SQL queries

- The learning objectives for this week are:
  - Knowing how to use the `SELECT` statement along with the `WHERE` clause to filter rows using **comparison and logical operators**
  - Knowing how to define **alias names** for selected columns
  - Knowing how to handle string values in queries
  - Knowing how to do **arithmetic operations**
  - Knowing how to define **conditional expressions** with the `CASE` expression
  - Knowing how to handle missing, `NULL` values in queries
  - Knowing how to **omit duplicate rows** with `SELECT DISTINCT` statement

The SQL syntax

> *"The syntax of a computer language is the rules that define the combinations of symbols that are considered to be correctly structured statements or expressions in that language"*

- So that the RDBMS can interpret our queries we need to closely follow a specific set of rules while defining them
- These rules are defined by the **SQL syntax**
- If we don’t follow the syntax in our SQL queries, the RDBMS will response to the query with an error message
- The "SQL DML Quick Reference" document in Moodle has the syntax definitions we need during the course

The SELECT statement syntax

```
SELECT [ DISTINCT ] { * |
 { column_expression [ AS column_alias ] [ { , column_expression [ AS column_alias ] }... ] }
}
FROM table_name [ [ AS ] table_alias ]
[ { [ INNER ] JOIN table_name [ [ AS ] table_alias ] ON join_condition }... ]
[ WHERE search_condition ]
[ GROUP BY column_list [ HAVING group_filtering_condition ] ]
[ ORDER BY sort_specification_list ]
```

```
-- ❌ wrong syntax for the SELECT statement, "Incorrect syntax near the keyword 'FROM'."
SELECT FROM Student first_name, surname
-- ✅ correct syntax for the SELECT statement
SELECT first_name, surname FROM Student
-- ✅ new lines can be used to improve readability
SELECT first_name, surname
FROM Student
-- ⚠️ uppercase keywords (e.g. SELECT) aren't required but they improve readability
select first_name, surname from Student
```

The SELECT statement

- The `SELECT` statement is used to **select rows from a table**
- We can define a group of columns we want to select from the target table, or select every column
- The result is a result table containing the rows from the target table with the specified columns

```
--- select all the students and display every column
SELECT * FROM Student
--- select all the students and display only the first_name and surname columns
SELECT first_name, surname FROM Student
```

The WHERE clause

- We can **filter the selected rows** of a table with the `WHERE` clause
- We can define a condition which the selected rows should satisfy
- The condition is quite similar as the `if` statement condition in programming languages
- The condition is commonly a combination of **comparison operators** (e.g. `=`, or `>`) and **logical operators** (e.g. `AND`, or `OR`)
- The result table only contains the rows that satisfy the condition

```
-- list all student's whose first name is Matti
SELECT first_name, surname FROM Student WHERE first_name = 'Matti'
```

Comparison operators

- The `WHERE` clause conditions support similar **comparison operators** as many programming languages

```
WHERE first_name = 'Matti' -- equal to. ⚠️ Note, just a single = symbol
WHERE first_name <> 'Matti' -- not equal to
WHERE start_year > 2020 -- greater than
WHERE start_year >= 2020 -- greater than or equal
WHERE start_year < 2025 -- less than
WHERE start_year <= 2025 -- less than or equal
-- 💡 greater than and less than operators work also for e.g. dates and strings
WHERE birth_date >= '1993-01-01'
WHERE finnish_proficiency_level <= 'B2'
```

Logical operators

- Comparisons can be combined with **logical operators** to achieve conditions such as *"age is greater than 18 and age is less than 30"*

```
WHERE start_year > 2020 AND start_year < 2025  -- AND operator
WHERE first_name = 'Matti' OR first_name = 'Kaarina' -- OR operator
WHERE NOT start_year < 2020 -- NOT operator
```

- The `AND` operator **is evaluated first**, before the `OR` operator, similarly as the multiplication operator (*) is evaluated before the sum operator (+)

```
-- these two conditions are the same
-- "either first name is Matti or it is Elina in city Helsinki"
first_name = 'Matti' OR first_name = 'Elina' AND city = 'Helsinki'
first_name = 'Matti' OR (first_name = 'Elina' AND city = 'Helsinki')
-- similarly as 1 + 2 * 3 = 1 + (2 * 3) = 7
```

Controlling the applying order of logical operators

- If we want to deviate from the default order of evaluation, we can use brackets to determine in which order the logical operators should be applied

```
--- true, when first name is Matti no matter the city
first_name = 'Matti' OR first_name = 'Elina' AND city = 'Helsinki'
--- false, when first name is Matti an city is not Helsinki
(first_name = 'Matti' OR first_name = 'Elina') AND city = 'Helsinki'
-- 💡 using brackets can clarify the condition even if they aren't strictly necessary
first_name = 'Matti' OR (first_name = 'Elina' AND city = 'Helsinki')
-- the condition above would be the same without the brackets
```

Order by

- The order of result table's rows is **unpredictable**, it might not be the same each time we execute the query
- We can use the `ORDER BY` clause to define **in which order** we want the rows to be in the result table
- The sorting is done based on one or many columns in the specified order

```
SELECT course_name, credits
FROM Course
ORDER BY credits -- rows will be sorted by the credits column's value
```

- **Sorting works for all kinds of data types**, e.g. numbers, strings, and dates:
  - Numbers are sorted from smallest to biggest number (e.g. 1, 2, 3)
  - Strings are sorted alphabetically (e.g. A, D, X)
  - Dates are sorted from oldest to newest date (e.g. 1990-01-01, 1990-06-13, 2025-08-15)

Order by with multiple columns

- Table might contain multiple rows with the same value in the column used in the `ORDER BY` clause
- To determine the order of such rows we can provide **multiple columns** to the `ORDER BY` clause

```
SELECT course_name, credits
FROM Course
-- when the credits is the same, the course_name is used to determine the order
ORDER BY credits, course_name
```

course_name	credits
Algorithms	⚠️ 4
Python Programming	⚠️ 4
Databases	3

Switching the sort order

- The `ORDER BY` sorts the records in **ascending order** (smallest value first) by default
- We can change the order by using either `ASC` (ascending order) or `DESC` (descending order) keyword

```
SELECT course_name, credits
FROM Course
-- use descending order for credits and ascending order for course_name
ORDER BY credits DESC, course_name ASC
```

Column aliases

- When we select columns from a table with the `SELECT` statement, the result table will have the same column names

```
SELECT first_name, surname FROM Student
```

first_name	surname
John	Doe
...	...

Column aliases

- We can have a different column name for the result table by defining an **alias name** for the column
- Column alias name is defined using the `column_name AS alias_column name` syntax

```
SELECT first_name AS given_name, surname AS family_name
FROM STUDENT
```

given_name	family_name
John	Doe
...	...

Column aliases

- Alias names are handy for renaming columns, but we can also use them to define **additional columns** for the result table
- The additional columns don't have to exist in the target table, they can be, for example computed from target table's columns

```
-- combine first_name and surname and name the column full_name
SELECT first_name + ' ' + surname AS full_name FROM Student
```

full_name
John Doe
…

Column aliases

- In fact, the content before the `AS alias_column_name` is an **column expression**
- Column expression can for example be a literal value, an arithmetic operation performed on target table columns, or a function call

```
-- a literal value 1
SELECT student_number, 1 AS literal_value FROM Student
-- an arithmetic operation grade * 20
SELECT course_code, grade * 20 AS zero_to_hundred_scale_grade FROM CourseGrade
-- a function call CONCAT(first_name, ' ', surname)
SELECT student_number, CONCAT(first_name, ' ', surname) AS full_name FROM Student
```

Column aliases

- It’s worth noting, that column aliases **can’t be used** in the `WHERE` clause

```
-- ❌ this won't work
SELECT first_name + ' ' + surname AS full_name
FROM Student
WHERE full_name = 'Matti Keto'

-- ✅ this will work
SELECT first_name + ' ' + surname AS full_name
FROM Student
WHERE first_name + ' ' + surname = 'Matti Keto'
```

String concatenation

- Combining string values to produce a new string is called **string concatenation**
- String concatenation can be done using the + operator similarly as in many programming languages, such as Java
- Alternatively, we can use the `CONCAT` function

```
-- combine first_name and surname using the + operator
SELECT first_name + ' ' + surname AS full_name FROM Student
-- combine first_name and surname using the CONCAT function
SELECT CONCAT(first_name, ' ', surname) AS full_name FROM Student
```

String functions

- SQL Server provides many built-in **functions**, which can be used e.g. to operate on strings and dates
- It is a good idea to always check if there is a built-in function for a specific general problem before trying to come up with a more complex solution ourselves
- Here are a few examples of built-in **string functions** in SQL Server:

```
-- LEFT and RIGHT return the left or right part of a string
-- with the specified number of characters
SELECT first_name, surname FROM Student WHERE LEFT(surname, 1) = 'K'
SELECT first_name, surname  FROM Student WHERE RIGHT(surname, 1) = 'a'
-- LEN returns the length of the strings
SELECT first_name, surname FROM Student WHERE LEN(surname) = 4
-- CHARINDEX Searches a substring for a string and returns its starting
-- position if found. Returns zero if not found.
SELECT first_name, surname FROM Student WHERE CHARINDEX('ta', surname) <> 0
```

- The "SQL DML Quick Reference" document in Moodle contains more such useful functions

Example of string functions

> *"List all the students (gender, birth date, surname, first name) whose surname starts with the letter L or K. Sort the results in descending order by birth date."*

- We'll need to know the *first letter* of the surname column and check if that is either "L" or "K"
- We can use the `LEFT` function to get the first letter of the surname column
- The `OR` logical operator can be used to construct the "starts with the letter L or K" condition

```
SELECT gender, birth_date, surname, first_name
FROM Student
-- we use the LEFT function to get the "first one letter from the left"
WHERE LEFT(surname, 1) = 'L' OR LEFT(surname, 1) = 'K'
ORDER BY birth_date DESC; -- "sort the results in descending order by birth date"
```

Arithmetic operations

- SQL supports similar arithmetic operators for calculations as many programming languages

```
-- the + operator for addition
SELECT credits, credits + 2 AS credits_calculation FROM Course
-- the - operator for substraction
SELECT credits, credits - 2 AS credits_calculation FROM Course
-- the * operator for multiplication
SELECT credits, credits * 2 AS credits_calculation FROM Course
-- the / operator for division
SELECT credits, credits / 2 AS credits_calculation FROM Course
-- the % operator for remainder of a division
SELECT credits, credits % 2 AS credits_calculation FROM Course
```

Arithmetic operations

- We can use brackets to determine the order operations

```
-- first calculate credits * 20, then dive the result with 2
SELECT (credits * 20) / 2 AS credits_calculation FROM Course
```

Conditional expressions

- The `CASE` expression allows us to use conditional logic in `SELECT` statements
- With the **simple variation** of `CASE` expression we compares a single expression (e.g. value of a column) to a set of simple expressions (e.g. literals) to determine the result
- We can use the simple variation to map column values to different values

```
SELECT
first_name,
surname,
CASE gender -- the gender column is used in comparisons
  WHEN 'F' THEN 'Female'
  WHEN 'M' THEN 'Male'
  ELSE 'Unknown' -- the ELSE clause is optional
END AS gender_description
FROM Student
```

Conditional expressions

- With the **searched variation** of `CASE` expression we can have a set of separate conditions to determine the result

```
SELECT
course_name,
credits,
CASE -- we don't define a expression here
  -- we have separate conditions, which could use any columns
  WHEN credits < 3 THEN 'Small amount of work'
  WHEN credits >= 3 AND credits <= 5 THEN 'Some amount of work'
  ELSE 'Big amount of work'
END AS workload_description
FROM Course
```

Handling missing values (NULLs)

- When column is missing a value, its value is `NULL`
- ⚠️ `NULL` is not the same as for example empty string `''` or zero
- Columns that do not have a value in the `INSERT INTO` statement will have a `NULL` value

```
-- ❌ missing surname violates NOT NULL constraint of the surname column
INSERT INTO Student (student_number, first_name, birth_date, gender)
VALUES ('o193', 'Kalle', '1993-01-19', 'M')

-- ✅ empty surname does not violate NOT NULL constraint of the surname column
INSERT INTO Student (student_number, first_name, surname, birth_date, gender)
VALUES ('o193', 'Kalle', '', '1993-01-19', 'M')
```

Queries with NULLs

- We can use `IS NULL` and `IS NOT NULL` operators to test for `NULL` values
- ⚠️ The equals `=` and not equals `<>` operators **cannot** be used to test for `NULL` values, because `NULL` value cannot be equal or unequal to any value (including `NULL`)

```
-- ❌ this won't work, we cannot use the equals = operator
SELECT student_number, email FROM Student WHERE email = NULL

-- ✅ instead, let's use the IS NULL operator
SELECT student_number, email FROM Student WHERE email IS NULL
```

Omitting duplicate rows

- A common query problem is that we want to know what are all **distinct values** for a column or group of columns
- For example, *"what are the available number of credits from courses?"*
- We can use the `SELECT DISTINCT` statement to select only distinct (different) values

```
-- ❌ many courses have the same number of credits
SELECT credits FROM Course

-- ✅ SELECT DISTINCT statement omits duplicate number of credits
SELECT DISTINCT credits FROM Course
```

Select distinct

- We can also define a **group columns** that needs to distinct in the result table
- For example, *"what are the courses teached by each teacher?"*

```
-- group of teacher_number and course_code
-- needs to be distinct in the result table
SELECT DISTINCT teacher_number, course_code FROM CourseInstance
ORDER BY teacher_number
```

Summary

- The `SELECT` statement is used to select rows from a table
- We can use a **column alias** `column_expression AS alias_name` to use different name for a target table column or to compute a new column
- **String concatenation** can be done using the + operator or the `CONCAT` function
- **Logical operators** `AND` and `OR` can be used to define conditions
- Calculations can be done using **arithmetic operators** + (addition), - (substraction), * (multiplication) and / (division)
- The `CASE` expression allows us to use conditional logic in `SELECT` statements
- We must use `IS NULL` and `IS NOT NULL` operators to test for `NULL` values
- We can use the `SELECT DISTINCT` statement to select **only distinct (different) values**