SQL essentials

Mastering database queries:
a comprehensive guide to SQL

Learning session 02

Selecting, filtering, ordering and limiting your data

Instructor
Péter Fülöp
peteristvan.fulop@hcl.com

SELECTing your data

Meet the SELECT command

The simplest version of the SQL SELECT command consists of:

SQL

```
SELECT <expression 1>, …,<expression n>;
```

Let's take a look at a few simple examples.

SQL

```
SELECT 1.23*4.56, 'Apple'<>'apple', current_time;
```

We can assign **aliases** to the expressions listed in the SELECT statement.

SQL

```
SELECT 2+5 AS MyLuckyNumber, 'Apple'<>'apple' AS string_comparison,
       current_time AS "The current time”;
```

Performing queries on your tables using the SELECT command

Utilizing the **FROM** clause in conjunction with the SELECT statement:

```SQL
SELECT <expression 1>, …,<expression n> FROM table
```

`<expression 1>, …,<expression n>` can represent any **valid combination** of SQL functions, operators, and operands that yield a result or value when applied to the data stored in the specified table.

These expressions can include:

- specific **column names** from the table
- **wildcard** selections like * to retrieve all columns
- mathematical calculations (with constants or the table columns)
- string / date manipulations
- conditional statements, or even aggregate functions like SUM, AVG, or COUNT

Selecting all columns

Simple examples of using the * wildcard:

SQL

```sql
-- To avoid specifying the schema, simply execute "SET search_path TO practice_02;"
SELECT * FROM practice_02.employees;
```

The query above will produce a result that includes all the rows and columns from the employees table.

OUTPUT

id	first_name	middle_name	last_name	hire_date	birth_date	unit_of_work
1	Gábor	Antal	Kovács	2021-05-15	1990-02-01	Consulting 1
2	Erzsébet	*<null>*	Tóth	2020-03-02	1992-04-05	Managed services
3	Zoltán	Péter	Szabó	2019-07-13	1988-11-10	Team augmentation 1
4	Anikó	Ilona	Varga	2018-10-21	1985-07-09	Team Augmentation 2

Note: in a production environment or when dealing with large tables, it is generally discouraged to use the asterisk (*) as it's considered a bad practice.

Specifying columns (explicit selecting)

To select specific columns, please list the desired columns after the SELECT keyword, separating them with commas.

Retrieve the first name, last name, and birth date of all employees.

SQL

```sql
SELECT first_name, last_name, birth_date FROM practice_02.employees;
```

The query above will produce the resulta below:

OUTPUT

first_name	last_name	birth_date
Zsófia	Szabó	1988-04-18
Attila	Horváth	1992-12-14
Bence	Nagy	1990-03-12

37 rows

Applying column operations, using aliases

The company has decided to implement a 10% salary increase for all employees. Provide a list of employee ids, first names, last names, current salaries, new salaries after the 10% increase, and the actual numeric increase in salary.

SQL

```sql
SELECT id, first_name, last_name,
       salary AS "Current salary", salary*1.1 "New salary",
       0.1*salary AS SalaryIncrease
FROM practice_02.employees;
```

OUTPUT

id	first_name	last_name	Current salary	New salary	salaryincrease
14	Zsófia	Szabó	4700	5170	470
15	Attila	Horváth	4500	4950	450
16	Bence	Nagy	3000	3300	

37 rows

Selecting DISTINCT values

Display the last names of all individuals in the employees table.

```sql
SELECT last_name FROM practice_02.employees;
```

OUTPUT

last_name
Szabó
Horváth
Nagy
Kovács
Tóth
Szabó
Horváth

37 rows

Since multiple individuals can share the same last name, executing the query above may result in duplicated rows in the output.

Utilizing the **DISTINCT** keyword can aid in eliminating duplicates.
However, it's advisable to exercise caution when using DISTINCT, as we will explore more effective solutions for duplicate removal later in this course.

Selecting DISTINCT values

Display the DISTINCT last names of all individuals in the employees table.

```sql
SELECT DISTINCT last_name FROM practice_02.employees;
```

OUTPUT

last_name
Török
Szabó
Horváth
Smith
Varga
Farkas

16 rows

In this instance, the query generated 16 distinct rows without any repetition.

Show a list of unique countries where the employees are currently working from

```sql
SELECT DISTINCT country_of_work
FROM practice_02.employees;
```

Selecting DISTINCT values

Display a list of unique combinations of places and countries where the employees are currently working.

SQL

```sql
SELECT DISTINCT place_of_work, country_of_work FROM practice_02.employees;
```

OUTPUT

place_of_work	country_of_work
Kaposvár	Hungary
New York	USA
Szeged	Hungary

15 rows

Explain the results of the `SELECT DISTINCT (place_of_work, country_of_work) FROM practice_02.employees;` query.

Operations with NULL values

Employees are required to undergo a medical examination annually. The date for the next medical examination will be set to one year after their last examination.
However, if an employee's "date_of_last_medical_analysis" is empty (NULL) for any reason, the next examination will be scheduled for two weeks from the current date.

SQL

```sql
SELECT first_name, last_name, date_of_last_medical_analysis,
       date_of_last_medical_analysis + INTERVAL '1 year' AS next_medical_analysis
FROM practice_02.employees;
```

OUTPUT

first_name	last_name	date_of_last_medical_analysis	next_medical_analysis
Márton	Varga	2022-05-15	2023-05-15 00:00:00.0000
Eszter	Kiss	<null>	<null>
Gábor	Kovács	<null>	<null>

37 rows

Managing NULL values

In the context of (Postgre)SQL, "**NULL**" is a special marker used to represent **missing or unknown data** in a database column. It indicates that the value for a particular field is not currently known or is undefined.

MISSING DATA: NULL is used to indicate the absence of a specific value in a column. It doesn't mean that the value is zero or empty; it signifies that the data is unknown or unavailable.

Methods and language components for **managing NULL values**:

- Check if an expression is NULL using "*`expression IS NULL`*."
- Verify if an expression is not NULL with "*`expression IS NOT NULL`*."
- Utilize the *`COALESCE(argument_1, argument_2,…)`* function which accepts an unlimited number of arguments. It returns the first argument that is not null.

Managing NULL values example

Display the first name, last name, and the most recent medical analysis date for every employee.
Examine whether employees possess a non-null value in the "date_of_last_medical_analysis" field.
Compute the "next_medical_analysis" date by adding one year to the last analysis date for employees with non-null entries and by adding two weeks to the current date for those with missing last analysis dates.

SQL

```sql
SELECT
  first_name,
  last_name,
  date_of_last_medical_analysis AS analysis_date,
  date_of_last_medical_analysis IS NULL AS no_analysis_yet,
  date_of_last_medical_analysis IS NOT NULL AS has_previous_analysis,
  CAST(
       COALESCE(date_of_last_medical_analysis + INTERVAL '1 year', CURRENT_DATE + INTERVAL '2 weeks')
  AS DATE) AS next_medical_analysis
FROM
  practice_02.employees;
```

Managing NULL values - analyze the results

OUTPUT

first_name	last_name	analysis_date	no_analysis_yet	has_previous_analysis	next_medical_analysis
Zsófia	Szabó	2022-04-22	false	true	2023-04-22
Attila	Horváth	<null>	true	false	2023-10-30
Bence	Nagy	2022-02-11	false	true	2023-02-11
Hanna	Kovács	<null>	true	false	2023-10-30
László	Tóth	<null>	true	false	2023-10-30
Eszter	Szabó	2022-03-03	false	true	2023-03-03
Márk	Horváth	<null>	true	false	2023-10-30

37 rows

SQL(fragment)

```sql
-- the use of COALESCE can be replaced as below, however the use of COALESCE is more concise
CASE WHEN date_of_last_medical_analysis IS NOT NULL THEN
    CAST(date_of_last_medical_analysis + INTERVAL '1 year' AS DATE)
ELSE
    CAST(CURRENT_DATE + INTERVAL '2 weeks' AS DATE)
END AS next_medical_analysis
```

Writing CASE statements

SQL

```
CASE
      WHEN condition_1  THEN result_1
      WHEN condition_2  THEN result_2
      ...
      ELSE  result_n
END
```

Furnish a list that includes the first names, last names, current salaries, new salaries, and the corresponding salary increases for employees.
The salary adjustments adhere to the following guidelines: For Junior1, there is an 8% increase; for Junior2, a 7% increase; for Medior, a 6% increase; and for Senior1 and above, there is a 5% increase if they were hired before 2020-01-01 and a 3% increase otherwise.

Writing CASE statements

You should substitute the ELSE sections with the suitable WHEN-THEN combinations.

SQL

```
SELECT first_name, last_name, band, hire_date,salary  as current_salary,
       CASE
           WHEN (SUBSTR(band, 1, 6) = 'Senior') AND (TO_DATE('2020-01-01', 'YMD') > hire_date)
               THEN salary * 1.05
           WHEN (SUBSTR(band, 1, 6) = 'Senior') AND (TO_DATE('2020-01-01', 'YMD') < hire_date)
               THEN salary * 1.03
           ELSE salary * 1.1
           END as new_salary,
       CASE
           WHEN (SUBSTR(band, 1, 6) = 'Senior') AND (TO_DATE('2020-01-01', 'YMD') > hire_date)
               THEN salary * 0.05
           WHEN (SUBSTR(band, 1, 6) = 'Senior') AND (TO_DATE('2020-01-01', 'YMD') < hire_date)
               THEN salary * 0.03
           ELSE salary * 1.1
           END as salary_increase
FROM practice_02.employees;
```

One more exercise involving the CASE statement

Provide a list containing employees' complete names, along with their corresponding email addresses and the working unit to which they are assigned. To derive the full name, concatenate the first name, middle name, and last name.

SQL

```sql
SELECT CONCAT(first_name, ' ', middle_name, ' ', last_name) AS full_name,
       email_address, unit_of_work
FROM practice_02.employees;
```

This solution requires minor adjustments (double spaces):

OUTPUT

full_name	email_address	unit_of_work
Hanna Ildikó Kovács	hanna.kovacs@hcl.com	Managed services
László Tóth	laszlo.toth@hcl.com	Team augmentation 1
Eszter Szabó	eszter.szabo@hcl.com	Team Augmentat

37 rows

One more exercise involving the CASE statement (solutions)

SOLUTION 1 — SQL

```sql
SELECT
    CASE
        WHEN middle_name IS NULL
            THEN CONCAT(first_name, ' ', last_name)
        ELSE CONCAT(first_name, ' ', middle_name, ' ', last_name)
    END as full_name,
        email_address, unit_of_work
FROM practice_02.employees;
```

SOLUTION 2 — SQL

```sql
SELECT REPLACE(CONCAT(first_name, ' ', middle_name, ' ', last_name), '  ', ' ',)  AS full_name,
        email_address, unit_of_work
FROM practice_02.employees;
```

FILTERing your data

The WHERE clause

SQL

```sql
SELECT
   expression_1, …,expression_n
FROM
   table
WHERE
   condition
```

The **WHERE clause** enables you to filter and retrieve specific rows from a database table based on a defined condition.

It serves as a powerful tool for narrowing down your query results **to only include the data that meets certain criteria**.

Examples using the WHERE clause

Think of the **WHERE clause** as your data's gatekeeper.

When you execute a **SELECT statement with a WHERE** clause, you're instructing the database to evaluate a condition for each row in the table.

Only the rows that meet the specified condition will be included in the result set, while the others are left out.

SQL

```sql
-- Task: Retrieve employees who are currently stationed at the 'Budapest' office.
SELECT
    first_name, last_name, place_of_work
FROM
    practice_02.employees
WHERE
    place_of_work = 'Budapest';
```

The query won't return any results when using the condition `place_of_work = 'budapest'` due to the case-sensitive nature of string literals. To make it case-insensitive, you can use `LOWER(place_of_work) = 'budapest'`

Examples using the WHERE clause

The "-" operation between two date objects computes the day-based difference between the two dates.

SQL

```sql
SELECT
  first_name,
  last_name,
  email_address
FROM
  practice_02.employees
WHERE (date_of_last_medical_analysis IS NULL)
  OR(CURRENT_DATE - COALESCE(date_of_last_medical_analysis, CURRENT_DATE) > 365)
```

Using the BETWEEN logical operator in the WHERE clause

SQL (fragment)

```
value BETWEEN lower_value AND upper_value;
```

The **BETWEEN** operator can be used with various types of values, including:

- **Numeric types**: you can use it with integer, real, double precision, and other numeric data types.
- **Date and time types**: it can be applied to date, timestamp, and interval data types.
- **Character Types**: You can also use it with character and text data types. Please note that the comparison is based on lexicographic (string) order.
- **Other Types:** you can use BETWEEN with other data types like boolean, UUID, and custom-defined types.

Using the BETWEEN logical operator in the WHERE clause

Retrieve the first names, last names, salaries, and hire dates of employees who were employed between January 1, 2022, and March 31, 2023, and whose salaries fall within the range of 2800 to 4000.

SQL

```sql
SELECT first_name, last_name, salary, hire_date
FROM practice_02.employees
WHERE (hire_date BETWEEN to_date('2022-01-01', 'YMD') AND TO_DATE('2023-03-31', 'Y-M-D'))
      AND salary BETWEEN 2800 AND 4000;
```

OUTPUT

first_name	last_name	salary	hire_date
Gergő	Tóth	2800	2022-05-08
Márton	Varga	4000	2022-11-23
László	Papp	3200	2022-08-17

33 rows

ORDERing your data

The ORDER BY clause

SQL

```
SELECT
    expression_1, …,expression_n
FROM
    table
WHERE
    condition
ORDER BY
    oexpression_1 [ASC|DESC], …, oexpression_m [ASC|DESC]
```

The **ORDER BY clause** allows you to sort the result set of your queries in a specified order, whether it's ascending (from the smallest value to the largest) or descending (from the largest value to the smallest).
With this powerful clause, you can easily arrange your data alphabetically, numerically, or by any other criteria you choose.

The ORDER BY clause

Please provide a list of employees' first names, last names, and hire dates, ordered by their date of joining the company. In case multiple employees joined on the same day, prioritize sorting them based on their last names and then their first names.

SQL

```sql
SELECT first_name, last_name, hire_date
FROM practice_02.employees
ORDER BY hire_date ASC, last_name ASC, first_name ASC;
```

OUTPUT

last_name	first_name	hire_date
Kiss	Zoltán	2017-02-12
Miller	William	2017-08-07
Nagy	József	2017-08-07

37 rows

The ORDER BY clause (optional ASC)

Display the first name, last name, work unit, and salary of the employees. Sort the list by the employees' work units, with those in the same unit arranged in descending order of their salaries. In case of employees within the same unit having the same salary, further sort them in ascending order based on their last names and then their first names.

SQL

```sql
SELECT last_name, first_name, unit_of_work,salary
FROM practice_02.employees
ORDER BY unit_of_work ASC, salary DESC, last_name, first_name
```

OUTPUT

last_name	first_name	unit_of_work	salary
Tóth	Anikó	Consulting 1	4900
Horváth	Attila	Consulting 1	4500
Horváth	Márk	Consulting 1	3200
Kovács	József	Consulting 1	3000

37 rows

Using aliases in the ORDER BY clause

Provide a list of employees who joined before '2023-01-01,' including their full name (formed by combining first name, middle name, and last name), hire date, work unit, and salary. Sort the list in ascending order based on the full names of the employees.

SQL

```sql
SELECT CASE
           WHEN middle_name IS NULL THEN CONCAT(first_name, ' ', last_name)
           ELSE CONCAT(first_name, ' ', middle_name, ' ', last_name)
       END as full_name,hire_date,unit_of_work, salary
FROM practice_02.employees
WHERE hire_date < TO_DATE('2023-01-01', 'YMD')
ORDER BY full_name ASC;
```

OUTPUT

full_name	hire_date	unit_of_work
Anikó Ilona Varga	2018-10-21	Team Augmentation

36 rows

LIMITing your data

The LIMIT clause

SQL

```
SELECT expression_1, …,expression_n
FROM table
LIMIT N
```

The **LIMIT** clause is used to get a subset of rows generated by a query. It is an optional clause of the SELECT statement.

SQL

```
SELECT expression_1, …,expression_n
FROM table
LIMIT N OFFSET M
```

The **OFFSET** clause placed after the LIMIT clause skips “m” number of rows before returning the result query.

Using the LIMIT clause

Obtain the information for the top 5 employees who have the highest salaries, which should include their first name, last name, and salary.

SQL

```sql
SELECT first_name, last_name, salary
FROM practice_02.employees
WHERE unit_of_work = 'Consulting 1'
ORDER BY salary DESC
LIMIT 5
```

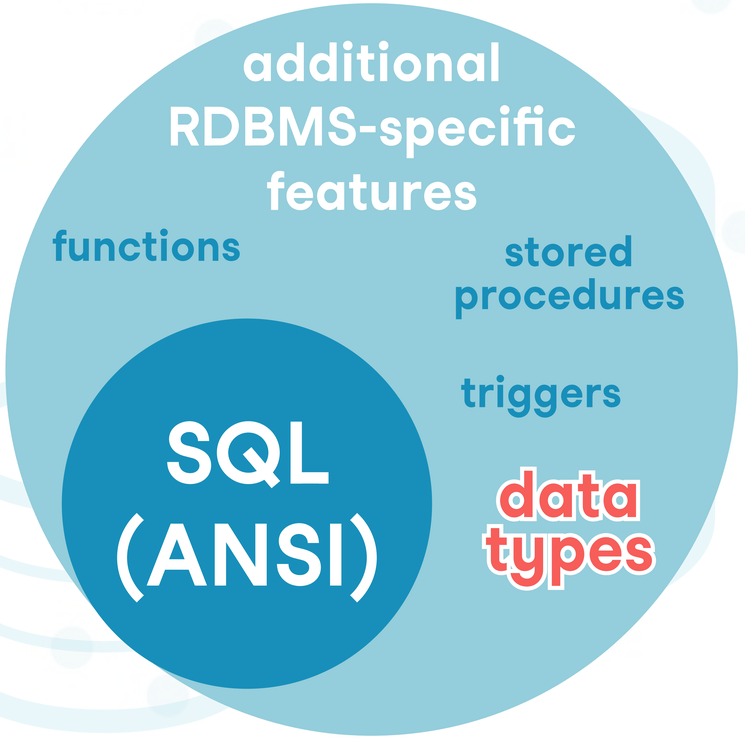
OUTPUT

first_name	last_name	unit_of_work	salary
Anikó	Tóth	Consulting 1	4900
Attila	Horváth	Consulting 1	4500
Márk	Horváth	Consulting 1	3200

5 rows

Data types (supported by PostgreSQL)

- **Boolean**
- **Character types** [such as char, varchar, and text]
- **Numeric types** [such as integer and floating-point number]
- **Temporal types** [such as **date, time, timestamp**, and **interval**]
- UUID [for storing UUID (Universally Unique Identifiers)]
- Array [for storing array strings, numbers, etc.]
- JSON [stores JSON data]
- hstore [stores key-value pair]
- Special Types [such as network address and geometric data]



Temporal data type (supported by PostgreSQL)

This data type is used to store date-time data. PostgreSQL has **5 temporal data type**:

DATE is used to store the dates only.

TIME is used to stores the time of day values.

TIMESTAMP is used to stores both date and time values.

TIMESTAMPTZ is used to store a timezone-aware timestamp data type.

INTERVAL is used to store periods of time.

Let's review these data types and become acquainted with the **functions that operate on them**.

The DATE data type

PostgreSQL offers the DATE data type for storing dates.
It occupies **4 bytes** of storage and accommodates dates ranging from 4713 BC to 5874897 AD.
PostgreSQL employs the **yyyy-mm-dd** format for both storing and inserting date values.

SQL

```sql
-- display the current date
SELECT CURRENT_DATE;
```

SQL

```sql
-- display the current datestyle
SHOW DATESTYLE;
```

For historical reasons, the **DATESTYLE** variable contains two independent components: **the output format** specification (ISO, Postgres, SQL, or German) and the **input/output** specification for year/month/day ordering (DMY, MDY, or YMD). These can be set separately or together.

Check the documentation at *https://www.postgresql.org/docs/16/datatype-datetime.html#DATATYPE-DATETIME-INPUT-DATES*

PostgreSQL DATE functions | AGE

AGE(timestamp,timestamp) → INTERVAL

The AGE() function accepts two TIMESTAMP values. It subtracts the second argument from the first one and returns an interval as a result.

SQL

```sql
-- display the first name, last name, hire date, and the number of years
-- elapsed since the hire date for each employee.
SELECT first_name, last_name, hire_date, AGE(current_date, hire_date)
FROM practice_02.employees;
```

OUTPUT

f...	la...	hire...	age
Zsófia	Szabó	2018-01-05	5 years 9 mons 13 days 0 hours 0 mins 0.0 s
Attila	Horváth	2022-08-08	1 years 2 mons 10 days 0 hours 0 mins 0.0 s
Bence	Nagy	2020-02-01	3 years 8 mons 17 days 0 hours 0 mins 0.0 s

37 rows

PostgreSQL DATE functions | EXTRACT

EXTRACT(field FROM source) → INTEGER

The **EXTRACT** function retrieves subfields such as year or hour from date/time values. **source** must be a value expression of type timestamp, time, or interval. (Expressions of type date are cast to timestamp and can therefore be used as well.) **field** is an identifier or string that selects what field to extract from the source value. The extract function returns values of type numeric.

Show the first name, last name, hire date, and the number of years that have passed since the hire date for every employee, both in a readable format and as an integer.

SQL

```sql
SELECT first_name, last_name, hire_date,
       AGE(current_date, hire_date) AS years_readable,
       EXTRACT( YEAR FROM AGE(current_date, hire_date)),
       EXTRACT( YEAR FROM current_date, hire_date),
FROM practice_02.employees;
```

PostgreSQL DATE functions | EXTRACT

Few more examples of using the **EXTRACT** function.

SQL

```sql
-- day of week based on ISO 8601 Monday (1) to Sunday (7)
SELECT EXTRACT(ISODOW FROM current_date);

-- extracting hour from a timestamp
SELECT EXTRACT(HOUR FROM TIMESTAMP '2016-12-31 13:30:15');

-- extracting the quarter from an interval
SELECT EXTRACT(QUARTER FROM INTERVAL '6 years 5 months 4 days 3 hours 2 minutes 1 second' );
```

1. *https://www.postgresql.org/docs/current/functions-datetime.html#FUNCTIONS-DATETIME-EXTRACT*
2. *https://www.postgresqltutorial.com/postgresql-date-functions/postgresql-extract/*

PostgreSQL essential date functions

In PostgreSQL, there are several important date functions that you can use to work with dates and times. Some of the most commonly used date functions include:

`CURRENT_DATE`: Returns the current date.

`CURRENT_TIME`: Returns the current time without the date portion.

`CURRENT_TIMESTAMP`: Returns the current date and time.

`DATE_TRUNC(field, source)`: Truncates a date or timestamp to a specified precision (e.g., truncating a timestamp to the nearest day).

`TO_DATE(text, format)`: Converts a text string into a date according to a specified format.

`DATE_PART(field, source)`: Similar to EXTRACT, it extracts specific components from a date or timestamp.

`DATE_ADD(source, interval)`: Adds an interval to a date or timestamp.

`DATE_SUB(source, interval)`: Subtracts an interval from a date or timestamp.

`NOW()`: Returns the current date and time (equivalent to CURRENT_TIMESTAMP).

`TO_TIMESTAMP(text, format)`: Converts a text string into a timestamp according to a specified format.

PostgreSQL date/time operators

`date + integer → date`

add a number of days to a date

`date - integer → date`

subtract a number of days from a date

`date - date → integer`

subtract dates, producing the number of days elapsed

`date + interval → timestamp`

add an interval to a date

`date + time → timestamp`

add a time-of-day to a date

`date - interval → timestamp`

subtract an interval from a date

https://www.postgresql.org/docs/current/functions-datetime.html#FUNCTIONS-DATETIME