row filter criteria to be used in the condition
  - `WHERE` clause as a correlated subquery
  - `SELECT` list as a column expression or as a part of a column expression
  - `HAVING` clause as a group filter criteria within a GROUP BY clause
  - `FROM` clause to create a temporary derived table
  - `WITH` clause to create a temporary named result set, known as common table expression (CTE)

Subquery as a row filter criteria

- The most common use of the subquery is to use it as a row filter criteria similarly as e.g. litrals:

```
-- using literal 5000 as a filter criteria
WHERE salary > 5000

-- using a subquery as a filter criteria (note the brackets wrapping the subquery)
WHERE salary > (
  -- the result of the subquery will be used in the comparison
  SELECT AVG(salary) FROM Teacher
)
```

Subquery as a row filter criteria

```sql
-- which countries have larger population than Australia?
SELECT country_name, population
FROM Country
WHERE population > (
  -- subquery for getting the population of Australia
  SELECT population
  FROM Country
  WHERE country_name = 'Australia'
)

-- ❓ what do we get from this query?
SELECT country_name, population
FROM Country
WHERE population = (SELECT MAX(population) FROM Country)

-- ❓ what do we get from this query?
SELECT empno, empname
FROM Employee
WHERE empno NOT IN (SELECT empno FROM Project_Employee)
```

Correlated subqueries

- A *correlated subquery* (inner query) uses one or more values from the outer query
- The correlated subquery is executed once for each row that is selected by the outer query

```
-- which students are from a city where there is a campus?
SELECT city, surname, given_name, student_number
FROM Student
-- using a correlated subquery
WHERE EXISTS (
  -- does any such row EXIST in the Campus table where Campus.city = Student.city
  SELECT * FROM Campus
  WHERE Campus.city = Student.city
)
```

Performance of correlated subqueries

- 🐌 In the example, the correlated subquery is executed once per each student, which will degrade the query performance
- 🚀 Sometimes It might be better to use a non-correlated subquery to improve readability and performance:

```
-- 🚀 same result, with using a non-correlated subquery
SELECT city, surname, given_name, student_number
FROM Student
WHERE city IN (SELECT city FROM Campus)
```

Subqueries within a SELECT list

- A subquery can be used in the `SELECT` list to calculate a value for a column that will be displayed in the result table:

```
-- what is the percentage of red cars?
SELECT (SELECT 100.0 * COUNT(*) FROM Car WHERE colour = 'red') / COUNT(*)
 AS percentage_of_red_cars
FROM Car

-- what is the average grade for each student?
SELECT
student_number, (
  SELECT AVG(grade) FROM CourseGrade
  WHERE CourseGrade.student_number = Student.student_number
) AS average_grade
FROM Student
```

Subqueries within a HAVING clause

- Similarly as in the `WHERE` clause, a subquery can be used in the `HAVING` clause to filter the groups based on the aggregate function result:

```
-- in which departments the average salary is higher than the average salary of all employees?
SELECT deptno, AVG(salary) AS average_salary
FROM Employee
GROUP BY deptno HAVING AVG(salary) > (SELECT AVG(salary) FROM Employee)
```

Subqueries within a FROM clause

- A subquery can be used in the `FROM` clause to create a temporary derived table that can be used in the outer query
- The subquery *must have an alias name* (`FROM (subquery) AS alias_name`)
- The subquery's result set can be used in the `FROM` clause similarly to a normal table
- Let's consider the following example:

> *"what is the count, and the minimum and the maximum grade point average (GPA) of such students who have passed more than 20 courses?"*

Subqueries within a FROM clause

- First, we define a query for the grade point average of students who have passed more than 20 courses:

```
SELECT AVG(grade) AS gpa
FROM CourseGrade
WHERE grade > 0
GROUP BY student_number
HAVING COUNT(*) > 20
```

Subqueries within a FROM clause

- Then, we use this query as a subquery in the `FROM` clause:

```
SELECT COUNT(*) AS count, MIN(gpa) AS min_gpa, MAX(gpa) AS max_gpa
FROM (
  -- our subquery from the previous slide
  SELECT AVG(grade) AS gpa
  FROM CourseGrade
  WHERE grade > 0
  GROUP BY student_number
  HAVING COUNT(*) > 20
) AS GpaTable -- ⚠️ alias name for the subquery is required
```

Subqueries within a WITH clause

- A subquery can be used in the `WITH` clause to create a temporary named result set, known as common table expression (CTE)
- CTEs are useful for improving the readability of the query and can be used multiple times in the query
- Let’s consider the following example:

> *"Which department has the highest number of employees"*

Subqueries within a WITH clause

- First, we define a CTE with the `WITH` clause for the number of employees in each department:

```
-- common table expression
WITH DeptInfo (deptno, employee_count) AS (
  SELECT deptno, COUNT(*) AS employee_count
  FROM Employee
  GROUP BY deptno
)
```

Subqueries within a WITH clause

- Then, we use this CTE in the main query to find the department with the highest number of employees:

```
-- common table expression from the previous slide
WITH DeptInfo (deptno, employee_count) AS (
  SELECT deptno, COUNT(*) AS employee_count
  FROM Employee
  GROUP BY deptno
)

-- main query
SELECT deptno, employee_count
FROM DeptInfo -- using the CTE
WHERE employee_count = (SELECT MAX(employee_count ) FROM DeptInfo) -- using the CTE again
```

Summary

- The `GROUP BY` clause is used to group the rows based on a specific column or columns
- The `HAVING` clause is used to filter the groups based on the aggregate function result
- Subqueries are queries within another query
- Subqueries can be used in a `SELECT` statement in the `WHERE` clause, `SELECT` list, `HAVING` clause, `FROM` clause, and `WITH` clause
- The `WITH` clause is used to create a temporary named result set, known as common table expression (CTE)