Take a sample database and show me DDL queries on it

Here are the key Data Definition Language (DDL) queries demonstrated on a simple, sample database.

DDL commands are the part of SQL that build and modify the *structure* (the "schema") of your database objects, like tables and the database itself. The three main commands are CREATE, ALTER, and DROP.

Let's imagine we're building a database for a small company, which we'll call CompanyDB. It will have two tables:

1. Departments: To store department info (e.g., "Sales", "Engineering").
2. Employees: To store employee info and link them to a department.

---

1. CREATE **Queries**

The CREATE command is used to build new database objects.

A. Create the Database

First, we create the database container itself.

SQL

```
CREATE DATABASE CompanyDB;
```

B. Create the Tables

Now, let's create our tables *inside* CompanyDB. We must create Departments first, because Employees will refer to it.

This query creates the Departments table with a primary key that automatically increments.

SQL

```
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY AUTO_INCREMENT,
    DepartmentName VARCHAR(100) NOT NULL
);
```

Next, we create the Employees table. Notice the FOREIGN KEY constraint, which links the DeptID column in this table to the DepartmentID column in the Departments table. This enforces referential integrity.

SQL

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Email VARCHAR(100) UNIQUE NOT NULL,
    HireDate DATE,
    DeptID INT,
    FOREIGN KEY (DeptID) REFERENCES Departments(DepartmentID)
);
```

---

2. ALTER **Queries** 🖍️

The ALTER command is used to modify an *existing* database object.

A. Add a Column

Let's say we forgot to add a Salary column to our Employees table. We can add it like this:

SQL

```
ALTER TABLE Employees
ADD COLUMN Salary DECIMAL(10, 2);
```

(This adds a column named Salary that can hold a number up to 10 digits total, with 2 digits after the decimal point.)

B. Modify a Column

We realize that employee FirstName might be longer than 50 characters. Let's change its data type to VARCHAR(75).

SQL

```
ALTER TABLE Employees
MODIFY COLUMN FirstName VARCHAR(75);
```

(Note: The exact syntax for modifying a column can vary slightly between SQL dialects, e.g., ALTER COLUMN in some systems.)

C. Drop a Column

On second thought, we decide Salary is too sensitive and we want to remove it.

SQL

```
ALTER TABLE Employees
DROP COLUMN Salary;
```

---

3. DROP **Queries** 🗑️

The DROP command is used to completely delete database objects. **Be careful, this action is permanent!**

A. Drop a Table

If we want to delete the Employees table, we can run:

SQL

```
DROP TABLE Employees;
```

Important Note: You must drop the Employees table *before* you can drop the Departments table. This is because Employees has a foreign key that *depends on* Departments. If you try to drop Departments first, the database will give you an error to protect the data integrity.

Once Employees is gone, we can safely drop Departments:

SQL

```
DROP TABLE Departments;
```

B. Drop the Entire Database

To delete the entire CompanyDB and all its contents (tables, data, everything):

SQL

```
DROP DATABASE CompanyDB;
```

Summary of Commands

Command	Purpose	Example
CREATE	Builds a new database or table.	CREATE TABLE Employees (...);
ALTER	Modifies an existing table.	ALTER TABLE Employees ADD COLUMN ...;

Command	Purpose	Example
DROP	Permanently deletes an object.	DROP TABLE Employees;

Would you now like to see the DML (Data Manipulation Language) queries, like INSERT, SELECT, UPDATE, and DELETE, to show how to add and manage data *within* these tables?

DML commands

Of course. Let's continue with our CompanyDB example.

While DDL builds the "house" (the tables), DML (Data Manipulation Language) is what you use to put "furniture" (the data) into the house and move it around.

DML commands are used to **create, read, update, and delete the data** *inside* your tables. The four main commands are INSERT, SELECT, UPDATE, and DELETE.

---

1. INSERT 📥

The INSERT command adds new rows (records) to a table.

A. Add Data to Departments

Let's add two departments: "Sales" and "Engineering".

SQL

INSERT INTO Departments (DepartmentName)

VALUES ('Sales'), ('Engineering');

(This adds two new rows. We don't need to specify DepartmentID because we set it to AUTO_INCREMENT.)

B. Add Data to Employees

Now, let's add two employees. We'll assume "Sales" is DepartmentID 1 and "Engineering" is DepartmentID 2.

SQL

INSERT INTO Employees (FirstName, LastName, Email, HireDate, DeptID)

VALUES

('Jane', 'Doe', 'jane.doe@company.com', '2023-01-15', 1), -- Jane in Sales

('Mark', 'Smith', 'mark.smith@company.com', '2022-05-30', 2); -- Mark in Engineering

---

2. SELECT 🔍

The SELECT command is used to retrieve data from one or more tables. It's the command you'll use most often.

A. Get All Data from a Table

To see everything in the Employees table:

SQL

SELECT * FROM Employees;

*(The * is a wildcard for "all columns".)*

B. Get Specific Columns

If you only want to see the names and emails:

SQL

SELECT FirstName, LastName, Email FROM Employees;

C. Filter Data with WHERE

To find a specific employee or group:

SQL

SELECT * FROM Employees WHERE FirstName = 'Jane';

...or...

SQL

SELECT * FROM Employees WHERE DeptID = 2; -- Get all employees in Engineering

D. Get Data from Multiple Tables (using JOIN**)**

This is the most powerful part of SELECT. Let's get the employee's name *and* their department's name.

SQL

SELECT

e.FirstName,

e.LastName,

d.DepartmentName

FROM

Employees e

JOIN

Departments d ON e.DeptID = d.DepartmentID;

Result: | FirstName | LastName | DepartmentName | | :--- | :--- | :--- | | Jane | Doe | Sales | | Mark | Smith | Engineering |

---

3. UPDATE 🔄

The UPDATE command modifies existing records in a table.

⚠️ **Warning:** Always use a WHERE clause with UPDATE! If you forget it, you will update **every single row** in the table.

Let's say Jane Doe gets married and changes her last name to "Jones". We also need to update her email.

SQL

UPDATE Employees

SET

LastName = 'Jones',

Email = 'jane.jones@company.com'

WHERE

EmployeeID = 1; -- Assuming Jane's ID is 1

---

4. DELETE ❌

The DELETE command removes one or more rows from a table.

⚠️ **Warning:** Just like UPDATE, always use a WHERE clause with DELETE! If you forget it, you will delete **all data** from your table.

Let's say Mark Smith (EmployeeID 2) leaves the company.

SQL

DELETE FROM Employees

WHERE EmployeeID = 2;

(Mark Smith's record is now gone. The Departments table is unaffected.)

Summary of Commands

Command	Purpose	Example
INSERT	Adds new rows of data.	INSERT INTO Employees (...) VALUES (...);
SELECT	Retrieves data.	SELECT * FROM Employees WHERE ...;
UPDATE	Modifies existing data.	UPDATE Employees SET ... WHERE ...;
DELETE	Removes rows of data.	DELETE FROM Employees WHERE ...;

Would you like to cover the next categories, DCL (Data Control Language) or TCL (Transaction Control Language)?

imple Queries: Simple

Queries with Aggregate functions, Queries with Aggregate functions (group by and having clause),

Queries involving- Date Functions, String Functions , Math Functions

Let's explore these query types using our CompanyDB database.

To make these examples work, let's assume our tables have a bit more data in them, and we've added a Salary column.

📈 **Our Sample Data**

Departments **Table:** | DepartmentID | DepartmentName | | :--- | :--- | | 1 | Sales | | 2 | Engineering | | 3 | Marketing |

Employees **Table:** | EmployeeID | FirstName | LastName | Email | HireDate | DeptID | Salary | | :--- | :--- | :--- | :--- | :--- | :--- | :--- | | 1 | Jane | Doe | jane.doe@co.com | 2023-01-15 | 1 | 70000 | | 2 | Mark | Smith | mark.smith@co.com | 2022-05-30 | 2 | 95000 | | 3 | Alice | Johnson | alice.j@co.com | 2023-03-10 | 1 | 72000 | | 4 | Bob | Brown | bob.brown@co.com | 2021-11-20 | 2 | 110000 | | 5 | Sarah | Lee | sarah.lee@co.com | 2023-08-01 | 3 | 60000 | | 6 | Chris | Walter | chris.w@co.com | 2022-07-15 | 2 | 105000 |

---

1. Simple Queries (Filtering & Sorting)

These are your basic SELECT statements, using WHERE to filter rows and ORDER BY to sort them.

Query: Get all employees from the 'Engineering' department (DeptID 2) and sort them by salary, highest to lowest.

SQL

SELECT FirstName, LastName, Salary

FROM Employees

WHERE DeptID = 2

ORDER BY Salary DESC;

Result: | FirstName | LastName | Salary | | :--- | :--- | :--- | | Bob | Brown | 110000 | | Chris | Walter | 105000 | | Mark | Smith | 95000 |

---

2. Queries with Aggregate Functions

Aggregate functions perform a calculation on a set of rows and return a single value.

- COUNT(): Counts the number of rows.
- SUM(): Adds all values.
- AVG(): Calculates the average.
- MAX(): Finds the highest value.
- MIN(): Finds the lowest value.

Query: Get the total number of employees, the average company salary, and the total amount paid in salaries.

SQL

SELECT

COUNT(EmployeeID) AS TotalEmployees,

AVG(Salary) AS AverageSalary,

SUM(Salary) AS TotalPayroll

FROM Employees;

Result: | TotalEmployees | AverageSalary | TotalPayroll | | :--- | :--- | :--- | | 6 | 85333.33 | 512000 |

---

3. Aggregate Queries with GROUP BY **and** HAVING

This is where you combine aggregation with grouping.

The GROUP BY **Clause**

GROUP BY splits the rows into groups based on a column and runs the aggregate function on each group.

Query: What is the average salary *per department*?

SQL

SELECT

DeptID,

AVG(Salary) AS AvgDeptSalary

FROM Employees

GROUP BY DeptID;

Result: | DeptID | AvgDeptSalary | | :--- | :--- | | 1 | 71000.00 | | 2 | 103333.33 | | 3 | 60000.00 |

The HAVING **Clause**

HAVING is used to filter the *groups* after they've been created.

Key Difference:

- WHERE filters **rows** *before* they are grouped.
- HAVING filters **groups** *after* they are grouped.

Query: Show me the departments whose average salary is over $80,000.

SQL

SELECT

DeptID,

AVG(Salary) AS AvgDeptSalary

FROM Employees

GROUP BY DeptID

HAVING AVG(Salary) > 80000;

Result: | DeptID | AvgDeptSalary | | :--- | :--- | | 2 | 103333.33 |

---

4. Queries with Date Functions 🗓️

These functions let you manipulate and extract information from date/time columns.

Query: Find all employees hired in 2023 and show how many days they've been with the company.

SQL

```
SELECT
    FirstName,
    LastName,
    HireDate,
    DATEDIFF(CURDATE(), HireDate) AS DaysEmployed
FROM Employees
WHERE YEAR(HireDate) = 2023;
```

- YEAR(HireDate) extracts the year from the HireDate.
- CURDATE() returns the current date.
- DATEDIFF(date1, date2) calculates the difference in days.

---

5. Queries with String Functions 🔡

These functions let you manipulate text data.

Query: Create a full name for each employee and show their email in all uppercase.

SQL

```
SELECT
    CONCAT(FirstName, ' ', LastName) AS FullName,
    UPPER(Email) AS EmailUpper
FROM Employees;
```

- CONCAT() joins two or more strings together.
- UPPER() converts a string to uppercase. (LOWER() does the opposite).

Query: Find all employees who have a .com email address (not .co.com).

SQL

```
SELECT Email
FROM Employees
WHERE Email LIKE '%.com';
```

- LIKE is a pattern-matching operator. The % is a wildcard for "any characters".

---

6. Queries with Math Functions 🧮

These functions perform mathematical operations.

Query: Show each employee's salary, their salary after a 5% raise, and the average salary rounded to two decimal places.

SQL

```
SELECT
    FirstName,
    Salary,
    Salary * 1.05 AS SalaryWithRaise,
    ROUND(AVG(Salary), 2) AS RoundedAverage
FROM Employees
GROUP BY EmployeeID, FirstName, Salary; -- Group by to show individual + aggregate
```

(Note: This is a slightly advanced query; a simpler version would just show the rounded average.)

A simpler example:

SQL

```
SELECT
    ROUND(AVG(Salary), 2) AS RoundedAvgSalary
FROM Employees;
```

- ROUND(number, decimals) rounds a number to a specified number of decimal places.

Would you like to move on to DCL (Data Control Language) to discuss user permissions, or TCL (Transaction Control Language) for managing changes?

Implementation of different types of Joins

• Inner Join

• Outer Join

• Natural Join etc

JOIN clauses are used to combine rows from two or more tables based on a related column between them.

To demonstrate, let's use two tables, Employees and Departments.

Departments **Table:** | DepartmentID | DepartmentName | | :--- | :--- | | 1 | Sales | | 2 | Engineering | | 3 | Marketing | | 4 | HR |

Employees **Table:** | EmployeeID | Name | DeptID | | :--- | :--- | :--- | | 101 | Jane | 1 | | 102 | Mark | 2 | | 103 | Alice | 1 | | 104 | Bob | NULL |

Note our data:

- The **HR** department (ID 4) has no employees.
- **Bob** (ID 104) does not belong to any department (DeptID is NULL).

This will help illustrate the differences between the joins.

🧩 Inner Join

This is the most common join. It returns **only the rows that have a match in both tables**. Any rows without a match (like "Bob" or the "HR" department) are excluded.

SQL

```
SELECT
    e.Name,
    d.DepartmentName
FROM
    Employees e
INNER JOIN
    Departments d ON e.DeptID = d.DepartmentID;
```

Result: | Name | DepartmentName | | :--- | :--- | | Jane | Sales | | Mark | Engineering | | Alice | Sales |

⬅️ **Left Outer Join (or** LEFT JOIN**)**

This returns **all rows from the *left* table** (the first one, Employees) and the matching rows from the *right* table (Departments).

If an employee has no match (like "Bob"), they are still included, but the department columns will be NULL.

SQL

SELECT

e.Name,

d.DepartmentName

FROM

Employees e

LEFT JOIN

Departments d ON e.DeptID = d.DepartmentID;

Result: (Includes "Bob", who has no department) | Name | DepartmentName | | :--- | :--- | | Jane | Sales | | Mark | Engineering | | Alice | Sales | | Bob | NULL |

➡️ **Right Outer Join (or** RIGHT JOIN**)**

This is the opposite of a LEFT JOIN. It returns **all rows from the *right* table** (Departments) and the matching rows from the *left* table (Employees).

If a department has no match (like "HR"), it is still included, but the employee columns will be NULL.

SQL

SELECT

e.Name,

d.DepartmentName

FROM

Employees e

RIGHT JOIN

Departments d ON e.DeptID = d.DepartmentID;

Result: (Includes "HR", which has no employees) | Name | DepartmentName | | :--- | :--- | | Jane | Sales | | Mark | Engineering | | Alice | Sales | | NULL | HR |

↔ Full Outer Join

This returns **all rows when there is a match in either the left or right table**. It's a combination of LEFT and RIGHT JOIN.

If a row has no match, the columns from the other table will be NULL. This query will include *both* "Bob" and the "HR" department.

SQL

SELECT

e.Name,

d.DepartmentName

FROM

Employees e

FULL OUTER JOIN

Departments d ON e.DeptID = d.DepartmentID;

(Note: MySQL doesn't natively support FULL OUTER JOIN. You would achieve it by combining a LEFT JOIN and a RIGHT JOIN with a UNION.)

Result: | Name | DepartmentName | | :--- | :--- | | Jane | Sales | | Mark | Engineering | | Alice | Sales | | Bob | NULL | | NULL | HR |

Natural Join

A NATURAL JOIN is a special type of INNER JOIN that doesn't use an ON clause. Instead, it **automatically joins on all columns that have the same name** in both tables.

In our example, this won't work because the columns are named DeptID and DepartmentID. Let's *pretend* both columns were named DeptID.

SQL

-- This only works if both tables have a column named "DeptID"

SELECT Name, DepartmentName

FROM Employees

NATURAL JOIN Departments;

⚠️ **Warning:** Most developers avoid NATURAL JOIN. It's dangerous because if someone adds a new column with the same name to both tables (e.g., DateCreated), the join logic will automatically change, which can break your query in unexpected ways. It's almost always better to be explicit and use an ON clause.

💥 Cross Join

A CROSS JOIN returns the **Cartesian product** of the two tables. This means it matches **every row from the left table with every row from the right table**. No ON clause is needed.

SQL

SELECT

e.Name,

d.DepartmentName

FROM

Employees e

CROSS JOIN

Departments d;

Result: (4 employees * 4 departments = 16 rows) | Name | DepartmentName | | :--- | :--- | | Jane | Sales | | Jane | Engineering | | Jane | Marketing | | Jane | HR | | Mark | Sales | | Mark | Engineering | | ...and so on for all 16 combinations. |

Would you like to see how to implement Subqueries (queries inside other queries)?

To perform SQL query that demonstrate following:

Search conditions, Summary queries, Sub- queries, Subqueries- With IN clause, With EXISTS clause

Here are SQL queries demonstrating those concepts, using our sample Employees and Departments tables.

Sample Data

Employees **Table:** | EmployeeID | Name | DeptID | Salary | | :--- | :--- | :--- | :--- | | 1 | Jane | 1 | 70000 | | 2 | Mark | 2 | 95000 | | 3 | Alice | 1 | 72000 | | 4 | Bob | 2 | 110000 | | 5 | Sarah | 3 | 60000 | | 6 | Chris | 2 | 105000 |

Departments **Table:** | DepartmentID | DepartmentName | | :--- | :--- | | 1 | Sales | | 2 | Engineering | | 3 | Marketing |

---

1. 🔍 Search Conditions

Search conditions use the WHERE clause to filter rows based on specific criteria, using operators like =, >, <, AND, OR, LIKE, and BETWEEN.

Query: Find all employees in the Sales department (DeptID 1) **OR** any employee (from any department) whose name starts with 'B'.

SQL

```
SELECT EmployeeID, Name, DeptID
FROM Employees
WHERE DeptID = 1 OR Name LIKE 'B%';
```

Result: | EmployeeID | Name | DeptID | | :--- | :--- | :--- | | 1 | Jane | 1 | | 3 | Alice | 1 | | 4 | Bob | 2 |

---

2. 📊 Summary Queries

Summary queries use aggregate functions (COUNT, SUM, AVG, MAX, MIN) to perform calculations on a set of rows, often combined with a GROUP BY clause.

Query: Find the number of employees in each department, the total salary for each department, and the average salary for each department.

SQL

```
SELECT
    DeptID,
    COUNT(EmployeeID) AS NumberOfEmployees,
    SUM(Salary) AS TotalPayroll,
    AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY DeptID;
```

Result: | DeptID | NumberOfEmployees | TotalPayroll | AverageSalary | | :--- | :--- | :--- | :--- | | 1 | 2 | 142000 | 71000.00 | | 2 | 3 | 310000 | 103333.33 | | 3 | 1 | 60000 | 60000.00 |

---

3. nested Subqueries

A subquery is a SELECT statement nested inside another query. This example uses a *scalar subquery*, which returns a single value.

Query: Find all employees who earn more than the company's average salary.

- The *inner query* (SELECT AVG(Salary) FROM Employees) runs first to find the average (which is 85333.33).
- The *outer query* then finds everyone who earns more than that value.

SQL

SELECT Name, Salary

FROM Employees

WHERE Salary > (SELECT AVG(Salary) FROM Employees);

Result: | Name | Salary | | :--- | :--- | | Mark | 95000 | | Bob | 110000 | | Chris | 105000 |

---

4. 📋 Subqueries with IN **Clause**

The IN operator checks if a value matches any value in a list. A subquery can be used to generate this list.

Query: Find the names of all employees who work in the 'Engineering' or 'Marketing' departments.

- The *inner query* finds the DepartmentIDs for 'Engineering' and 'Marketing' (which are 2 and 3).
- The *outer query* finds all employees whose DeptID is **IN** that list (2, 3).

SQL

SELECT Name, DeptID

FROM Employees

WHERE DeptID IN (SELECT DepartmentID

FROM Departments

WHERE DepartmentName = 'Engineering' OR DepartmentName = 'Marketing');

Result: | Name | DeptID | | :--- | :--- | | Mark | 2 | | Bob | 2 | | Sarah | 3 | | Chris | 2 |

---

5. ❓ Subqueries with EXISTS Clause

The EXISTS operator checks if a subquery returns *any rows* (evaluates to TRUE) or *no rows* (evaluates to FALSE). It's used to check for existence.

This is often a *correlated subquery*, where the inner query refers to the outer query.

Query: Find all departments that *have at least one employee*.

- The *outer query* iterates through each Department.
- For each department, the *inner query* checks: "Does there EXISTS any employee whose DeptID matches this department's ID?"

SQL

SELECT DepartmentName

FROM Departments d

WHERE EXISTS (SELECT 1

FROM Employees e

WHERE e.DeptID = d.DepartmentID);

(The SELECT 1 is a convention; you can select any column. It just checks if any row is returned.)

Result: | DepartmentName | | :--- | | Sales | | Engineering | | Marketing |

Would you like to see how to implement TCL (Transaction Control Language) commands like COMMIT and ROLLBACK?

Perform Following

• Creating Database /Table Space

• Managing Users: Create User, Delete User

• Managing roles:-Grant, Revoke

Here are the commands for performing those database administration tasks.

These commands are part of **DCL (Data Control Language)** and **DDL (Data Definition Language)**, used for defining and managing access to your database.

---

1. 🏗️ Creating a Database

This DDL command creates a new, empty database container.

SQL

CREATE DATABASE CompanyDB;

You can also specify character sets and collation (rules for sorting text) for more advanced setups:

SQL

CREATE DATABASE CompanyDB

CHARACTER SET utf8mb4

COLLATE utf8mb4_unicode_ci;

A Note on "Table Space": A TABLESPACE is a more advanced concept, primarily in database systems like Oracle or PostgreSQL. It refers to the *physical file on the disk* where your table data is stored. While you *can* create them, for most day-to-day work (especially in MySQL), CREATE DATABASE is the command you will use.

---

2. 👤 Managing Users

You need to create users to control who can access the database. A user account is defined by a username and a host (where the user is connecting *from*).

Create User

This command creates a new user, 'app_user', who can *only* connect from localhost (the same machine as the database) and sets their password.

SQL

CREATE USER 'app_user'@'localhost'

IDENTIFIED BY 'a-very-strong-password123';

- 'app_user'@'localhost': User 'app_user' connecting from the local machine.
- 'app_user'@'%': User 'app_user' connecting from **any** IP address (less secure).

Delete User

This command completely removes a user account.

SQL

DROP USER 'app_user'@'localhost';

---

3. 🛡️ Managing Roles and Permissions (Grant & Revoke)

Just creating a user isn't enough; they have no permissions by default. You must GRANT them specific privileges.

A. Granting Privileges (Grant)

GRANT gives permissions to a user.

Example 1: Grant specific permissions on one table. Let's give our 'app_user' the ability to SELECT and INSERT data, but only on the Employees table in CompanyDB.

SQL

```
GRANT SELECT, INSERT
ON CompanyDB.Employees
TO 'app_user'@'localhost';
```

Example 2: Grant all permissions on a whole database. Let's create an admin user and give them full control over the CompanyDB.

SQL

```
GRANT ALL PRIVILEGES
ON CompanyDB.* TO 'admin_user'@'localhost';
```

- CompanyDB.* means "all tables within CompanyDB."

B. Revoking Privileges (Revoke)

REVOKE takes permissions away.

Example 1: Revoke a specific permission. Let's decide our 'app_user' should no longer be able to INSERT records.

SQL

```
REVOKE INSERT
ON CompanyDB.Employees
FROM 'app_user'@'localhost';
```

(The user can still SELECT data, but INSERT will now be denied.)

Example 2: Revoke all permissions. To revoke all privileges on CompanyDB from our admin:

SQL

```
REVOKE ALL PRIVILEGES
```

ON CompanyDB.* FROM 'admin_user'@'localhost';

👑 Managing Roles (A More Powerful Method)

Your prompt specifically mentioned "roles." A **role** is a named *collection* of privileges. This is much more efficient than granting permissions one by one. You grant permissions *to the role*, and then grant *the role* to users.

Here is the full workflow:

Step 1: Create a Role Let's create a 'read_only' role.

SQL

CREATE ROLE 'read_only';

Step 2: Grant Privileges to the Role Give this role SELECT permission on all tables in CompanyDB.

SQL

GRANT SELECT ON CompanyDB.* TO 'read_only';

Step 3: Grant the Role to a User Now, let's create a new user and assign them this 'read_only' role.

SQL

CREATE USER 'analyst'@'localhost' IDENTIFIED BY 'password';

GRANT 'read_only' TO 'analyst'@'localhost';

(The 'analyst' user now has all the permissions of the 'read_only' role.)

Step 4: Revoke a Role from a User If the analyst's job changes, you can simply revoke the role.

SQL

REVOKE 'read_only' FROM 'analyst'@'localhost';

⭐ Important Final Command

After creating users or granting permissions, many database systems (especially older MySQL versions) require you to run this command to reload the permission tables:

SQL

FLUSH PRIVILEGES;

Would you like to move on to TCL (Transaction Control Language) commands like COMMIT and ROLLBACK?

Implementation of Database Backup & Recovery commands, Rollback, Commit, Savepoint

Here's an implementation of Transaction Control Language (TCL) commands and an explanation of backup/recovery commands.

🛡️ Transaction Control Language (TCL)

TCL commands are used to manage *transactions* in a database. A transaction is a group of SQL statements that are treated as a single unit of work. They must **all** succeed, or **none** of them do.

Think of it as editing a document. The commands you run are your *changes*, but they aren't permanent until you hit "Save" (COMMIT). If you don't like your changes, you can "Close without Saving" (ROLLBACK).

Let's use our Employees table for these examples.

COMMIT 💾

Makes all changes in the current transaction permanent.

ROLLBACK 🔄

Undoes all changes made in the current transaction, restoring the data to its state at the last COMMIT.

Simple COMMIT **and** ROLLBACK **Example:**

Let's try to delete an employee, then change our minds.

SQL

-- Start a new transaction

START TRANSACTION;

-- Delete Sarah from the Employees table (ID 5)

DELETE FROM Employees WHERE EmployeeID = 5;

-- At this point, Sarah appears to be gone *for you*,

-- but the change is not permanent.

-- Let's check the table (in this same session).

SELECT * FROM Employees;

-- Sarah's row is missing.

-- Now, we change our mind. We don't want to delete her.

ROLLBACK;

-- Let's check the table again.

SELECT * FROM Employees;

-- Sarah is back! The DELETE was undone.

If we had typed COMMIT; instead of ROLLBACK;, the deletion would have been permanent.

SAVEPOINT

Sets a named checkpoint *within* a transaction. This allows you to ROLLBACK to a specific point without undoing the entire transaction.

SAVEPOINT **and** ROLLBACK TO **Example:**

Let's give two employees a raise, but we make a mistake on the second one.

SQL

-- Start a new transaction

START TRANSACTION;

-- 1. Give Jane (ID 1) a raise

UPDATE Employees SET Salary = 75000 WHERE EmployeeID = 1;

-- 2. Create a checkpoint before we continue

SAVEPOINT before_bob_raise;

-- 3. Give Bob (ID 4) a raise

UPDATE Employees SET Salary = 115000 WHERE EmployeeID = 4;

-- Let's check our work...

-- Oh no! Bob's raise was supposed to be 112000, not 115000.

-- We want to undo *only Bob's raise*, not Jane's.

-- 4. Roll back to our savepoint

ROLLBACK TO SAVEPOINT before_bob_raise;

-- At this point, Jane's raise (Salary = 75000) is still staged.

-- Bob's raise (Salary = 115000) has been undone.

-- 5. Now, let's apply the *correct* raise for Bob

UPDATE Employees SET Salary = 112000 WHERE EmployeeID = 4;

-- 6. Everything looks good. Let's make all changes permanent.

COMMIT;

Now, both Jane's and Bob's raises are saved to the database.

📥 📤 Database Backup & Recovery (Utilities)

These are **not** SQL commands. They are high-level utility programs that you run from your computer's **command line (terminal)**, not from inside a SQL editor.

Here are the most common examples using **MySQL/MariaDB**.

1. Database Backup (mysqldump**)**

mysqldump is a command-line utility that "dumps" the entire structure (CREATE TABLE) and data (INSERT INTO) of your database into a single .sql file.

Command:

Bash

```
mysqldump -u [username] -p [database_name] > [backup_file_name].sql
```

Example: To back up our CompanyDB:

Bash

```
mysqldump -u root -p CompanyDB > companydb_backup_2025-11-04.sql
```

- mysqldump: The program.
- -u root: Use the database user root.
- -p: Prompt for the user's password.
- CompanyDB: The name of the database to back up.
- >: A standard terminal operator that means "redirect the output to..."
- companydb_backup...sql: The new file that will be created with all the SQL commands.

2. Database Recovery (mysql**)**

To "recover," you do the opposite. You use the mysql client utility to read the .sql backup file and execute all the commands inside it, rebuilding the database.

Command:

Bash

```
mysql -u [username] -p [database_name] < [backup_file_name].sql
```

Example: To restore CompanyDB from our backup:

Bash

```
mysql -u root -p CompanyDB < companydb_backup_2025-11-04.sql
```

- mysql: The client program.
- -u root -p: Sign in as the root user.
- CompanyDB: The name of the database you want to restore *into*. (This database usually needs to be created first, e.g., CREATE DATABASE CompanyDB;).
- <: A standard terminal operator that means "get input from..."
- companydb_backup...sql: The backup file we want to read.

This command will run all the CREATE TABLE and INSERT INTO statements from the backup file, effectively restoring your database.

Create View and index for database tables with a large number of records.

Here's how to create views and indexes, especially for tables with a large number of records.

📈 **Indexes**

An index is a special data structure that your database uses to find rows much faster. Think of it as the index at the back of a large textbook: instead of reading the whole book to find a topic, you look it up in the index and go straight to the correct page.

For a table with millions of records, an index is **essential**. Without an index, a query like SELECT * FROM Employees WHERE LastName = 'Smith'; would force the database to scan *every single row* (a "full table scan") to find the 'Smith' records.

With an index on the LastName column, the database can instantly find those records, making the query thousands of times faster.

How to Create an Index

You create an index on one or more columns that you frequently use in your WHERE clauses or JOIN conditions.

SQL

CREATE INDEX idx_lastname

ON Employees (LastName);

- idx_lastname: This is the name we are giving to our new index.
- ON Employees (LastName): This specifies that the index should be created on the LastName column of the Employees table.

You can also create an index on a foreign key column, which is highly recommended for speeding up joins.

SQL

CREATE INDEX idx_deptid

ON Employees (DeptID);

(This will make JOIN operations between Employees and Departments much faster.)

When to use Indexes:

- On **Primary Key** columns (most databases do this automatically).
- On **Foreign Key** columns used in JOINs.

- On columns frequently used in WHERE clauses (like LastName, Email, Status).
- On columns used in ORDER BY clauses to speed up sorting.

Warning: Don't just add indexes to every column. Indexes take up disk space and can slightly slow down INSERT, UPDATE, and DELETE operations, because the index has to be updated along with the data.

👓 Views

A **View** is a *virtual table* whose content is defined by a query. It doesn't store any data itself; it's a saved SELECT statement that you can query just like a regular table.

For large tables, views are great for simplifying complex queries and restricting access to data.

How to Create a View

Let's say you frequently need to see employee names and their department names, which requires a JOIN every time. You can save this join as a view.

SQL

```
CREATE VIEW v_EmployeeDetails AS
SELECT
    e.EmployeeID,
    e.FirstName,
    e.LastName,
    e.Email,
    d.DepartmentName
FROM
    Employees e
JOIN
    Departments d ON e.DeptID = d.DepartmentID;
```

- v_EmployeeDetails: This is the name of our new view.
- AS SELECT ...: This is the query that defines the view.

Now, instead of writing that whole JOIN query every time, you can just do this:

SQL

-- This simple query...

SELECT FirstName, DepartmentName

FROM v_EmployeeDetails

WHERE DepartmentName = 'Sales';

...will automatically run the complex JOIN query in the background.

Benefits of Views:

- **Simplicity:** They hide complex joins and calculations. A user can query the view without needing to know the underlying table structure.
- **Security:** You can create a view that only shows certain "safe" columns (e.g., hiding the Salary column). You can then give users permission to the view, but not the base table.
- **Consistency:** All users get the same, consistent view of the data, as the join logic is stored in one place.