SQL Miscellaneous Topics

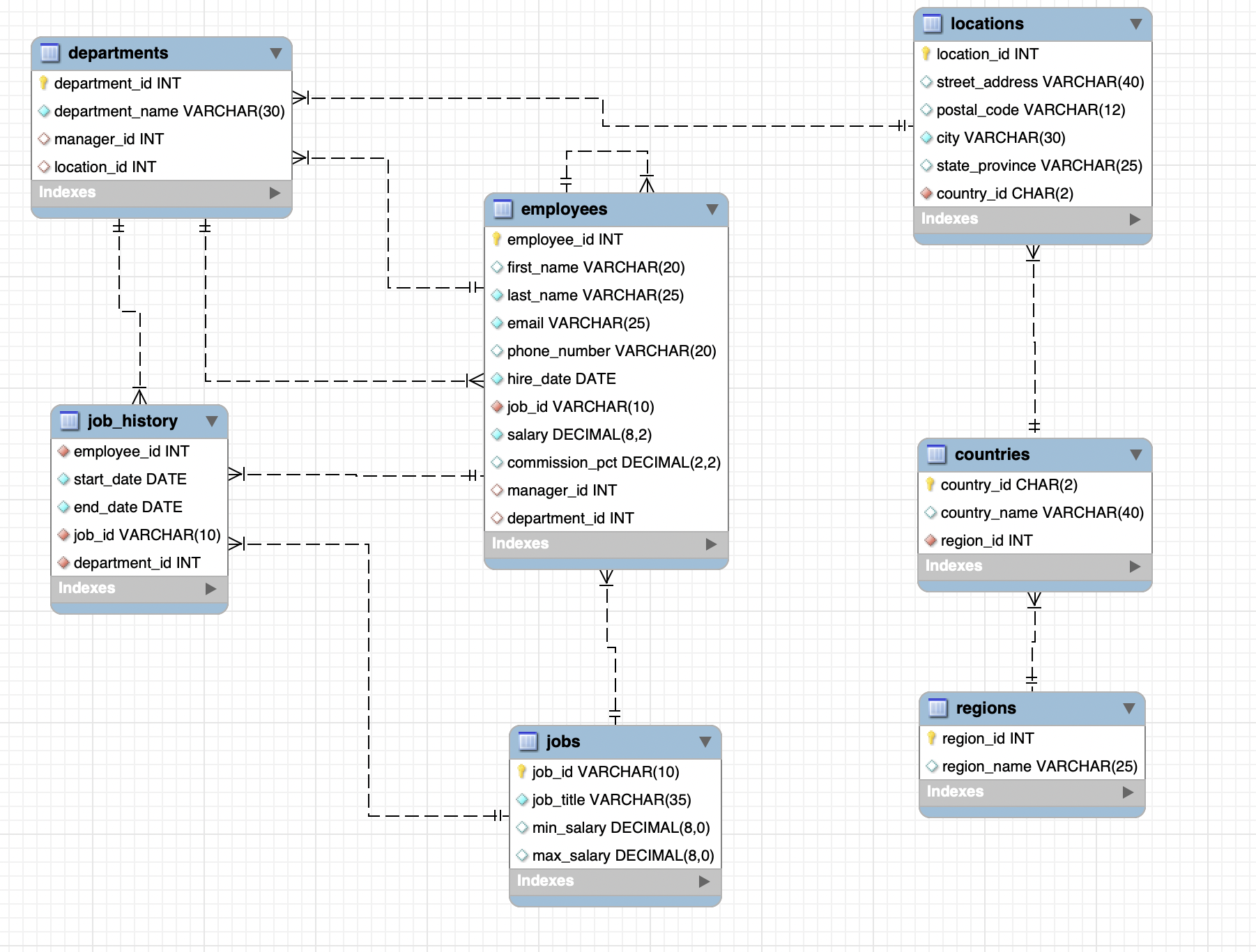
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### **Agenda (for the instructor only)**

1. Correlated Subqueries
2. EXISTS keyword
3. ANY and ALL operators
4. Fetch & delete duplicate records
5. COALESCE()
6. GROUP\_CONCAT()

### **Dataset:** [**link**](https://drive.google.com/file/d/1SKTg9dwNZFqbB1rWptDs-rcvmwm8AJ-S/view?usp=drive_link)

### **ER Diagram**

****

**Important:**

* Queries demonstrated in this lecture have been tested on **MySQL Workbench**.
* Please note that these queries may not work on platforms like Google BigQuery.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Correlated (or Synchronized) Subqueries**

**Illustration:**

* Use the **Macros** created in this [**GSheet**](https://docs.google.com/spreadsheets/d/1a2wQJ0vh4hNY3i6q1SgJ02Da7RycsXilMmDm30PDkcU/edit#gid=1087140509) to demonstrate how a **Correlated Subquery** works.
* Make sure you **Reset the resultant table** once done with the demonstration.

## **Question:** Retrieve the details of employees whose salary is higher than the average salary of employees in their respective departments.

Let’s first check the average salary for each department -

**Query:**

SELECT

department\_id,

AVG(salary) AS avg\_dept\_sal

FROM hr.employees

GROUP BY department\_id;

**Soln using JOIN :**

SELECT

e1.last\_name,

e1.first\_name,

e1.department\_id,

e1.salary

FROM hr.employees e1

JOIN

(SELECT department\_id, AVG(salary) AS avg\_salary

FROM hr.employees

GROUP BY department\_id) e2

ON e1.department\_id = e2.department\_id

WHERE

e1.salary > e2.avg\_salary;

**Correlated Subquery:** It gets its name because the two queries are related; the inner query uses information obtained from the outer query.

* We need a temporary table to compute the average salary for each department.
* Additionally, we ensure that the department of employees matches the department for which we are calculating the average salary.

**Query:**

SELECT

e1.last\_name,

e1.first\_name,

e1.department\_id,

e1.salary

FROM hr.employees e1

WHERE e1.salary >

(SELECT AVG(salary)

FROM hr.employees e2

WHERE e2.department\_id = e1.department\_id);

**In this query:**

* The WHERE clause filters the rows based on the condition that the salary of each employee (e1.salary) is greater than the average salary of employees in the same department.
* The Subquery calculates the average salary for each department in the outer query.
  + It is correlated to the outer query by matching the department IDs (e1.department\_id).
* The result of the subquery is used as a condition in the WHERE clause to compare with the salary of each employee in the outer query.

The main difference between a SQL correlated subquery and a simple subquery is that **correlated subqueries** **reference columns from the outer table**. In the above example, **e1.dept\_id** is a reference to the outer subquery table.

To identify a correlated query, just look for these kinds of references. If you find at least one, you have a SQL correlated subquery!

| Simple Subquery | Correlated Subquery |
| --- | --- |
| Can only reference columns from the inner query. | Can reference columns from the outer query as well. |
| Independent of the outer query, executed only once before the outer query runs. | Dependent on the outer query, executed once for each row of the outer query. |

**Question:** Retrieve the details of employees whose commission is lower than the average commission of employees in the same job category (job\_id).

* Similar to the question above, we need a temporary table to compute the average commission percentage for each job category.
* Additionally, we ensure that the job id of employees matches the job ID for which we are calculating the average commission percentage.

**Query:**

SELECT

e1.first\_name,

e1.last\_name,

e1.job\_id,

e1.commission\_pct

FROM hr.employees e1

WHERE e1.commission\_pct <

(SELECT AVG(e2.commission\_pct)

FROM hr.employees e2

WHERE e1.job\_id = e2.job\_id);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **EXISTS keyword**

**Illustration:**

* Use the **Macros** created in this [**GSheet**](https://docs.google.com/spreadsheets/d/1a2wQJ0vh4hNY3i6q1SgJ02Da7RycsXilMmDm30PDkcU/edit#gid=495197474) to demonstrate how **EXISTS** works.
* Make sure you **Reset the resultant table** once done with the demonstration.

# **Question:** Retrieve the details of employees who have at least one employee reporting to them.

**Query:**

SELECT

employee\_id, first\_name, last\_name,

salary, department\_id

FROM hr.employees e

WHERE EXISTS (

SELECT 1

FROM hr. employees

WHERE manager\_id = e.employee\_id

);

**In this query:**

* The WHERE clause filters the rows based on a condition using the EXISTS keyword and a Correlated Subquery.
* The Subquery checks if there exists at least one row in the hr.employees table where the manager\_id column matches the employee\_id of the current row in the outer query (e.employee\_id).
* The condition manager\_id = e.employee\_id ensures that the subquery is correlated to the outer query by matching the manager\_id column in the subquery's table (hr.employees) with the employee\_id column in the outer query's table (e).

**Note:**

* The SELECT 1 in the subquery is just a placeholder value, used to indicate the existence of any row in the subquery result set.
* The actual values returned by the subquery aren't needed for the EXISTS condition to be satisfied. The primary purpose is to check for the presence of at least one row.
* Even if we replace the 1 with any other column, the subquery will still function the same way. It will return a row whenever there is at least one employee with the manager\_id equal to e.employee\_id.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Quiz:** What does the following SQL query achieve?

**Query:**

SELECT department\_name

FROM demo.departments d

WHERE EXISTS (

SELECT 1

FROM demo.employees e

WHERE e.employee\_id = d.manager\_id

AND e.salary <= (

SELECT AVG(salary)

FROM demo.employees e2

WHERE e2.department\_id = d.department\_id

)

);

**Choices:**

1. Identifies the departments where the manager is earning less compared to the average earning of an employee from that department.
2. Identifies the departments where the manager's salary is less than or equal to the average salary of employees in that department.
3. Identifies the departments where the average salary of an employee is less than or equal to the manager's salary within that department.
4. Identifies the departments where the manager's salary is greater than or equal to the average salary of employees in that department.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**ANY and ALL operators**

These operators are used in combination with comparison operators to compare a value with a set of values returned by a subquery.

* The ANY operator returns true if the comparison holds true for **at least one value** in the set.
* The ALL operator returns true if the comparison holds true for **all values** in the set.

# **Question:** Retrieve the details of employees who get a commission equal to the commission percentage of any manager level employee from the Sales department.

**Query with JOIN :**

SELECT

DISTINCT e.employee\_id, e.first\_name, e.last\_name, e.commission\_pct

FROM demo.employees e

JOIN demo.employees m ON e.commission\_pct = m.commission\_pct

WHERE

m.department\_id = 80

AND m.job\_id LIKE '%MAN%';

**Query with ANY :**

SELECT

e.employee\_id, e.first\_name, e.last\_name, e.commission\_pct

FROM demo.employees e

WHERE e.commission\_pct = ANY (

SELECT m.commission\_pct

FROM demo.employees m

WHERE

m.department\_id = 80

AND m.job\_id LIKE '%MAN%'

);

**In this query:**

* The WHERE clause filters rows by comparing each employee's commission\_pct with the subquery result.
* The subquery selects the commission\_pct of employees who are in department 80 and have 'MAN' in their job\_id.
* The ANY keyword compares each employee's commission\_pct with any commission\_pct value returned by the subquery.
* It returns true if an employee's commission\_pct equals any commission\_pct value returned by the subquery.

# **Question:** Retrieve the details of employees who have a higher salary than all employees in both of these departments - Human Resource and Public Relations.

**Query with JOIN :**

SELECT

e1.employee\_id, e1.first\_name, e1.last\_name, e1.salary, e1.department\_id

FROM hr.employees e1

JOIN (

SELECT

MAX(salary) AS max\_salary

FROM hr.employees

WHERE department\_id IN (40, 70)

) e2 ON e1.salary > e2.max\_salary;

**Query with ALL :**

SELECT

employee\_id, first\_name, last\_name, salary, department\_id

FROM hr.employees

WHERE salary > ALL (

SELECT salary

FROM hr.employees e

JOIN hr.departments d

ON e.department\_id = d.department\_id

WHERE department\_name IN ('Human Resource', 'Public Relations')

);

**In this query:**

* The WHERE clause filters rows by comparing each employee's salary with the subquery result.
* The subquery selects the salary of employees who belong to departments Human Resource (40) or Public Relations (70).
* The ALL keyword compares each employee's salary with all the salary values returned by the subquery.
* It returns true if an employee's salary is greater than all the salaries returned by the subquery.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Fetching duplicate records**

# **Important:**

As our dataset currently lacks any duplicate records, we will create a new table `**job\_hist**`and intentionally add some duplicates for the purpose of demonstration.

**Query:**

CREATE TABLE job\_hist (

employee\_id INT,

start\_date DATE,

end\_date DATE,

job\_id VARCHAR(20),

department\_id INT

);

INSERT INTO job\_hist (employee\_id, start\_date, end\_date, job\_id, department\_id) VALUES

(101, '1989-09-21', '1993-10-27', 'AC\_ACCOUNT', 110),

(101, '1993-10-28', '1997-03-15', 'AC\_MGR', 110),

(101, '1993-10-28', '1997-03-15', 'AC\_MGR', 110),

(102, '1993-01-13', '1998-07-24', 'IT\_PROG', 60),

(114, '1998-03-24', '1999-12-31', 'ST\_CLERK', 50),

(122, '1999-01-01', '1999-12-31', 'ST\_CLERK', 50),

(122, '1999-01-01', '1999-12-31', 'ST\_CLERK', 50),

(122, '1999-01-01', '1999-12-31', 'ST\_CLERK', 50),

(122, '1999-01-01', '1999-12-31', 'ST\_CLERK', 50),

(122, '1999-01-01', '1999-12-31', 'ST\_CLERK', 50),

(176, '1998-03-24', '1998-12-31', 'SA\_REP', 80),

(176, '1999-01-01', '1999-12-31', 'SA\_MAN', 80),

(200, '1987-09-17', '1993-06-17', 'AD\_ASST', 90),

(200, '1987-09-17', '1993-06-17', 'AD\_ASST', 90),

(200, '1987-09-17', '1993-06-17', 'AD\_ASST', 90),

(200, '1994-07-01', '1998-12-31', 'AC\_ACCOUNT', 90),

(201, '1996-02-27', '1999-12-19', 'MK\_REP', 20);

# **Question:** How would you identify if there are any duplicate records present in a table?

**Query:** This query retrieves records from our table that occur more than once, along with their respective occurrence count.

-- Create a CTE to identify duplicate records

WITH cte AS (

SELECT

employee\_id,

start\_date,

end\_date,

job\_id,

department\_id,

COUNT(\*) AS dup\_cnt

FROM hr.job\_hist

GROUP BY

employee\_id,

start\_date,

end\_date,

job\_id,

department\_id

HAVING dup\_cnt > 1

)

-- Select the records identified as duplicates from the CTE

SELECT \*

FROM cte;

**In this query:**

* The CTE named "cte" is defined using the WITH clause.
* It consists of a SELECT statement that retrieves data from the hr.employees table.
* The SELECT statement within the CTE aggregates the data by all employee attributes.
* It counts the number of occurrences for each combination of these attributes using COUNT(\*).
* The outer SELECT statement retrieves data from the CTE.
* It selects all duplicate records along with the count of their occurrences (stored in the "dup\_cnt" column).

# **Question:** How would you fetch all duplicate records (if any) present in a table?

**Query:** This query retrieves all duplicate records existing in our table, excluding the initial occurrence of each duplicate entry.

-- Define a CTE to assign row numbers to each record

WITH cte AS (

SELECT

employee\_id,

start\_date,

end\_date,

job\_id,

department\_id,

ROW\_NUMBER() OVER (

PARTITION BY

employee\_id,

start\_date,

end\_date,

job\_id,

department\_id

) AS row\_num

FROM hr.job\_hist

)

-- Filter for records with row numbers greater than 1

SELECT \*

FROM cte

WHERE row\_num > 1

ORDER BY employee\_id;

**In this query:**

* TheCTE named "cte" is defined using the WITH clause.
* It consists of a SELECT statement that retrieves data from the employee's table.
* The SELECT statement assigns a unique row number (row\_num) to each row within groups of duplicate records based on ALL employee attributes.
* The main SELECT statement selects all columns from the CTE.
* It retrieves only those rows from the CTE where the row number (row\_num) is greater than 1, indicating duplicate records.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Deleting duplicate records**

# **Question:** How can we get rid of these duplicate records?

1. **Using DISTINCT keyword**

CREATE TABLE new\_job\_hist AS

SELECT DISTINCT

employee\_id,

start\_date,

end\_date,

job\_id,

department\_id

FROM hr.job\_hist;

1. **Using temporary table**

SET SQL\_SAFE\_UPDATES = 0;

CREATE TABLE hr.temp AS

SELECT DISTINCT \* FROM hr.job\_hist;

DELETE FROM hr.job\_hist;

INSERT INTO hr.job\_hist (SELECT \* FROM temp);

DROP TABLE hr.temp;

1. **Using ROW\_NUMBER()**

CREATE TABLE new\_job\_hist AS

SELECT \*

FROM (

SELECT \*,

ROW\_NUMBER() OVER (PARTITION BY employee\_id, job\_id, department\_id ORDER BY employee\_id) AS rn

FROM demo.job\_hist

) AS subquery

WHERE rn = 1;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **GROUP\_CONCAT()**

* It is a string aggregation function available in many RDBMS such as **MySQL**, **SQLite**, etc.
* It is also available in **BigQuery** but with a different name - **STRING\_AGG()**
* It allows you to **concatenate values from multiple rows into a single string** based on a specified delimiter.
* It is particularly useful when you need to combine multiple values from grouped rows into a single, comma-separated list.

**Syntax:**

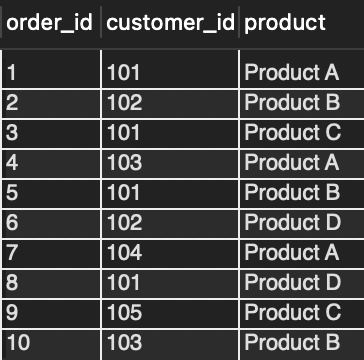
GROUP\_CONCAT(column\_name [ORDER BY expression] [SEPARATOR separator])

**Parameters:**

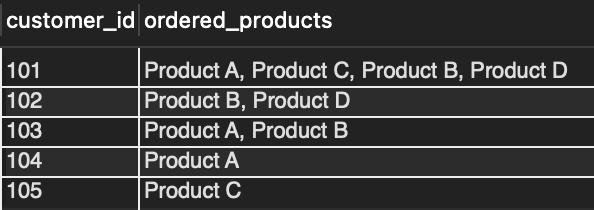
* column\_name: The column whose values you want to concatenate.
* ORDER BY expression (optional): Specifies the order of values within the concatenated string.
* SEPARATOR separator (optional): Specifies the delimiter used to separate concatenated values. **The default separator is a comma (,)**.

**Illustration:**

Input ->



Output ->



# **Question:** Let's say we want to concatenate the first\_name of employees within each department\_id into a single string separated by commas.

**Query:**

SELECT

department\_id,

GROUP\_CONCAT(first\_name ORDER BY first\_name SEPARATOR ' , ') AS employees\_list

FROM hr.employees

GROUP BY department\_id;

**In this query:**

* We group the rows by department\_id.
* For each group, GROUP\_CONCAT concatenates the first\_name values of employees into a single string separated by commas.
* The result is a list of first\_name values for each department\_id **sorted lexicographically**.

**Note:**

* Be cautious when using GROUP\_CONCAT with **large datasets**, as it can potentially result in large concatenated strings and **impact performance**.
* The maximum length of the concatenated string may be limited by the RDBMS configuration, so be mindful of **potential truncation issues**.

# **Question:**

* We want two new columns with concatenated email IDs and phone numbers of employees within each city.
* The concatenated strings should be separated by commas, ensuring that the order of email IDs and phone numbers aligns with the employee IDs.

**Query:**

SELECT

l.city,

GROUP\_CONCAT(e.email ORDER BY e.employee\_id ASC SEPARATOR ', ') AS concatenated\_emails,

GROUP\_CONCAT(e.phone\_number ORDER BY e.employee\_id ASC SEPARATOR ', ') AS concatenated\_phone\_numbers

FROM

hr.employees e

JOIN

hr.departments d ON e.department\_id = d.department\_id

JOIN

hr.locations l ON d.location\_id = l.location\_id

GROUP BY l.city;

# **Question:**

* We want two new columns with concatenated employee names and hire dates of employees from each department.
* The concatenated strings should be separated by commas, ensuring that the order of employee names and phone numbers should be aligned together.

**Query:**

SELECT

d.department\_name,

GROUP\_CONCAT(CONCAT(first\_name, ' ', last\_name) ORDER BY e.employee\_id ASC SEPARATOR ', ') AS employee\_names,

GROUP\_CONCAT(e.hire\_date ORDER BY e.employee\_id ASC SEPARATOR ', ') AS hire\_dates

FROM

hr.employees e

JOIN

hr.departments d ON e.department\_id = d.department\_id

GROUP BY d.department\_name;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **COALESCE()**

* It evaluates a list of expressions in the order they are provided as arguments.
* And returns the first non-NULL value from that list of expressions.
* If all expressions are NULL, COALESCE() returns NULL.
* You can also specify a default value to be returned if all expressions are NULL.

**Syntax:** COALESCE(expression1, expression2, ...)

# **Question:** Let's say we want to retrieve the name and salary of employees, but if the salary is NULL, we want to display a default value of 0.

**Query:**

SELECT

first\_name, last\_name,

COALESCE(salary, 0) AS salary

FROM hr.employees;

**In this query:**

* COALESCE() is used to replace NULL values in the salary column with the default value of 0.
* If the salary is not NULL, it is returned as is. If it is NULL, it is replaced with 0.

Please note that this question can be solved using the **IFNULL()** function as well which was covered in the earlier lectures.

**Query:**

SELECT

first\_name, last\_name,

IFNULL(salary, 0) AS salary

FROM hr.employees;

Then why do we need the COALESCE() function?

**Question:** Retrieve the names of all employees from our database according to the following guidelines -

* If the employee's first name is provided, retrieve it.
* If the employee's first name is not available, retrieve their last name.
* If neither the employee's first name nor their last name is specified, set the default value to "UNKNOWN".

**Query:**

SELECT

COALESCE(first\_name, last\_name, 'UNKNOWN') AS employee\_name

FROM

hr.employees;

**In this query:**

* COALESCE() evaluates the expressions in the order they are provided.
* If first\_name is not NULL, it is returned as the employee's name.
* If the first\_name is NULL, the function returns the last\_name.
* If both first\_name and last\_name are NULL, the function returns "UNKNOWN" as the default value for the employee's name.

**Question:** Retrieve the location details of all employees based on these guidelines -

* If an employee's city is provided, we will include this information.
* If the employee's city is unavailable, we will include their state or province.
* If neither the employee's city nor their state/province is specified, we will display their country name.
* If no location details are available, we will display "-" to signify the absence of data.

**Query:**

SELECT

e.employee\_id,

e.first\_name,

e.last\_name,

COALESCE(l.city, l.state\_province, c.country\_name, "-") AS location

FROM

hr.employees e

LEFT JOIN

hr.departments d ON e.department\_id = d.department\_id

LEFT JOIN

hr.locations l ON d.location\_id = l.location\_id

LEFT JOIN

hr.countries c ON l.country\_id = c.country\_id;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Question:** List departments where the average salary is greater than 8,000 and there are at least two employees earning more than 5,000.

**Query:**

SELECT department\_name

FROM demo.departments d

WHERE EXISTS (

SELECT 1

FROM demo.employees e

WHERE e.department\_id = d.department\_id

AND salary >= 5000

HAVING COUNT(\*) >= 2

) AND EXISTS (

SELECT 1

FROM demo.employees e2

WHERE e2.department\_id = d.department\_id

HAVING AVG(salary) >= 8000

);

**Question:** Identify employees who are not managers but earn more than their department's average salary.

**Query:**

SELECT \*

FROM demo.employees e

WHERE NOT EXISTS (

SELECT 1

FROM demo.departments

WHERE manager\_id = e.employee\_id

) AND EXISTS (

SELECT 1

FROM demo.employees e2

WHERE e.department\_id = e2.department\_id

HAVING e.salary > AVG(e2.salary)

);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_