SQL Notes

Asaad Cheema

SQL Commands and Operations

Below are detailed SQL commands based on the provided examples:

Stopping Database Alterations

To make a database read-only:

```sql
ALTER DATABASE myDB READ_ONLY = 1;
```

Creating a Table

Example of creating a table named `employees`:

```sql
CREATE TABLE employees (
    employee_id INT,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    hourly_pay DECIMAL(5,2),
    hire_date DATE
);
```

Renaming a Table

Rename the table `employee` to `worker`:

```sql
RENAME TABLE employee TO worker;
```

Altering a Table

Adding a Column

Add a `phone_number` column to `employees`:

```sql
ALTER TABLE employees
ADD phone_number VARCHAR(15);
```

Renaming a Column

Rename `phone_number` to `email`:

```sql
ALTER TABLE employees
RENAME COLUMN phone_number TO email;
```

Modifying a Column

Change the datatype of `email` to `VARCHAR(100)`:

```
ALTER TABLE employees
MODIFY COLUMN email VARCHAR(100);
```

Move the `email` column to be after `last_name`:

```
ALTER TABLE employees
MODIFY COLUMN email VARCHAR(100)
AFTER last_name;
```

Dropping a Column

Remove the `email` column:

```
ALTER TABLE employees
DROP COLUMN email;
```

Inserting Data

Inserting All Columns

Insert a complete record into `employees`:

```
INSERT INTO employees
VALUES (1, 'squid', 'krab', 5.50, '2023-02-01');
```

Inserting Specific Columns

Insert a record with specific columns:

```
INSERT INTO employees (employee_id, first_name, last_name, hire_date)
VALUES (1, 'squid', 'krab', '2023-02-01');
```

Selecting Data

Selecting All Records

Retrieve all records from `employees`:

```
SELECT * FROM employees;
```

Selecting Records with Conditions

Find records where `hire_date` is `NULL`:

```
SELECT * FROM employees
WHERE hire_date IS NULL;
```

Updating Data

Update the `hourly_pay` for an employee with `employee_id = 1`:

```
UPDATE employees
SET hourly_pay = 10.25
WHERE employee_id = 1;
```

Deleting Data

Delete a record with `employee_id = 6`:

```sql
DELETE FROM employees
WHERE employee_id = 6;
```

Transaction Control

Disabling Autocommit

Disable `AUTOCOMMIT`:

```sql
SET AUTOCOMMIT = OFF;
```

Committing Changes

Commit all changes made during the transaction:

```sql
COMMIT;
```

Rolling Back Changes

Roll back to the last committed state:

```sql
ROLLBACK;
```

Unique Constraints

Creating a Unique Column During Table Creation

A unique column ensures that no duplicate values exist in the column. Example:

```sql
CREATE TABLE products (
    product_id INT,
    product_name VARCHAR(100) UNIQUE,
    price DECIMAL(10,2)
);
```

Adding a Unique Constraint to an Existing Table

To make an existing column unique after table creation:

```sql
ALTER TABLE products
ADD CONSTRAINT unique_product_name UNIQUE (product_name);
```

Example Usage

Consider a table `products` where we ensure that `product_name` is unique:

```sql
-- Add unique constraint to an existing column
ALTER TABLE products
ADD CONSTRAINT unique_product_name UNIQUE (product_name);

-- Attempting to insert duplicate product names will fail
INSERT INTO products (product_id, product_name, price)
VALUES (1, 'Laptop', 999.99);

-- This will throw an error as 'Laptop' already exists
INSERT INTO products (product_id, product_name, price)
VALUES (2, 'Laptop', 899.99);
```

NOT NULL Constraint

Defining a Column as NOT NULL During Table Creation

The `NOT NULL` constraint ensures that a column cannot have `NULL` values. Example:

```sql
CREATE TABLE orders (
    order_id INT NOT NULL,
    customer_name VARCHAR(100) NOT NULL,
    order_date DATE
);
```

Adding NOT NULL Constraint to an Existing Column

To make an existing column `NOT NULL`:

```sql
ALTER TABLE orders
MODIFY COLUMN customer_name VARCHAR(100) NOT NULL;
```

Example Usage

Consider a table `orders` where the `customer_name` column must always have a value:

```sql
-- Add a NOT NULL constraint
ALTER TABLE orders
MODIFY COLUMN customer_name VARCHAR(100) NOT NULL;

-- Attempting to insert a row without customer_name will fail
INSERT INTO orders (order_id, order_date)
VALUES (1, '2023-01-01'); -- Error: customer_name cannot be NULL
```

CHECK Constraint

Defining a Column with a CHECK Constraint During Table Creation

The `CHECK` constraint ensures that a column's values meet a specific condition. Example:

```sql
CREATE TABLE employees (
    employee_id INT,
    employee_name VARCHAR(100),
    age INT CHECK (age >= 18),
    salary DECIMAL(10,2) CHECK (salary > 0)
);
```

Adding a CHECK Constraint to an Existing Column

To add a `CHECK` constraint to an existing column:

```sql
ALTER TABLE employees
ADD CONSTRAINT check_age CHECK (age >= 18);
```

Example Usage

Consider a table `employees` where the `age` column must always have a value of 18 or greater:

```sql
-- Add a CHECK constraint
ALTER TABLE employees
ADD CONSTRAINT check_age CHECK (age >= 18);

-- Attempting to insert a row with age less than 18 will fail
INSERT INTO employees (employee_id, employee_name, age, salary)
VALUES (1, 'John Doe', 17, 3000.00); -- Error: CHECK constraint failed
```

DEFAULT Constraint

Defining a Column with a DEFAULT Value During Table Creation

The DEFAULT constraint assigns a default value to a column if no value is specified during insertion. Example:

```
CREATE TABLE employees (
    employee_id INT,
    employee_name VARCHAR(100),
    department VARCHAR(50) DEFAULT 'General',
    salary DECIMAL(10,2) DEFAULT 3000.00
);
```

Adding a DEFAULT Constraint to an Existing Column

To add a DEFAULT constraint to an existing column:

```
ALTER TABLE employees
ALTER COLUMN department SET DEFAULT 'General';
```

Example Usage

Consider a table employees where the department column has a default value of 'General':

```
-- Insert without specifying the department
INSERT INTO employees (employee_id, employee_name, salary)
VALUES (1, 'John Doe', 5000.00);

-- The department will automatically be set to 'General'.
SELECT * FROM employees;
```

PRIMARY KEY Constraint

Defining a Primary Key During Table Creation

The PRIMARY KEY constraint uniquely identifies each record in a table. Example:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    order_date DATE
);
```

Adding a Primary Key to an Existing Table

To add a PRIMARY KEY constraint to an existing table:

```
ALTER TABLE orders
ADD CONSTRAINT pk_order_id PRIMARY KEY (order_id);
```

Example Usage

Consider a table orders where the order_id column is the primary key:

```
-- Insert valid rows
INSERT INTO orders (order_id, customer_name, order_date)
VALUES (1, 'Alice', '2023-01-01');

-- Attempting to insert a duplicate primary key will fail
INSERT INTO orders (order_id, customer_name, order_date)
VALUES (1, 'Bob', '2023-01-02'); -- Error: Duplicate entry for primary key
```

FOREIGN KEY Constraints

Defining a FOREIGN KEY During Table Creation

The `FOREIGN KEY` constraint establishes a relationship between two tables. Example:

```sql
CREATE TABLE orders (
    order_id INT,
    customer_id INT,
    order_date DATE,
    PRIMARY KEY (order_id),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

Adding a FOREIGN KEY to an Existing Table

To add a `FOREIGN KEY` constraint to an existing table:

```sql
ALTER TABLE orders
ADD CONSTRAINT fk_customer FOREIGN KEY (customer_id)
REFERENCES customers(customer_id);
```

Example Usage

Consider a table `orders` linked to a table `customers` through a foreign key:

```sql
-- Insert records into customers
INSERT INTO customers (customer_id, customer_name)
VALUES (1, 'John Doe'), (2, 'Jane Smith');

-- Insert records into orders
INSERT INTO orders (order_id, customer_id, order_date)
VALUES (101, 1, '2023-01-01'), (102, 2, '2023-01-02');
```

SQL Joins

INNER JOIN

Retrieve records with matching values in both tables:

```sql
SELECT orders.order_id, customers.customer_name
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

LEFT JOIN

Retrieve all records from the left table and matching records from the right table:

```sql
SELECT orders.order_id, customers.customer_name
FROM orders
LEFT JOIN customers ON orders.customer_id = customers.customer_id;
```

RIGHT JOIN

Retrieve all records from the right table and matching records from the left table:

```sql
SELECT orders.order_id, customers.customer_name
FROM orders
RIGHT JOIN customers ON orders.customer_id = customers.customer_id;
```

FULL JOIN

Retrieve all records when there is a match in either table:

```sql
SELECT orders.order_id, customers.customer_name
FROM orders
FULL OUTER JOIN customers ON orders.customer_id = customers.customer_id;
```

Logical Operators: AND, OR, NOT

Logical operators are used in SQL to combine conditions in the `WHERE` clause.

AND

The `AND` operator combines conditions and returns rows where all conditions are true.

```sql
SELECT * FROM employees
WHERE department = 'HR' AND employee_name = 'Alice';
```

OR

The `OR` operator combines conditions and returns rows where at least one condition is true.

```sql
SELECT * FROM employees
WHERE department = 'HR' OR department = 'IT';
```

NOT

The `NOT` operator negates a condition and returns rows where the condition is false.

```sql
SELECT * FROM employees
WHERE NOT department = 'HR';
```

Combined Example

You can combine `AND`, `OR`, and `NOT` operators for complex conditions.

```sql
SELECT * FROM employees
WHERE (department = 'HR' OR department = 'IT')
AND NOT employee_name = 'Alice';
```

This query retrieves all employees from the HR or IT departments except those named Alice.

SQL Functions

SQL provides various functions to perform calculations on data. Some commonly used functions are:

Aggregate Functions

- `SUM()`: Calculates the total of a numeric column.
- `COUNT()`: Counts the number of rows.
- `AVG()`: Calculates the average value of a numeric column.
- `MAX()`: Returns the maximum value in a column.
- `MIN()`: Returns the minimum value in a column.

Examples

```
-- Calculate the total salary of all employees
SELECT SUM(salary) AS total_salary FROM employees;

-- Count the number of employees in each department
SELECT department, COUNT(*) AS num_employees
FROM employees
GROUP BY department;

-- Find the average salary in the company
SELECT AVG(salary) AS average_salary FROM employees;

-- Get the maximum and minimum salaries in the company
SELECT MAX(salary) AS highest_salary, MIN(salary) AS lowest_salary
FROM employees;
```

Wildcards with LIKE

Wildcards are used with the `LIKE` operator to search for patterns in text columns.

Wildcards

- %: Represents zero or more characters.
- _: Represents a single character.

Examples

```
-- Find employees whose names start with 'A'
SELECT * FROM employees
WHERE employee_name LIKE 'A%';

-- Find employees whose names end with 'n'
SELECT * FROM employees
WHERE employee_name LIKE '%n';

-- Find employees whose names contain 'ali'
SELECT * FROM employees
WHERE employee_name LIKE '%ali%';

-- Find employees whose names have 'a' as the second character
SELECT * FROM employees
WHERE employee_name LIKE '_a%';
```

Combined Example

Using functions and wildcards together:

```
-- Find the total salary of employees whose names start with 'A'
SELECT SUM(salary) AS total_salary
FROM employees
WHERE employee_name LIKE 'A%';

-- Count the number of employees in departments starting with 'S'
SELECT COUNT(*) AS num_employees
FROM employees
WHERE department LIKE 'S%';
```

LIMIT Clause

The `LIMIT` clause is used to restrict the number of rows returned by a query. It can be combined with the `OFFSET` keyword to skip a specific number of rows before starting to return rows.

Examples

```sql
-- Retrieve the first 5 rows from the employees table
SELECT * FROM employees
LIMIT 5;

-- Retrieve the next 5 rows after skipping the first 5 rows
SELECT * FROM employees
LIMIT 5 OFFSET 5;

-- Retrieve rows 11 to 20 from the employees table
SELECT * FROM employees
LIMIT 10 OFFSET 10;

-- MySQL-specific syntax to retrieve rows 11 to 20
SELECT * FROM employees
LIMIT 10, 10;
```

Note

In MySQL, `LIMIT 10 OFFSET 10` is equivalent to `LIMIT 10, 10`. While the former is widely supported across various databases, the latter is specific to MySQL.

ORDER BY Clause

The `ORDER BY` clause is used to sort the result set in ascending or descending order.

Examples

```sql
-- Retrieve all employees sorted by their names in ascending order
SELECT * FROM employees
ORDER BY employee_name ASC;

-- Retrieve employees sorted by salary in descending order
SELECT * FROM employees
ORDER BY salary DESC;
```

GROUP BY Clause

The `GROUP BY` clause groups rows that have the same values in specified columns. It is often used with aggregate functions (Where is replaced by Having).

Examples

```sql
-- Count the number of employees in each department
SELECT department, COUNT(*) AS num_employees
FROM employees
GROUP BY department;

-- Calculate the total salary for each department
SELECT department, SUM(salary) AS total_salary
FROM employees
GROUP BY department;
```

Indexes in MySQL

Indexes are used to optimize the performance of queries by allowing faster data retrieval. MySQL supports several types of indexes:

Types of Indexes

- **PRIMARY KEY**: A unique index where each value must be unique and cannot be NULL.
- **UNIQUE**: Ensures all values in the indexed column are unique.
- **FULLTEXT**: Used for text searching.
- **INDEX (Non-unique)**: Speeds up data retrieval but does not enforce uniqueness.

Creating Indexes

```sql
-- Create a simple index
CREATE INDEX idx_employee_name ON employees(employee_name);

-- Create a unique index
CREATE UNIQUE INDEX idx_unique_department ON departments(department_name);

-- Add an index when creating a table
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    department_id INT,
    INDEX idx_department_id (department_id)
);
```

Dropping Indexes

```sql
-- Drop an index
DROP INDEX idx_employee_name ON employees;
```

Subqueries in MySQL

Subqueries are nested queries used within another SQL query to perform operations that depend on the results of the nested query.

Types of Subqueries

- **Single-row Subqueries**: Return a single value.
- **Multi-row Subqueries**: Return multiple rows.
- **Correlated Subqueries**: Refer to columns from the outer query.

Examples

```sql
-- Single-row subquery to find the employee with the highest salary
SELECT * FROM employees
WHERE salary = (SELECT MAX(salary) FROM employees);

-- Multi-row subquery to find employees in specific departments
SELECT employee_name FROM employees
WHERE department_id IN (SELECT department_id FROM departments WHERE location = 'New York
    ');

-- Correlated subquery to find employees earning above the average salary in their
    department
SELECT employee_name, salary FROM employees e1
WHERE salary > (
    SELECT AVG(salary) FROM employees e2
    WHERE e1.department_id = e2.department_id
);
```

Views in MySQL

Views are virtual tables based on the result set of a query. They do not store data but simplify complex queries and enhance security by controlling data access.

Creating Views

```sql
-- Create a view for employee details with department names
CREATE VIEW employee_department AS
SELECT e.employee_id, e.employee_name, d.department_name
FROM employees e
JOIN departments d
ON e.department_id = d.department_id;
```

Using Views

```sql
-- Query the view like a regular table
SELECT * FROM employee_department;

-- Use the view with filtering
SELECT employee_name, department_name
FROM employee_department
WHERE department_name = 'HR';
```

Dropping Views

```sql
-- Drop a view
DROP VIEW employee_department;
```

ROLLUP in MySQL

The ROLLUP operator is used in conjunction with the GROUP BY clause to generate subtotals and grand totals.

Example

```sql
-- Calculate total salary by department with a grand total
SELECT department_id, SUM(salary) AS total_salary
FROM employees
GROUP BY department_id WITH ROLLUP;

-- Output example:
-- | department_id | total_salary |
-- | 1             | 50000        |
-- | 2             | 75000        |
-- | NULL          | 125000       | -- Grand total
```

Notes

- The NULL value in the ROLLUP result represents the grand total.
- Useful for generating reports with aggregated data.

Triggers in MySQL

Triggers are database objects that automatically execute a specified SQL statement when a specific event occurs on a table.

Creating Triggers

```
-- Create a trigger to log changes to a table
CREATE TRIGGER after_employee_update
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO employee_changes (employee_id, change_time, old_salary, new_salary)
    VALUES (OLD.employee_id, NOW(), OLD.salary, NEW.salary);
END;
```

Using Triggers

```
-- Example: Update an employee's salary
UPDATE employees
SET salary = 60000
WHERE employee_id = 1;

-- The trigger logs the change in the employee_changes table.
SELECT * FROM employee_changes;
```

Dropping Triggers

```
-- Drop a trigger
DROP TRIGGER after_employee_update;
```

Notes

- Triggers can be defined for `INSERT`, `UPDATE`, or `DELETE` events.
- Each trigger is associated with a specific table.

Notes

- Triggers can be defined for `INSERT`, `UPDATE`, or `DELETE` events.
- Each trigger is associated with a specific table.

ON DELETE Clause in MySQL

The `ON DELETE` clause is used to define actions that should occur when a referenced row in a parent table is deleted. Common options include `SET NULL` and `CASCADE`.

SET NULL

If `ON DELETE SET NULL` is used, the foreign key column in the child table will be set to `NULL` when the referenced row in the parent table is deleted.

```
-- Example: Create tables with ON DELETE SET NULL
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
    ON DELETE SET NULL
);

-- Delete a department
```

```sql
DELETE FROM departments WHERE department_id = 1;

-- The department_id column in the employees table will be set to NULL.
SELECT * FROM employees;
```

CASCADE

If `ON DELETE CASCADE` is used, deleting a row in the parent table will automatically delete all related rows in the child table.

```sql
-- Example: Create tables with ON DELETE CASCADE
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
    ON DELETE CASCADE
);

-- Delete a department
DELETE FROM departments WHERE department_id = 2;

-- All employees in department_id = 2 will also be deleted.
SELECT * FROM employees;
```

Stored Procedures in MySQL

Stored procedures are precompiled SQL statements that can be executed as a single unit. They allow for complex operations and can accept parameters.

Creating a Stored Procedure

```sql
-- Create a stored procedure to calculate the total salary by department
CREATE PROCEDURE GetTotalSalaryByDepartment (IN dep_id INT, OUT total_salary DECIMAL
    (10,2))
BEGIN
    SELECT SUM(salary) INTO total_salary
    FROM employees
    WHERE department_id = dep_id;
END;
```

Using a Stored Procedure

```sql
-- Declare a variable to store the output
SET @total = 0;

-- Call the procedure
CALL GetTotalSalaryByDepartment(1, @total);

-- Display the result
SELECT @total AS TotalSalary;
```

Dropping a Stored Procedure

```sql
-- Drop a stored procedure
DROP PROCEDURE GetTotalSalaryByDepartment;
```

Notes

- Stored procedures can have input (`IN`), output (`OUT`), or input-output (`INOUT`) parameters.
- They are useful for encapsulating complex logic and reusing it across multiple queries.