SQL Tutorial: Basic Commands, Creating Databases, Tables, and Inserting Data

In this tutorial, we will cover some of the basic SQL commands and concepts including how to:

- Create a database
- Create tables
- Insert data into tables
- Understand Primary and Foreign keys and their uses

Let's get started!

---

1. Creating a Database

The first step when working with SQL is to create a database. You can think of a database as a container that holds your data, in the form of tables.

Command: `CREATE DATABASE`

```
CREATE DATABASE SchoolDB;
```

This command creates a database called `SchoolDB`. After creating the database, you would typically want to use it.

Command: `USE`

```
USE SchoolDB;
```

This command selects the `SchoolDB` database as the active database, so that all subsequent commands will be executed within it.

---

2. Creating Tables

After creating a database, the next step is to create tables. Tables are used to store data in a structured way. Each table consists of columns and rows.

Command: `CREATE TABLE`

Here's an example of how to create a table for storing student information:

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,    -- Unique identifier for each student
    first_name VARCHAR(50),        -- First name of the student
    last_name VARCHAR(50),         -- Last name of the student
    date_of_birth DATE,            -- Date of birth of the student
    email VARCHAR(100)             -- Email address of the student
);
```

Explanation of the columns:

- `student_id`: An integer column, marked as the **Primary Key**. It uniquely identifies each student.
- `first_name` and `last_name`: Strings that store the student's first and last names.
- `date_of_birth`: A date column for the student's date of birth.
- `email`: A string column for the student's email address.

3. Inserting Data into a Table

Once a table is created, you can add data to it using the `INSERT INTO` statement.

Command: `INSERT INTO`

```
INSERT INTO Students (student_id, first_name, last_name, date_of_birth, email)
VALUES (1, 'John', 'Doe', '2000-05-15', 'john.doe@example.com');

INSERT INTO Students (student_id, first_name, last_name, date_of_birth, email)
VALUES (2, 'Jane', 'Smith', '2001-08-22', 'jane.smith@example.com');
```

This command inserts two rows into the `Students` table with the details for two students.

4. Querying Data

To see the data in your table, you can use the `SELECT` statement.

Command: `SELECT`

```
SELECT * FROM Students;
```

This will return all rows and columns from the `Students` table. You can also specify which columns to return:

```
SELECT first_name, last_name FROM Students;
```

This will return only the `first_name` and `last_name` columns.

---

5. Primary Key and Foreign Key

What is a Primary Key?

- A **Primary Key** is a column (or a combination of columns) that uniquely identifies each row in a table.
- It cannot contain `NULL` values.
- Each table can only have one **Primary Key**.

In the `Students` table above, `student_id` is the **Primary Key**, because each student has a unique ID. This helps to uniquely identify each student and avoid duplicate records.

Example of Primary Key:

```
CREATE TABLE Courses (
    course_id INT PRIMARY KEY,       -- Unique identifier for each course
    course_name VARCHAR(100),        -- Name of the course
```

```
    credits INT                          -- Number of credits for the course
);
```

Here, `course_id` is the **Primary Key** of the `Courses` table, ensuring each course has a unique identifier.

What is a Foreign Key?

- A **Foreign Key** is a column (or a set of columns) that establishes a relationship between two tables.
- It refers to the **Primary Key** of another table.
- The purpose of a **Foreign Key** is to maintain referential integrity between the two tables.

For example, if we want to link students to the courses they enroll in, we would need a **Foreign Key** in one table (e.g., `Enrollments`) that references the `student_id` in the `Students` table.

Example of Foreign Key:

Let's create an `Enrollments` table that tracks which student is enrolled in which course.

```
CREATE TABLE Enrollments (
    enrollment_id INT PRIMARY KEY,       -- Unique identifier for each enrollment
    student_id INT,                      -- The ID of the student
    course_id INT,                       -- The ID of the course
    enrollment_date DATE,                -- The date the student enrolled
    FOREIGN KEY (student_id) REFERENCES Students(student_id),  -- Foreign Key
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)     -- Foreign Key
);
```

Here:

- `student_id` is a **Foreign Key** that refers to the `student_id` in the `Students` table.
- `course_id` is a **Foreign Key** that refers to the `course_id` in the `Courses` table.

By using foreign keys, we ensure that every enrollment record corresponds to a valid student and a valid course.

---

6. Why Use Primary and Foreign Keys?

Primary Key:

- **Uniqueness**: The primary key guarantees that each record in a table is unique.
- **Data Integrity**: It helps maintain consistency and accuracy in the database by ensuring no duplicate rows.

Foreign Key:

- **Referential Integrity**: A foreign key ensures that a record in one table corresponds to a valid record in another table, maintaining the logical relationships between tables.
- **Prevent Orphan Records**: By enforcing the foreign key relationship, you prevent "orphan" records (e.g., an enrollment in a non-existent student or course).

---

7. Updating and Deleting Data

You can also update or delete data in a table.

Update Data:

```
UPDATE Students
SET email = 'john.newemail@example.com'
WHERE student_id = 1;
```

This command updates the email address of the student with `student_id` 1.

Delete Data:

```
DELETE FROM Students
WHERE student_id = 2;
```

This command deletes the student with `student_id` 2 from the `Students` table.

8. Alter Table

The `ALTER TABLE` command allows you to modify the structure of an existing table, such as adding, removing, or changing columns.

Add a Column

```
ALTER TABLE Students
ADD phone_number VARCHAR(20);
```

This command adds a new column `phone_number` to the `Students` table.

Modify a Column

You can change the definition of an existing column, for example, changing the data type of `phone_number` from `VARCHAR(20)` to `VARCHAR(30)`.

```
ALTER TABLE Students
MODIFY phone_number VARCHAR(30);
```

Drop a Column

To remove a column from a table:

```
ALTER TABLE Students
DROP COLUMN phone_number;
```

Rename a Column

```
ALTER TABLE Students
RENAME COLUMN email TO student_email;
```

---

9. Dropping a Table

If you no longer need a table, you can remove it using the `DROP TABLE` command.

Command: `DROP TABLE`

```
DROP TABLE Enrollments;
```

This command deletes the `Enrollments` table and all of its data permanently. **Be cautious** when using `DROP` as it cannot be undone.

---

10. Indexing

An index in SQL is used to speed up query execution. It is particularly useful for columns that are frequently searched or used in join conditions. While creating an index may improve read performance, it can degrade write performance (due to the additional work involved in updating the index).

Creating an Index

```
CREATE INDEX idx_lastname ON Students(last_name);
```

This creates an index on the `last_name` column of the `Students` table. Now, searches based on `last_name` will be faster.

Dropping an Index

To remove an index, you can use the `DROP INDEX` command.

```
DROP INDEX idx_lastname;
```

Note that the syntax for dropping an index may vary slightly depending on the database system (e.g., MySQL, PostgreSQL, SQL Server).

---

11. Constraints

Constraints are rules that help ensure the integrity of data in your database. We already touched on **Primary Keys** and **Foreign Keys**, but here are some more common constraints.

NOT NULL

A `NOT NULL` constraint ensures that a column cannot have a `NULL` value.

```
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,    -- Name cannot be NULL
    department VARCHAR(50)         -- Department can be NULL
);
```

UNIQUE

The `UNIQUE` constraint ensures that all values in a column are distinct.

```
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE   -- Email must be unique
);
```

CHECK

The `CHECK` constraint ensures that the values in a column meet a specific condition.

```
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    price DECIMAL(10, 2),
    CHECK (price > 0)   -- Price must be greater than 0
);
```

DEFAULT

The `DEFAULT` constraint is used to provide a default value for a column if no value is specified during an `INSERT`.

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    order_date DATE DEFAULT CURRENT_DATE   -- Default to the current date
);
```

---

12. Joins

In SQL, joins are used to combine rows from two or more tables based on a related column. Here are the most common types of joins:

INNER JOIN

The `INNER JOIN` returns only the rows where there is a match in both tables.

```
SELECT Students.first_name, Students.last_name, Courses.course_name
FROM Students
INNER JOIN Enrollments ON Students.student_id = Enrollments.student_id
INNER JOIN Courses ON Enrollments.course_id = Courses.course_id;
```

This query returns a list of students along with the courses they are enrolled in.

LEFT JOIN (or LEFT OUTER JOIN)

The `LEFT JOIN` returns all rows from the left table (in this case, `Students`) and the matching rows from the right table (`Courses`). If there's no match, the result will contain `NULL` for the right table's columns.

```
SELECT Students.first_name, Students.last_name, Courses.course_name
FROM Students
LEFT JOIN Enrollments ON Students.student_id = Enrollments.student_id
LEFT JOIN Courses ON Enrollments.course_id = Courses.course_id;
```

This will return all students, including those who are not enrolled in any courses (in which case `course_name` will be `NULL`).

RIGHT JOIN (or RIGHT OUTER JOIN)

The `RIGHT JOIN` is the opposite of the `LEFT JOIN` — it returns all rows from the right table (e.g., `Courses`) and the matching rows from the left table (e.g., `Students`). If no match exists, `NULL` values will appear for the left table's columns.

```
SELECT Students.first_name, Students.last_name, Courses.course_name
FROM Students
RIGHT JOIN Enrollments ON Students.student_id = Enrollments.student_id
RIGHT JOIN Courses ON Enrollments.course_id = Courses.course_id;
```

FULL OUTER JOIN

A `FULL OUTER JOIN` returns all rows when there is a match in either the left or right table. Rows with no match will have `NULL` values for the non-matching side.

```
SELECT Students.first_name, Students.last_name, Courses.course_name
FROM Students
FULL OUTER JOIN Enrollments ON Students.student_id = Enrollments.student_id
FULL OUTER JOIN Courses ON Enrollments.course_id = Courses.course_id;
```

---

13. Group By and Aggregate Functions

SQL provides aggregate functions like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()` to perform calculations on data. You often use these functions in combination with the `GROUP BY` clause.

COUNT

```
SELECT COUNT(*) AS total_students FROM Students;
```

This returns the total number of students in the `Students` table.

SUM

```
SELECT course_id, SUM(credits) AS total_credits
FROM Courses
GROUP BY course_id;
```

This query calculates the total number of credits for each course.

AVG

```
SELECT AVG(price) AS average_price
FROM Products;
```

This returns the average price of all products in the `Products` table.

GROUP BY

The `GROUP BY` statement groups rows that have the same values into summary rows, like "total" or "average" calculations.

```
SELECT student_id, COUNT(course_id) AS total_courses
FROM Enrollments
```

```
GROUP BY student_id;
```

This query gives the number of courses each student is enrolled in.

---

14. HAVING Clause

The `HAVING` clause is used to filter records after the `GROUP BY` has been applied. It is similar to the `WHERE` clause but works with grouped data.

```
SELECT student_id, COUNT(course_id) AS total_courses
FROM Enrollments
GROUP BY student_id
HAVING COUNT(course_id) > 2;
```

This query returns the students who are enrolled in more than 2 courses.

---

15. Subqueries

A subquery is a query inside another query. Subqueries can be used in the `WHERE`, `FROM`, or `SELECT` clauses.

Subquery in WHERE Clause

```
SELECT first_name, last_name
FROM Students
WHERE student_id IN (SELECT student_id FROM Enrollments WHERE course_id = 101);
```

This query returns the names of students who are enrolled in the course with `course_id` 101.

Subquery in FROM Clause

```
SELECT subquery.student_id, subquery.total_courses
FROM (SELECT student_id, COUNT(course_id) AS total_courses
      FROM Enrollments
      GROUP BY student_id) AS subquery
WHERE subquery.total_courses > 2;
```

This query returns students who are enrolled in more than 2 courses, using a subquery in the `FROM` clause.

Certainly! Let's continue exploring more advanced SQL commands and concepts that are valuable for working with databases and managing data. Here are additional SQL topics and commands you may find useful.

---

16. Transactions

Transactions allow you to group multiple SQL statements into a single unit of work. If something goes wrong during the transaction, you can roll back the entire operation, ensuring data consistency.

Start a Transaction

```
BEGIN TRANSACTION;
```

This begins a new transaction. After this point, you can execute several SQL commands.

Commit a Transaction

```
COMMIT;
```

The `COMMIT` command saves all changes made during the transaction. Once committed, the changes become permanent.

Rollback a Transaction

```
ROLLBACK;
```

The `ROLLBACK` command undoes all changes made during the transaction, effectively reverting the database to its state before the `BEGIN TRANSACTION` was issued.

Example of Using Transactions

```
BEGIN TRANSACTION;

UPDATE Students
SET email = 'new.email@example.com'
WHERE student_id = 1;

DELETE FROM Enrollments
WHERE student_id = 1;

COMMIT;
```

In this example, if anything goes wrong (e.g., a constraint violation), you can roll back the transaction and none of the changes will be applied.

---

17. Views

A **view** is a virtual table created by a query that combines data from one or more tables. It doesn't store data itself but presents a result set as if it were a table.

Creating a View

```
CREATE VIEW StudentCourses AS
SELECT Students.first_name, Students.last_name, Courses.course_name
FROM Students
INNER JOIN Enrollments ON Students.student_id = Enrollments.student_id
INNER JOIN Courses ON Enrollments.course_id = Courses.course_id;
```

This creates a view named `StudentCourses` that returns the list of students and the courses they are enrolled in.

Using a View

Once the view is created, you can query it just like a table.

```
SELECT * FROM StudentCourses;
```

Dropping a View

To remove a view, you use the `DROP VIEW` command.

```
DROP VIEW StudentCourses;
```

---

18. Stored Procedures

A **stored procedure** is a precompiled collection of SQL statements that you can execute on the database. It allows for more complex operations, reusability, and abstraction from the end user.

Creating a Stored Procedure

```
CREATE PROCEDURE GetStudentCourses (IN student_id INT)
BEGIN
    SELECT Students.first_name, Students.last_name, Courses.course_name
    FROM Students
    INNER JOIN Enrollments ON Students.student_id = Enrollments.student_id
    INNER JOIN Courses ON Enrollments.course_id = Courses.course_id
    WHERE Students.student_id = student_id;
END;
```

This stored procedure returns the courses for a given student. The `IN` parameter `student_id` is used to pass the value to the procedure.

Executing a Stored Procedure

To execute the stored procedure:

```
CALL GetStudentCourses(1);
```

This will execute the procedure for the student with `student_id = 1`.

Dropping a Stored Procedure

To remove a stored procedure from the database:

```
DROP PROCEDURE GetStudentCourses;
```

---

19. Triggers

A **trigger** is a special kind of stored procedure that automatically runs when a specific event (like `INSERT`, `UPDATE`, or `DELETE`) occurs on a table.

Creating a Trigger

For example, let's create a trigger that automatically updates the `last_updated` timestamp in the `Students` table whenever a student's email address is changed.

```
CREATE TRIGGER UpdateLastUpdated
BEFORE UPDATE ON Students
FOR EACH ROW
BEGIN
    IF OLD.email <> NEW.email THEN
        SET NEW.last_updated = NOW();
    END IF;
END;
```

This trigger runs **before** any update on the `Students` table. If the `email` field is updated, the `last_updated` column is set to the current timestamp.

Dropping a Trigger

To remove a trigger:

```
DROP TRIGGER UpdateLastUpdated;
```

---

20. User-Defined Functions (UDFs)

A **user-defined function** (UDF) is a function that you can define and use to perform calculations or data transformations. It can return a single value (scalar function) or a table (table-valued function).

Creating a Simple Function

For example, a function to calculate the full name of a student:

```
CREATE FUNCTION GetFullName(first_name VARCHAR(50), last_name VARCHAR(50))
RETURNS VARCHAR(100)
BEGIN
    RETURN CONCAT(first_name, ' ', last_name);
END;
```

This function takes two arguments (`first_name` and `last_name`) and returns them as a concatenated string (full name).

Using the Function

```
SELECT GetFullName(first_name, last_name) AS full_name
FROM Students;
```

This will return the full name for each student.

Dropping a Function

```
DROP FUNCTION GetFullName;
```

---

21. Normalization and Denormalization

Normalization

Normalization is the process of organizing a database to minimize redundancy and dependency. It involves breaking down tables into smaller ones and establishing relationships between them using foreign keys. The goal is to reduce data duplication and maintain data integrity.

Common normal forms are:

- **1NF (First Normal Form)**: No repeating groups or arrays in a column; each column contains atomic values.
- **2NF (Second Normal Form)**: Achieves 1NF and removes partial dependencies (non-prime attributes should be fully dependent on the primary key).
- **3NF (Third Normal Form)**: Achieves 2NF and removes transitive dependencies (no non-prime attribute depends on another non-prime attribute).

Example of normalization:

- You might split a table with `student_id`, `first_name`, `last_name`, and `course_name` into two tables: one for students and another for courses, with a third table for enrollments to capture the many-to-many relationship between students and courses.

Denormalization

Denormalization is the process of combining tables that have been split through normalization. It can improve query performance, but it may lead to redundant data.

Example: In a highly normalized system, you might denormalize data by storing a student's full name along with course information in a single table for fast queries at the cost of data redundancy.

---

22. Window Functions

Window functions perform calculations across a set of rows that are related to the current row. They are often used for tasks like calculating running totals, ranking rows, or finding moving averages.

ROW_NUMBER()

The `ROW_NUMBER()` function assigns a unique number to each row in a result set, based on a specified ordering.

```
SELECT student_id, first_name, last_name,
       ROW_NUMBER() OVER (ORDER BY last_name) AS row_num
FROM Students;
```

This query returns a list of students, along with a unique row number assigned based on their last name.

RANK() and DENSE_RANK()

These functions are used to assign ranks to rows, with `RANK()` leaving gaps in the ranking sequence when there are ties, while `DENSE_RANK()` does not.

```
SELECT student_id, first_name, last_name,
       RANK() OVER (ORDER BY first_name) AS rank
```

```
FROM Students;
```

This query ranks students based on their first name.

SUM() with OVER() for Running Totals

```
SELECT student_id, first_name, last_name, grade,
       SUM(grade) OVER (ORDER BY student_id) AS running_total
FROM Students;
```

This query calculates a running total of grades for each student, ordered by `student_id`.

---

23. Common Table Expressions (CTEs)

A **Common Table Expression (CTE)** is a temporary result set that can be referred to within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement.

Creating a Simple CTE

```
WITH StudentCourses AS (
    SELECT Students.first_name, Students.last_name, Courses.course_name
    FROM Students
    INNER JOIN Enrollments ON Students.student_id = Enrollments.student_id
    INNER JOIN Courses ON Enrollments.course_id = Courses.course_id
)
SELECT * FROM StudentCourses;
```

The `WITH` clause defines a CTE named `StudentCourses`, which is then used in the `SELECT` statement.

Recursive CTEs

A **recursive CTE** is used when you need to perform operations like hierarchical data retrieval (e.g., employee-manager relationships).

```
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT employee_id, manager_id, name
    FROM Employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.employee_id, e.manager_id, e.name
    FROM Employees e
    INNER JOIN EmployeeHierarchy eh ON e.manager_id = eh.employee_id
)
SELECT * FROM EmployeeHierarchy;
```

This recursive CTE retrieves the hierarchy of employees in an organization, starting with employees who have no manager (`manager_id IS NULL`), then recursively including those who report to them.

Conclusion

We've now covered even more advanced SQL topics, including:

- **Transactions** for managing groups of SQL operations.
- **Views** to simplify complex queries.
- **Stored Procedures** for reusable SQL code.
- **Triggers** for automatically executing actions on data changes.
- **User-Defined Functions (UDFs)** for custom calculations.
- **Normalization and Denormalization** for designing efficient database schemas.
- **Window Functions** for advanced row-based calculations like running

Intermediate SQL Tutorial

In this intermediate SQL tutorial, we'll explore more advanced concepts that will help you work with databases more effectively. These topics will cover practical applications of SQL in the real world, including joins, subqueries, window functions, advanced filtering, and more.

---

1. Advanced Joins

As you've seen in basic SQL, joins are used to combine data from multiple tables. Let's dive into more advanced join techniques, including `SELF JOIN`, `CROSS JOIN`, and using `JOIN` with aggregate functions.

1.1 SELF JOIN

A **self join** is a join where a table is joined with itself. It's useful when you have hierarchical data in a table, such as employees reporting to managers.

Example:

Imagine you have an `Employees` table with the following columns:

- `employee_id`
- `name`
- `manager_id` (which is the `employee_id` of their manager)

To list employees and their managers:

```
SELECT e1.name AS employee, e2.name AS manager
FROM Employees e1
LEFT JOIN Employees e2 ON e1.manager_id = e2.employee_id;
```

Here:

- `e1` is the alias for the employee.
- `e2` is the alias for the manager (which is also an employee in this case).
- The `LEFT JOIN` ensures that employees without managers are also included in the result.

1.2 CROSS JOIN

A **CROSS JOIN** returns the Cartesian product of two tables. It will return all possible combinations of rows from the two tables.

Example:

```
SELECT Products.product_name, Categories.category_name
FROM Products
CROSS JOIN Categories;
```

This query returns every combination of products and categories. **Be cautious** with `CROSS JOIN`, as it can generate a large result set.

1.3 JOIN with Aggregate Functions

You can use aggregate functions like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()` with joins to summarize data.

Example:

Let's say we want to know how many students are enrolled in each course. We have two tables: `Courses` and `Enrollments`.

```
SELECT c.course_name, COUNT(e.student_id) AS number_of_students
FROM Courses c
LEFT JOIN Enrollments e ON c.course_id = e.course_id
GROUP BY c.course_name;
```

Here:

- `COUNT(e.student_id)` counts the number of students per course.
- `GROUP BY` is used to group the result by `course_name`.

---

2. Subqueries

Subqueries (also known as nested queries) are queries within queries. They can be used in `WHERE`, `FROM`, and `SELECT` clauses to perform complex operations.

2.1 Subqueries in the WHERE Clause

A subquery can be used in the `WHERE` clause to filter results based on values from another table.

Example:

Let's find students who are enrolled in courses with more than 30 students:

```
SELECT student_id, first_name, last_name
FROM Students
WHERE student_id IN (
    SELECT student_id
    FROM Enrollments
    GROUP BY student_id
    HAVING COUNT(course_id) > 30
);
```

2.2 Subqueries in the FROM Clause

You can use a subquery in the `FROM` clause to create a temporary table for further analysis.

Example:

```
SELECT temp.student_id, temp.total_courses
FROM (
    SELECT student_id, COUNT(course_id) AS total_courses
    FROM Enrollments
    GROUP BY student_id
) AS temp
WHERE temp.total_courses > 2;
```

In this example, the subquery in the `FROM` clause calculates the total number of courses each student is enrolled in. The outer query filters for students enrolled in more than two courses.

2.3 Correlated Subqueries

A **correlated subquery** references columns from the outer query. The subquery is executed for each row in the outer query.

Example:

Find students who are enrolled in courses with a total enrollment greater than 50.

```
SELECT s.first_name, s.last_name
FROM Students s
WHERE EXISTS (
    SELECT 1
    FROM Enrollments e
    WHERE e.student_id = s.student_id
    GROUP BY e.course_id
    HAVING COUNT(e.student_id) > 50
);
```

In this query, the subquery is executed once for each student in the outer query. The `EXISTS` operator checks if the subquery returns any rows (i.e., if the student is enrolled in a course with more than 50 students).

---

3. Window Functions

Window functions are advanced functions that perform calculations across a set of table rows related to the current row. They are useful for tasks such as running totals, moving averages, and ranking.

3.1 ROW_NUMBER()

The `ROW_NUMBER()` function assigns a unique integer to rows in a result set, based on a specified order.

Example:

Let's rank students based on their total grade in descending order:

```
SELECT student_id, first_name, last_name, total_grade,
       ROW_NUMBER() OVER (ORDER BY total_grade DESC) AS rank
FROM Students;
```

This assigns a rank to each student, where the student with the highest `total_grade` gets rank 1.

3.2 RANK() and DENSE_RANK()

- `RANK()` assigns a rank to each row, but if there are ties (i.e., rows with the same value), it leaves gaps in the rank numbers.
- `DENSE_RANK()` also handles ties but does not leave gaps in the rank numbers.

Example:

```
SELECT student_id, first_name, last_name, total_grade,
       RANK() OVER (ORDER BY total_grade DESC) AS rank,
       DENSE_RANK() OVER (ORDER BY total_grade DESC) AS dense_rank
FROM Students;
```

Here, students with the same `total_grade` will receive the same rank. However, `RANK()` will leave gaps in the rank (e.g., 1, 1, 3), while `DENSE_RANK()` will not (e.g., 1, 1, 2).

3.3 SUM() with OVER() for Running Totals

The `SUM()` window function can calculate running totals across rows.

Example:

```
SELECT student_id, total_grade,
       SUM(total_grade) OVER (ORDER BY student_id) AS running_total
FROM Students;
```

This query calculates the running total of `total_grade` for each student, ordered by `student_id`.

3.4 PARTITION BY with Window Functions

You can partition the result set into groups, and the window function will operate on each group separately.

Example:

Let's calculate the rank of students within each course.

```
SELECT course_id, student_id, total_grade,
       RANK() OVER (PARTITION BY course_id ORDER BY total_grade DESC) AS rank
FROM Enrollments;
```

This query calculates the rank of students within each course, ordered by their `total_grade`.

---

4. Grouping and Aggregating Data

4.1 GROUP BY with Multiple Columns

You can group data by more than one column to perform more complex aggregations.

Example:

Let's calculate the number of students enrolled in each course, broken down by department:

```
SELECT c.course_name, c.department, COUNT(e.student_id) AS number_of_students
FROM Courses c
LEFT JOIN Enrollments e ON c.course_id = e.course_id
GROUP BY c.course_name, c.department;
```

This groups the results by both `course_name` and `department`.

4.2 HAVING Clause

The `HAVING` clause is used to filter results after aggregation has occurred (like `GROUP BY`). It is similar to the `WHERE` clause but works with grouped data.

Example:

Find courses with more than 10 students enrolled:

```
SELECT c.course_name, COUNT(e.student_id) AS number_of_students
FROM Courses c
LEFT JOIN Enrollments e ON c.course_id = e.course_id
GROUP BY c.course_name
HAVING COUNT(e.student_id) > 10;
```

The `HAVING` clause filters groups where the number of students is greater than 10.

---

5. Advanced Filtering Techniques

5.1 Using `IN` with Subqueries

You can use the `IN` operator to check whether a value exists in a set of values returned by a subquery.

Example:

Find students who are enrolled in any of the courses taught by a particular instructor (let's say instructor_id = 5):

```
SELECT s.student_id, s.first_name, s.last_name
FROM Students s
WHERE s.student_id IN (
    SELECT e.student_id
    FROM Enrollments e
    INNER JOIN Courses c ON e.course_id = c.course_id
    WHERE c.instructor_id = 5
);
```

This subquery returns the `student_id` values of students enrolled in courses taught by instructor 5, and the main query uses this to filter students.

5.2 Using `EXISTS`

The `EXISTS` operator checks whether a subquery returns any rows. It's often used when you want to test for the existence of related data.

Example:

Find students who are enrolled in at least one course:

```
SELECT s.student_id, s.first_name, s.last_name
FROM Students s
WHERE EXISTS (
    SELECT 1
    FROM Enrollments e
    WHERE e

.student_id = s.student_id
);
```

This query returns only the students who are enrolled in at least one course.

5.3 `LIKE` and Wildcards

The `LIKE` operator is used for pattern matching with wildcards.

- `%` matches zero or more characters.
- `_` matches exactly one character.

Example:

Find students whose last name starts with "S":

```
SELECT first_name, last_name
FROM Students
WHERE last_name LIKE 'S%';
```

This query matches all students whose last name starts with "S".

Conclusion

In this **Intermediate SQL Tutorial**, we've covered the following concepts:

- **Advanced Joins**: Using self joins, cross joins, and joins with aggregate functions.
- **Subqueries**: Subqueries in `WHERE`, `FROM`, and `SELECT` clauses, including correlated subqueries.
- **Window Functions**: Using `ROW_NUMBER()`, `RANK()`, and `SUM()` with `OVER()` and `PARTITION BY` for running totals and rankings.
- **Grouping and Aggregating**: Using `GROUP BY` with multiple columns, `HAVING` for post-aggregation filtering, and aggregate functions.
- **Advanced Filtering**: Using `IN`, `EXISTS`, and pattern matching with `LIKE`.

These intermediate techniques provide the building blocks for writing more complex and efficient SQL queries. As you practice these concepts, you'll be able to handle more sophisticated data manipulation and analysis tasks.