# Database fundamentals
# Part 3

# Agenda

- Limit
- Group by
- Having
- SQL Alias
- Subqueries
- UNION, UNION ALL
- Join
- Views
- Case ...when
- Stored Procedure

# LIMIT Clause

# The SQL LIMIT clause

- To return a specific number of records use limit clause.
- Useful for pagination or when you don’t need the full result.
- The LIMIT clause is useful on large tables with thousands of records.
- Returning a large number of records can impact performance.
- Note:
  - Not all database systems support the Limit clause.
  - MySQL supports the LIMIT clause to select a limited number of records.

# LIMIT Clause…

- MySQL Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

# LIMIT… examples

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |

- The following SQL statement selects the first three records from the "Customers" table (for MySQL):
  - `SELECT * FROM Customers LIMIT 3;`

Result:

Number of Records: 3

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

# LIMIT…OFFSET Clause

- **OFFSET:** tells MySQL where to start returning rows.
- MySQL Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number
OFFSET number;
```

- Example: Skip the first 5 rows, then return the next 5.
- Skips rows 1–5
- Returns rows 6–10

```
SELECT *
FROM employees
LIMIT 5 OFFSET 5;
```

# Group By and Having Clauses

# The SQL Group by clause

- For use with aggregate functions (min, max, count, sum, avg).
- multiple values returned from SQL query with aggregate function (via GROUP BY)

- **Example:** Count employees in each department

```sql
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department;
```

# The SQL Having clause

- For use with GROUP BY.
- Like WHERE clause, but used to filter groups after aggregation, not on individual rows.
- You **can't** use **WHERE** with aggregate functions, that's why we use **HAVING**.
- **Example:** Show departments with more than 5 employees

```sql
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

- Explanation:
  - First groups employees by department.
  - Then filters only groups where COUNT(*) > 5.

# Difference Between WHERE and HAVING

- WHERE: filters rows before grouping (on raw data).
- HAVING: filters groups after aggregation (on aggregated results).
- Example combining both:

```sql
SELECT department, COUNT(*) AS total_employees
FROM employees
WHERE salary > 3000        -- filter rows first
GROUP BY department
HAVING COUNT(*) > 2;       -- then filter groups
```

# SQL Aliases

# SQL Aliases

- SQL aliases are used to give a table, or a column in a table, a temporary name.
- Aliases are often used to make column names more readable.
- An alias only exists for the duration of that query.
- An alias is created with the **AS** keyword.
- Alias Column Syntax
  - ```
    SELECT column_name AS alias_name
    FROM table_name;
    ```
- Alias Table Syntax
  - ```
    SELECT column_name(s)
    FROM table_name AS alias_name;
    ```

# SQL Aliases example

- The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:
  - ```
    SELECT CustomerID AS ID, CustomerName AS Customer
    FROM Customers;
    ```

Result:

Number of Records: 91

| ID | Customer |
|---|---|
| 1 | Alfreds Futterkiste |
| 2 | Ana Trujillo Emparedados y helados |
| 3 | Antonio Moreno Taquería |
| 4 | Around the Horn |
| 5 | Berglunds snabbköp |
| 6 | Blauer See Delikatessen |
| 7 | Blondel père et fils |
    ```

# SQL Aliases example

- The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column.
- **Note**: It requires double quotation marks or square brackets if the alias name contains spaces
  - `SELECT CustomerName AS Customer, ContactName AS [Contact Person] FROM Customers;`

Result:

Number of Records: 91

| Customer | Contact Person |
|---|---|
| Alfreds Futterkiste | Maria Anders |
| Ana Trujillo Emparedados y helados | Ana Trujillo |
| Antonio Moreno Taquería | Antonio Moreno |
| Around the Horn | Thomas Hardy |

# SQL Aliases example

- The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):
  - ```
    SELECT CustomerName, concat(Address , ', ' , PostalCode , ' ' , City , ', ' , Country) AS Address
    FROM Customers;
    ```

Result:

Number of Records: 91

| CustomerName | Address |
|---|---|
| Alfreds Futterkiste | Obere Str. 57, 12209 Berlin, Germany |
| Ana Trujillo Emparedados y helados | Avda. de la Constitución 2222, 05021 México D.F., Mexico |
| Antonio Moreno Taquería | Mataderos 2312, 05023 México D.F., Mexico |
| Around the Horn | 120 Hanover Sq., WA1 1DP London, UK |
| Berglunds snabbköp | Berguvsvägen 8, S-958 22 Luleå, Sweden |
    ```

# SQL Aliases

- Aliases can be useful when:
  - There are more than one table involved in a query
  - Functions are used in the query
  - Column names are big or not very readable
  - Two or more columns are combined together

# Subqueries

# The SQL Subquery

- A subquery is a query inside another query.
- It is enclosed in parentheses ( ) and can be used in SELECT, INSERT, UPDATE, or DELETE statements.
- Think of it like:

  First run the inner query, then use its result in the outer query.

# The SQL Subquery examples

- **Example:** Find employees who earn more than the average salary.
  - The inner query: SELECT AVG(salary) FROM employees→ returns the average salary.
  - The outer query: selects employees with salary above that value.

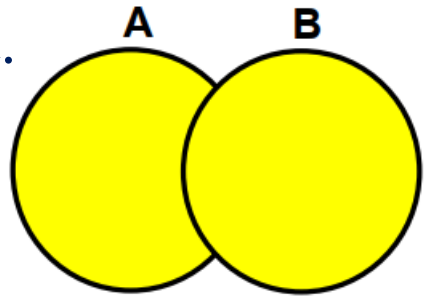
```sql
SELECT first_name, last_name, salary
FROM employees
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
);
```

- **Example:** Find employees who work in departments that are located in New York.

```sql
SELECT first_name, last_name
FROM employees
WHERE department IN (
    SELECT department_id
    FROM departments
    WHERE location = 'New York'
);
```

# UNION and UNION ALL Clauses

# SQL UNION…




- Combines the results of two or more SELECT queries into one result set.
- Removes duplicate rows by default.
- The expressions in the SELECT lists must match in number.
- The data type of each column in the second query must match the data type of its corresponding column in the first query.

- **Example:** Display the current and previous job details of all employees.
- Display each employee only once.

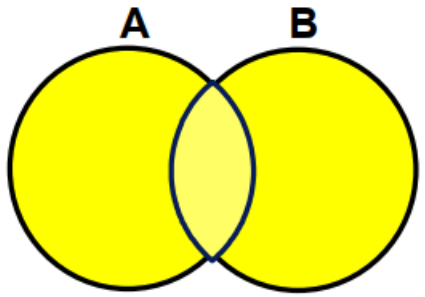
```
SELECT employee_id, job_id
FROM employees

UNION

SELECT employee_id, job_id
FROM job_history;
```

# SQL UNION All…

- Same as **UNION**, but it **does not** remove duplicates.
- Keeps all rows, even if repeated.




- If You Use UNION ALL This will keep duplicates.

```sql
SELECT employee_id, job_id
FROM employees

UNION ALL

SELECT employee_id, job_id
FROM job_history;
```

## 🔹 Example with Different Tables

Suppose we have two tables:

- `customers(id, name, email)`
- `suppliers(id, name, email)`

👉 Get a list of all people (customers + suppliers):

sql

```sql
SELECT name, email FROM customers
UNION
SELECT name, email FROM suppliers;
```

👉 Get them **with duplicates included**:

sql

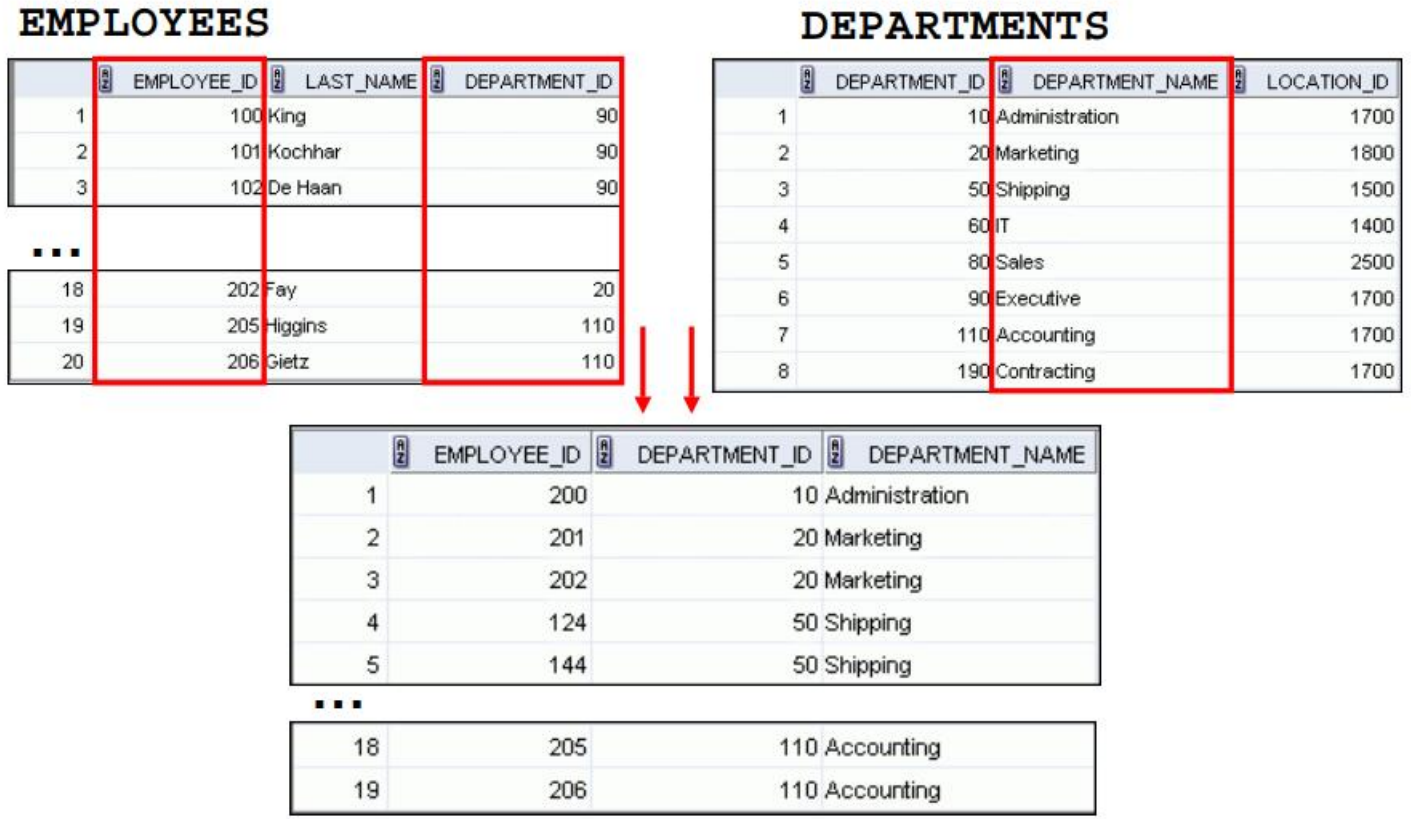
```sql
SELECT name, email FROM customers
UNION ALL
SELECT name, email FROM suppliers;
```

# SQL JOIN

Types of Joins with Examples

# SQL JOIN

- Obtaining Data from Multiple Tables.
- Write SELECT statements to access data from more than one table using join.



**EMPLOYEES**

| | EMPLOYEE_ID | LAST_NAME | DEPARTMENT_ID |
|---|---|---|---|
| 1 | 100 | King | 90 |
| 2 | 101 | Kochhar | 90 |
| 3 | 102 | De Haan | 90 |
| ... | | | |
| 18 | 202 | Fay | 20 |
| 19 | 205 | Higgins | 110 |
| 20 | 206 | Gietz | 110 |

**DEPARTMENTS**

| | DEPARTMENT_ID | DEPARTMENT_NAME | LOCATION_ID |
|---|---|---|---|
| 1 | 10 | Administration | 1700 |
| 2 | 20 | Marketing | 1800 |
| 3 | 50 | Shipping | 1500 |
| 4 | 60 | IT | 1400 |
| 5 | 80 | Sales | 2500 |
| 6 | 90 | Executive | 1700 |
| 7 | 110 | Accounting | 1700 |
| 8 | 190 | Contracting | 1700 |

| | EMPLOYEE_ID | DEPARTMENT_ID | DEPARTMENT_NAME |
|---|---|---|---|
| 1 | 200 | 10 | Administration |
| 2 | 201 | 20 | Marketing |
| 3 | 202 | 20 | Marketing |
| 4 | 124 | 50 | Shipping |
| 5 | 144 | 50 | Shipping |
| ... | | | |
| 18 | 205 | 110 | Accounting |
| 19 | 206 | 110 | Accounting |

# SQL JOINS

- A JOIN clause is used to combine rows from two or more tables, based on a related column between them (usually a primary key / foreign key relationship).
  - Let's look at a selection from the "Orders" table:

| OrderID | CustomerID | OrderDate |
|---|---|---|
| 10308 | 2 | 1996-09-18 |
| 10309 | 37 | 1996-09-19 |
| 10310 | 77 | 1996-09-20 |

  - Then look at a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Country |
|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mexico |

# SQL JOINS…

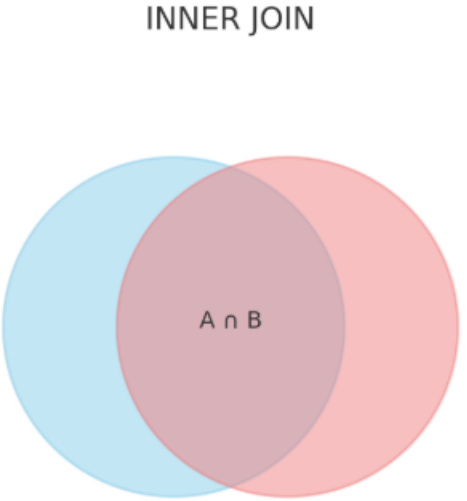
- Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.
- Then, we can create the following SQL statement (that contains a JOIN), that selects records that have matching values in both tables:
  - 
```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

| OrderID | CustomerName | OrderDate |
|---|---|---|
| 10308 | Ana Trujillo Emparedados y helados | 9/18/1996 |
| 10365 | Antonio Moreno Taquería | 11/27/1996 |
| 10383 | Around the Horn | 12/16/1996 |
| 10355 | Around the Horn | 11/15/1996 |
| 10278 | Berglunds snabbköp | 8/12/1996 |

# Types of JOIN

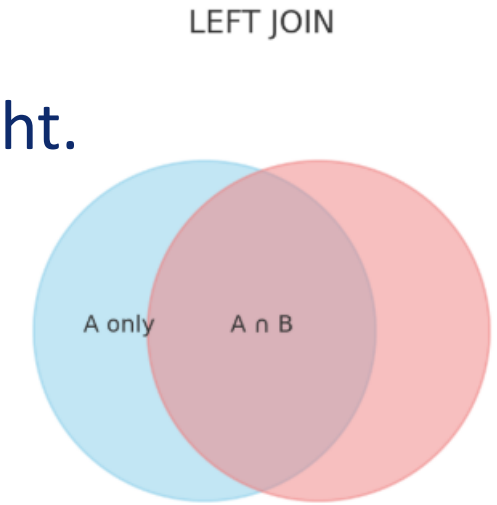
# INNER JOIN

- Returns rows with matching values in both tables.
- Example: select employee with its department name.
- Shows only employees that belong to a department.




```sql
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
INNER JOIN departments d
       ON e.department_id = d.department_id;
```

# LEFT JOIN

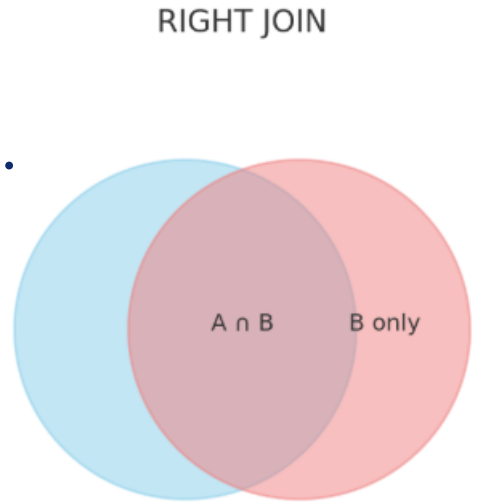



- Returns all rows from the left table + matching rows from right.
- If no match, right table columns are NULL appear.

- Example:
- Shows all employees, even those not assigned to any department.

```
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
LEFT JOIN departments d
       ON e.department_id = d.department_id;
```

# RIGHT JOIN

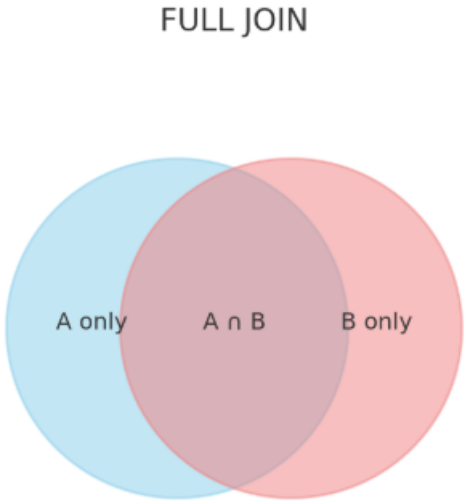



- Returns all rows from the right table + matching rows from left.
- If no match, right table columns are NULL appear.

- Example:
- Shows all departments, even if no employees work in them.

```sql
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
RIGHT JOIN departments d
       ON e.department_id = d.department_id;
```

# FULL JOIN (Simulated in MySQL)




- Returns all rows from both tables.
- MySQL **does not support** FULL JOIN directly.
- Can be simulated with **UNION** of LEFT JOIN and RIGHT JOIN.

- Example:

```sql
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
LEFT JOIN departments d
       ON e.department_id = d.department_id

UNION

SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
RIGHT JOIN departments d
       ON e.department_id = d.department_id;
```

# SELF JOIN

- A table joined with **itself** (useful for hierarchical data, like managers and employees).
- It’s often used to find relationships inside the same table.

- Example:
- Shows each employee with their manager’s name.

```
SELECT e.employee_id, e.first_name, m.first_name AS manager_name
FROM employees e
JOIN employees m
    ON e.manager_id = m.employee_id;
```

# ON Keyword

- Used in the JOIN clause to define join conditions (how two tables are linked).
- Used with all types of joins (INNER JOIN, LEFT JOIN, RIGHT JOIN, etc.).
- It defines the relationship/condition between the two tables.
- Keeps join condition separate from filtering conditions.
- If the condition in ON is not satisfied, the row will not appear (except in outer joins where NULLs may appear).
- Without ON, SQL wouldn’t know which columns to use to connect the tables.

# JOIN Summery

- INNER JOIN → matching rows only.
- LEFT JOIN → all left + matches from right.
- RIGHT JOIN → all right + matches from left.
- FULL JOIN → all rows from both (MySQL: simulate with UNION)
- SELF JOIN → table joined to itself

# VIEWS

# The SQL VIEWS

- Views provide users-controlled access to tables.
- Base Table–table containing the raw data.
- Dynamic View
  - A “virtual table” created dynamically upon request by a user.
  - No data actually stored; instead, data from base table made available to user.
  - Based on SQL SELECT statement on base tables or other views.
- Materialized View
  - Copy or replication of data.
  - Data actually stored.
  - Must be refreshed periodically to match corresponding base tables.

# The SQL VIEWS example

- View has a name.
- View is based on a SELECT statement.
- **Create view**

```
-- Create a view to show employee full names and their departments
CREATE VIEW employee_summary AS
SELECT id,
       CONCAT(first_name, ' ', last_name) AS full_name,
       department
FROM employees;
```

- Now you can query it like a table.

```
SELECT * FROM employee_summary;
```

# The SQL VIEWS

- **Modify View**

```
ALTER VIEW employee_summary AS
SELECT id,
       CONCAT(first_name, ' ', last_name) AS full_name,
       department,
       salary
FROM employees;
```

- **Drop view**

```
-- Drop view view_name;

Drop view employee_summary;
```

# Advantages of Views

- To restrict data access.
- To make complex queries easy.
- To provide data independence.
- To present different views of the same data.

# Case …when

# CASE…WHEN

- The CASE WHEN statement in MySQL is used to add conditional logic inside SQL queries — like an IF…ELSE in programming.

- Syntax:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE default_result
END
```

Example:

```
SELECT first_name,
       salary,
       CASE
         WHEN salary > 10000 THEN 'High'
         WHEN salary BETWEEN 5000 AND 10000 THEN 'Medium'
         ELSE 'Low'
       END AS salary_level
FROM employees;
```

# Stored Procedure

# Stored Procedure

- A Stored Procedure is a set of SQL statements that you save in the database and run later by calling it.
- It’s like a function in programming: reusable, parameterized, and stored on the server.
- Encapsulates complex queries.
- Improves performance (stored on server).
- Reusable by multiple apps.
- Can include loops, conditions, variables.

# Stored Procedure

- Creating Syntax

```
DELIMITER $$

CREATE PROCEDURE procedure_name()
BEGIN
    -- SQL statements
END$$

DELIMITER ;
```

Using or calling syntax

```
-- 🔹 Basic call (no parameters)
CALL procedure_name();

-- 🔹 With one IN parameter (just pass a value)
CALL procedure_name(123);

-- 🔹 With IN and OUT parameter
-- IN → pass a value
-- OUT → capture result in a user variable
CALL procedure_name(123, @out_value);

-- 🔹 With multiple parameters (mix of IN, OUT, INOUT possible)
CALL procedure_name('Alice', 5000, @status);
```

- **DELIMITER $$** → temporarily changes the statement terminator so MySQL doesn't stop at the first ;.
- **CREATE PROCEDURE** → defines the procedure.
- **BEGIN … END** → block of SQL code.

# Stored Procedure examples

- Example 1: Simple Procedure

```
DELIMITER $$

CREATE PROCEDURE GetAllEmployees()
BEGIN
    SELECT * FROM employees;
END$$

DELIMITER ;
```

- Calling it

```
CALL GetAllEmployees();
```

# Stored Procedure examples

- Example 2: With Input Parameter

```
DELIMITER $$

CREATE PROCEDURE GetEmployeesByDept(IN dep_id INT)
BEGIN
    SELECT employee_id, first_name, salary
    FROM employees
    WHERE department_id = dep_id;
END$$

DELIMITER ;
```

- Calling it

```
CALL GetEmployeesByDept(10);
```

# Stored Procedure examples

- Example 3: With Output Parameter

```
DELIMITER $$

CREATE PROCEDURE CountEmployeesByDept(IN dep_id INT, OUT emp_count INT)
BEGIN
    SELECT COUNT(*) INTO emp_count
    FROM employees
    WHERE department_id = dep_id;
END$$

DELIMITER ;
```

- Calling it

```
CALL CountEmployeesByDept(10, @total);
SELECT @total;  -- shows number of employees in dept 10
```

# Stored Procedure examples

- Example 4: Procedure with IF / CASE

```
DELIMITER $$

CREATE PROCEDURE SalaryLevel(IN emp_id INT, OUT level VARCHAR(10))
BEGIN
    DECLARE emp_salary DECIMAL(10,2);

    SELECT salary INTO emp_salary
    FROM employees
    WHERE employee_id = emp_id;

    CASE
        WHEN emp_salary > 10000 THEN SET level = 'High';
        WHEN emp_salary BETWEEN 5000 AND 10000 THEN SET level = 'Medium';
        ELSE SET level = 'Low';
    END CASE;
END$$

DELIMITER ;
```

Calling it

```
-- 1 Initialize the OUT variable (to store the procedure's result)
SET @emp_level = '';

-- 2 Call the stored procedure with:
--      - IN parameter: employee_id = 101
--      - OUT parameter: @emp_level (to capture the level result)
CALL SalaryLevel(101, @emp_level);

-- 3 Display the value of the OUT parameter returned by the procedure
SELECT @emp_level;
```

# Any Questions ?