kintsugi-stack-sql

> The relational model is founded on logic.

- Author: Kintsugi-Programmer

> Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

Table of Contents

Fundamentals

What is a Database?

A database is a collection of interrelated data stored in a easily accessible, organized format. In modern systems, databases are digital collections stored on computer systems, allowing for easy retrieval, modification, and deletion of data.

Database Characteristics:

- Organized collection of related data
- Digital format storage
- Easily accessible and manageable
- Supports multiple tables and relationships
- Examples: Company employee records, college student information, e-commerce product catalogs

What is DBMS?

DBMS stands for **Database Management System**. It is a software application designed to manage databases efficiently.

Primary Functions of DBMS:

- Create and organize databases
- Add new data to databases
- Delete old or unnecessary data
- Update existing data
- Search and retrieve data from databases
- Manage database access and security

How DBMS Works:

```
User ⟷ DBMS ⟷ Database
```

The user does not access the database directly. Instead, the DBMS acts as an intermediary layer, processing user requests and performing corresponding operations on the database.

Types of Databases

1. Relational Databases

- Data is stored in **table format** (rows and columns)
- Follows strict structure with predefined schemas
- Tables contain interrelated data
- Use SQL for interaction
- Examples:
  - MySQL
  - Oracle Database
  - PostgreSQL
  - Microsoft SQL Server

2. Non-Relational (NoSQL) Databases

- Data is stored **without table structure**
- More flexible and schema-less
- Data can be stored as documents, key-value pairs, graphs, etc.
- Do not use SQL
- Examples:
  - MongoDB
  - Cassandra
  - Redis

Note: This course focuses on **Relational Databases** since we are learning SQL.

What is RDBMS?

RDBMS stands for **Relational Database Management System**.

RDBMS Characteristics:

- Based on the concept of tables (relations)
- Data organized into rows (records) and columns (attributes)
- Tables can have relationships with other tables
- Uses SQL as the query language
- Examples: MySQL, PostgreSQL, Oracle, SQL Server

Databases and DBMS

Database Structure

A database contains multiple tables, each with its own data structure:

```
Database
├── Table 1
│   ├── Column 1
│   ├── Column 2
│   └── Column 3
├── Table 2
│   ├── Column 1
│   └── Column 2
└── Table 3
    ├── Column 1
    ├── Column 2
    └── Column 3
```

What is a Table?

A table is the fundamental data structure in a relational database.

Table Components:

- **Columns (Attributes/Fields)**: Vertical structure defining what data is stored
- **Rows (Tuples/Records)**: Horizontal structure representing individual data entries

Example Student Table:

Roll_No	Full_Name	Class	DOB	Gender	City	Marks
101	Raj Kumar	10A	2005-05-15	M	Mumbai	85
102	Priya Sharma	10A	2005-08-22	F	Delhi	92
103	Amit Patel	10B	2006-01-10	M	Bangalore	78

Table Concepts:

- **Rows**: Each row represents one complete student's data
- **Columns**: Each column represents a specific attribute (Roll_No, Name, etc.)
- **Schema**: The design/structure defined by columns
- **Data**: The actual values stored in rows

SQL Basics

What is SQL?

SQL stands for **Structured Query Language**. It is a programming language used to interact with relational databases.

Key Points about SQL:

- SQL is NOT a database; it is a language for database interaction
- SQL keywords are NOT case sensitive (SELECT = select)
- Used to perform CRUD operations
- Standardized language understood by all RDBMS

CRUD Operations

SQL enables four major operations:

1. **CREATE** - Create databases, tables, insert new data
2. **READ** - Retrieve and view data from the database
3. **UPDATE** - Modify existing data
4. **DELETE** - Remove database objects or data

SQL vs MySQL

Aspect	SQL	MySQL
Type	Programming Language	Database Management System (DBMS)
Purpose	Used to interact with databases	Manages and organizes relational databases
Functionality	Provides commands for CRUD operations	Implements SQL language
Independence	Not a database itself	Uses SQL to operate
Usage	Applies to multiple RDBMS systems	Specific RDBMS software

Note: SQL is the language; MySQL is the system that uses SQL.

Original vs Current Naming

- **Original Name**: Structured **English** Query Language (SEQUEL)
- **Current Name**: Structured Query Language (SQL)
- **Pronunciation**: Both "SQL" (pronounced as letters) and "Sequel" are acceptable

---

Data Types

SQL data types define the kind of data that can be stored in columns or variables.

Numeric Data Types

Integer Types

Data Type	Range	Usage	Example
TINYINT	-128 to 127	Very small integers	TINYINT
TINYINT UNSIGNED	0 to 255	Small positive integers	TINYINT UNSIGNED
INT	-2,147,483,648 to 2,147,483,647	Standard integers	INT
BIGINT	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Large integers	BIGINT
BIT	x-bit values (x: 1-64)	Binary values	BIT(2)

Decimal Types

Data Type	Description	Usage	Example
FLOAT	Decimal with precision to 23 digits	Single precision decimals	FLOAT
DOUBLE	Decimal with 24 to 53 digits	Double precision decimals	DOUBLE
DECIMAL	Fixed-point decimal numbers	Exact decimal calculations	DECIMAL(10, 2)

String Data Types

Data Type	Range	Description	Usage	Example
CHAR	0-255 characters	Fixed length strings	Names, codes	CHAR(50)
VARCHAR	0-65,535 characters	Variable length strings	Addresses, descriptions	VARCHAR(50)
BLOB	0-65,535 bytes	Binary large object data	Images, files	BLOB(1000)
TEXT	0-65,535 characters	Long text data	Comments, paragraphs	TEXT

Date and Time Data Types

Data Type	Format	Range	Usage	Example
DATE	YYYY-MM-DD	1000-01-01 to 9999-12-31	Dates	DATE
TIME	HH:MM:SS	Time values	Time records	TIME
YEAR	YYYY	1901 to 2155	Year values	YEAR
DATETIME	YYYY-MM-DD HH:MM:SS	Date and time combined	Timestamps	DATETIME
TIMESTAMP	YYYY-MM-DD HH:MM:SS	Automatic timestamps	Update tracking	TIMESTAMP

Boolean Data Type

Data Type	Values	Usage	Example

Data Type	Values	Usage	Example
BOOLEAN	0 or 1	True/False values	BOOLEAN

Important Notes on Data Types

String Length Considerations:

- **CHAR**: Fixed length - reserves all space even if not fully used
- **VARCHAR**: Variable length - uses only necessary space
- **Recommendation**: Use VARCHAR for better memory efficiency and performance

UNSIGNED Modifier:

- Used when you only need positive values
- Example: `UNSIGNED INT` for ages, counts, IDs
- Increases positive value range at the expense of negative values

Reference

For all MySQL data types, visit: https://dev.mysql.com/doc/refman/8.0/en/data-types.html

---

SQL Commands Types

SQL commands are categorized into five main types based on their functionality:

1. DQL - Data Query Language

Purpose: Retrieve and query data from databases

Commands: SELECT

Description: Used to select and retrieve data from one or more tables

2. DDL - Data Definition Language

Purpose: Create, modify, and delete database objects

Commands: CREATE, DROP, ALTER, RENAME, TRUNCATE

Description:

- Manages the structure and schema of database objects
- Creates tables and databases
- Modifies existing structures
- Removes database objects

3. DML - Data Manipulation Language

Purpose: Modify and manage data within tables

Commands: INSERT, UPDATE, DELETE

Description:

- Adds new records to tables
- Updates existing records
- Removes records from tables

4. DCL - Data Control Language

Purpose: Control access rights and permissions

Commands: GRANT, REVOKE

Description:

- Grants privileges to users or roles
- Revokes previously granted privileges
- Manages database security and access control

5. TCL - Transaction Control Language

Purpose: Manage transactions and ensure data consistency

Commands: COMMIT, ROLLBACK, SAVEPOINT, START TRANSACTION

Description:

- Controls transaction execution
- Ensures data integrity
- Manages rollback and commit operations

Data Definition Language (DDL)

DDL is a subset of SQL responsible for defining and managing the structure of databases and their objects.

CREATE DATABASE

Used to create a new database in the system.

Syntax:

```
CREATE DATABASE database_name;
```

Example:

```
CREATE DATABASE college_db;
```

Safe Creation:

```
CREATE DATABASE IF NOT EXISTS college_db;
```

DROP DATABASE

Permanently deletes a database and all its contents.

Syntax:

```
DROP DATABASE database_name;
```

Example:

```
DROP DATABASE college_db;
```

Safe Deletion:

```
DROP DATABASE IF EXISTS college_db;
```

USE DATABASE

Selects a database for use in subsequent operations.

Syntax:

```
USE database_name;
```

Example:

```
USE college_db;
```

SHOW DATABASES

Displays all existing databases in the system.

Syntax:

```
SHOW DATABASES;
```

SHOW TABLES

Displays all tables in the currently selected database.

Syntax:

```
SHOW TABLES;
```

CREATE TABLE

Creates a new table with specified columns and constraints.

Syntax:

```
CREATE TABLE table_name (
    column_name1 datatype constraint,
    column_name2 datatype constraint,
    column_name3 datatype constraint
);
```

Example:

```
CREATE TABLE students (
    roll_no INT PRIMARY KEY,
    full_name VARCHAR(50) NOT NULL,
    class VARCHAR(10),
    dob DATE,
    gender CHAR(1),
    city VARCHAR(50),
    marks INT
);
```

DROP TABLE

Removes a table along with its structure and data.

Syntax:

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE students;
```

Key Difference from TRUNCATE:

- DROP removes table structure and data
- TRUNCATE removes only data but keeps structure

TRUNCATE TABLE

Removes all data from a table but preserves the table structure.

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE TABLE students;
```

Advantages of TRUNCATE over DELETE:

- Faster execution
- Uses less memory
- Resets identity seeds
- Cannot be rolled back in some systems

ALTER TABLE

Modifies the structure of an existing table.

ADD COLUMN

Adds a new column to the table.

Syntax:

```
ALTER TABLE table_name
ADD COLUMN column_name datatype constraint;
```

Example:

```
ALTER TABLE students
ADD COLUMN email VARCHAR(100);
```

DROP COLUMN

Removes a column from the table.

Syntax:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Example:

```
ALTER TABLE students
DROP COLUMN email;
```

RENAME TABLE

Changes the name of a table.

Syntax:

```
ALTER TABLE table_name
RENAME TO new_table_name;
```

Example:

```
ALTER TABLE students
RENAME TO class_students;
```

MODIFY COLUMN

Changes the data type or constraints of a column.

Syntax:

```
ALTER TABLE table_name
MODIFY column_name new_datatype new_constraint;
```

Example:

```
ALTER TABLE students
MODIFY full_name VARCHAR(100);
```

CHANGE COLUMN

Renames a column and changes its data type or constraints.

Syntax:

```
ALTER TABLE table_name
CHANGE COLUMN old_column_name new_column_name new_datatype new_constraint;
```

Example:

```
ALTER TABLE students
CHANGE COLUMN full_name student_name VARCHAR(100);
```

CREATE INDEX

Creates an index on one or more columns to improve query performance.

Syntax:

```
CREATE INDEX index_name ON table_name (column_name);
```

Example:

```
CREATE INDEX idx_student_name ON students (full_name);
```

DROP INDEX

Removes an index from a table.

Syntax:

```
DROP INDEX index_name;
```

Example:

```
DROP INDEX idx_student_name;
```

CREATE and DROP CONSTRAINT

Creates or removes constraints to ensure data integrity.

Common Constraints:

- PRIMARY KEY
- FOREIGN KEY
- UNIQUE
- NOT NULL
- CHECK
- DEFAULT

Example - Add Constraint:

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

Example - Drop Constraint:

```
ALTER TABLE orders
DROP CONSTRAINT fk_customer;
```

Constraints in SQL

SQL constraints are rules applied to columns to ensure data integrity and quality.

NOT NULL

Ensures that a column cannot contain NULL values.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype NOT NULL
);
```

Example:

```
CREATE TABLE students (
    full_name VARCHAR(50) NOT NULL,
    roll_no INT NOT NULL
);
```

UNIQUE

Ensures all values in a column are unique; no duplicates allowed.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype UNIQUE
);
```

Example:

```
CREATE TABLE students (
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(15) UNIQUE
);
```

PRIMARY KEY

Uniquely identifies each row in a table. A column with PRIMARY KEY is both NOT NULL and UNIQUE. Each table can have only one PRIMARY KEY.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype PRIMARY KEY
);
```

Example:

```
CREATE TABLE students (
    roll_no INT PRIMARY KEY,
    full_name VARCHAR(50) NOT NULL
);
```

FOREIGN KEY

Establishes a relationship between two tables by referencing the PRIMARY KEY of another table. Prevents actions that would destroy links between tables.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype,
```

```
    FOREIGN KEY (column_name) REFERENCES other_table(primary_key)
);
```

Example:

```
CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    student_id INT,
    course_name VARCHAR(100),
    FOREIGN KEY (student_id) REFERENCES students(roll_no)
);
```

Characteristics of FOREIGN KEY:

- Can have duplicate values
- Can contain NULL values
- Multiple FOREIGN KEYs allowed in a table
- Prevents referential integrity violations

DEFAULT

Sets a default value for a column if no value is provided during insertion.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype DEFAULT default_value
);
```

Example:

```
CREATE TABLE students (
    city VARCHAR(50) DEFAULT 'Unknown',
    country VARCHAR(50) DEFAULT 'India'
);
```

CHECK

Limits the values allowed in a column based on a condition.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype CHECK (condition)
);
```

Example:

```
CREATE TABLE students (
    age INT CHECK (age >= 5 AND age <= 25),
    marks INT CHECK (marks >= 0 AND marks <= 100)
);
```

Data Query Language (DQL)

DQL is focused on retrieving data from databases using the SELECT statement.

SELECT Statement

Retrieves specific columns from a table.

Basic Syntax:

```
SELECT column1, column2, ... FROM table_name;
```

Select All Columns:

```
SELECT * FROM table_name;
```

Examples:

```
-- Select specific columns
SELECT full_name, city FROM students;

-- Select all columns
SELECT * FROM students;

-- Select with column order
SELECT city, full_name, marks FROM students;
```

WHERE Clause

Filters records based on specified conditions.

Syntax:

```

```

```
SELECT column1, column2, ... FROM table_name
WHERE condition;
```

Example:

```
SELECT * FROM students WHERE city = 'Mumbai';
SELECT * FROM students WHERE marks > 80;
```

Operators in WHERE Clause

Comparison Operators

Operator	Meaning	Example
=	Equal to	marks = 90
!= or <>	Not equal to	city != 'Delhi'
>	Greater than	marks > 80
<	Less than	age < 25
>=	Greater than or equal	marks >= 80
<=	Less than or equal	age <= 20

Logical Operators

AND:

- Returns records where ALL conditions are true
- Syntax: `condition1 AND condition2 AND condition3`
- Example: `SELECT * FROM students WHERE city = 'Mumbai' AND marks > 80;`

OR:

- Returns records where ANY condition is true
- Syntax: `condition1 OR condition2 OR condition3`
- Example: `SELECT * FROM students WHERE city = 'Mumbai' OR city = 'Delhi';`

NOT:

- Negates a condition
- Syntax: `NOT condition`
- Example: `SELECT * FROM students WHERE NOT city = 'Delhi';`

Advanced Operators

IN:

- Matches any value in a list
- Syntax: `SELECT * FROM table WHERE column IN (value1, value2, value3)`
- Example: `SELECT * FROM students WHERE city IN ('Mumbai', 'Delhi', 'Bangalore');`

BETWEEN:

- Selects values within a given range
- Syntax: `SELECT * FROM table WHERE column BETWEEN value1 AND value2`
- Example: `SELECT * FROM students WHERE marks BETWEEN 70 AND 90;`

LIKE:

- Searches for a specified pattern in a column
- Wildcards:
  - `%` represents zero, one, or multiple characters
  - `_` represents one single character

LIKE Examples:

```
-- Names starting with 'A'
SELECT * FROM students WHERE full_name LIKE 'A%';

-- Names ending with 'a'
SELECT * FROM students WHERE full_name LIKE '%a';

-- Names containing 'ar'
SELECT * FROM students WHERE full_name LIKE '%ar%';

-- Names with 'r' as second character
SELECT * FROM students WHERE full_name LIKE '_r%';

-- Names starting with 'a' and at least 2 characters
SELECT * FROM students WHERE full_name LIKE 'a_%';

-- Names starting with 'a' and ending with 'o'
SELECT * FROM students WHERE full_name LIKE 'a%o';
```

IS NULL:

- Checks for NULL values
- Syntax: `SELECT * FROM table WHERE column IS NULL`
- Example: `SELECT * FROM students WHERE email IS NULL;`

Bitwise Operators

Operator	Name	Example
&	Bitwise AND	marks & 5

Operator	Name	Example
|	Bitwise OR	marks | 5

Arithmetic Operators

Operator	Operation	Example
+	Addition	marks + 10
-	Subtraction	marks - 5
*	Multiplication	marks * 2
/	Division	marks / 2
%	Modulus	marks % 10

DISTINCT Keyword

Removes duplicate rows from query results.

Syntax:

```
SELECT DISTINCT column1, column2 FROM table_name;
```

Example:

```
-- Get unique cities
SELECT DISTINCT city FROM students;

-- Get unique combinations of city and marks range
SELECT DISTINCT city, marks FROM students;
```

LIMIT Clause

Sets an upper limit on the number of rows returned.

Syntax:

```
SELECT column1, column2 FROM table_name
LIMIT number;
```

Example:

```sql
-- Get only first 5 students
SELECT * FROM students LIMIT 5;

-- Get first 10 high scorers
SELECT * FROM students ORDER BY marks DESC LIMIT 10;
```

ORDER BY Clause

Sorts the result set in ascending or descending order.

Syntax:

```sql
SELECT column1, column2 FROM table_name
ORDER BY column_name ASC|DESC;
```

Examples:

Ascending Order

```sql
-- Default ascending order
SELECT * FROM students ORDER BY marks;
SELECT * FROM students ORDER BY marks ASC;
```

Descending Order

```sql
-- Highest marks first
SELECT * FROM students ORDER BY marks DESC;
```

Multiple Column Sorting

```sql
-- Sort by city first, then by marks
SELECT * FROM students ORDER BY city ASC, marks DESC;
```

Sorting by Expression

```sql
-- Sort by calculated values
SELECT full_name, marks, marks * 1.1 AS adjusted_marks
FROM students
ORDER BY adjusted_marks DESC;
```

Sorting by Column Position

```
-- Sort by second column (descending), then first column (ascending)
SELECT full_name, marks FROM students ORDER BY 2 DESC, 1 ASC;
```

Handling NULL Values

```
-- Place NULL values at the end
SELECT * FROM students ORDER BY email NULLS LAST;

-- Place NULL values at the beginning
SELECT * FROM students ORDER BY email NULLS FIRST;
```

AS Keyword

Renames columns or expressions in query results (creates aliases).

Syntax:

```
SELECT column_name AS "Alias Name" FROM table_name;
```

Examples:

```
-- Rename single column
SELECT full_name AS "Student Name", marks AS "Score" FROM students;

-- Rename with expressions
SELECT full_name, marks * 1.1 AS "Adjusted Marks" FROM students;

-- Use in calculations
SELECT full_name, marks / 10 AS "Grade Points" FROM students;
```

GROUP BY Clause

Groups rows that have the same values into summary rows, typically used with aggregate functions.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Key Concepts:

- Collects data from multiple records
- Groups results by one or more columns
- Usually paired with aggregate functions (COUNT, SUM, AVG, MAX, MIN)
- Returns one row per group

Examples:

```
-- Count students per city
SELECT city, COUNT(*) as student_count
FROM students
GROUP BY city;

-- Average marks per city
SELECT city, AVG(marks) as average_marks
FROM students
GROUP BY city;

-- Count students per gender
SELECT gender, COUNT(*)
FROM students
GROUP BY gender;
```

Grouping by Multiple Columns

```
-- Count students by city and gender
SELECT city, gender, COUNT(*)
FROM students
GROUP BY city, gender;

-- Average marks by city and gender
SELECT city, gender, AVG(marks)
FROM students
GROUP BY city, gender;
```

GROUP BY with ORDER BY

```
-- Get cities with most students
SELECT city, COUNT(*) as count
FROM students
GROUP BY city
ORDER BY count DESC;
```

HAVING Clause

Filters grouped data based on aggregate function results. Used with GROUP BY.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

Key Differences from WHERE:

- WHERE filters rows before grouping
- HAVING filters groups after grouping
- HAVING works with aggregate functions

Examples:

```
-- Get cities with more than 5 students
SELECT city, COUNT(*) as count
FROM students
GROUP BY city
HAVING COUNT(*) > 5;

-- Get cities where average marks exceed 80
SELECT city, AVG(marks) as avg_marks
FROM students
GROUP BY city
HAVING AVG(marks) > 80;

-- Get cities where max marks is greater than 90
SELECT city, MAX(marks)
FROM students
GROUP BY city
HAVING MAX(marks) > 90;
```

Aggregate Functions

Perform calculations on groups of rows or entire result sets, returning a single value.

COUNT()

Counts the number of rows in a group or result set.

Syntax:

```
SELECT COUNT(column) FROM table_name;
SELECT COUNT(*) FROM table_name;
SELECT COUNT(DISTINCT column) FROM table_name;
```

Examples:

```
-- Total students
SELECT COUNT(*) FROM students;

-- Students in Mumbai
SELECT COUNT(*) FROM students WHERE city = 'Mumbai';

-- Unique cities
SELECT COUNT(DISTINCT city) FROM students;

-- Students with recorded emails
SELECT COUNT(email) FROM students;
```

SUM()

Calculates the total sum of numeric values.

Syntax:

```
SELECT SUM(column) FROM table_name;
```

Examples:

```
-- Total marks of all students
SELECT SUM(marks) FROM students;

-- Total marks per city
SELECT city, SUM(marks) FROM students GROUP BY city;
```

AVG()

Computes the average of numeric values.

Syntax:

```
SELECT AVG(column) FROM table_name;
```

Examples:

```
-- Average marks
SELECT AVG(marks) FROM students;
```

```sql
-- Average marks per city
SELECT city, AVG(marks) FROM students GROUP BY city;

-- Average marks excluding values below 50
SELECT AVG(marks) FROM students WHERE marks > 50;
```

MAX()

Finds the maximum value in a set.

Syntax:

```sql
SELECT MAX(column) FROM table_name;
```

Examples:

```sql
-- Highest marks
SELECT MAX(marks) FROM students;

-- Highest marks per city
SELECT city, MAX(marks) FROM students GROUP BY city;

-- Name of student with highest marks
SELECT full_name, marks FROM students
WHERE marks = (SELECT MAX(marks) FROM students);
```

MIN()

Retrieves the minimum value in a set.

Syntax:

```sql
SELECT MIN(column) FROM table_name;
```

Examples:

```sql
-- Lowest marks
SELECT MIN(marks) FROM students;

-- Lowest marks per city
SELECT city, MIN(marks) FROM students GROUP BY city;
```

General SELECT Query Order

The proper order of clauses in a SELECT statement:

```
SELECT column(s)
FROM table_name
WHERE condition
GROUP BY column(s)
HAVING condition
ORDER BY column(s) ASC|DESC
LIMIT number;
```

Execution Order (differs from written order):

1. FROM - Select the table
2. WHERE - Filter rows
3. GROUP BY - Group filtered rows
4. HAVING - Filter groups
5. SELECT - Select columns
6. ORDER BY - Sort results
7. LIMIT - Limit results

Data Manipulation Language (DML)

DML encompasses commands that manipulate data within a database.

INSERT Statement

Adds new records to a table.

Syntax:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Insert Single Row:

```
INSERT INTO students (roll_no, full_name, city, marks)
VALUES (101, 'Raj Kumar', 'Mumbai', 85);
```

Insert Multiple Rows:

```
INSERT INTO students (roll_no, full_name, city, marks)
VALUES
```

```
(102, 'Priya Sharma', 'Delhi', 92),
(103, 'Amit Patel', 'Bangalore', 78),
(104, 'Neha Singh', 'Pune', 88);
```

Insert Without Specifying Columns (all columns required in order):

```
INSERT INTO students
VALUES (105, 'Vikram Verma', '2006-03-15', 'M', 'Chennai', 81);
```

Insert Default Values:

```
INSERT INTO table_name VALUES (DEFAULT, DEFAULT, ...);
```

UPDATE Statement

Modifies existing records in a table.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Important: Always use WHERE clause to specify which records to update. Without WHERE, ALL records will be updated.

Examples:

```
-- Update single student's marks
UPDATE students SET marks = 90 WHERE roll_no = 101;

-- Update multiple columns
UPDATE students
SET marks = 95, city = 'Delhi'
WHERE roll_no = 102;

-- Update based on condition
UPDATE students SET marks = marks + 5 WHERE city = 'Mumbai';

-- Update all records (use with caution)
UPDATE students SET status = 'Active';
```

DELETE Statement

Removes records from a table.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Important: Always use WHERE clause. Without WHERE, ALL records will be deleted.

Examples:

```
-- Delete specific student
DELETE FROM students WHERE roll_no = 101;

-- Delete students with low marks
DELETE FROM students WHERE marks < 40;

-- Delete students from specific city
DELETE FROM students WHERE city = 'Unknown';

-- Delete all records (use with extreme caution)
DELETE FROM students;
```

DELETE vs TRUNCATE vs DROP:

- **DELETE**: Removes rows; can specify conditions; slower; can be rolled back in transactions
- **TRUNCATE**: Removes all rows; no conditions; faster; resets identity; cannot select specific rows
- **DROP**: Removes entire table structure and data; fastest; cannot be rolled back easily

---

Data Control Language (DCL)

DCL focuses on managing access rights, permissions, and security-related aspects.

GRANT Command

Provides specific privileges or permissions to users or roles.

Syntax:

```
GRANT privilege_type
ON object_name
TO user_or_role;
```

Parameters:

- **privilege_type**: SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, etc.

- **object_name**: Table name, view name, or database name
- **user_or_role**: Username or role name

Examples:

```
-- Grant SELECT privilege on Employees table to Analyst user
GRANT SELECT ON Employees TO Analyst;

-- Grant multiple privileges
GRANT SELECT, INSERT, UPDATE ON Students TO Manager;

-- Grant all privileges
GRANT ALL ON Database_Name TO Admin;

-- Grant with GRANT option (user can grant to others)
GRANT SELECT ON Employees TO Analyst WITH GRANT OPTION;
```

REVOKE Command

Removes or revokes previously granted privileges.

Syntax:

```
REVOKE privilege_type
ON object_name
FROM user_or_role;
```

Examples:

```
-- Revoke SELECT privilege
REVOKE SELECT ON Employees FROM Analyst;

-- Revoke multiple privileges
REVOKE SELECT, INSERT ON Students FROM Manager;

-- Revoke all privileges
REVOKE ALL ON Database_Name FROM User;

-- Revoke GRANT option
REVOKE GRANT OPTION FOR SELECT ON Employees FROM Analyst;
```

DCL and Database Security

DCL plays a crucial role in database security:

- Controls who can access data
- Prevents unauthorized data modifications

- Restricts sensitive information access
- Ensures compliance with security policies
- Maintains data integrity through access control

Best Practices:

- Grant minimum necessary privileges (Principle of Least Privilege)
- Regularly audit user permissions
- Revoke unnecessary privileges
- Use roles for easier permission management
- Separate read-only and write access when possible

Transaction Control Language (TCL)

TCL manages transactions to ensure data consistency and integrity.

What is a Transaction?

A transaction is a sequence of one or more SQL statements that are executed as a single unit of work. Either all statements are completed successfully or none of them are applied.

COMMIT Command

Permanently saves changes made during a transaction.

Syntax:

```
COMMIT;
```

Characteristics:

- Makes all changes permanent
- Cannot be undone after execution
- Marks successful completion of transaction
- Releases database locks

Example:

```
UPDATE Employees
SET Salary = Salary * 1.10
WHERE Department = 'Sales';

COMMIT;
```

ROLLBACK Command

Undoes changes made during a transaction, reverting to the previous state.

Syntax:

```
ROLLBACK;
```

Characteristics:

- Reverts all changes since last COMMIT or SAVEPOINT
- Returns database to consistent state
- Used when errors occur
- Releases database locks

Example:

```
BEGIN;
UPDATE Inventory
SET Quantity = Quantity - 10
WHERE ProductID = 101;

-- An error occurs here
ROLLBACK;
-- Changes are undone
```

SAVEPOINT Command

Creates a named point within a transaction to allow partial rollbacks.

Syntax:

```
SAVEPOINT savepoint_name;
```

Characteristics:

- Creates markers within transactions
- Allows selective rollback to specific points
- Multiple SAVEPOINTs can exist in one transaction
- Useful for complex multi-step transactions

Example:

```
BEGIN;

UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 123;
```

```
SAVEPOINT before_withdrawal;

UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountID = 456;

-- An error occurs here
ROLLBACK TO before_withdrawal;
-- First update is still applied
-- Second update is rolled back

COMMIT;
```

Transaction Example Workflow

```
-- Start transaction
BEGIN;

-- Statement 1: Debit from account A
UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1;

-- Create savepoint
SAVEPOINT sp1;

-- Statement 2: Credit to account B
UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 2;

-- If error occurs, rollback to savepoint
ROLLBACK TO sp1;

-- Or if all is well, commit transaction
COMMIT;
```

TCL and Transaction Management

TCL commands are essential for:

- **Data Consistency**: Ensures related changes succeed or fail together
- **Data Integrity**: Prevents partial updates
- **Error Handling**: Allows rollback on errors
- **Atomicity**: Guarantees all-or-nothing execution
- **Reliability**: Maintains database in valid state

Joins

A join is an operation that combines rows from two or more tables based on related columns.

Importance of Joins

In normalized databases:

- Related data is stored in separate tables
- Joins retrieve connected data from multiple tables
- Essential for querying related information
- Maintain data integrity through relationships

Primary Key and Foreign Key

Primary Key (PK):

- Uniquely identifies each row in a table
- Must be unique and NOT NULL
- Only one per table
- Example: `roll_no INT PRIMARY KEY`

Foreign Key (FK):

- Column referencing another table's primary key
- Establishes relationship between tables
- Can have duplicate and NULL values
- Multiple foreign keys allowed in a table
- Example: `student_id INT FOREIGN KEY REFERENCES students(roll_no)`

Types of Joins

1. INNER JOIN

Returns only rows where the join condition is met in all participating tables.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Characteristics:

- Returns matching rows only
- Excludes non-matching rows from both tables
- Most commonly used join type
- Can use = or other comparison operators

Example:

Students Table:

StudentID	StudentName

StudentID	StudentName
1	Alice
2	Bob
3	Carol

Courses Table:

CourseID	StudentID	CourseName
101	1	Math
102	3	Science
103	2	English

Query:

```
SELECT Students.StudentName, Courses.CourseName
FROM Students
INNER JOIN Courses
ON Students.StudentID = Courses.StudentID;
```

Result:

StudentName	CourseName
Alice	Math
Bob	English
Carol	Science

2. LEFT JOIN (Left Outer Join)

Returns all rows from the left table and matching rows from the right table. Non-matching right table rows show NULL.

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

Characteristics:

- Returns all rows from left table

- Returns matching rows from right table
- Non-matching right rows appear as NULL
- Useful for finding unmatched left table records

Example:

```
SELECT Students.StudentName, Courses.CourseName
FROM Students
LEFT JOIN Courses
ON Students.StudentID = Courses.StudentID;
```

Result (with additional unmatched students):

StudentName	CourseName
Alice	Math
Bob	English
Carol	Science
David	NULL

3. RIGHT JOIN (Right Outer Join)

Returns all rows from the right table and matching rows from the left table. Non-matching left table rows show NULL.

Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

Characteristics:

- Returns all rows from right table
- Returns matching rows from left table
- Non-matching left rows appear as NULL
- Inverse of LEFT JOIN

Example:

```
SELECT Students.StudentName, Courses.CourseName
FROM Students
RIGHT JOIN Courses
ON Students.StudentID = Courses.StudentID;
```

4. FULL OUTER JOIN (Full Join)

Returns all rows from both tables. Non-matching rows show NULL values.

Syntax (MySQL uses UNION of LEFT and RIGHT JOINs):

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column
UNION
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

Characteristics:

- Returns all rows from both tables
- Includes both matched and unmatched rows
- Non-matching columns show NULL
- Most comprehensive join type

Example:

```
SELECT Students.StudentName, Courses.CourseName
FROM Students
FULL JOIN Courses
ON Students.StudentID = Courses.StudentID;
```

5. CROSS JOIN

Produces Cartesian product - combines every row from one table with every row from another table.

Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

Characteristics:

- No join condition needed
- Generates all combinations
- Result set = (rows in table1) × (rows in table2)

- Can create very large result sets

Example:

Students Table (2 rows):

StudentID	StudentName
1	Alice
2	Bob

Subjects Table (3 rows):

SubjectID	SubjectName
101	Math
102	Science
103	English

Query:

```
SELECT Students.StudentName, Subjects.SubjectName
FROM Students
CROSS JOIN Subjects;
```

Result (2 × 3 = 6 rows):

StudentName	SubjectName
Alice	Math
Alice	Science
Alice	English
Bob	Math
Bob	Science
Bob	English

6. SELF JOIN

Joins a table with itself. Used to find relationships within the same table.

Syntax:

```
SELECT columns
FROM table AS alias1
```

```
JOIN table AS alias2
ON alias1.column = alias2.column;
```

Characteristics:

- Table joined with itself
- Requires table aliases for differentiation
- Useful for hierarchical data
- Common with employee-manager relationships

Example:

Employees Table:

EmployeeID	EmployeeName	ManagerID
1	Alice	3
2	Bob	3
3	Carol	NULL
4	David	1

Query (find employees and their managers):

```
SELECT e1.EmployeeName AS Employee,
       e2.EmployeeName AS Manager
FROM Employees AS e1
JOIN Employees AS e2
ON e1.ManagerID = e2.EmployeeID;
```

Result:

Employee	Manager
Alice	Carol
Bob	Carol
David	Alice

Join Comparison Table

Join Type	Returns	Use Case
INNER	Matching rows only	Common data between tables
LEFT	All left + matching right	Include all from left table
RIGHT	All right + matching left	Include all from right table

Join Type	Returns	Use Case
FULL	All rows from both	Complete data from both tables
CROSS	All combinations	Generate all possibilities
SELF	Internal relationships	Hierarchical or related data

Set Operations

Set operations combine or manipulate result sets of multiple SELECT queries using set theory principles.

UNION

Combines result sets of two or more SELECT queries into a single result set, removing duplicates.

Syntax:

```
SELECT columns FROM table1
UNION
SELECT columns FROM table2;
```

Rules:

- SELECT queries must have same number of columns
- Column data types must be compatible
- Columns must be in same order
- Duplicate rows are automatically removed

Example:

Customers Table:

CustomerID	CustomerName
1	Alice
2	Bob

Suppliers Table:

SupplierID	SupplierName
101	SupplierA
102	SupplierB

Query:

```
SELECT CustomerName FROM Customers
UNION
SELECT SupplierName FROM Suppliers;
```

Result:

CustomerName
Alice
Bob
SupplierA
SupplierB

UNION ALL

Combines result sets without removing duplicates. Simply concatenates all rows.

Syntax:

```
SELECT columns FROM table1
UNION ALL
SELECT columns FROM table2;
```

Characteristics:

- Retains all rows including duplicates
- Faster than UNION (no duplicate checking)
- Used when duplicates are acceptable or desired

Example:

```
SELECT CustomerName FROM Customers
UNION ALL
SELECT SupplierName FROM Suppliers;
```

If there were duplicate names, they would all appear in the result.

INTERSECT

Returns common rows that exist in result sets of two SELECT queries.

Syntax:

```

```

```
SELECT columns FROM table1
INTERSECT
SELECT columns FROM table2;
```

Characteristics:

- Returns only rows present in ALL result sets
- Automatically removes duplicates
- Useful for finding common data

Example:

```
SELECT CustomerName FROM Customers
INTERSECT
SELECT SupplierName FROM Suppliers;
```

Returns names that appear in both Customers and Suppliers.

EXCEPT (or MINUS)

Returns rows from first query that are NOT in second query.

Syntax:

```
SELECT columns FROM table1
EXCEPT
SELECT columns FROM table2;
```

Note: Called MINUS in some databases (Oracle, MySQL alternatives).

Characteristics:

- Returns rows in first set but not in second
- Automatically removes duplicates
- Order of queries matters

Example:

```
SELECT CustomerName FROM Customers
EXCEPT
SELECT SupplierName FROM Suppliers;
```

Returns customer names that are not in suppliers.

Set Operations vs Joins

Aspect	Set Operations	Joins
Purpose	Combine result sets based on set theory	Combine rows from tables by condition
Data Source	Result sets of SELECT queries	Columns from related tables
Combining Rows	Combines entire rows from different result sets	Combines rows by matching condition
Output Columns	SELECT queries must have same column count	Can combine any columns
Data Types	Must be compatible	Can differ
Duplicate Handling	UNION removes; UNION ALL keeps	Depends on join type
Performance	Generally faster for simple operations	More efficient for large datasets
Use Cases	Combining data from different sources	Retrieving related data

Subqueries

Subqueries (also called inner queries or nested queries) use one query's result as input for another query.

Importance of Subqueries

- Break complex problems into manageable steps
- Filter data based on aggregate results
- Compare values dynamically
- Create temporary result sets
- Improve readability of complex queries

Basic Subquery Syntax

```
SELECT columns
FROM table
WHERE column OPERATOR (
    SELECT column
    FROM table
    WHERE condition
);
```

Components:

- **Outer Query**: Main query using subquery result
- **Inner Query**: Subquery providing filtered/calculated data
- **Operator**: Comparison operator (=, >, <, IN, NOT IN, etc.)

Subquery Types and Examples

Single Value Subquery

Returns a single value used in comparison.

Example - Find students with marks above class average:

Step 1: Calculate class average

```
SELECT AVG(marks) FROM students;  -- Returns 82
```

Step 2: Find students above average

```
SELECT * FROM students
WHERE marks > (SELECT AVG(marks) FROM students);
```

List-Based Subquery with IN

Returns multiple values for membership checking.

Example - Find students with even roll numbers:

```
SELECT * FROM students
WHERE roll_no IN (
    SELECT roll_no FROM students
    WHERE roll_no % 2 = 0
);
```

Example - Find courses taken by Alice:

```
SELECT * FROM courses
WHERE student_id IN (
    SELECT student_id FROM students
    WHERE full_name = 'Alice'
);
```

Subquery with FROM Clause

Uses subquery result as temporary table.

Example - Find max marks from students in Delhi:

```
SELECT MAX(marks)
FROM (
```

```
    SELECT * FROM students
    WHERE city = 'Delhi'
) AS delhi_students;
```

More simply:

```
SELECT MAX(marks) FROM students WHERE city = 'Delhi';
```

Example - Find average marks per city:

```
SELECT city, AVG(marks) as avg_marks
FROM students
GROUP BY city;
```

Subquery with HAVING Clause

Filters groups using subquery results.

Example - Find cities with average marks above overall average:

```
SELECT city, AVG(marks)
FROM students
GROUP BY city
HAVING AVG(marks) > (SELECT AVG(marks) FROM students);
```

Subquery Operators

Operator	Meaning	Use Case
=	Equal	Single value comparison
>	Greater than	Numeric comparison
<	Less than	Numeric comparison
>=	Greater or equal	Numeric comparison
<=	Less or equal	Numeric comparison
<> or !=	Not equal	Inequality check
IN	Value in list	Multiple value match
NOT IN	Value not in list	Exclusion
BETWEEN	Within range	Range check
EXISTS	Subquery returns results	Existence check

Operator	Meaning	Use Case
ALL	Compare to all values	All-comparison
ANY	Compare to any value	Any-comparison

Subqueries vs Joins

Aspect	Subqueries	Joins
Purpose	Filter/calculate using another query	Combine rows from multiple tables
Data Source	Query result set	Table columns
Combining Rows	Not for combining; for filtering	For combining related data
Result Structure	Can return scalar or sets	Multi-column result sets
Performance	Can be slower with large datasets	Generally more efficient
Complexity	Easier for simple logic	Better for complex multi-table queries
Readability	Clear step-by-step logic	More compact
Use Cases	Filtering by aggregate results	Retrieving related data

Correlated Subqueries

A subquery that references columns from outer query.

Example - Find students with marks above their city's average:

```
SELECT full_name, city, marks
FROM students s1
WHERE marks > (
    SELECT AVG(marks)
    FROM students s2
    WHERE s2.city = s1.city
);
```

Views

A view is a virtual table based on the result set of a SQL statement.

Characteristics of Views

- Virtual table created from query result
- Not physically stored (except materialized views)
- Automatically updated when underlying data changes
- Database recreates view every time queried
- Shows current data always

Advantages of Views

- **Data Abstraction**: Hide complex queries
- **Security**: Restrict column/row access
- **Simplicity**: Simplify complex queries
- **Consistency**: Ensure consistent query logic
- **Reusability**: Use same view in multiple queries
- **Encapsulation**: Hide database complexity

Disadvantages of Views

- **Performance**: Recreated every query
- **Limitations**: Cannot use ORDER BY (some systems)
- **Maintenance**: Update logic when tables change
- **Restrictions**: Cannot insert/update through complex views
- **Storage**: Extra metadata storage

Creating Views

Basic Syntax:

```
CREATE VIEW view_name AS
SELECT columns FROM table WHERE condition;
```

Example:

```
CREATE VIEW high_scorers AS
SELECT full_name, marks, city
FROM students
WHERE marks > 80
ORDER BY marks DESC;
```

Using Views

Query a View:

```
SELECT * FROM high_scorers;
```

View with Aggregate Data:

```
CREATE VIEW city_statistics AS
SELECT city, COUNT(*) as student_count, AVG(marks) as avg_marks
FROM students
GROUP BY city;
```

Dropping Views

Syntax:

```
DROP VIEW view_name;
```

Example:

```
DROP VIEW high_scorers;
```

View Best Practices

- Use descriptive view names
- Document view purpose
- Avoid unnecessary complexity
- Update views when table structure changes
- Use for frequently-used queries
- Consider performance impact
- Avoid creating views on views when possible

Installing and Setting Up MySQL and MySQL Workbench

System Requirements

- Operating System: Windows, macOS, Linux
- RAM: Minimum 2GB (4GB recommended)
- Disk Space: 500MB minimum
- Administrator privileges

Installation on macOS

Step 1: Download MySQL Community Server

1. Visit https://www.mysql.com/downloads/
2. Click on "Downloads" section
3. Scroll down to "MySQL Community Server"
4. Select macOS version
5. Click "Download" (no login required)
6. Select "No thanks, just start my download"

Step 2: Install MySQL Server

1. Find downloaded DMG file
2. Double-click to mount the image

3. Double-click the installer package
4. Follow installation wizard:
    - Click "Continue"
    - Review license, click "Agree"
    - Click "Continue"
    - Click "Install"
5. Enter system password when prompted
6. Installation completes with "Install Succeeded"
7. Set MySQL password (remember this for later)
8. Optional: Click "Move to Bin" to delete installer

Step 3: Download MySQL Workbench

1. Go back to mysql.com/downloads/
2. Select "MySQL Workbench" from community section
3. Click "Download"
4. Click "No thanks, just start my download"

Step 4: Install MySQL Workbench

1. Double-click downloaded DMG file
2. Drag MySQL Workbench icon to Applications folder
3. Wait for copying to complete
4. Close the installer window
5. Find MySQL Workbench in Applications

Installation on Windows

Step 1: Download MySQL Installer

1. Visit https://www.mysql.com/downloads/
2. Click "Downloads"
3. Find MySQL Community Downloads
4. Select Windows installer
5. Click "Download"

Step 2: Run Installation Wizard

1. Double-click downloaded EXE file
2. Choose "Full" setup option
3. Review and select packages for download
4. Click "Next"
5. Click "Execute" to download packages
6. Follow installation prompts

Step 3: Configure MySQL Server

1. Select setup type (Developer Default recommended)

2. Click "Next"
3. Set port number (default 3306)
4. Configure MySQL server
5. Enter MySQL password (save this!)
6. Complete setup wizard
7. Note the port number and password

Step 4: Install MySQL Workbench

1. In installer, select MySQL Workbench
2. Click "Install"
3. Wait for installation
4. Workbench starts automatically

Connecting to MySQL in Workbench

Creating a Connection (macOS)

1. Open MySQL Workbench
2. On startup, find connection options
3. Click "+" icon to add new connection
4. In dialog:
    - Connection Name: "Local Instance"
    - Connection Method: "Standard"
    - Hostname: "localhost" or "127.0.0.1"
    - Port: 3306 (default)
    - Username: "root"
5. Click "Store in Vault"
6. Enter MySQL password (set during installation)
7. Click "Test Connection"
8. Success message appears: "Successfully made the MySQL connection"
9. Click "OK"
10. Click "OK" again to save

Creating a Connection (Windows)

1. Open MySQL Workbench
2. Click "+" icon to create new connection
3. Fill connection details:
    - Connection Name: "Local Instance"
    - Hostname: localhost
    - Port: 3306
    - Username: root
    - Password: [your MySQL password]
4. Click "Test Connection"
5. Verify success message
6. Click "OK"

First Steps in MySQL Workbench

Workbench Interface

Main Areas:

- **Left Panel**: Database navigation, admin tools
- **Center Panel**: SQL query editor
- **Right Panel**: Object details, help
- **Bottom Panel**: Query results and messages

Opening a Connection

1. Double-click connection name in left panel
2. MySQL Workbench connects to server
3. Query tab opens
4. Ready to write SQL

Writing Your First Query

```
-- This is a comment
SELECT 'Hello SQL!' as message;
```

To Execute:

- Click lightning bolt icon or
- Press Ctrl+Enter (Windows) / Command+Enter (macOS)

Organizing Your Queries

1. Click document icon to create new SQL script
2. Write multiple queries
3. Save with Ctrl+S
4. Give meaningful filename
5. Use multiple tabs for different files

Best Practices for Setup

- **Password Security**: Use strong passwords
- **Port Configuration**: Note your port number
- **Backups**: Enable automatic backups if available
- **Updates**: Keep MySQL and Workbench updated
- **Connection Testing**: Test connections regularly
- **Script Backup**: Save important SQL scripts
- **Version Tracking**: Know which MySQL version you're using

---

Common SQL Patterns and Examples

Practice Problem Example: Student Table Modifications

Initial Setup:

```
CREATE TABLE students (
    roll_no INT PRIMARY KEY,
    full_name VARCHAR(50),
    class VARCHAR(10),
    dob DATE,
    gender CHAR(1),
    city VARCHAR(50),
    marks INT,
    grade CHAR(1)
);

INSERT INTO students VALUES
(101, 'Raj Kumar', '10A', '2005-05-15', 'M', 'Mumbai', 85, 'A'),
(102, 'Priya Sharma', '10A', '2005-08-22', 'F', 'Delhi', 92, 'A'),
(103, 'Amit Patel', '10B', '2006-01-10', 'M', 'Bangalore', 78, 'B'),
(104, 'Neha Singh', '10B', '2005-12-03', 'F', 'Pune', 88, 'A'),
(105, 'Vikram Verma', '10A', '2006-03-15', 'M', 'Chennai', 72, 'C');
```

Task 1: Rename column name to full_name:

```
ALTER TABLE students
CHANGE COLUMN full_name full_name VARCHAR(50);
```

Or rename from "name" to "full_name":

```
ALTER TABLE students
CHANGE COLUMN name full_name VARCHAR(50);
```

Task 2: Delete students with marks less than 80:

```
DELETE FROM students WHERE marks < 80;
```

Task 3: Drop grade column:

```
ALTER TABLE students DROP COLUMN grade;
```

Real-World Query Examples

E-Commerce Database Query:

```sql
SELECT c.customer_name, COUNT(o.order_id) as total_orders, SUM(o.total) as
total_spent
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id
HAVING total_spent > 1000
ORDER BY total_spent DESC;
```

HR Database Query:

```sql
SELECT d.department_name, e.employee_name, e.salary,
       AVG(e.salary) OVER (PARTITION BY d.department_id) as dept_avg_salary
FROM employees e
JOIN departments d ON e.dept_id = d.department_id
WHERE e.salary > (SELECT AVG(salary) FROM employees)
ORDER BY d.department_name, e.salary DESC;
```

Analysis Query:

```sql
SELECT
    YEAR(order_date) as year,
    MONTH(order_date) as month,
    COUNT(DISTINCT customer_id) as unique_customers,
    COUNT(order_id) as total_orders,
    SUM(order_amount) as revenue,
    AVG(order_amount) as avg_order_value
FROM orders
WHERE order_date >= DATE_SUB(NOW(), INTERVAL 1 YEAR)
GROUP BY YEAR(order_date), MONTH(order_date)
ORDER BY year DESC, month DESC;
```

SQL Quick Reference

Database Commands

```sql
CREATE DATABASE db_name;
DROP DATABASE db_name;
USE db_name;
SHOW DATABASES;
SHOW TABLES;
```

Table Commands

```
CREATE TABLE table_name (column datatype constraint);
DROP TABLE table_name;
TRUNCATE TABLE table_name;
ALTER TABLE table_name ADD COLUMN column_name datatype;
ALTER TABLE table_name DROP COLUMN column_name;
ALTER TABLE table_name MODIFY column_name new_datatype;
ALTER TABLE table_name CHANGE old_name new_name datatype;
```

Data Commands

```
INSERT INTO table_name (col1, col2) VALUES (val1, val2);
SELECT * FROM table_name;
SELECT col1, col2 FROM table_name WHERE condition;
UPDATE table_name SET col1 = val1 WHERE condition;
DELETE FROM table_name WHERE condition;
```

Filtering and Sorting

```
SELECT * FROM table WHERE column = value;
SELECT * FROM table WHERE column BETWEEN val1 AND val2;
SELECT * FROM table WHERE column IN (val1, val2, val3);
SELECT * FROM table WHERE column LIKE 'pattern%';
SELECT * FROM table ORDER BY column ASC/DESC;
SELECT DISTINCT column FROM table;
SELECT * FROM table LIMIT number;
```

Aggregation

```
SELECT COUNT(*) FROM table;
SELECT SUM(column) FROM table;
SELECT AVG(column) FROM table;
SELECT MAX(column) FROM table;
SELECT MIN(column) FROM table;
SELECT column, COUNT(*) FROM table GROUP BY column;
SELECT column, COUNT(*) FROM table GROUP BY column HAVING COUNT(*) > x;
```

Joins

```
SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 LEFT JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 RIGHT JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 CROSS JOIN t2;
SELECT * FROM t1 JOIN t1 AS t2 ON t1.id = t2.parent_id;
```

Set Operations

```
SELECT col FROM t1 UNION SELECT col FROM t2;
SELECT col FROM t1 UNION ALL SELECT col FROM t2;
SELECT col FROM t1 INTERSECT SELECT col FROM t2;
SELECT col FROM t1 EXCEPT SELECT col FROM t2;
```

Subqueries and Views

```
SELECT * FROM table WHERE col IN (SELECT col FROM table2);
SELECT * FROM table WHERE col > (SELECT AVG(col) FROM table);
CREATE VIEW view_name AS SELECT * FROM table WHERE condition;
DROP VIEW view_name;
```

Key Concepts Summary

Normalization

- Organize data to reduce redundancy
- Follow normal forms (1NF, 2NF, 3NF)
- Improves data integrity

Relationships

- **One-to-One**: Single record relates to one other record
- **One-to-Many**: Single record relates to multiple records
- **Many-to-Many**: Multiple records relate to multiple records

Indexes

- Speed up data retrieval
- Slow down INSERT, UPDATE, DELETE
- Use on frequently searched columns

Transactions

- Group related operations
- Ensure atomicity (all or nothing)
- Maintain consistency
- ACID properties (Atomicity, Consistency, Isolation, Durability)

Database Security

- Use strong passwords

- Limit user privileges
- Regular backups
- Encrypt sensitive data
- Audit access logs

Tips for SQL Mastery

1. **Practice Regularly**: Write queries daily
2. **Start Simple**: Begin with SELECT statements
3. **Understand Data**: Know your schema
4. **Use Aliases**: Improve readability
5. **Comment Code**: Document complex queries
6. **Test Thoroughly**: Verify results
7. **Optimize Performance**: Use appropriate indexes
8. **Follow Naming Conventions**: Consistent naming
9. **Backup Data**: Always have backups
10. **Learn Best Practices**: Follow SQL standards

Important Websites and Resources

- MySQL Official: https://www.mysql.com
- MySQL Documentation: https://dev.mysql.com/doc/
- Data Types Reference: https://dev.mysql.com/doc/refman/8.0/en/data-types.html
- SQL Tutorial Sites: W3Schools, SQLZoo, LeetCode

End-of-File

The KintsugiStack repository, authored by Kintsugi-Programmer, is less a comprehensive resource and more an Artifact of Continuous Research and Deep Inquiry into Computer Science and Software Engineering. It serves as a transparent ledger of the author's relentless pursuit of mastery, from the foundational algorithms to modern full-stack implementation.

> Made with 💚 Kintsugi-Programmer