SQL-Refresher-Notes

1. Create Command

Borrow table with foreign key isbn which references the table book.

```
CREATE TABLE borrow (
    borrow_id varchar(20) PRIMARY KEY,
    isbn varchar(20),
    borrow_date date,
    due_date date,
    return_date date,
    fine decimal(20,2),
    FOREIGN KEY(isbn) REFERENCES Book(isbn)
);
```

2. Alter Table Command

Change Name of column

Command to change name of column email to mail_info

```
ALTER TABLE USER
CHANGE email mail_info varchar(20);
```

Change Datatype size and add column

Changes genre datatype size and adds a column genre_spec immediately after genre column

```
ALTER TABLE Book
MODIFY genre varchar(10),
ADD COLUMN genre_spec varchar(50)
AFTER genre;
```

3. TRUNCATE table

```
TRUNCATE TABLE Borrowing;
```

4. INSERT Command

```
INSERT INTO user VALUES
('UN033', 'Helen', 'helen@hotmail.com', '1989-07-01', 'F', 8033307890, '152
Orax Road'),
('UN034', 'Marry', 'marry@gmail.com', '1985-05-20', 'F', 7204560191, '63
Oredo Square'),
('UN035', 'Richard', 'richard@yahoo.com', '1993-11-16', 'M', 5230567448, '16
Rudolf St');
```

5. Update Command

UPDATE statement to modify existing data in the Borrowing table's Return_Date column will be updated to '2023-09-10' and fine amount is set as null to the borrower's account. Only the row where Borrowing_ID is 'BR009' will be affected by this update.

```
UPDATE BORROWING
SET return_date = '2023-09-10', fine = null
WHERE borrowing_id = 'BR009';
```

6. Delete Record

The delete statement is used to remove rows from the book table where the value in the genre column is equal to 'Horror'.

```
DELETE from Book where genre='Horror';
```

7. SELECT and ORDER BY

The SELECT statement retrieves specific columns from the book table and orders the result set by the first column in ascending order.

```
SELECT title, author, publication_date
FROM Book
```

```
ORDER BY title;
```

8. WHERE and LIKE

The SELECT statement retrieves specific columns from the user table where the email ends with the word gmail.com. The results are then sorted in ascending order based on the gender column.

```
SELECT user_id, username, gender
FROM USER
WHERE email LIKE '%gmail.com'
ORDER BY gender;
```

9. SELECT and WHERE

The SELECT statement retrieves specific columns from the book table, specifically selecting books with prices between 500 and 900. The results are then ordered by the publication date column in descending order.

```
SELECT isbn, title, publication_date, price
FROM BOOK
WHERE price BETWEEN 500 AND 900
ORDER BY publication_date DESC;
```

10. WHERE NOT IN

The SELECT statement retrieves specific columns from the book table for books that do not belong to the 'Fiction' or 'Mystery' genres. The results are then sorted in descending order based on the isbn column, listing books with higher ISBN numbers first

```
SELECT isbn, title, author, publication_date
FROM BOOK
WHERE genre NOT IN ('Fiction', 'Mystery')
ORDER BY isbn DESC;
```

11. != Operator

The SELECT statement retrieves specific columns from the borrowing table for records where the return_date is not same as due_date. The results are then sorted in ascending order based on the user_id, which represents the user's unique identifiers.

```
SELECT borrowing_id, user_id, borrow_date
FROM BORROWING
WHERE return_date != due_date
ORDER BY user_id;
```

12. CONCAT and SUBSTRING

- The SELECT statement retrieves specific columns from the user table.
- It takes the first four characters of the phone column and the first five characters of the username column, concatenates them, and assigns the result to the password column in the output.
- The results are then sorted in ascending order based on the user name.

```
SELECT user_id, username,
       CONCAT(SUBSTRING(phoneno, 1, 4), SUBSTRING(username, 1, 5))
FROM user
ORDER BY username ASC;
```

13. COALESCE

- The SELECT statement retrieves the user_id, username, and phone number or email details of the user
- if both phone number and email are NULL then display as 'Not Available', and alias it as contact from the user table.
- It further filters the results to include only records where the month of the dob column is 'May'.
- The results are then sorted in ascending order based on the username.

```
SELECT user_id, username,
       COALESCE(phoneno, email, 'Not Available') AS contact
FROM USER
WHERE MONTH(dob) = 5
```

```
ORDER BY username ASC;
```

- Here COALESCE will check the first non-null value from phoneno, email, and 'Not Available'
- If phoneno is null, email is selected
- If phoneno and email is null then 'Not Available' is selected

14. ROUND(), DATEDIFF(), CURDATE()

- The SELECT statement retrieves the user_id, username, email, and calculates the age in years using the date function.
- It retrieves this information from the user table, specifically for users whose age is lesser than 35 years.
- The results are then sorted in ascending order based on the user name.

```
SELECT user_id, username, dob,
       ROUND(DATEDIFF(CURDATE(), dob) / 365) AS age_in_years
FROM USER
WHERE ROUND(DATEDIFF(CURDATE(), dob) / 365) = 35
ORDER BY dob DESC;
```

- Calculates the age in years by finding the difference in days between the current date (Using CURDATE()) and the user's date of birth (dob) (Using DATEDIFF), then dividing by 365 and rounding the result (Using ROUND()).
- Filters users who are exactly 35 years old.
- Sorts the results by dob in descending order (youngest among the 35-year-olds first).

15. CONCAT() and Year()

- The SELECT statement concatenates the strings.
- It creates a string named book_info that contains information about the author and the publication year for each book in the BOOK table.
- The results are then sorted in descending order based on the publication year.

```
SELECT CONCAT('Mr/Ms.', author, ' published a book on ',
YEAR(publication_date)) AS book_info
```

```
FROM BOOK
ORDER BY YEAR(publication_date) DESC;
```

- What the query does?
  - Constructs a string for each book in the format: "Mr/Ms.[author] published a book on [year]"
  - Uses YEAR(publication_date) to extract the year from the publication_date.
  - Sorts the results by publication year in descending order (most recent first).

16. CASE

The SELECT statement retrieves the isbn, title, author and categorises books based on their price into three categories. The results are then sorted in descending order based on the ISBN (isbn) column.

```
SELECT isbn,
       title,
       author,
       CASE
           WHEN price < 400 THEN 'Affordable'
           WHEN price >= 400 AND price < 600 THEN 'Moderate'
           ELSE 'Expensive'
       END AS price_category
FROM BOOK
ORDER BY isbn DESC;
```

- What this query does
  - Retrieves the isbn, title, and author of books from the BOOK table.
  - Uses a CASE statement to categorize each book's price into:
    - 'Affordable' if price < 400
    - 'Moderate' if price is between 400 and 599
    - 'Expensive' if price ≥ 600
  - Sorts the results by isbn in descending order.

17. SUBSTRING_INDEX()

This Select statement retrieves user information from the user table and create an email_info column that modifies the email by replacing the 'lausd.com' for users with 'Los Angeles'

addresses and retains the original email for others. The results are then ordered by user name in descending order.

```
SELECT user_id, username,
    CASE
        WHEN address = 'Los Angeles'
        THEN CONCAT(SUBSTRING_INDEX(email, '@', 1), '@lausd.com')
        ELSE email
    END AS email_info
FROM USER
ORDER BY username DESC;
```

- Retrieves user_id, username, and a modified email_info field.
- If the user's address is 'Los Angeles', it replaces their email domain with '@lausd.com' while keeping the username part.
- Otherwise, it keeps the original email.
- Results are sorted by username in descending order.

Syntax

```
SUBSTRING_INDEX(str, delim, count)
```

- str: The string you want to extract from.
- delim: The delimiter that separates parts of the string.
- count:
  - If positive, it returns everything before the nth occurrence of the delimiter.
  - If negative, it returns everything after the nth occurrence from the end.

18. COUNT(), YEAR()

- The query counts the number of books each user borrowed from the year other than 2024 by grouping userid's.
- It uses the Scalar function to extract the year from borrow_date, filters records for the year other than 2024, calculates the count of borrowing IDs, and presents the result as count_borrowed per user, sorting the output by user id

```sql
SELECT user_id, COUNT(*) AS count_borrowed
FROM BORROWING
WHERE YEAR(borrow_date) != 2024
GROUP BY user_id
ORDER BY user_id DESC;
```

- What this query does:
  - Counts the number of borrowings (COUNT(*)) for each user_id.
  - Filters out borrowings that occurred in the year 2024.
  - Groups the results by user_id so each user's borrow count is calculated.
  - Orders the results by user_id in descending order.

19. GROUP BY, HAVING, COUNT(), ORDER BY

The query calculates the total number of books (calculation_books) written by a particular author in the book table by grouping records based on the author. It includes only authors who have written fewer than 2 books and orders the results in descending order based on the author.

```sql
SELECT author, COUNT(*) AS calculation_books
FROM BOOK
GROUP BY author
HAVING calculation_books < 2
ORDER BY author DESC;
```

- What this query does:
  - Counts the number of books written by each author.
  - Groups the results by author.
  - Filters the results to include only authors who have written fewer than 2 books.
  - Sorts the results by author in descending alphabetical order.

20. Subquery

The query retrieves data from user information in the user table and calculates the maximum fine associated with each user by using a subquery from the borrowing table. If a user has no

fines, the query returns null as the maximum fine. The results are then ordered based on the user_id in descending order.

```
SELECT
    u.user_id,
    u.username,
    (
        SELECT MAX(b.fine)
        FROM borrowing b
        WHERE b.user_id = u.user_id
    ) AS max_fine
FROM user u
ORDER BY u.user_id DESC;
```

- What this query does
  - Retrieves each user's user_id and username.
  - Uses a subquery to find the maximum fine (MAX(b.fine)) from the borrowing table for each user.
  - The subquery runs once per user, matching b.user_id with u.user_id.
  - The result is sorted by user_id in descending order.

21. Subquery 2

- The query retrieves user id from the user table for users who have borrowed books.
- It performs a join between the borrowing and user tables.
- The results are then grouped and filtered to include only users who have borrowed the maximum number of books compared to all other users, as determined by a subquery that calculates the count of borrowing IDs grouped by users.
- Order the results with respect to userid in descending order.

```
SELECT user_id
FROM borrowing
GROUP BY user_id
HAVING COUNT(*) = (
    SELECT COUNT(*)
    FROM borrowing
    GROUP BY user_id
```

```
    ORDER BY COUNT(*) DESC
    LIMIT 1
)
ORDER BY user_id DESC;
```

- What this query does:
  - Groups the borrowing records by user_id.
  - Counts how many borrowings each user has made.
  - Filters to include only those users whose borrow count is equal to the highest borrow count among all users.
  - The subquery:
    - Finds the maximum borrow count by grouping and ordering users by their borrow count in descending order and taking the top one.
    - The outer query returns all users who match that maximum count (in case of a tie).
    - Results are sorted by user_id in descending order.

22. Subquery 3

- The query retrieves the author and genre of books along with the count of user_id associated with each book.
- It joins the BOOK and BORROWING tables based on matching ISBN values and filters records to only include books that have been borrowed by a specific user ('UN002').
- The results are then grouped by author and genre, aggregating the count of unique users for each book.

```
SELECT b.author, b.genre, COUNT(br.user_id) AS user_total
FROM book b
JOIN BORROWING br ON b.isbn = br.isbn
WHERE b.isbn IN (
    SELECT isbn
    FROM BORROWING
    WHERE user_id = 'UN002'
)
GROUP BY b.author, b.genre, b.isbn;
```

- What this query does:
  - Joins the book and BORROWING tables on isbn.
  - Filters to include only those books that were borrowed by the user with user_id = 'UN002'.
  - Groups the results by author, genre, and isbn.
  - Counts how many times each book (by author and genre) was borrowed by that user.

23. Date functions and CONCAT

- The joins are used to combine rows from two or more tables based on a related column between them.
- The concept of joins allows you to retrieve data that is spread across multiple tables in a database by linking records that have matching values in the specified columns.

```
SELECT u.username, borrowing_id,
       CONCAT(MONTHNAME(borrow_date), "_", YEAR(borrow_date)) AS
borrowed_date_info
FROM BORROWING AS b
JOIN USER u ON u.user_id = b.user_id
ORDER BY borrowing_id ASC;
```

- Joins the BORROWING table (b) with the USER table (u) using user_id.
- Retrieves:
  - username from the USER table.
  - borrowing_id from the BORROWING table.
  - A formatted string borrowed_date_info combining the month name and year of the borrow_date, separated by an underscore (e.g., "June_2025").
  - Orders the results by borrowing_id in ascending order.

24. JOINS

The query retrieves the username, genre, and borrow_date by joining data from the users, borrowing, and book tables based on matching user_id and isbn columns. The results are sorted in descending order based on the username.

```
SELECT username, genre, borrow_date
FROM BORROWING AS br
JOIN USER u ON u.user_id = br.user_id
JOIN BOOK bk ON br.isbn = bk.isbn
ORDER BY username DESC;
```

- What this query does:
  - Joins the BORROWING, USER, and BOOK tables.
  - Retrieves:
    - username from the USER table.
    - genre from the BOOK table.
    - borrow_date from the BORROWING table.
    - Orders the results by username in descending order.

25. JOINS 2

The query retrieves data from the users, borrowing, and book tables based on matching user_id and isbn columns. It selects specific columns from these tables, filters records of the users who have returned the book and orders the results based on the borrowing id.

```
SELECT u.user_id, u.username, br.isbn, bk.genre, br.borrow_date,
br.due_date, br.fine
FROM USER AS u
JOIN BORROWING br ON br.user_id = u.user_id
JOIN BOOK bk ON bk.isbn = br.isbn
WHERE br.return_date IS NOT NULL
ORDER BY borrowing_id DESC;
```

- What this query does:
  - Joins the USER, BORROWING, and BOOK tables.
  - Retrieves:
    - user_id and username from the USER table.
    - isbn, borrow_date, due_date, and fine from the BORROWING table.
    - genre from the BOOK table.
    - Filters to include only records where the book has been returned (return_date IS NOT NULL).

        - Orders the results by borrowing_id in descending order (most recent borrowings first)

26. JOINS 3

- The query retrieves distinct pairs of book titles (title1 and title2) along with their common publisher and publication date.
- It joins the BOOK table with itself based on the condition that books have the same author.
- Additionally, it ensures that the books have the same publisher but different titles.
- Finally, it orders the results in descending order based on the second book's title and in ascending order based on first book's title.

```
SELECT
    b1.title AS title1,
    b2.title AS title2,
    b1.publisher,
    b1.publication_date
FROM BOOK b1
JOIN BOOK b2
    ON b1.author = b2.author
    AND b1.publisher = b2.publisher
    AND b1.title != b2.title
ORDER BY
    title2 DESC,
    title1 ASC;
```

- What this query does:
    - Performs a self-join on the BOOK table to find pairs of books (b1 and b2) that:
        - Have the same author
        - Have the same publisher
        - Have different titles
    - Selects:
        - title1 and title2 (the titles of the two books)
        - publisher and publication_date from b1
        - Orders the results by title2 in descending order and title1 in ascending order.

27. JOINS 4

- The query retrieves the gender details from the USER table and counts the number of borrowing IDs associated with each gender.
- It uses a particular join that joins the BORROWING table based on matching user values.
- The results are then grouped and finally ordered in descending order based on their gender.
- This approach ensures that all gender from the USERS table are included in the result, even if they have no borrowing records in the BORROWING table

```
SELECT u.gender, COUNT(*) AS Total_books
FROM BORROWING AS b
JOIN USER u ON u.user_id = b.user_id
GROUP BY u.gender
ORDER BY u.gender DESC;
```

- What this query does:
  - Joins the BORROWING and USER tables using user_id.
  - Groups the results by gender.
  - Counts the total number of books borrowed by users of each gender.
  - Orders the results by gender in descending order.

28. Add Primary key

When Creating

```
CREATE TABLE Employees (
    EmployeeID INT NOT NULL,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    PRIMARY KEY (EmployeeID)
);
```

Adding to an existing table

```
ALTER TABLE Employees
ADD CONSTRAINT PK_Employees PRIMARY KEY (EmployeeID);
```

29. Remove Primary key

```
ALTER TABLE Employees
DROP CONSTRAINT PK_Employees;
```

30. Modify Column

Add Column

```
ALTER TABLE Employees
ADD DateOfBirth DATE;
```

Drop Column

```
ALTER TABLE Employees
DROP COLUMN DateOfBirth;
```

Modify Data type

```
ALTER TABLE Employees
MODIFY COLUMN FirstName VARCHAR(100);
```

Rename

```
ALTER TABLE Employees
RENAME COLUMN FirstName TO GivenName;
```

31. Add Foreign Key

```
ALTER TABLE Employees
ADD CONSTRAINT FK_Employees_Departments
FOREIGN KEY (DepartmentID)
REFERENCES Departments(DepartmentID);
```

32. Change Column name via change and rename

Using CHANGE

```
ALTER TABLE Employees
CHANGE FirstName GivenName VARCHAR(50);
```

Using RENAME

```
ALTER TABLE Employees
RENAME COLUMN FirstName TO GivenName;
```

33. All Date related functions

1. Current Date and Time

Function	Description
`NOW()`	Returns the current date and time (`YYYY-MM-DD HH:MM:SS`)
`CURDATE()`	Returns the current date (`YYYY-MM-DD`)
`CURTIME()`	Returns the current time (`HH:MM:SS`)
`UTC_DATE()`	Returns the current UTC date
`UTC_TIME()`	Returns the current UTC time
`UTC_TIMESTAMP()`	Returns the current UTC date and time

2. Date Arithmetic

Function	Description
`DATE_ADD(date, INTERVAL expr unit)`	Adds a time interval to a date
`DATE_SUB(date, INTERVAL expr unit)`	Subtracts a time interval from a date

Function	Description
`ADDDATE(date, INTERVAL expr unit)`	Synonym for `DATE_ADD`
`SUBDATE(date, INTERVAL expr unit)`	Synonym for `DATE_SUB`
`DATEDIFF(date1, date2)`	Returns the number of days between two dates
`TIMESTAMPDIFF(unit, datetime1, datetime2)`	Returns the difference between two dates in the specified unit (e.g., SECOND, MINUTE, HOUR, DAY, MONTH, YEAR)

3. Extracting Parts of Dates

Function	Description
`YEAR(date)`	Returns the year
`MONTH(date)`	Returns the month (1–12)
`DAY(date)` or `DAYOFMONTH(date)`	Returns the day of the month
`HOUR(time)`	Returns the hour
`MINUTE(time)`	Returns the minute
`SECOND(time)`	Returns the second
`DAYOFWEEK(date)`	Returns the weekday index (1 = Sunday, 7 = Saturday)
`DAYOFYEAR(date)`	Returns the day of the year (1–366)
`WEEK(date)`	Returns the week number
`QUARTER(date)`	Returns the quarter (1–4)
`EXTRACT(unit FROM date)`	Extracts a part of the date (e.g., `EXTRACT(YEAR FROM '2025-06-14')`)

4. Formatting and Parsing

Function	Description
`DATE_FORMAT(date, format)`	Formats a date using format specifiers (e.g., `%Y-%m-%d`)

Function	Description
`STR_TO_DATE(str, format)`	Parses a string into a date using the given format
`TIME_FORMAT(time, format)`	Formats a time value

34. LIMIT

The LIMIT clause in SQL is used to restrict the number of rows returned by a query. It's especially useful when you're working with large datasets and only want to see a subset of the results.

```
SELECT * FROM Employees
LIMIT 5;
```

35. Auto_increment

```
CREATE TABLE Employees (
    EmployeeID INT AUTO_INCREMENT,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    PRIMARY KEY (EmployeeID)
);
```

36. CTE

A CTE is like a temporary named result that you can use in your main SQL query. Think of it as a shortcut or a nickname for a subquery.

Let’s say you want to find employees who earn more than ₹1,00,000.

```
WITH HighEarners AS (
    SELECT FirstName, Salary
    FROM Employees
    WHERE Salary > 100000
)
```

```
SELECT * FROM HighEarners;
```

HighEarners is the name of the CTE.

You can now use it like a table in your main query.

37. Unique, NOT NULL and CHECK Constraints

UNIQUE Constraint

- Ensures that all values in a column are different.
- Allows NULL values (but only one NULL per column in most databases).

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE
);
```

NOT NULL Constraint

- Ensures that a column cannot have NULL values.
- Used when a field is mandatory.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(50) NOT NULL
);
```

CHECK Constraint

- Ensures that values in a column meet a specific condition.
- Can be used to enforce business rules.

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    Price DECIMAL(10, 2) CHECK (Price > 0)
```

```
);
```

38. Types of Subqueries(Correlated and non correlated subqueries)

Non-Correlated Subquery

A non-correlated subquery is independent of the outer query. It runs once, and its result is used by the outer query.

Example

```
SELECT Name
FROM Employees
WHERE DepartmentID = (
    SELECT DepartmentID
    FROM Departments
    WHERE DepartmentName = 'Sales'
);
```

- The subquery finds the DepartmentID for 'Sales'.
- The outer query uses that result to find employees in that department.
- The subquery runs once.

Correlated Subquery

A correlated subquery depends on the outer query. It runs once for each row processed by the outer query.

```
SELECT Name
FROM Employees e
WHERE Salary > (
    SELECT AVG(Salary)
    FROM Employees
    WHERE DepartmentID = e.DepartmentID
);
```

- The subquery calculates the average salary for each employee's department.

- It uses e.DepartmentID from the outer query.
- The subquery runs for each row in the outer query

39. Stored Procedures

A stored procedure is a precompiled set of SQL statements that you can save and reuse in a database. It's like a function in programming—once created, you can call it whenever needed, often with parameters.

```
CREATE PROCEDURE GetEmployeeByID
    @EmployeeID INT
AS
BEGIN
    SELECT * FROM Employees
    WHERE EmployeeID = @EmployeeID;
END;
```

```
EXEC GetEmployeeByID @EmployeeID = 101;
```

40. Triggers

A trigger in SQL is a special kind of stored procedure that automatically executes in response to certain events on a table or view, such as INSERT, UPDATE, or DELETE.

```
CREATE TRIGGER trg_AuditInsert
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO AuditLog (Action, EmployeeID, ActionTime)
    VALUES ('INSERT', NEW.EmployeeID, NOW());
END;
```

41. Window functions

1. MAX Salary – Whole Table (Without window function)

emp_id	emp_name	salary	dept_name
1	Asha	60000	Sales
2	Biju	50000	Sales
3	Chitra	50000	Sales
4	Dinesh	40000	Sales
5	Esha	40000	Sales
6	Farhan	30000	Sales

SQL:

```
SELECT MAX(salary) AS max_salary
FROM employee;
```

Output:

max_salary
60000

2. MAX Salary by Department (without window function)

emp_id	emp_name	salary	dept_name
1	Asha	60000	Sales
2	Biju	50000	Sales
3	Chitra	50000	Sales
4	Dinesh	40000	Sales
5	Esha	40000	Sales
6	Farhan	30000	Sales
7	Gayathri	62000	HR
8	Hari	55000	HR

SQL:

```sql
SELECT dept_name, MAX(salary) AS max_salary
FROM employee
GROUP BY dept_name;
```

Output:

dept_name	max_salary
Sales	60000
HR	62000

❌ But this doesn't show other employee details. So we will start with window functions.

3. MAX Salary Using `OVER()` – Keep All Rows

SQL:

```sql
SELECT e.*,
       MAX(salary) OVER() AS max_salary
FROM employee e;
```

Output:

emp_name	dept_name	salary	max_salary
Asha	Sales	60000	62000
Biju	Sales	50000	62000
...	...	...	62000
Hari	HR	55000	62000

`OVER()` creates a **window over the entire table**.

4. MAX Salary per Department using `PARTITION BY`

SQL:

```sql
SELECT e.*,
       MAX(salary) OVER(PARTITION BY dept_name) AS max_salary
FROM employee e;
```

Output:

emp_name	dept_name	salary	max_salary
Asha	Sales	60000	60000
Biju	Sales	50000	60000
...	...	...	...
Gayathri	HR	62000	62000

Each **department gets its own window**.

5. ROW_NUMBER() — Assign Unique Numbers

SQL:

```
SELECT e.*,
       ROW_NUMBER() OVER() AS rn
FROM employee e;
```

Output:

emp_name	dept_name	salary	rn
Asha	Sales	60000	1
Biju	Sales	50000	2
...	...	...	..

6. ROW_NUMBER per Department

SQL:

```
SELECT e.*,
       ROW_NUMBER() OVER(PARTITION BY dept_name ORDER BY emp_id) AS rn
FROM employee e;
```

Output:

emp_name	dept_name	salary	rn
Asha	Sales	60000	1

emp_name	dept_name	salary	rn
Biju	Sales	50000	2
...	...	...	..
Gayathri	HR	62000	1
Hari	HR	55000	2

7. RANK() — Skips Numbers for Ties

SQL:

```
SELECT emp_name, dept_name, salary,
       RANK() OVER(PARTITION BY dept_name ORDER BY salary DESC) AS rnk
FROM employee;
```

Output (Sales):

emp_name	salary	rnk
Asha	60000	1
Biju	50000	2
Chitra	50000	2
Dinesh	40000	4
Esha	40000	4
Farhan	30000	6

⛔ Skips **3** and **5**.

8. DENSE_RANK() — No Skipping

SQL:

```
SELECT emp_name, dept_name, salary,
       DENSE_RANK() OVER(PARTITION BY dept_name ORDER BY salary DESC) AS
dense_rnk
FROM employee;
```

Output (Sales):

emp_name	salary	dense_rnk
Asha	60000	1
Biju	50000	2
Chitra	50000	2
Dinesh	40000	3
Esha	40000	3
Farhan	30000	4

✅ No skipped values.

9. LAG() and LEAD() — Previous and Next Row

SQL:

```
SELECT emp_id, emp_name, dept_name, salary,
       LAG(salary) OVER(PARTITION BY dept_name ORDER BY emp_id) AS
prev_salary,
       LEAD(salary) OVER(PARTITION BY dept_name ORDER BY emp_id) AS
next_salary
FROM employee;
```

Output (Sales):

emp_id	emp_name	salary	prev_salary	next_salary
1	Asha	60000	NULL	50000
2	Biju	50000	60000	50000
3	Chitra	50000	50000	40000
4	Dinesh	40000	50000	40000
5	Esha	40000	40000	30000
6	Farhan	30000	40000	NULL

Shows salary of the previous and next employee **in the same department**.

42. Offset

`OFFSET` is used in SQL to **skip a number of rows** before returning results. It's commonly used with `LIMIT` to implement **pagination**.

Syntax

```
SELECT columns
FROM table_name
ORDER BY some_column
LIMIT number_of_rows
OFFSET number_to_skip;
```

Use Case: Pagination

Let's say you have this `employee` table:

emp_id	emp_name	salary
1	Asha	60000
2	Biju	55000
3	Chitra	50000
4	Dinesh	45000
5	Esha	40000
6	Farhan	35000

Example 1: Get First 3 Employees

```
SELECT *
FROM employee
ORDER BY emp_id
LIMIT 3;
```

Output:

emp_id	emp_name	salary
1	Asha	60000
2	Biju	55000

emp_id	emp_name	salary
3	Chitra	50000

Example 2: Get Next 3 Employees (Skip first 3)

```
SELECT *
FROM employee
ORDER BY emp_id
LIMIT 3 OFFSET 3;
```

📊 Output:

emp_id	emp_name	salary
4	Dinesh	45000
5	Esha	40000
6	Farhan	35000

Quick Summary

Keyword	Purpose
`LIMIT`	How many rows to return
`OFFSET`	How many rows to skip