MySQL DML. SELECT Constructions 1.

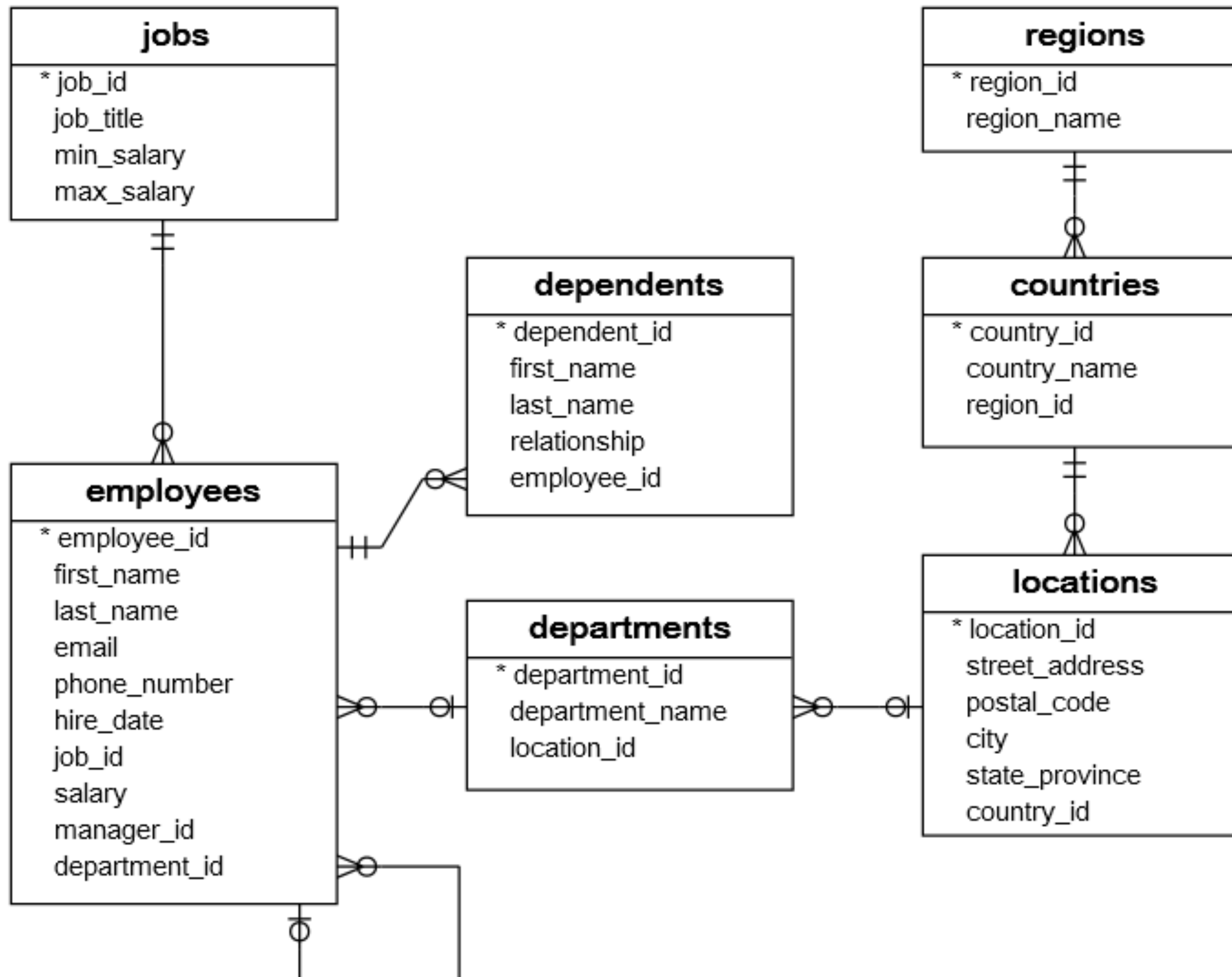
Summary

- 1. HR sample database
- 2. SELECT Statement Full Syntax
- 3. SELECT - Filtering Data
    - 3.1. DISTINCT - Remove Duplicates from the result set.
    - 3.2. LIMIT / OFFSET of returned rows
    - 3.3. WHERE Clause Constructions
        - 3.3.1. Comparison operators (=, !=, <>, <, >, <=, >=)
        - 3.3.2. SQL Triple Logic Overview
            - 3.3.2.1. IS NULL operator and the NULL concepts
            - 3.3.2.2. AND operator
            - 3.3.2.3. OR operator
            - 3.3.2.4. NOT operator
        - 3.3.3. BETWEEN operator
        - 3.3.4. IN operator
        - 3.3.5. LIKE operator
- 4. SELECT - Aliases
- 5. SELECT - Sorting Data
    - 5.1. ORDER BY Clause
- 6. SELECT - Grouping Data
    - 6.1. GROUP BY Clause
- 7. SELECT - Filtering Group
    - 7.1. HAVING Clause
- 8. SELECT - Joining Multiple Tables
    - 8.1. INNER JOIN
    - 8.2. LEFT [OUTER] JOIN
    - 8.3. RIGHT [OUTER] JOIN
    - 8.4. FULL [OUTER] JOIN
    - 8.5. [CROSS] JOIN
    - 8.6. Self Joins
- 9. SELECT UNION and UNION ALL
- 10. SELECT Subqueries

© Yuriy Shamshin

1. HR sample database.

You can use SQL Tutorial site https://www.sqltutorial.org/seeit/ for online testing examples and exercises on real DB.



© Yuriy Shamshin

2. SELECT Statement Full Syntax

To query data from a table, you use the SQL SELECT statement, where contains the syntax for selecting columns, selecting rows, grouping data, joining tables, and performing simple calculations.

```
-- Complete SELECT query
SELECT DISTINCT column, AGG_FUNC(column_or_expression), …
FROM mytable
    JOIN another_table
      ON mytable.column = another_table.column
    WHERE constraint_expression
    GROUP BY column
    HAVING constraint_expression
    ORDER BY column ASC/DESC
    LIMIT count OFFSET COUNT;
```

Each query begins with finding the data that we need in a database, and then filtering that data down into something that can be processed and understood as quickly as possible. Because each part of the query is executed sequentially, it's important to understand the order of execution so that you know what results are accessible where.

The SELECT statement is one of the most complex commands in SQL include many clauses:

- **SELECT** − This is one of the fundamental query command of SQL. It is similar to the projection operation of relational algebra. It selects the attributes based on the condition described by WHERE clause.
- **FROM** − This clause takes a relation name as an argument from which attributes are to be selected/projected. In case more than one relation names are given, this clause corresponds to Cartesian product.
- **JOIN** – for querying data from one, two or multiple related tables
- **WHERE** – This clause defines predicate or conditions for filtering data based on a specified condition.
- **GROUP BY** – for grouping data based on one or more columns
- **HAVING** – for filtering groups
- **ORDER BY** – for sorting the result set
- **LIMIT** – for limiting rows returned

You will learn about these clauses in the subsequent tutorials on Practice Works PW-01, PW-02, PW-03 and PW-04.

3. SELECT - Filtering Data

3.1. DISTINCT - Remove Duplicates from the result set.

The primary key ensures that the table has no duplicate rows. However, when you use the SELECT statement to query a portion of the columns in a table, you may get duplicates.

To remove duplicates from a result set, you use the DISTINCT operator in the SELECT clause as follows:

```
SELECT DISTINCT
  column1, column2, ...
FROM
  table1;
```

If you use one column after the DISTINCT operator, the database system uses that column to evaluate duplicate. **In case you use two or more columns, the database system will use the combination of value in these columns for the duplication check.**

To remove the duplicates, the database system first sorts the result set by every column specified in the SELECT clause. It then scans the table from top to bottom to identify the duplicates that are next to each other. In case the result set is large, the sorting and scanning operations may reduce the performance of the query.

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, salary)

```
SELECT
  salary
FROM
  employees
ORDER BY salary DESC;
```

salary
24000.00
17000.00
17000.00
14000.00
13500.00
13000.00
12000.00

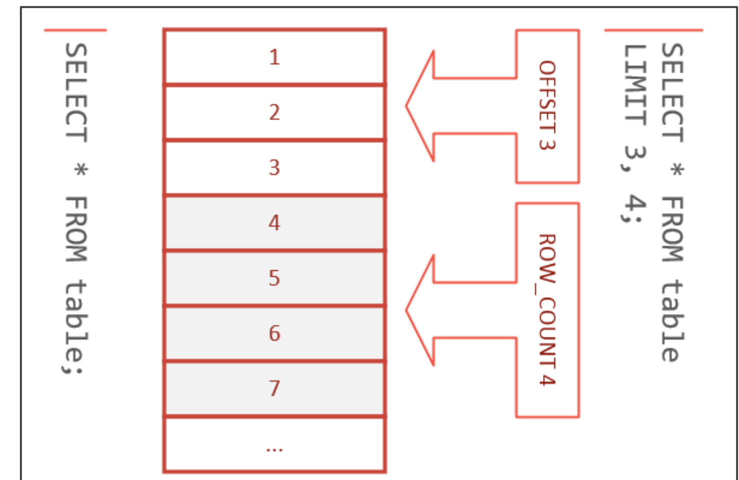
```
SELECT
  DISTINCT salary
FROM
  employees
ORDER BY salary DESC;
```

salary
24000.00
17000.00
14000.00
13500.00
13000.00
12000.00

3.2. LIMIT / OFFSET of returned rows

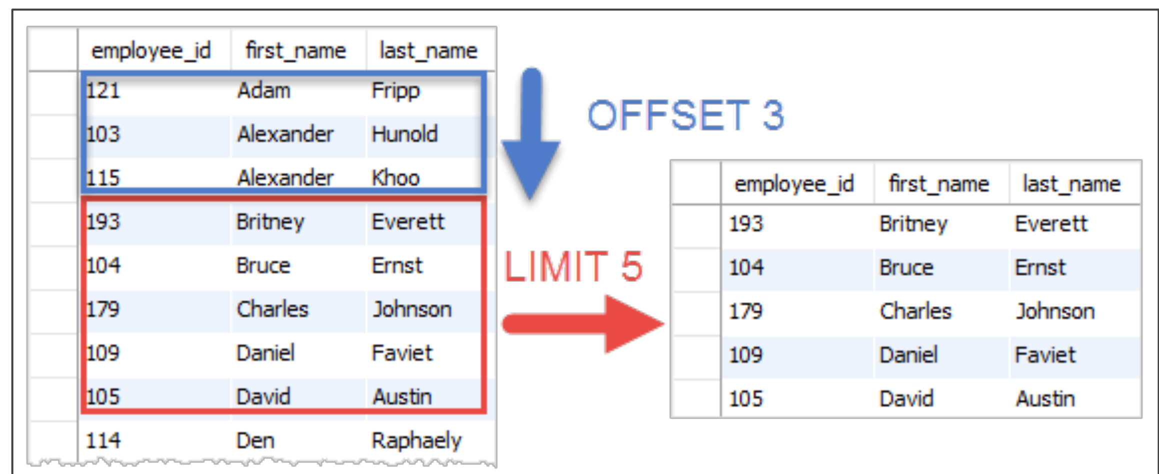
To retrieve a portion of rows returned by a query, you use the LIMIT and OFFSET clauses. The following illustrates the syntax of these clauses:

```
SELECT
  column_list
FROM
  table1
ORDER BY column_list
--or
LIMIT row_count;
--or
LIMIT row_count OFFSET offset;
--or
LIMIT offset, row_count;
-- row_count determines the number of rows that will be returned.
-- OFFSET clause skips the offset rows before return the rows.
```



Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(employee_id, first_name, last_name, salary)

```
SELECT
  employee_id, first_name, last_name
FROM
  employees
ORDER BY first_name
LIMIT 5 OFFSET 3;
--or
LIMIT 3, 5;
```



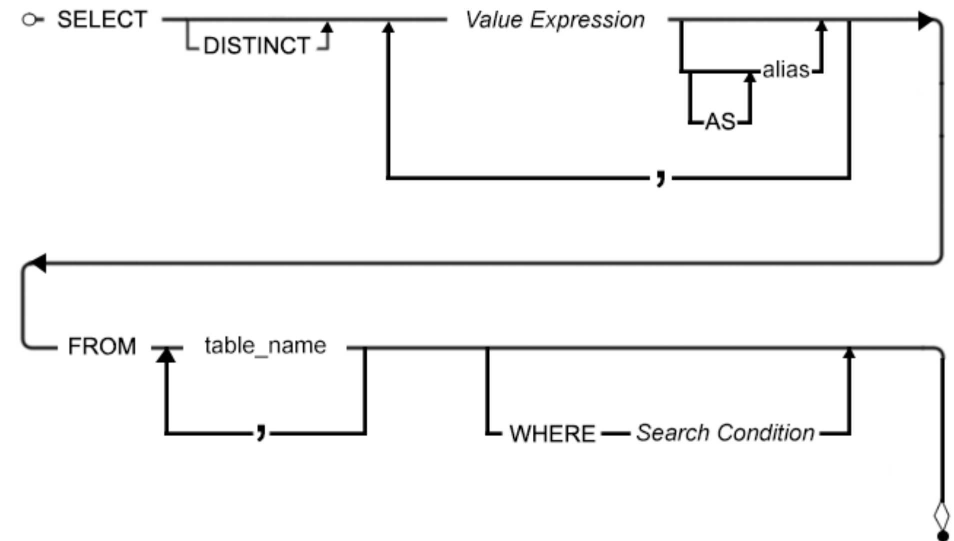
© Yuriy Shamshin

3.3. WHERE Clause to filter rows

See more on https://www.w3resource.com/sql/where-clause.php

- To select interesting rows from a table based on specified conditions, you use a WHERE clause in the SELECT statement.
- Besides the SELECT statement, the WHERE clause use in UPDATE or DELETE statement to specify which rows to be updated or deleted.
- The WHERE clause appears immediately **after** the FROM / SET clause.

```
SELECT
  column1, column2, ...
FROM table
WHERE
  condition;
```



- The WHERE clause contains one or more **logical expressions** (predicate) that evaluate each row in the table.
- SQL has **three-valued logic** which is TRUE, FALSE, and UNKNOWN (NULL).
- If a row that causes the condition evaluates to **true**, it will be included in the result set; otherwise (**false and unknown**), it will be excluded.
- You can use various **comparison and logical operators** to form logical expressions.

SQL Comparison Operators		
Operator	**Meaning**	**Syntax**
=	Equal	expr1 = expr2
<> (!=)	Not equal to	expr1 <> expr2
>	Greater than	expr1 > expr2
>=	Greater than or equal to	expr1 >= expr2
<	Less than	expr1 < expr2
<=	Less than or equal to	expr1 <= expr2

SQL Logical Operators		
Operator	**Meaning**	**Syntax**
IS NULL	Return true if the compared value is null	expr IS NULL
AND	Return true if both expressions are true	expr1 AND expr2
OR	Return true if either expression is true	expr1 OR expr2
NOT	Reverse the result of any other Boolean operator	NOT [Boolean_expression]
BETWEEN	Return true if the operand is within a range	expression BETWEEN low AND high
IN	Return true if operand is equal to one value in a list	expression IN (value1,value2,...)
LIKE	Return true if the operand matches a pattern	expression LIKE pattern
ANY	Return true if any one of the comparisons is true	Used for subquery
ALL	Return true if all comparisons are true	Used for subquery
EXISTS	Return true if a subquery contains any rows	Used for subquery
SOME	Return true if some of the expressions are true	expr1 SOME expr2

3.3.1. Comparison Operators (=, !=, <>, <, >, <=, >=).

Examples for DB: employees(employee_id, first_name, last_name, email, phone_number, hire_date, job_id, salary, manager_id, department_id)

To form a **simple expression** (my be include different SQL **functions**: length, sqrt, month…), you use one of the operators (>, <, =, …) above with two **operands** that can be either **column name** on one side and a **literal value** on the other, for example:

```
salary > 1000
```

It asks a question: "Is salary greater than 1000?".

Or you can use column names on **both sides** of an operator such as:

```
length(first_name) < length(last_name)
```

This expression asks another question: "Is the first_name length less than the last_name length?".

The **literal values** that you use in an expression can be numbers, characters, dates, and times, depending on the format you use:

- **Number**: use a number that can be an integer or a decimal without any formatting e.g., 100, 200.5
- **Character**: use characters surrounded by either single or double quotes e.g., "100", "John Doe".
- **Date**: use the format that the database stores. It depends on the DBMS e.g., MySQL uses 'yyyy-mm-dd' format to store the date data.
- **Time**: use the format that the database system uses to store the time. For example, MySQL uses 'HH:MM:SS' to store time data.

Finds employees who have the salaries greater than 14,000 (number literal) and sorts the result set based on the salary in descending order.

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE salary > 14000
ORDER BY salary DESC;
```

employee_id	first_name	last_name	salary
100	Steven	King	24000.00
101	Neena	Kochhar	17000.00
102	Lex	De Haan	17000.00

The following query finds all employees who work in the department id 5 (number literal).

```
SELECT employee_id, first_name, last_name, department_id
FROM   employees
WHERE  department_id = 5
ORDER BY first_name;
```

employee_id	first_name	last_name	department_id
121	Adam	Fripp	5
193	Britney	Everett	5
126	Irene	Mikkilineni	5

© Yuriy Shamshin

SQL is case-insensitive. However, when it comes to the values in the comparisons, it is case-sensitive. For instance, the following query finds the employees whose last name is Chen (char literal).

```
SELECT employee_id, first_name, last_name
FROM employees
WHERE last_name = 'Chen';
```

employee_id	first_name	last_name
110	John	Chen

To get all employees who joined the company after January 1st, 1999 (date literal), you use the following query:

```
SELECT employee_id, first_name, last_name, hire_date
FROM employees
WHERE
  hire_date >= '1999-01-01'
ORDER BY
  hire_date DESC;
```

employee_id	first_name	last_name	hire_date
179	Charles	Johnson	2000-01-04
113	Luis	Popp	1999-12-07
119	Karen	Colmenares	1999-08-10
178	Kimberely	Grant	1999-05-24
107	Diana	Lorentz	1999-02-07

Find the employees who joined the company in 1999 (number/char literal)

```
SELECT
  employee_id, first_name, last_name, hire_date
FROM employees
WHERE
  YEAR(hire_date) = 1999
-- strftime('%Y', hire_date) = '1999'
ORDER BY
  hire_date DESC;
```

employee_id	first_name	last_name	hire_date
113	Luis	Popp	1999-12-07
119	Karen	Colmenares	1999-08-10
178	Kimberely	Grant	1999-05-24
107	Diana	Lorentz	1999-02-07

You can **combine simple expressions** that use various comparison operators using the logical operator (AND, OR, NOT, …).

For example, the following statement finds employees in department 8 and have the salary greater than 10,000:

```
SELECT
  employee_id, first_name, last_name, salary
FROM employees
WHERE
  salary > 10000 AND department_id = 8
ORDER BY
  salary DESC;
```

3.3.2. SQL Triple Logic Overview.

3.3.2.1. IS [NOT] NULL operator and the NULL concepts.

NULL value is special in SQL, because any comparisons with a NULL can never result in true or false, but in a third logical result - **unknown**. NULL indicates that the data is: unknown, inapplicable or even does not exist. In other words, NULL represents that data is missing in the DB.

For example. If an employee does not have any phone number, you can store it as an empty string (‘’). However, if we don’t know his or her phone number at the time we insert the employee record, we will use the NULL value for the unknown phone numbers.

General properties of NULL:

```
NULL != NULL
```

Therefore you cannot use the comparison operator (=, !=, <>, <, <=, >, >=) to compare a value to a NULL value. For example, the following statement return not correct **empty** result.

Bad Expression with **NULL.**

```
-- Find all employees who do not have the phone numbers
SELECT employee_id, first_name, last_name, phone_number
FROM employees
WHERE phone_number = NULL;
```

Good Expression with **IS NULL**.

```
-- Find all employees who do not have the phone numbers
SELECT employee_id, first_name, last_name, phone_number
FROM employees
WHERE phone_number IS NULL;
```

Good Expression with **IS NOT NULL**.

```
-- Find all employees who have phone numbers
SELECT employee_id, first_name, last_name, phone_number
FROM employees
WHERE phone_number IS NOT NULL;
```

© Yuriy Shamshin

3.3.2.2. AND operator.

```
Logical_expression1 AND Logical_expression2
```

The following table illustrates the results of the AND operator when comparing true, false, and NULL values; and properties of AND:

x AND y		y		
		true	**false**	**null**
x	**true**	true	false	null
	false	false	false	false
	null	null	false	null

AND interesting properties	
commutativity	x AND y := y AND x
idempotency	x AND x := x
neutrality, here true is a neutral element	true AND y := y
absorption, here false an absorbing element	false AND y := false

- Notice that the AND operator returns true only if both expression is true.

SQL AND operator and short-circuit evaluation.

The short-circuit feature use **absorption properties** and allows the database system to stop evaluating the remaining parts of a logical expression as soon as it can determine the result.

Example to get a better understanding of how the to short-circuit evaluation feature works. See the following condition:

```
1=0 AND 1=1;
```

The database system processes the two comparisons first (1=0, 1=1) and uses the AND operator to evaluate the two results.

However, with the short-circuit evaluation feature, the database system just has to evaluate the left part of the expression, because the left part (1=0) returns false, that causes the whole **condition returns false regardless** of the result of the right part (1=1) of the condition.

The short-circuit feature, therefore, decreases the CPU computation time and in some cases helps prevent runtime-error. Consider the following condition:

```
1=0 AND 1/0;
```

If the DBMS supports the short-circuit feature, it will not evaluate the right part of the expression (1/0) that causes the division by zero error.

3.3.2.3. OR operator.

```
Logical_expression1 OR Logical_expression2
```

The following table illustrates the results of the OR operator when comparing true, false, and NULL values; and properties of OR:

x OR y		**y**				**OR interesting properties**	
		true	**false**	**null**		commutativity	x OR y := y OR x
x	**true**	true	true	true		idempotency	x OR x := x
	false	true	false	null		neutrality, here false is a neutral element	false OR y := y
	null	true	null	null		absorption, here true an absorbing element	true OR y := true

- Notice that the OR operator always returns true if either expression is true.

```
YEAR(hire_date) = 1997 OR YEAR(hire_date) = 1998
```

- If the database system use **absorption properties** and supports the short-circuit feature, the OR operator stops evaluating the remaining parts of the condition as soon as one expression is **true**.

- When you use the OR operator with the AND operator, the database system evaluates the OR operator after the AND operator.
- This is known as the rule of precedence. However, you can use parentheses to change the order of evaluation.

```
(expr1 OR expr2 AND expr3)
-- is equivalent of
expr1 OR (expr2 AND expr3))
```

```
department_id = 3 AND (YEAR(hire_date) = 1997 OR YEAR(hire_date) = 1998)
-- is not equivalent of
department_id = 3 AND  YEAR(hire_date) = 1997 OR YEAR(hire_date) = 1998
```

```
YEAR(hire_date) = 2000 OR YEAR(hire_date) = 1999 OR YEAR(hire_date) = 1990;
-- is equivalent of
YEAR(hire_date) IN(1990, 1999, 2000)
```

3.3.2.4. NOT operator.

To reverse the result of any Boolean expression, you use the NOT operator.

NOT use with various logical operators such as AND, OR, LIKE, BETWEEN, IN, EXISTS and with NULL value.

```
NOT [Logical_expression]
NOT NULL
```

The following table illustrates the results of the NOT operator when comparing true, false, and NULL values; and properties of NOT:

x	**NOT**		**OR interesting properties**	
true	false		The law of double negation	NOT NOT x := x
false	true			
null	null		absorption, here null an absorbing element	NOT null := null

- When you use the NOT operator with OR, AND operator, the database system evaluates NOT operator before the OR, AND operator.

Examples

```
department_id = 5 AND NOT salary > 5000
```

```
department_id NOT IN (1, 2, 3)
```

```
first_name NOT LIKE 'D%'
```

```
salary NOT BETWEEN 3000 AND 5000
```

```
-- Find all employees who have phone numbers
SELECT employee_id, first_name, last_name, phone_number
FROM employees
WHERE phone_number IS NOT NULL;
```

© Yuriy Shamshin

3.3.2.5. Examples for Logical Operators.

A logical operator allows you to test for the truth of a condition. Similar to a comparison operator, a logical operator returns a value of true, false, or unknown (null).

Examples for DB: employees(employee_id, first_name, last_name, email, phone_number, hire_date, job_id, salary, manager_id, department_id)

IS NULL - returns true if the compared value is null

The following statement finds all employees who do not have a phone number

```
SELECT first_name, last_name, phone_number
FROM employees
WHERE
  phone_number IS NULL
ORDER BY
  first_name, last_name;
```

first_name	last_name	phone_number
Charles	Johnson	NULL
Jack	Livingston	NULL
John	Russell	NULL
Jonathon	Taylor	NULL
Karen	Partners	NULL

AND – combine multiple Boolean expressions

The following statement finds all employees whose salaries are greater than 5,000 and less than 7,000

```
SELECT first_name, last_name, salary
FROM
  employees
WHERE
  salary > 5000 AND salary < 7000
ORDER BY
  salary;
```

first_name	last_name	salary
Bruce	Ernst	6000.00
Pat	Fay	6000.00
Charles	Johnson	6200.00
Shanta	Vollman	6500.00
Susan	Mavris	6500.00

OR – combine multiple Boolean expressions

The following statement finds employees whose salary is either 7,000 or 8,000

```
SELECT first_name, last_name, salary
FROM employees
WHERE salary = 7000 OR salary = 8000
ORDER BY salary;
```

first_name	last_name	salary
Kimberely	Grant	7000.00
Matthew	Weiss	8000.00

3.3.3. BETWEEN operator.

The BETWEEN operator select data within a range of values, given the minimum value and maximum value.

```
test_expr BETWEEN low_expr AND high_expr
```

In this syntax:

- test_exp is the expression to test for in the range defined by low and high.
- low_expr and high_expr can be either expressions or literal values with a requirement that the value of low is less than value of high.
- if the low value is greater than the high value, you will get an empty result set.
- between is equivalent to the following condition:

```
test_expr >= low_expr AND test_expr <= high_expr
```

- To negate the result of the BETWEEN operator, you add the NOT operator:

```
test_expr NOT BETWEEN low AND high
```

- The NOT BETWEEN is equivalent to the following condition:

```
test_expr < low OR test_expr > high
```

For example, the following statement finds all employees whose salaries are between 9,000 and 12,000.

```
SELECT
    first_name, last_name, salary
FROM
    employees
WHERE
    salary BETWEEN 9000 AND 12000
ORDER BY salary;
```

first_name	last_name	salary
Alexander	Hunold	9000.00
Daniel	Faviet	9000.00
Hermann	Baer	10000.00
Den	Raphaely	11000.00
Nancy	Greenberg	12000.00
Shelley	Higgins	12000.00

Notice that the value 9,000 and 12,000 are included in the output.

Using the BETWEEN operator with the DATETIME data.

Simple Examples.

To find all employees who joined the company between January 1, 2019, and December 31, 2000

```
hire_date BETWEEN '2019-01-01' AND '2020-12-31'
```

To retrieve all employees who have not joined the company from January 1, 2009 to December 31, 2019

```
hire_date NOT BETWEEN '2009-01-01' AND '2019-12-31'
```

Where is my rows?

Consider the following t1 table:

	id	created_at
	1	2016-05-31 23:59:59
	2	2016-06-29 15:32:00
	3	2016-06-29 23:59:59
	4	2016-06-30 00:00:00
	5	2016-06-30 23:59:59
	6	2016-07-01 00:00:00

There are **four rows** created between June 29, 2016 and June 30, 2016.

If you use the BETWEEN operator to query the rows whose created_at values are between June 29, 2016, and June 30, 2016, you will get what you may expect.

```
SELECT id, created_at FROM t1
WHERE created_at BETWEEN
'20160629' AND '20160630';
```

	id	created_at
	2	2016-06-29 15:32:00
	3	2016-06-29 23:59:59
	4	2016-06-30 00:00:00

The result shows only **three rows** returned.

This is because when you used the following condition:

```
WHERE created_at BETWEEN
'20160629' AND '20160630'
```

The database system translates it into something like:

```
WHERE created_at BETWEEN
'20160629 00:00:00.000000' AND
'20160630 00:00:00.000000'
```

Therefore the row with value 2016-06-30 23:59:59 was **not included** in the result set.

Using the Between operator with a STRING data.

```
SELECT *
FROM product
WHERE ProductCode BETWEEN 'A' AND 'C'
ORDER BY ProductCode;
```

3.3.4. IN operator.

- The IN operator check whether a value is in the list of specified values.
- The IN operator is often used in the WHERE clause of the SELECT, UPDATE, and DELETE statements. It is also used in subqueries.
- The IN operator returns true if the compared value matches at least one value in the list; otherwise, it returns false.

```
expression IN (value1,value2,...)
```

- Notice that if any value in the list (value1,value2,...) is null, the IN operator returns no rows.
- The condition that uses the IN operator can be rewritten using one or more OR operators as follows:

```
expression = value1 OR expression = value2 OR ...
```

- To negate the result of the IN operator, you use the NOT operator:

```
expression NOT IN (value1, value2,...)
```

- The NOT IN operator returns true if the expression does not match any value in the list, otherwise, it returns false.
- Similarly, you can rewrite the NOT IN operator by using the AND operators as shown below:

```
expression != value1 AND expression != value2 AND...
```

Example. The following statement finds all employees who work in the department id 8 or 9.

```
SELECT
  first_name, last_name, department_id
FROM
  employees
WHERE
  department_id IN (8, 9)
ORDER BY
  department_id;
```

first_name	last_name	department_id
John	Russell	8
Karen	Partners	8
Jonathon	Taylor	8
Jack	Livingston	8
Kimberely	Grant	8
Charles	Johnson	8
Steven	King	9
Neena	Kochhar	9
Lex	De Haan	9

3.3.5. LIKE operator.

LIKE - query data based on a specified pattern. See the following syntax:

```
expression LIKE pattern
```

- The LIKE operator compares a value to similar values using a **pattern**.
- The LIKE operator is often used in the WHERE clause of the SELECT, UPDATE, and DELETE statements.
- SQL provides two wildcards used in conjunction with the LIKE operator:
    - The percent sign **(%)** represents **zero, one, or multiple characters**.
    - The underscore sign **(_)** represents a **single character**.
- If you want to match the wildcards % or _, you must use the backslash character **(\)** to escape it.
- In case you want to use a different escape character rather than the (\), you use **ESCAPE** clause in the LIKE expression as follows:

```
expression LIKE pattern ESCAPE escape_character
```

- To negate the result of the LIKE operator, you use the **NOT** operator as follows:

```
expression NOT LIKE pattern ESCAPE escape_character
```

Examples.

To find all employees whose first name starts with the string jo:

```
SELECT employee_id, first_name, last_name
FROM employees
WHERE
  first_name LIKE 'jo%'
ORDER BY first_name;
```

employee_id	first_name	last_name
110	John	Chen
145	John	Russell
176	Jonathon	Taylor
112	Jose Manuel	Urman

To find all employees with the first names whose the second character is h:

```
SELECT employee_id, first_name, last_name
FROM employees
WHERE
  first_name LIKE '_h%'
ORDER BY first_name;
```

employee_id	first_name	last_name
179	Charles	Johnson
123	Shanta	Vollman
205	Shelley	Higgins
116	Shelli	Baida

To find all employees whose first names begin with S but not begin with Sh:

```
SELECT employee_id, first_name, last_name
FROM employees
WHERE
    first_name LIKE 'S%'
AND first_name NOT LIKE 'Sh%'
ORDER BY first_name;
```

employee_id	first_name	last_name
192	Sarah	Bell
117	Sigal	Tobias
100	Steven	King
203	Susan	Mavris

The following table illustrates some patterns and their meanings:

Expression	Meaning
LIKE ‘Databases'	Is value Databases only
LIKE 'Kim%'	Begins with Kim
LIKE '%er'	Ends with er
LIKE '%ch%'	Contains ch
LIKE 'Le_'	Begins with Le and is followed by at most one character e.g., Les, Len…
LIKE '_uy'	Ends with uy and is preceded by at most one character e.g., guy
LIKE '%are_'	Contains are, begins with any number of characters and ends with at most one character
LIKE '_are%'	Contains are, begins with at most one character and ends with any number of characters