Here's a comprehensive overview of subqueries in SQL, along with detailed notes on how to find the largest, second largest, third largest, and smallest salaries using both subqueries and without subqueries. Each concept will be explained clearly with examples based on the provided data.

---

1. Understanding Subqueries

Definition

A **subquery** is a query nested inside another query. It can be used to retrieve data that will be used in the main query or to filter results based on certain criteria. Subqueries can return single or multiple values and can be found in the `SELECT`, `FROM`, `WHERE`, and `HAVING` clauses.

Types of Subqueries

1. **Single-row subquery**: Returns only one row and one column.
2. **Multiple-row subquery**: Returns one column but multiple rows.
3. **Multiple-column subquery**: Returns multiple columns.
4. **Correlated subquery**: Depends on the outer query for its values.

General Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name OPERATOR (SELECT column_name FROM table_name
WHERE condition);
```

---

2. Finding Salaries Using Subqueries

a. Second Highest Salary

To find the second highest salary using a subquery:

```
SELECT * FROM emp
WHERE salary = (
    SELECT MAX(salary)
    FROM emp
    WHERE salary < (SELECT MAX(salary) FROM emp)
);
```

Explanation:

- The inner subquery `(SELECT MAX(salary) FROM emp)` retrieves the maximum salary.
- The middle subquery `(SELECT MAX(salary) FROM emp WHERE salary < ...)` retrieves the maximum salary that is less than the highest salary.

b. Third Highest Salary

To find the third highest salary:

```
SELECT * FROM emp
WHERE salary = (
    SELECT MAX(salary)
    FROM emp
    WHERE salary < (
        SELECT MAX(salary)
        FROM emp
        WHERE salary < (SELECT MAX(salary) FROM emp)
    )
);
```

Explanation:

- Similar to the second highest, this query adds an additional layer to filter the maximum salary that is less than the second highest.

c. Smallest Salary

To find the smallest salary:

```
SELECT * FROM emp
WHERE salary = (
    SELECT MIN(salary)
    FROM emp
);
```

Explanation:

- This subquery simply retrieves the minimum salary from the `emp` table.

d. Third Smallest Salary

To find the third smallest salary:

```
SELECT * FROM emp
WHERE salary = (
    SELECT MIN(salary)
    FROM emp
    WHERE salary > (
        SELECT MIN(salary)
        FROM emp
        WHERE salary > (
            SELECT MIN(salary) FROM emp
        )
    )
);
```

Explanation:

- Each layer of the subquery retrieves the next higher minimum salary until the third smallest salary is identified.

---

3. Finding Salaries Without Subqueries

a. Second Highest Salary

To find the second highest salary without using subqueries:

```
SELECT * FROM emp
ORDER BY salary DESC
LIMIT 1 OFFSET 1; -- or LIMIT 1, 1;
```

Explanation:

- `ORDER BY salary DESC` sorts the salaries in descending order.
- `LIMIT 1 OFFSET 1` skips the highest salary and retrieves the second highest.

b. Third Highest Salary

To find the third highest salary:

```
SELECT * FROM emp
ORDER BY salary DESC
LIMIT 1 OFFSET 2; -- or LIMIT 1, 2;
```

Explanation:

- This uses the same logic but skips the first two highest salaries.

c. Smallest Salary

To find the smallest salary:

```
SELECT * FROM emp
ORDER BY salary ASC
LIMIT 1;
```

Explanation:

- `ORDER BY salary ASC` sorts the salaries in ascending order, and `LIMIT 1` retrieves the smallest salary.

d. Third Smallest Salary

To find the third smallest salary:

```
SELECT * FROM emp
ORDER BY salary ASC
LIMIT 1 OFFSET 2; -- or LIMIT 1, 2;
```

Explanation:

- This orders the salaries in ascending order and skips the first two, retrieving the third smallest salary.

Summary

Understanding how to use subqueries is crucial for retrieving complex data from SQL databases. They allow you to nest queries within one another, facilitating more advanced data manipulation and retrieval. Conversely, leveraging sorting and limits can achieve similar outcomes without the overhead of additional queries.

Key Takeaways:

- **Subqueries** can be used for hierarchical data retrieval, but they can also affect performance for large datasets.
- **Direct sorting and limiting** can often be a more efficient approach for finding specific rank values.

Feel free to ask if you need more examples or details on any specific concept!

Here's a detailed overview of `LIMIT` and `OFFSET` in SQL, including their definitions, syntax, usage, and examples.

`LIMIT` and `OFFSET`

Definition

- **LIMIT**: This clause is used to specify the maximum number of records that the result set should return. It helps to control the number of rows displayed in a query result.
- **OFFSET**: This clause is used in conjunction with `LIMIT` to specify the number of rows to skip before starting to return rows from the result set.

Key Points

- The `LIMIT` clause is useful for paginating results, retrieving a specific subset of data, or controlling output for performance.
- The `OFFSET` clause allows you to skip a specified number of rows before returning results.
- Both clauses are widely used in applications that require displaying data in chunks or pages, such as in web applications.

Syntax

```sql
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name
LIMIT number_of_rows OFFSET number_of_rows_to_skip;
```

- **Example**:

  ```sql
  SELECT * FROM employees ORDER BY id LIMIT 10 OFFSET 5;
  ```

  This would return 10 rows, starting from the 6th row (skipping the first 5 rows).

Usage

1. Using LIMIT

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

Explanation:

- This query retrieves the first 5 records from the `employees` table.

2. Using LIMIT with ORDER BY

```
SELECT *
FROM employees
ORDER BY salary DESC
LIMIT 3;
```

Explanation:

- This retrieves the top 3 employees with the highest salaries.

3. Using OFFSET with LIMIT

```
SELECT *
FROM employees
ORDER BY age ASC
LIMIT 5 OFFSET 10;
```

Explanation:

- This query skips the first 10 rows and returns the next 5 rows, effectively getting the 11th to 15th employees based on age in ascending order.

Pagination Example

Pagination is a common use case for `LIMIT` and `OFFSET`. Below is an example of how to implement pagination to display employee data.

Sample Pagination Query

Assume you want to display 10 records per page.

To get page 1:

```
SELECT *
FROM employees
ORDER BY id
LIMIT 10 OFFSET 0;  -- Page 1
```

To get page 2:

```
SELECT *
FROM employees
ORDER BY id
LIMIT 10 OFFSET 10;  -- Page 2
```

To get page 3:

```
SELECT *
FROM employees
```

```sql
ORDER BY id
LIMIT 10 OFFSET 20;  -- Page 3
```

Combining LIMIT and OFFSET

You can combine both clauses in a single query to retrieve a specific subset of results:

Example

```sql
SELECT *
FROM products
ORDER BY price ASC
LIMIT 5 OFFSET 15;  -- Fetches products 16 to 20 based on price
```

Explanation:

- This retrieves 5 products, starting from the 16th product when ordered by price in ascending order.

Summary

- `LIMIT` is used to specify the maximum number of rows returned by a query.
- `OFFSET` allows you to skip a specified number of rows before returning results.
- Together, they are powerful tools for managing large datasets and implementing pagination in applications.

Here's a refined and comprehensive set of notes on SQL keys, focusing on clarity, depth, and practical applications. These notes aim to serve as an effective reference for both understanding the concepts and implementing them in SQL.

Comprehensive Notes on SQL Keys

Introduction to SQL Keys

- **Definition**: SQL keys are essential attributes in a database that help to uniquely identify rows in a table and maintain relationships between tables.
- **Importance**:
  - Ensure data integrity.
  - Optimize data retrieval.
  - Facilitate relationships between tables.

Types of SQL Keys

1. Super Key

- **Definition**: A super key is any set of columns that can uniquely identify a row in a table. It may include extra columns that are not necessary for uniqueness.
- **Characteristics**:

- Can consist of one or more columns.
- Every table must have at least one super key.

- **Example**:
  - For a table `Employees`, a super key could be `{employee_id}`, `{employee_id, name}`, or `{employee_id, age, salary}`.

2. Candidate Key

- **Definition**: A candidate key is a minimal super key, meaning it is a set of columns that can uniquely identify a row, and no subset of those columns can do so.
- **Characteristics**:
  - A table can have multiple candidate keys.
  - One candidate key is selected as the **Primary Key**.
- **Example**:
  - In the `Employees` table, both `employee_id` and `email` could serve as candidate keys if they uniquely identify each employee.

3. Primary Key

- **Definition**: The primary key is a specific candidate key selected to uniquely identify records in a table. It must contain unique values and cannot be null.
- **Characteristics**:
  - Enforces entity integrity.
  - Each table should have only one primary key.
  - Can be a single column or a composite key.
- **Example**:

```
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
);
```

- **Notes**: A primary key ensures that every record in the table can be uniquely identified.

4. Unique Key

- **Definition**: A unique key ensures that all values in a column are distinct. Unlike primary keys, unique keys can contain null values.
- **Characteristics**:
  - A table can have multiple unique keys.
  - Ensures uniqueness but allows nulls.
- **Example**:

```
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE
);
```

- **Notes**: Useful for columns that need to be unique but are not the primary means of identifying records.

5. Alternate Key

- **Definition**: An alternate key is a candidate key that is not selected as the primary key. It can uniquely identify records.
- **Characteristics**:
  - There can be multiple alternate keys.
- **Example**: If `employee_id` is the primary key, then `email` could serve as an alternate key.

6. Foreign Key

- **Definition**: A foreign key is a column or set of columns in one table that uniquely identifies a row in another table, creating a relationship between the two tables.
- **Characteristics**:
  - Can accept duplicate values and nulls.
  - Ensures referential integrity.
- **Example**:

```
CREATE TABLE Departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100)
);

CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES
Departments(department_id)
);
```

- **Notes**: Foreign keys help maintain consistency across related tables.

7. Composite Key (Compound Key)

- **Definition**: A composite key consists of two or more columns used together to uniquely identify a row in a table.
- **Characteristics**:
  - Useful when a single column cannot uniquely identify records.
- **Example**:

```
CREATE TABLE Enrollments (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```

- **Notes**: Composite keys are used in many-to-many relationships.

Summary Table of SQL Keys

Key Type	Uniqueness	Null Allowed	Description	Example
Super Key	Yes	Yes	Any set of columns that can uniquely identify rows.	{employee_id, name}
Candidate Key	Yes	No	Minimal super key; uniquely identifies rows.	{employee_id}, {email}
Primary Key	Yes	No	Chosen candidate key that uniquely identifies rows.	{employee_id}
Unique Key	Yes	Yes	Ensures all values are different; allows nulls.	{email}
Alternate Key	Yes	No	Candidate key not chosen as primary key.	{email}
Foreign Key	No	Yes	References a primary key in another table; allows duplicates.	{department_id}
Composite Key	Yes	No	Combination of two or more columns to uniquely identify rows.	{student_id, course_id}

Practical Usage of SQL Keys

Defining Keys

1. **Primary Key**: Always define a primary key for each table.

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

2. **Foreign Key**: Use foreign keys to establish relationships between tables.

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

3. **Unique Key**: Define unique keys for attributes that must remain unique but do not serve as the primary identifier.

```
CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50) UNIQUE
);
```

4. **Composite Key**: Implement composite keys when necessary to identify records uniquely using multiple columns.

```
CREATE TABLE CourseEnrollments (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```

Conclusion

Understanding SQL keys is crucial for designing robust and efficient databases. They enforce data integrity, ensure relationships between tables, and optimize data retrieval. Properly implementing and managing these keys can significantly enhance the performance and reliability of a database system.

-

```
In [ ]:
```