Date: __/__/20__ Day: _____

Hand written Notes (SQL MySQL, NoSQL)

Written by:

Mirza Yasir Abdullah Baig

Databases

- Basics
- Functions of DBMS
- Types of databases
- Databases in Different technologies
- Database Jobs
- Tip and tricks

SQL

- Introduction
- Data types
- Operators
- Commands
- SQL Database
- SQL Tables
- SQL Queries
- SQL Clauses
- SQL Operators
- SQL Aggregate Functions
- Data Constraints
- SQL Joins
- SQL Functions
- SQL Views
- SQL Indexes
- SQL Subquery

Date: __/__/20__ Day: _____

- Advanced SQL Topics

MYSQL

- MySQL Basics
- MySQL User Management
- MySQL Managing Databases
- MySQL Managing Tables
- MySQL Query
- MySQL Clauses
- MySQL operators
- MySQL Aggregate Functions
- My SQL Data Constraints
- MySQL Joining Data
- MySQL Functions
- MySQL Views
- MySQL Indexes
- My SQL Triggers
- MySQL Advanced Topics
- MySQL Jobs and Opportunities

NoSQL (MongoDB)

- MongoDB Introduction
- MongoDB Query Api
- MongoDB Create DB
- MongoDB Collection
- MongoDB Insert
- MongoDB Find
- MongoDB Update
- MongoDB Delete
- MongoDB Query Operation
- MongoDB Update Operators

Date: __/__/20 Day: _____

- MongoDB Aggregations
- MongoDB Indexing Search
- MongoDB Validation
- MongoDB Data Api
- MongoDB Drivers
- MongoDB Node.js Drivers
- MongoDB Charts.

DataBases

What is Data and Database?

- Data: Any information - numbers, text, images, sounds.
- Database: A structured collection of data, like a digital library. Helps store, organize, and manage information efficiently.

What is DBMS?

DBMS (Database Management System) = Software to manage databases).

- Main Functions:
  - Data Definition: Define tables, columns & relationships.
  - Data Manipulation: Add, update, delete and read data.
  - Data Security: Allow only authorized access
  - Data Integrity: Keep data accurate and consistent
  - Concurrency Control: Manage Multiple users at once.
  - Backup and recovery: Save and restore data in case of failure.

Types of Databases

- Relational Databases (SQL):

Data in tables (rows and columns). Use SQL

Date: __/__/20__ Day: _____

Example: MySQL, PostgreSQL, Oracle, SQL server etc.

- NoSQL databases:

Handle big, flexible, or unstructured data

- Document: MongoDB, Couchbase
- Key-Value: Cassan Redis, DynamoDB
- Column-Family: Cassandra, HBase
- Graph: Neo4j, Amazon Neptune

- Cloud Database:

Online, Scalable, reliable

Example: Amazon RDS, Aurora, Azure SQL, Google Cloud SQL, Firestore

- In-Memory Databases: -

Data in RAM, → very fast

Example: Redis, Memcached

. Time-Series Databases:

For time-stamped data (IoT, sensors)

Examples: InfluxDB, Prometheus.

- NewSQL Databases:

Mix of SQL + NoSQL Features

Examples: CockroachDB, Google Spanner

- Object-Oriented Databases:

Store data as objects (OOP style)

- Example: db4o, ObjectDB

Date: __/__/20__ Day: _____

• Hierarchical Databases:

Tree-like structure (parent-child)

Example: IMS

• Network Databases

Record connections like a network (many-to-many)

Example: IDMS

• Centeralized Database:

All data stored in one central server.

Example: ISAM

• Operational Databases

For daily business operations

Example: Aurora, Teradata

Learning Path

Start → Basics → SQL → Design → Advanced

• Learn SQL and databases

• Database design (E-commerce, social media, flight system)

• Practice

• Built projects

Database Design

• Proper design helps in performance and scaling.

• Steps: ER diagrams → Schema design → Normalization

. Database Connectivity

• Frontend to DB

• Backend to DB

• API to DB

• Mechanisms: ORM, ODM, ODBC

Date: __/__/20__ Day: _____

Databases in Different Technologies

- Web development - Blogs, e-commerce, → SQL/NoSQL
- Mobile Development: Fast, lightweight databases
- Devops: High availability, scalable
- Data Engineering: Handle big data, pipelines
- Artifical Intelligence: Large scalable data for training model
- Cloud Computing: Reliable, integrates with cloud.
- Block Chain/ Web3 - Focus on security, transparency.

Summary

Databases may look complex at first, but with practice they become easy.

- Start small → Learn SQL, → Explore DBMS → Built projects
- Keep ~~practin~~ practicing, stay update, & you will master databases.

SQL

What is SQL?

- SQL (Structured Query Language) is the standard language for working with relational databases (data in tables)

It lets you: store, read, update, delete, and manage data easily.

Known For being easy to learn yet very powerful

Used everywhere: business, datascience, web, AI, finance, healthcare etc

How SQL works?

When you write an SQL query, this is what happens:

Date: __/__/20 Day: _____

- Input → You type a query (SELECT, INSERT, UPDATE, DELETE)
- Parsing → SQL engine check syntax and structure
- Optimization → Finds the Fastest way to run Query
- Execution → Database performs the task
- Output → shows results or success message

Setting Up SQL (MySQL Example)

Before writing query, set up your database environment.

- Install MySQL on your Computer
- Start MySQL server
- Open MySQL command line

Now you can write queries

First SQL Program(Hello world Example)

- Create a database

```
CREATE DATABASE test_db;
```

- Use that database

```
USE test_db;
```

- Create a table

```
CREATE TABLE greetings (
    id INT AUTO_INCREMENT PRIMARY KEY,
    message VARCHAR (255)
);
```

- Insert Data

```
INSERT INFO greetings (message)
VALUES "Hello world"
```

Date: __/__/20__ Day: _____

• Show Data:

SELECT message FROM greetings;

* output: Hello, world.

Why Learn SQL?

SQL is everywhere! It's used in almost all fields involving data:

- Data Science and Analytics:
- Machine Learning and AI
- Web development
- Cloud and big Data.
- Blockchain and Web3
- E-Commerce
- Healthcare
- Banking / Finance

Key Parts of SQL System

- Database : Collection of organized data
- Tables : Like spreadsheets (rows = records, columns = field)
- Indexes : Speed up search
- Views : Saved queries that act like virtual Table
- Stored Procedures : Pre-written SQL programs stored in DB
- Transaction : Groups of operations that run together
- Security and Permissions : Controls who can access / change data
- Joins : Combine data from multiple tables

Rules For Writing SQL

- End each command with semicolon (;)
- Keywords (SELECT, INSERT) are not case-sensitive
- Use spaces / indentation to make queries readable
- Don't use reserved words as table/column names.

Date: __/__/20__ Day: _____

- Comments
  - -- single line
  - /* multi line */
- Constraints: Ensure accuracy (e.g NOT NULL, UNIQUE)
- String Value: go in single quotes 'text'
- Naming rule: Start with letter, max 30 char, only letters/numbers/underscore

Types of SQL commands

- DDL (Data Defination Language) - define Structure
  - CREATE - make new table, view, object
  - ALTER - change exisiting clo' table
  - DROP - delete table/object
  - TRUNCATE - remove all rows but keep table structure

- DML (Data Manipulation Language) - work with data
  - INSERT - add new record
  - UPDATE - change data
  - DELETE - remove data

- DQL (Data Query Language) - get data
  - SELECT - retrieve data from tables

- DCL (Data Control language) - Control access
  - GRANT - give premissions
  - REVOKE - remove premissions

- TCL (Transaction Control language) - manage transaction
  - COMMIT - Save changes permanently
  - ROLLBACK - undo changes
  - SAVEPOINT - mark a point to roll back later

Date: __/__/20__ Day: _____

Benefits of SQL

- Efficient → Handles big data and complex queries fast
- Standardized → Works across platform (ANSI/ISO standard)
- Scalable → Works for small apps or enterprise systems
- Flexible → Supports custom functions, bussiness logic.

Limitations of SQL

- Advanced features (indexing, tuning) need enterprise
- Scalability issues with unstructured / huge distributed data
- Different versions - MySQL, Oracle etc have unique features
- Not real-time: Traditional SQL struggles with real-time analytics

SQL Data Types

In SQL, every column must have a data type, (decides what kind of values can be stored)

correct choice = better storage, speed, accuracy, consistency

Benefits of Choosing right Data type

- Save memory (efficient storage)
- Faster queries (better indexing / sorting)
- Ensures correct values (validation and integrity)
- Prevents errors in calculations.

Main SQL Data Types.

- Numeric Data types
  - Used for numbers (counts, money, calculations)
  - Two types
    - Exact Numeric: when accuracy is important (finance, quantities)
      - BIGINT → very large Integers

Date: __/__/20 Day: _____

- INT → Standard integers
- SMALLINT, TINYINT → smaller numbers
- DECIMAL / NUMERIC → exact decimal numbers
- MONEY, SMALLMONEY → currency value

- Approximate Numeric: when small errors are okay (science, ML, big data)
  - FLOAT → large range, not exact
  - REAL → smaller version of FLOAT

- Character and String Data types
- For text (name, emails, description)
- Non-Unicode (English/limited chars):
  - CHAR: fixed length text
  - VARCHAR: Variable length text
  - TEXT: very long text
- Uni-code (multi-language support):
  - NCHAR, NVARCHAR, NVARCHAR(MAX)

Data and Time Data types
- For dates and timestamps
- DATE → only year month-day
- TIME → only time (hours-mins-secs)
- DATE/TIME → both date and time

Binary data type
- For storing files (not text/numbers)
- BINARY → fixed length binary data
- VARBINARY → variable length binary data
- IMAGE → store images, videos files.

Date __/__ /20 Day: _____

BOOLEAN DATA Type

- Strores TRUE or FALSE(Yes/No, 0/1)
- Used For flags, binary conditions.

Special DATA TYPES

- XML → Store XML DAta
- Spatial (Geometry) → Stores maps, coordinates

SQL OPerators

SQL operators are symbols/keywords used in queries to calculate, compare, or filter data They are essential for data analysis, cleaning and preparation in ML/AL projects.

Types of SQL operators

Arthemabic Operators (Math operations)

used for calculations

Add, Sub, multiply, divide, modulus (reminder)

Comparison Operators (compare values)

Equal to (=), Greater than (>), less than (<), Greater or equal (>=), less or equal (<=), Not equal (<>)

Logical Operators (combine conditions)

- Add → both must be true
- OR → at least one true
- NOT → reverse condition

Bitwise Operators (work on Binary values)

And (&), OR (|), xOR(^), NOT(~), left Shift(<<)

right shift(>>)

Date: __/__/20__ Day: _____

Compound operators (Assign + calculate together)

Add and assign (+=)

Subsbract and assign (-=)

Multiply and assign (*=)

Divide and assign (/=)

Modulo and assign (%=)

Special Operators

- ALL - Compare with all values in a set
- ANY - true if matches at least 1 value
- BETWEEN - check value lies in range
- IN - check if value exisits in list
- EXISTS - check if subquery returns rows
- SOME - similar to any
- UNIQUE - return unique row

SQL Commands

SQL commands are inter instructions we give to the database to create, update, query or control data They are divided into 5 categories

DDL - Data Defination Language

Used to define or change database structure (tables, schemas)

Commands:

- CREATE → make new database objects (tables, views, indexes)
- DROP → delete objects
- ALTER → change structure (add/remove columns)
- TRUNCATE → remove all rows (faster than DELETE)
- COMMENT → add notes
- RENAME → rename objects

Date: _/_ /20 Day: _____

DQL → Data Query Language

Used to get data from database (mainly with SELECT)

Commands:

- SELECT - retrieve data
- FROM - pick table(s)
- WHERE - filter rows
- GROUP BY - group rows
- HAVING - filter grouped results
- DISTINCT - remove duplicates
- ORDER BY - sort results
- LIMIT - restrict number of rows.

DML - Data Manipulation Language

Used to insert, update, delete data

Commands:

- INSERT - add new data
- UPDATE - change data
- DELETE - remove rows
- LOCK - control concurrent access
- CALL - call procedures
- EXPLAIN PLAN - see how queries run.

DCL - Data Control Language

Commands:

- GRANT → give user permission
- REVOKE → remove permission

TCL - Transaction Control Language

Manages transaction (a group of queries that succeed/fail together)

Date: __/__/20__ Day: _____

Commands:

- BEGIN TRANSACTION → start transaction
- COMMIT → save changes
- ROLLBACK → undo changes
- SAVEPOINT → set checkpoint to rollback if needed

Most Important SQL commands

- SELECT → get data
- INSERT, UPDATE, DELETE → manage record
- CREATE, TABLE, ALTER TABLE, DROP TABLE, TRUNCATE TABLE
- WHERE, ORDER BY, GROUP BY, HAVING → filtering & aggregation
- JOIN → combine data from tables
- DISTINCT, IN, BETWEEN, LIKE → advanced filtering
- UNION → combine queries
- GRANT, REVOKE → permissions
- COMMIT, ROLLBACK, SAVEPOINT → transaction control.

SQL CREATE Database

CREATE DATABASE is the first step in SQL. It makes a new container to store tables, views and data

What is CREATE Database?

- Command to make a new database
- Database = container for tables, queries, stored procedures.
- Helps organize and manage data properly.

Syntax

```
CREATE DATABASE database_name;
```

Rules:

- Name must be unique
- No space → use underscore (_)
- Max-length ~ 128 characters

Date: __/__/20__ Day: _____

Example

CREATE DATABASE GeeksForGeeks;

create an empty database named: GeeksForGeeks

Verify Database

Check if it was created:

SHOW DATABASE;

Switch to DAtaBase

Work inside it

USE GeeksforGeeks

Delete Database

Remove permanently

DROP DATABASE GeeksForGeeks;

Delete all database — cannot be undone

Database Compability

- MySQL → Supports extra options (CHARACTER SET, COLLATE)
- PostgreSQL → allow owner and template options
- SQL Server → can set files sizes and storage location

Common Errors and Fixes

- Database already Exists — use

CREATE DATABASE IF NOT EXISTS db_name;

- No permission → need admin or special privileges.

Important Points

- Keep names unique, short, clear.
- Use lowercase or consistent format
- Avoid spaces use underscores
- Need correct permissions to create DB

Date: __/__/20__ Day: _____

SQL DROP Database

DROP DATABASE = deletes a database permanently (including all its tables, views, and data)

What is DROP Database?

- Removes database + all its objects (tables, indexes, views)
- Cannot be undone → always take backup first.
- Needs admin permissions
- Use when DB is no longer required or needs re-organization

Syntax:

```
DROP DATABASE database_name;
```

Example WorkFlow

- Create a DB (for testing)

```
CREATE DATABASE Yasir;
```

- Verify DB exists

```
SHOW DATABASES;
```

- Delete DB

```
DROP DATABASE Yasir;
```

- Verify delection

```
SHOW DATABASES;
```

Important Points

- Permanent action → deletes everything
- Backup before deleting
- Requires admin privileges
- Works even if DB is offline, ready-only or supsect

Date: __/__/20__ Day: _____

SQL RENAME Database

Used when we want to change the name of an existing database. Useful for reorganizing, fixing naming mistakes, or updating project structure.

How to Rename Database

- SQL Server

```
ALTER DATABASE old_name MODIFY NAME = new_name;
```

- PostgreSQL

```
ALTER DATABASE old_name RENAME TO new_name;
```

- MySQL (Rename not supported after v5.1.23)

Instead, we must:

- Create a new DB

```
CREATE DATABASE new_name;
```

- Move all tables

```
RENAME TABLE old_name.table1 TO new_name.table1;
RENAME TABLE old_name.table2 TO new_name.table1;
```

Example (SQL Server)

```
CREATE DATABASE Test;
ALTER DATABASE Test MODIFY NAME = Example;
```

Renames Test → Example

Important Points

- Availability: DB may be temporarily unavailable during rename
- Dependencies: Update apps, ML scripts, pipeline,
- Permissions: Needs admin privileges
- Backup: Always backup before renaming.

Date: __/__/20__ Day: _____

SQL SELECT Database

- Purpose: Choose a database to work with in your current SQL session
- Syntax

USE database_name;

- Note: Not supported in PostgreSQL → must connect directly to the DB

Example:

USE Yasir;

Now all queries will run inside Yasir

- SELECT Statement

Used to retrieve data from tables inside the selected DB.

Examples

- Select all data

SELECT * FROM employees;

Get all rows and columns

- Select specific columns

SELECT name, age FROM employees;

get only name and age.

- Filter rows (WHERE)

SELECT name, age FROM employees WHERE age >= 35;

gets employees aged 35+

- Sort results (ORDER By)

SELECT name, age FROM employees ORDER By age DESC;

sorts employees by age (descending)

Date: _/_ /20 Day: _____

• Limit rows (LIMIT)

SELECT name, salary FROM employees ORDER BY Salary DESC LIMIT 3;

Gets top highest-paid employees.

• Group and Aggregate (Group by)

SELECT department, AVG(salary) FROM employees GROUP BY department;

Find average salary per department

Key Notes

• Use Database: Sets the active database

• Select: Retrieves data with filters, sorting, limits & group

SQL CREATE TABLE

What is CREATE TABLE?

Used to make a new table in the database

Defines:

- Table name
- Columns + Data types
- Rules (constraints) → NOT NULL, PRIMARY KEY, Check etc.

Ensures data is organized, accurate, consistent.

Syntax:

```
CREATE TABLE table_name (
    column1 datatype (size) constraint,
    column2 datatype (size) constraint,
    ...
);
```

Date: __/__/20__ Day: _____

Example: Customer Table

```
CREATE TABLE Customer (
    CustomerIO INT PRIMARY KEY,
    CustomerName VARCHAR (50),
    LastName VARCHAR (50),
    Counby VARCHAR(50),
    Age INT CHECK (Age >= 0 AND Age <= 99),
    Phone VARCHAR(15)
);
```

Explanation

- CustomerID → Unique (PRIMARY KEY)
- Name, Counby → Text (VARCHAR)
- Age → Must be between 0-99 (CHECK)
- Phone → Better as VARCHAR (not INT) because of formatting

SQL DROP TABLE

What is DROP TABLE?

- Deletes a table permenently (data + structure + rules)
- Once dropped → cannot be recovered (unless backup exists)
- Removes:
  - Data
  - Structure (columns)
  - Constraints (primary keys, foreign keys, check)
  - Indexes, triggers, permissions.

Use with caution

Syntax

```
DROP TABLE table_name;
```

Works in MySQL, SQL server, Oracle (same syntax)

Date __/__/20__ Day: _____

Example

```
CREATE DATABASE NewCafe;
USE NewCafe;
CREATE TABLE categories(
    category ID INT PRIMARY KEY,
    CategoryName NVARCHAR(50) NOT NULL,
    ItemDescription NAVARCHAR(50) NOT NULL
);
```

ALTER (RENAME) in SQL

What is Alter?

ALTER TABLE = change the structure of an existing table (without losing data)

Common Uses:

- Rename a table
- Rename a column
- Add a new column
- Modify a column data type
- Remove a column
- Change default values

Syntax and Example

- Rename Table

```
ALTER TABLE old_table RENAME To new_table;
```

- Rename Column

```
ALTER TABLE table_name RENAME COLUMN oldcol To newcol;
```

- Add new Column

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

Date: __/__/20__ Day: _____

- Modify Data type

ALTER TABLE table_name MODIFY COLUMN column_name new_datatype;

- Drop (Remove) Column

ALTER TABLE table_name DROP COLUMN column_name;

- Set Default Value

ALTER TABLE tablename AITER COLUMN column_name SET DEFAULT value;

SQL TRUNCATE TABLE

TRUNCATE table = removes all rows from a table, but keeps the table structure Much faster than delete for large data. Cannot be used on a column or entire database (only works on table)

Syntax

TRUNCATE TABLE table_name;

Example

```
CREATE TABLE EMPLOYEE(
    EMP_ID INT,
    NAME VARCHAR(20),
    AGE INT,
    SALARY DECIMAL(7, 2)
);
INSERT INTO EMPLOYEE VALUES (121, 'Yosir', 24, 0.07);
TRUNCATE TABLE EMPLOYEE;
SELECT * FROM EMPLOYEE;
```

Data is gone, but the table + structure remain.

Date: __/__/20__ Day: _____

TRUNCATE vs DELETE

Feature	TRUNCATE TABLE	DELETE
Action	Removes all rows	Remove row
Speed	Faster	Slower for large data
Rollback	Usually cannot rollback	Can rollback in transactions
WHERE Clause	Not allowed	Allowed
Triggers	Not fired	Fired
Identity reset	Reset auto-increment	Does not reset

TRUNCATE Vs DROP

Feature	Truncate	DROP
Action	Remove all data, keep structure	Remove data + structure
Speed	Very Fast	Very Fast
Table Structure	Retained	Deleted
Usage	Empty table for reuse	Remove table permantly

Important Point

- TRUNCATE = fast way to clear a table
- Keeps columns, schema, constraints, indexes
- Cannot truncate if foriegn key constraints exist
- Cannot be rolled back in many DBMS
- Requires ALTER permission

SQL Copy Table

What is Cloning a Table?

- Cloning = making a copy of a table
- like taking a photocopy of a table
- Can copy only structure or structure + data
- Useful for
  - Testing new Features
  - Creating backups
  - Experimentaing without changing orginal data

Date: __/__/20 Day: _____

Real-life Example

library system → create a clone of student table to test changes safely, without touching orginal data.

Methods of Cloning

- Simple Cloning
  - Copies structure + data
  - Does Not copy constraints (Primary key, Unique)

Syntax:

CREATE TABLE clone_table AS SELECT * FROM orginal_table;

Problem: Many cause duplicate values since constraints are lost.

Shallow Cloning

- Copies only structure (no data)
- Keeps constraints (Primary key, Unique)
- Useful when you want an empty table but with same rules.

Syntax:

CREATE TABLE clone_table LIKE original_table;

Deep Cloning

- Best method → Copies structure + constraints + data
- Fully independent duplicate of original table

Syntax:

CREATE TABLE clone_table LIKE original_table;

INSERT INTO clone_table SELECT * FROM orginal_table;

Date: _/_ /20 Day: _____

Comparison of Methods

Method	Structure	Data	Constraints
Simple	✓	✓	lost
Shallow	✓	x	Preserved
Deep	✓	✓	Preserved

SQL TEMP TABLE

What is Temporary Table?

- A temporary table = a table created for short-term use
- Stored in a special database (e.g TempDB, in SQL)
- Automatically deleted when session or procedure ends
- Used for storing intermediate results without affecting permanent tables.

Syntax

```
CREATE TABLE #EmpDetails (id INT, name VARCHAR(25));
INSERT INFO #EmpDetails VALUES (1, 'lalit'), (2, 'Athena')
SELECT * FROM #EmpDetails;
```

Types of Temporary Tables

Local Temporary Table

- Prefix: # (single hash)
- Only available in session that create it
- Delete when session ends
- Example

```
CREATE TABLE #EmpDetails ( id INT, name VARCHAR(25));
```

Global Temporary Table

- Prefix: ## (double hash)
- Visible to all sessions
- Deleted when the last session using it ends.

Date: __/__/20 Day: _____

Why use Temporary Tables?

- Store intermediate results during queries
- Perform calculations or transformations safely
- Avoid changing permanent tables
- Helpful in complex queries, stored procedures, and testing.

SQL ALTER TABLE

ALTER TABLE used to change an existing table without losing data

we can:

- Add columns
- Remove (drop) columns
- Modify coloumn datatype / size
- Rename coloums
- Rename whole table
- And many more things.

Syntax

```
AITER TABLE table_name:
[ADD | DROP | MODIFY] column_name datatype;
```

Example

```
ALTER TABLE Students ADD Gmail VARCHAR(255));
ALTER TABLE Students MODIFY COURSE VARCHAR(20);
```

Why AILTER TABLE is Important?

- Keeps tables flexible as requirements change
- Lets you update schema without lossing data
- Useful for cleaning, updating, restructuring datasets.

Date: __/__/20__ Day: _____

SQL Query

Select Statement (SELECT)

It Retrieves data from a table

It is useful for core command to explore or extract data.

Syntax

```
SELECT column1, column2
FROM table_name
WHERE condition
ORDER BY column
LIMIT n;
```

Example

```
SELECT name, age
FROM employees
WHERE age > 30
ORDER BY age DESC
LIMIT 5;
```

ML/DS: Ideal for building model datasets, filtering for specific observations and preparing data for training or analysis.

INSERT INTO (single Row and Multiple Row)

Adds new records to tables

loads data into your tables - For example, appending training data

Syntax: Single

```
INSERT INTO table_name (col1, col2) VALUES (val1, val2);
```

Date: __/__/20 Day: _____

For multiple

INSERT INTO table_name (col1, col2) VALUES (v1a, v1b), (v2a, v2b)

Example

INSERT INTO employees (name, age) VALUES ('Yasir', 24), ('Bob', 21);

Summary: Useful for augmenting datasets, including adding new labeled examples or simulation data.

UPDATE Statement

Changes values of existing records

Allows correction or transformation without re-importing

Syntax

UPDATE table_name

SET column = expression

WHERE condition;

Example:

UPDATE employees

SET Salary = Salary * 1.1

WHERE performance = 'Excellent';

Summary: Often used to scale features, filling missing values, or create new engineered features like ratios.

DELETE Statement

Removes records from a table

Cleans data by removing outdated or irrelevant

Date: __/__/20__ Day: _____

rows

Syntax

DELETE FROM table_name WHERE condition;

Example:

DELETE FROM employees WHERE resigned = True;

Summary: Helps maintain data integrity by removing noise, anomalies, or outdated samples before modeling

DELETE Duplicates ROWs

Cleaning duplicates is essential for data quality

A - Using GROUP BY + MIN (keep one copy)

```
DELETE FROM TABLE_name
WHERE id NOT IN (
    SELECT MIN(id)
    FROM table_name
    GROUP BY col1, col2
);
```

-B Using ROW_NUMBER() and CTE (advanced, more control)

```
WITH CTE AS (
  SELECT *, ROW_NUMBER() OVER (
    PARTITION BY col1, col2 ORDER BY id
  ) AS rn
  FROM table_name
)
DELETE FROM CTE WHERE rn > 1;
```

Summary: Eliminates duplicates entries that can bias models or confused analysis.

Date: __/__/20 Day: _____

SQL Clauses

- WHERE Clause.
- Used to filter rows based on a condition

Helps get only thedt data you need instead of the whole table

Syntax:

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

Example

```
SELECT name, age
FROM students
WHERE age > 18;
```

(shows students older than 18)

WITH Clause (common table Expression. CTE)

Creates a temporary result (like a virtual table) that can be used in a query

Make complex queries easier to read and reuse.

Syntax

```
WITH cte_name AS(
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT * FROM cte_name;
```

Example:

```
WITH high_salary AS(
```

Date: __/__/20__ Day: _____

```
SELECT name, salary
FROM employees
WHERE salary > 50000
);
SELECT * FROM high_salary;
```

(Finds employees earning more than 50,000)

HAVING Clause:

Filters groups created by GROUP BY

Works like WHERE but for grouped data.

Syntax:

```
SELECT column, COUNT(*)
FROM table_name
GROUP BY column
HAVING condition;
```

Example:

```
SELECT department, COUNT(*)
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

(shows only departments with more than 5 employees)

ORDER BY Clause

Sorts the result in ascending (ASC) or descending (DESC) order.

Makes results more organized and readable.

Syntax

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

Date: __/__/20__ Day: _____

Example:

```
SELECT name, age
FROM students
ORDER BY age DESC;
```

(Lists students from oldest to youngest)

GROUP BY Clause:

Groups rows that have the same values

Used with aggregate functions like COUNT, SUM, AVG

Syntax

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

Example:

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department;
```

(shows average salary in each department)

LIMIT Clause

Restricts the number of rows returned

Helpful for testing or when you only need a few results

Syntax

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

Example:

```
SELECT *
FROM students
LIMIT 5;
```

} (shows only the first 5 students)

Date: __/__/20__ Day: _____

DISTINCT Clause

Removes duplicate values from results

Ensures uniqueness in query output

Syntax

SELECT DISTINCT column1

FROM table-name;

Example:

SELECT DISTINCT department

FROM employees;

(lists each department only once)

FETCH Clause (works with OFFSET)

Gets a fixed number of rows after skipping some

Often used for pagination (like showing 10 results per page)

Syntax

SELECT column1, column2

FROM table_name

OFFSET x ROWS

FETCH NEXT y ROWS ONLY;

Example:

SELECT *

FROM students

OFFSET 5 ROWS

FETCH NEXT 3 ROWS ONLY;

(skips first 5 students, then shows next 3)

Date: __/__/20__ Day: _____

Aliases (AS)

Gives a temporary name to a column or table

Makes queries shorter and easier to read

Syntax

```
SELECT column AS alias_name
FROM table_name AS t;
```

Example:

```
SELECT name AS student_name, age
FROM students AS s;
```

SQL Operators

AND / OR / NOT (logical operators)

Combine multiple conditions is a query

Helps narrow down or broaden filters

Syntax

```
SELECT * FROM table
WHERE condition1 AND condition2;
WHERE condition1 OR condition2;
WHERE NOT condition;
```

Example:

```
SELECT * FROM products
WHERE category = 'Books' AND price < 20;
SELECT * FROM users
WHERE country = 'USA' OR country = 'UK';
SELECT * FROM customers
WHERE NOT vip = TRUE;
```

Date: / /20 Day:

LIKE

Matches a text pattern in a column

Great for partial matches ("Starts with", "contains")

Syntax

```
SELECT * FROM table
WHERE column LIKE 'Pattern';
```

Example:

```
SELECT * FROM employees
WHERE name LIKE 'A%';
```

IN / NOT IN

Checks if a value exists (or not) in a list

Simplifiers multiple OR condition

Syntax

```
WHERE column IN (value1, value2, ...)
WHERE column NOT IN (value1, value2, ...)
```

Example:

```
SELECT * FROM students
WHERE grade IN ('A', 'B', 'C');
```

IS NULL

Checks if a value is missing (NULL)

NULL isn't equal to anything, so use this

Syntax

```
WHERE column IS Null
WHERE column IS NOT NULL
```

BETWEEN

Checks if a value falls within a range (inclusive)

Date: __/__/20 Day: _____

Neater than using multiple comparsions

Syntax

WHERE column BETWEEN min AND max

Example:

SELECT * FROM orders

WHERE date BETWEEN '2025-08-24' AND '2025-06-30';

ALL / ANY

Compare a value to a set of values from a subquery

Helps with complex comparisons in filters.

Syntax

WHERE value > ALL (SELECT ...)

WHERE value = ANY (SELECT ...)

UNION / UNION ALL

Combine rows from multiply SELECT queries

- UNION : removes duplicates
- UNION ALL : keeps all rows

Merge datasets from different tables

Syntax

SELECT ... FROM table1

UNION

SELECT ... FROM table2;

Example;

SELECT name FROM employees

UNION

SELECT nome FROM customers;

Date: __/__/20__ Day: _____

INTERSECT

Returns rows common to both SELECT queries

Finds overlapping records b/w datasets

Syntax

```
SELECT ... FROM table1
INTERSECT
SELECT ... FROM table2;
```

EXCEPT (or MINUS)

Returns rows from the first query that don't appear in the second

Ideal for finding differences b/w sets

Syntax

```
SELECT ... FROM table1
EXCEPT
SELECT ... FROM table2;
```

EXISTS

Checks if a subquery returns any row

Efficient way to test presence, not actual data

Syntax

```
WHERE EXISTS (SELECT ... FROM other_table WHERE condition)
```

CASE (conditional Expression)

Similar to IF-ELSE, returns values based on condition

Handle complex conditional logic inline

Syntax

```
SELECT CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ELSE default
END AS alias
FROM table;
```

Date: __/__/20 Day: _____

Example:

```
SELECT name,
    CASE
    WHEN age < 18 THEN 'Minor'
    ELSE 'Adult'
  END AS age_group
FROM users;
```

SQL Aggregate Functions

Aggregate Functions

Perform calculations on groups of rows to return a single summary value (e.g total, average).

Simplifies summarizing large datasets for insights.

Often used with GROUP BY

COUNT()

Returns the number of rows or non-null values

Great for counting records or checking data presence.

Syntax

```
COUNT(*)
COUNT( column_name)
COUNT(DISTINCT col)
```

Example:

```
SELECT COUNT(*)
SELECT COUNT(salary) AS IknowSalaries FRO
SELECT COUNT(DISTINCT department) AS NumDepartment
  FROM EMPOYEE;
```

Date: __/__/20__ Day: _____

SUM()

Adds up all non-null numeric values in a column

Calulates totals like sales or salaries

Syntax

SUM (column_name)

SUM (DISTINCT column_name)

Example

SELECT SUM(salary) AS TotalSalary FROM ~~EMPLOYEE~~ Employee;

SELECT SUM(DISTINCT salary) AS UniqueSalarySum FROM Employee;

AVG()

Computes the average of non-null numeric values

Quickly finds averages like average price / rating.

Syntax

AVG (column_name)

AVG (DISTNICT column_name)

Example:

SELECT AVG(salary) AS AverageSalary FROM Employee;

MIN() and MAX()

MIN() return smallest values, MAX() returns the largest

Useful for finding ranges, top values, or boundaries

Syntax

MIN (column_name)

MAX (column_name

Example

SELECT MIN(salary) AS LowestSalary, MAX(salary) AS HighestSalary
FROM EMPLOYE employee;

Date: __/__/20 Day: _____

Using GROUP BY with Aggregates

Group BY groups rows by sep specified columns to run aggregate functions on each group

Enables pre-group analyses (e.g total sales per region)

Syntax

```
SELECT group_column, AGG_FUNC(target_column)
FROM table
GROUP BY group_column;
```

Example

```
SELECT department, AVG(salary), AS Avgsalary
FROM Employee
GROUP BY department;
```

Aggregate Functions + HAVING

HAVING filters groups based on aggregate condition (like a WHERE for groups)

Useful: Narrow down results after aggregation

Syntax

```
SELECT group_column, AGG_FUNC(target_column)
FROM table
GROUP BY group_column
HAVING AGG_FUNC(target_column) condition;
```

Example

```
SELECT department, SUM(salary) AS TotalSalary
FROM Employee
GROUP BY department
HAVING SUM(salary) > 50000;
```

Date: __/__/20__ Day: _____

SQL Data Constraints

NOT NULL Constraint

Ensures a column cannot have a NULL (no value)

Guarantees essential data is always provided.

Syntax

```
column_name datatype NOT NULL
```

Example

```
CREATE TABLE Users (
    user_id INT NOT NULL,
    username VARCHAR(50) NOT NULL
);
```

PRIMARY KEY

Uniquely identifies each row; combines constraints

NOT NULL + UNIQUE

Essential for identifying records and referenced by foreign keys

Syntax

Within CREATE TABLE

```
PRIMARY KEY (column1, column2)
```

Or inline

```
column_name datatype PRIMARY KEY
```

Example:

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

Date: __/__/20 Day: _____

FOREIGN KEY

Links a column to the primary (or unique) key of another table

Maintains referential integrity - only valid references allowed

Syntax

```
FOREIGN KEY (column_name)
    REFERENCES otherTable (other_column)
```

Example

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    student_id INT,
    FORGIGN KEY (student_id) REFERENCES Students(studentid)
);
```

UNIQUE Constraint

Ensures values in a column or group of columns are different

Pervents duplicate data when PK isn't appropiate

Syntax

```
column_name datatype UNIQUE
```

Example

```
CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE
);
```

Date: __/__/20__ Day: _____

ALTERNATE KEY

A candidate not selected as the primary key, often implemented using UNIQUE

Offers another way to uniquely identify records.

Syntax

CONSTRAINT alt_key_name UNIQUE (column1, column2)

Example

```
CREATE TABLE Employees(
    emp_id INT PRIMARY KEY,
    ssn VARCHAR(11) UNIQUE,
    email VARCHAR(255) UNIQUE
);
```

COMPOSITE KEY

A key made up of two or more columns together uniquely identifying a row

Used when a single column isn't enough for uniqueness

Syntax

```
PRIMARY KEY (col1, col2)
```

Example

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

CHECK Constraints

Ensures values meet specific conditions

Validates data at the database level to avoid

Date: __/__/20 Day: _____

invalid entries

Syntax

Column_name datatype CHECK (condition)

or

CONSTRAINT check_name CHECK (condition)

Example

```
CREATE TABLE Products (
    price DECIMAL(10, 2),
    CHECK (price >= 0)
);
```

DEFAULT Constraint

Specifies a default value when none is provided.

Ensures consistent default values, easing data entry

Syntax

column_name datatype DEFAULT default_value

Example

```
CREATE TABLE Products (
    price DECIMAL(10, 2) DEFAULT 0.00
);
```

SQL Joins

INNER JOIN (JOIN):

Returns only the rows with matching values in both tables.

Retrieves combined data where relationships exists

Syntax

```
SELECT t1.*, t2.*
FROM table1 As t1
```

Date: __/__ /20 Day: _____

INNER JOIN table2 AS t2
ON t1.key = t2.key;

Example:

SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers
ON Orders.CustomerID = Customers.CustomerID;

LEFT (OUTER) JOIN

Returns all rows from the left table, and matching rows from the right; NULL when no match

Keeps unmatched left-side data intact

Syntax

SELECT t1.*, t2.*
FROM table1 AS t1
LEFT JOIN table2 AS t2
ON t1.key = t2.key;

Example

All customers, with their orders if available

RIGHT (OUTER) JOIN

Returns all rows from the right table, and matching rows From the left; NULL when no match

Shows all data from right table even if no left-side match exists.

Syntax

SELECT t1.*, t2.*
FROM table1 As t1
RIGHT JOIN table2 AS t2
ON t1.key = t2.key;

Date: __/__/20__ Day: _____

FULL (OUTER) JOIN

Returns all rows when there is a match in either left or right table

Comprehesive view combining both tables fully

Syntax

```
SELECT t1.*, t2.*
FROM table1 AS t1
FULL JOIN table2 AS t2
    ON t1.key = t2.key;
```

Example:

Shows all students and course matches (or NULL)

CROSS JOIN

Produces the Cartesian product of both tables (every combination)

Useful for generating all possible pairings or test data.

Syntax

```
SELECT *
FROM TableA
CROSS JOIN TableB;
```

Example: Pair every meal with every drink in a menu.

SEE JOIN

Joins a table to itself, using aliases to compare rows within the same table

Useful for hierarchical comparisons or duplicate lookups

Date: _/_ /20 Day: _____

Syntax

```
SELECT A.*, B.*
FROM table AS A
JOIN table AS B
    ON A.key = B.key;
```

Example:

Find employees inthe same country

UPDATE with JOIN

Updates a table's values using data from another table

Convenient cross-table updates.

Note: Syntax differ accross SQL dialects, often uses a FROM clause

DELETE With JOIN

Deletes rows from one table based on matching conditions From another table

Efficient way to delete rows related to data from another table

Syntax

```
DELETE A
FROM TableA AS A
LEFT JOIN TableB AS B
    ON A.id = B.id
WHERE B.id IS NULL;
```

Recursive Join (Recursive CTE)

Uses a common Table Expression (WITH RECURSIVE)

Date: __/__/20__ Day: _____

to repeatedly join a table to itself. Great for exploring hierarchical data (e.g organization charts or ancestry trees)

Why useful: Simplifies queries on nested or hierarchical data.

SQL Functions

Date Functions

Built in tools to handle dates and times - likely finding the current date, extracting parts, or calculating differences

Essential for time-based analysis, reporting, scheduling

NOW()/ GETDATE()

Returns current data / time

```
SELECT NOW() As current_datetime;
```

EXTRACT(), DAY(), MONTH(), YEAR()

Pull parts from data.

```
SELECT MONTH ("2025-08-28") As month;
```

DATA(), ADD(), DATA_SUB(), DATADIFF()

Add/substract time, get difference

```
SELECT DATE_ADD(CURDATE(), INTERVAL 5 DAY) As
    five_days_later;
```

String Functions

Help modify and analyze text - like changing, case trimming, spaces, or extracting parts.

Vital for cleaning and processing textual data

LTRIM() / RTRIM()

Remove leading or trailing spaces

Date: _/_ /20 Day: _____

SELECT LTRIM (' Hello') As trimmed;

SELECT RTRIM ('Hello ') As trimed;

UPPER/ LOWER()

Change text case consistenthy

SELECT UPPER ('hello') As uppercase;

SUBSTRING()

Get part of a string

SELECT SUBSTRING('Hello world', 1, 5) As part;

Numeric Functions

Perform calculations on numbers (not detailed in sources)

Useful for rounding, absolute values, etc.

Statistical Functions

Calculations like average, sum or count

Fundamental for data summaries and analytics.

JSON Fonctions

Work with JSON data stored in databases

Important for semi-structured data in modern applications

Conversion Function / Data type Functions

Change one data to another (e.g; String → DATE)

Essential when working with mismatched or formatted data, but e

Date: __/__/20__ Day: _____

SQL Views

What is a SQL View?

A view is like a virtual table created from a saved SQL query. It doesn't store data itself when you query a view, it runs the underlying query dynamically

Why Useful:

- Simplifies complex joins or filters
- Enhances data security by hidding sensitive columns.
- Allows different presentations of the same data for different users.

CREATE VIEW

Creates a virtual table from a query expression

Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE Country = 'Pakistan'
```

Why Useful: Make complex queries easy to use and hides unnecessary details.

UPDATE VIEW

Modifies data in the view if the view is ~~updated~~ updatable (database-dependent)

Not all views allow updates. Show some systems support this, others treats view as read-only.

RENAME VIEW

Changes the name of an existing view

Date: __/__/20__ Day: _____

Syntax Examples:

- SQL sewer (using sp rename): EXEC sp name 'old_view', 'new_view';
- MySQL (via RENAME TABLE): RENAME TABLE old view;
- Postre.SQL (via ALTER. VIEW): ALTER VIEW old view RENAME To new_view;

Why Useful:

Keeps code up-to-date if view names need to change without losing structure or dependencies

DROP VIEW

Deletes an existing view from the databse.

Syntax

DROP VIEW view name;

Why Useful: Removal / Removes obsolete or unused views from the system

SQL Indexes

What is an Index?

Defination:

An Index is a special data structure that speeds up data lookup - like an index in a book, pointing to locations in a table

Providing - Provides fast query results by avoiding full table scans.

CREATE INDEX

Creates an index on one or more table columns to improve search speed.

Date: __/__/20 Day: _____

Syntax:

CREATE INDEX index_name

ON table_name (column1, column2, ...);

• Unique Index Syntax

CREATE UNIQUE INDEX index_name

ON table_name (column_name);

Example

CREATE INDEX idx_lastname

ON Persons (LastName);

Useful for speeding up queries on LastName

DROP INDEX

Deletes an index that's no longer needed or is slowing down updates

Syntax Varies by DBMS

DROP INDEX table_name.index_name;

ALTER TABLE table_name DROP INDEX index_name;

: contentReference [oaicite:2]{index=2}

SHOW INDEXES

Lists all indexes on a table (varies by database)

Example in MySQL

SHOW INDEX FROM table_name;

UNIQUE INDEX

Prevents duplicate values in an index columns, ensuring data integrity.

Often created automatically when defining UNIQUE constraints.

Date: _/_ /20 Day: _____

Clustered Vs Non-Clustered Indexes

- Clustered Index:

Sorts and stores table rows in index order; one per table, Ideal for range queries.

- Non-Clustered Index:

Separate structure pointing to table rows; allows multiple per table. Great for lookups.

SQL Subquery

What is a Subquery?

A subquery is a query placed inside another SQL query (in the SELECT, FROM, WHERE or other clauses)
It lets you perform multi-step logic like filtering based on results of another query, without needing separate queries.

Nested Subquery

A subquery fully separate from the outer query.
The inner query runs first, and its result is used by the outer query

Syntax Example

```
SELECT name
FROM Customers
WHERE CustomerID IN (
    SELECT CustomerID
    FROM Orders
);
```

Why Useful: Make queries modular and easier to read. especially handy when joins would be overakill

Date: __/__/20 Day: _____

Correlated Subquery

A subquery that refers to values from the outer query. It runs once per row of the outer query.

Syntax Example

```
SELECT e.last_name, e.salary
FROM employees e
WHERE e.salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department_id = e.department_id
);
```

Why Useful: Useful for dynamic comparisons like checking if an employee earns more than their department average.

Types of Subqueries by Output

- Scalar:

Returns a single (1 row, 1 column)

Use Case Example: (SELECT AVG(salary) FROM employees)

- Column:

Returns one column, multiple rows

IN (SELECT department_id FROM departments)

- Derived Table:

Subquery used in FROM, treated as a temp table

FROM (SELECT ...) AS alias

Advanced SQL

Wildcard Operators (LIKE)

Definition: Use % or _ to filter parts of text

Great for searching patterns like names or codes

Date __/__/20__ Day: _____

Syntax

SELECT * FROM table WHERE column LIKE "Pattern";

Example:

SELECT * FROM users WHERE name LIKE 'A%';

Coments

Add human-friendly notes in SQL scripts

Helps document logic and remind yourself or teammates

Syntax

-- Single line

/* Multi-line comment */

PIVOT / UNPIVOT

Transform rows to columns (PIVOT) or columns to rows (UNPIVOT)

Helps reshape data for reporting or databoards

Syntax (SQL server)

SELECT * FROM table

PIVOT (SUM(Value) FOR column IN(List)) AS alias;

Example:

--Diagram demo not shown

Triggers

Automatically run code when certain events happen (like INSERT, UPDATE, DELETE)

Enforce business rules or audit changes

Syntax

CREATE TRIGGER trigger_name

AFTER INSERT ON table

BEGIN

-- statement

END;

Date: __/__/20__ Day: _____

Stored Procedures

Stored blocks of code that can take inputs and run queries

Simplifiers reusable logic or operations

Syntax

```
CREATE PROCEDURE proc_name (params)
AS
BEGIN
  -- code
END;
```

Transactions

Group queries into a single set to ensure data consistency

Ensures all steps succeed - or none are applied.

ACID properties

Syntax

```
BEGIN TRANSACTION;
-- operations
COMMIT;
-- or ROLLBACK;
```

Sequences (AUTO - Increment)

Automatically generate uniques IDs

Ideal for primary keys

Syntax (MySQL)

```
CREATE TABLE table(
  id INT AUTO_INCREMENT PRIMARY KEY
);
```

Date: _/_ /20 Day: _____

Window Functions

Perform calculations accross a specified set of rows related to the current row

Allows ranking, running tasks/totals, moving averages, etc.

Syntax:

```
SELECT col,
    RANK() OVER (PARTITION By category ORDER BY score
                 DESC) AS rank.
FROM table;
```

Cursors

Retrieve and process rows one at a time

Needed for row-by-row operations when set-based logic isn't enough.

Syntax (SQL)

```
DECLARE cursor_name CURSOR FOR SELECT ...
OPEN cursor_name;
FETCH cursor name INTO vars;
CLOSE cursor_name;
```

Common Table Expressions (CTE)

Define a tempory result set for better query structuring.

Makes recursive queries and complex logic cleaver

```
WITH cte AS(
    SELECT ...
)
SELECT * FROM cte;
```

Performance / Database Tuning

Optimize queries and database structure for better

Date: __/__/20__ Day: _____

Speed.

Critical for large datasets or slow queries.

Example:

- Index tuning
- Analyzing execution plans
- Partitioning large tables

Dynamic SQL

SQL code that's constructed and executed at runtime.

Handles unknown table/field names or query conditions dynamically.

Syntax (SQL Server)

```
DECLARE @sql NVARCHAR(MAX);
SET @sql = 'SELECT ...';
EXEC sp_executesql @sql;
```

Regular Expressions (Regex)

Pattern matching within strings

Allows flexible filtering, formatting, and validation

Syntax

```
SELECT * FROM table WHERE column ~ 'pattern';
```

Date: _/__/20__ Day: _____

MySQL

Introduction to MySQL

What is SQL?

- MySQL is a Relational Database Management System (RDBMS)
- It is open-source and free to use
- Works for small and large applications
- Fast, reliable, scalable, and easy to use
- Cross-platform (works on Window, Linux, macOS)
- Follows ANSI SQL standard
- First released in 1995, developed and supported by Oracle
- Named after the daughter of co-founder Monty Widenius ("My")

Who Use MySQL?

- Big Companies: FaceBook, Twitter, YouTube, Uber, Github, Airbnd, Booking.com
- Popular CMS: Wordpress, Joomla, Drupal
- Millions of developers worldwide

Showing Data on a Website

To display data from a database on a website, you need:

- An RDBMS (like MySQL)
- A server-side language (like PHP, Python)
- SQL queries to fetch data
- Html/CSS to display and style the page

Date: __/__/20 Day: _____

MySQL RDBMS

What is RDBMS?

- RDBMS : Relational Database Management System
- It stores data in tables (row & column)
- All modern DBs like MySQL, Oracle, SQL server, MS Access are RDBMS
- Uses SQL to access are manage data.

What is a table?

- A table = collection of data (rows and columns)
- Column = type of information (like Name, Address)
- Row = (record) → only one entry of data.

Example (customers Table):

CustomerID	CustomerName	City	Country
1	Alfreds F.	Berlin	Germany
2	Ana Trujillo	Mexico	Mexico
3	Antonio M.	Mexico	Mexico
4	Around Horn	London	UK
5	Berglunds	Lulea	Sweden

What is a Relational Database?

- A relational database connects tables using common columns

Example Relationships

- Customers → Orders (Linked by CustomerID)
- Orders → Shippers (Linked by ShipperID)

Orders Table (sample)

OrderID	CustomerID	ShipperID	OrderDate
10278	5	2	1996-08-12
10308	2	3	1996-09-18
10383	4	3	1996-12-16

Date: __/__/20__ Day: _____

Shippers Table (sample)

ShipperID	Shipper Name	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

Summary

- MySQL is a free database system
- Stores data in tables (rows + columns)
- Tables can be linked (related) with Keys
- Used by websites, apps, and big companies worldwide

MySQL and SQL Basics

What is SQL?

- SQL = Structural Query language
- Used with relational databases (like MySQL)
- Main tasks: Insert, Search, Update, Delete Data.
- Keywords are not case sensitive (SELECT = select)
- Use semicolon (;) at end of statements.

Common SQL Commands

- SELECT → get data from a table
- INSERT INTO → add new data
- UPDATE → change data
- DELETE → remove data
- CREATE DATABASE → make new database
- CREATE TABLE → make new table
- ALTER TABLE → change table
- DROP TABLE → delete table
- CREATE INDEX → make search Key
- DROP INDEX → remove search Key

Date: __/__/20 Day: _____

MySQL SELECT

The SELECT statement is used to select data from a database. The data returned is stored in a result table, called the result-set.

Syntax:

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

→ SELECT * selects all columns

Example:

```
SELECT CustomerName, City, Country
FROM Customers;
```

```
SELECT * FROM Customers;
```

SELECT DISTINCT

Used to get unique values (no duplicates)

Syntax:

```
SELECT DISTINCT column1
FROM table_name;
```

Example:

```
SELECT DISTINCT Country FROM Customers;
```

→count unique value:

```
SELECT COUNT (DISTINCT Country) FROM Customers;
```

MySQL Where

The WHERE clause is used to filter records.

It is used to extract only those records that fulfill a specific condition.

Date: __/__/20__ Day: _____

Syntax

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

Examples

```
-- Customers From Mexico
SELECT * FROM Customers
WHERE Country = 'Pakistan/Mexico';
-- Find customer by ID
SELECT * FROM Customers
WHERE CustomerID = 1;
```

Operators in WHERE

- = → equal
- > → greater than
- < → less than
- >= → greater or equal
- <= → less or equal
- <> or != → not equal
- BETWEEN → between range
- LIKE → pattern search
- IN → match multiple values

Summary

- SQL is the language for databases
- SELECT → get data
- DISTINCT → remove duplicates
- WHERE → filter results

Date: __/__/20__ Day: _____

MySQL AND, OR, NOT Operators

We can use AND, OR, NOT in the WHERE clause to filter data

- AND → All conditions must be true
- OR → Any condition can be true
- NOT → Opposite (exclude values)

AND Syntax

```
SELECT column1, column2
FROM tablename
WHERE condition1, AND condition2;
```

Example: Customers in Germany and Berlin

```
SELECT * FROM Customers
WHERE Country = 'Germany' AND City = 'Berlin';
```

OR Syntax

```
SELECT column1, column2
FROM table_name
WHERE condition1 OR condition2;
```

Examples:

- City is Berlin or Stuttgart

```
SELECT * FROM Customers
WHERE City = 'Berlin' OR City = 'Stuttgart';
```

- Country is Germany or Spain

```
SELECT * FROM Customers
WHERE Country = 'Germany' OR Country = 'Spain';
```

NOT Syntax

```
SELECT column1, column2
FROM table_name
WHERE NOT condition;
```

Date: __/__/20 Day: _____

Example: Customers not in Germany

```
SELECT * FROM Customers
WHERE NOT Country = 'Germany';
```

Combine AND, OR, NOT

We can mix them together (use brackets for clarity)

Example 1: Germany + (Berlin or Stuttgart)

```
SELECT * FROM Customers
WHERE Country = 'Germany' AND (city = 'Berlin' OR city = 'Stuttgart');
```

Example 2: Exclude Germany and USA

```
SELECT * FROM Customers
WHERE NOT Country = 'Germany' AND NOT country = 'USA';
```

MySQL ORDER BY, INSERT INFO, UPDATE, NULL

LIMIT (SQL Basic)

ORDER BY

Used to sort results in ascending (ASC) or decending (DESC) order

- Default is ASC

Syntax

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

Examples:

```
--sort by country (A-Z)
SELECT * FROM customers ORDER BY Country;


--sort by country (Z-A)
SELECT * FROM Customers ORDER BY Country DESC;
```

Date: __/__/20__ Day: _____

```
-- Sort by country, then CustomerName
SELECT * FROM Customers ORDER BY Country, CustomerName;

-- SELECT sort by Country ASC and CustomerName DESC
SELECT * FROM Customers ORDER BY Country ASC, CustomerName
DESC;
```

INSERT INTO

Used to add new records into a table

Syntax (two ways)

--1 With column names

```
INSERT INTO table_name (col1, col2, col3)
VALUES (val1, val2, val3);
```

--2 Without column names (all columns)

```
INSERT INTO table_name
VALUES (val1, val2, val3, ...);
```

Examples:

```
-- Insert full data
INSERT INTO Customers (CustomerName, ContactName, Address)
VALUES ('cardinal', 'Yasir', 'Lahore');
-- Insert only specific columns
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
```

Note: CustomerID is auto-increment, so no need to insert it manually

NULL Values

A Null means no value. It is different from 0 or empty space

Date: __/__/20__ Day: _____

How to check Null?

- Use IS NULL or IS NOT NULL

Examples:

-- Customers with NULL Address

```
SELECT CustomerName, Address
FROM Customers
INHERE Address IS NULL;
```

-- Customers with NOT NULL Address

```
SELECT CustomerName, Address
FROM Customers
INHERE Address IS NOT NULL;
```

UPDATE

Used to modify existing records in a table

Syntax

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

∴ If you omit WHERE, all records will be updated

Examples:

-- Update one record

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City = 'Frank'
INHERE CustomerID = 1;
```

-- update multiple records

```
UPDATE Customers
SET PostalCode = 00000
WHERE Country = "Mexico";
```

Date: __/__/20 Day: _____

-- update all records (dangerous)

UPDATE Customers

SET PostalCode = 00000;

DELETE

Used to delete records from a table

Syntax

DELETE FROM table_name WHERE condition;

∴ If you omit WHERE, all records will be deleted

Example

--Delete one record

DELETE FROM Customers

WHERE CustomerName = 'Alfreds ";

-- Delete all records (table stays empty but structure remains)

DELETE FROM Customers;

LIMIT

Used to limit the number of rows returned.

Helpful for large tables

Syntax

SELECT column name(s)

FROM table name

WHERE condition:

LIMIT number OFFSet Start;

Example

-- First 3 records

SELECT * FROM Customers LIMIT 3;

Date: __/__/20__ Day: _____

```
-- Records 4 to 6
SELECT * FROM Customers LIMIT 3 OFFSET 3;


-- First 3 German customers
SELECT * FROM Customers
WHERE Country = 'Germany'
LIMIT 3;


-- First 3 sorted by country
SELECT * FROM Customers
ORDER BY country
LIMIT 3;
```

MySQL MIN, MAX, COUNT, AVG, SUM

MIN()

Finds the smallest value in a column

Syntax

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

Example:

```
-- cheapest product price
SELECT MIN(Price) AS smallestPrice
FROM Products;
```

MAX()

Find the largest value in a column

Syntax

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

Date: __/__/20__ Day: _____

Example;

```
-- Most expensive product price
SELECT MAX(Price) AS LargestPrice
FROM Products;
```

COUNT()

Counts the number of rows (Ignores NULL values)

Syntax

```
SELECT COUNT (column_name)
FROM table name
WHERE condition;
```

Example

```
-- Total number of products
SELECT COUNT (ProductID)
FROM Products;
```

AVG()

Finds the average value of a numeric column (Ignores NULL)

Syntax

```
SELECT AVG (column_name)
FROM table_name
WHERE condition;
```

Example

```
-- Average product price
SELECT AVG (Price)
FROM Products;
```

Date: __/__/20__ Day: _____

SUM()

Adds up (total) all values in a numeric column

Syntax

```
SELECT SUM( column_name)
FROM table_name
WHERE condition;
```

Example

```
-- Total quantity of all order details
SELECT SUM( Quantity)
FROM OrderDetails;
```

Summary

- MIN() → Smallest value
- MAX() → Largest value
- COUNT() → Number of rows (ignores Null)
- AVG() → Average value (ignores NULL)
- SUM() → Total Sum.

MySQL LIKE, Wildcards & IN Operators

LIKE Operator

Used with WHERE to search for a pattern in a column

- % → zero, one or many characters
- _ → exactly one character

Syntax

```
SELECT column1, column2
FROM table_name
WHERE column_name LIKE pattern;
```

Date: __/__/20 Day: _____

Examples

-- Starts with "a"

WHERE CustomerName LIKE 'a%';

-- Ends with "a"

WHERE CustomerName LIKE 'a%';

-- Contains "or"

WHERE CustomerName LIKE '%or%';

-- "r" in 2nd position

WHERE CustomerName LIKE '_r%';

-- Starts with "a" and min 3 chars

WHERE CustomerName LIKE 'a__%';

-- ContactName starts "a" and "ends "o"

WHERE ContactName LIKE 'a%o';

-- Does Not start with "a"

WHERE CustomerName NOT LIKE 'a%';

Wildcards In LIKE

Wildcards help match unknown or variable characters.

Wildcard	Meaning	Example	Matches
%	0 or more chars	WHERE City LIKE 'ber%'	Berlin,
%	Anywhere in text	" " " '%os%'	Estes
_	Single character	" " " '_ondon'	London
Combination	Mix % & _	" " " 'L_n%'	London

IN Operators

lets you check if a column's value matches any value in a list.

- Works like many OR conditions
- Can also a subquery inside IN

Date: __/__/20__ Day: _____

Syntax

```
SELECT column_name(s)
FROM table_name
INHERE column_name IN (values, value2, ...);
-- OR with subquery
SELECT Column_name(s)
FROM table_name
WHERE column_name IN (SELECT column FROM other_table);
```

Examples

```
-- Customers from Germany, France, or UK
SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');
-- Same as multiple OR
WHERE Country = 'Germany' OR country = 'France' or Country = 'UK';
-- Customers with IDs returned from another query
SELECT * FROM customers
WHERE CustomerID IN (SELECT CustomerID FROM
    orders WHERE OrderID = 10308);
```

Summary

- LIKE → search with patterns
- % → many characters, _ → one character
- IN → match against a list (shortcut for multiple OR)

MySQL (Between, Aliases)

Between Operator

The BETWEEN operator is used to select values within a given range (numbers, text or data).
It is inclusive (includes the start and end value)

Date: __/__/20__ Day: _____

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Example

Numbers

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

(gets products with price from 10 to 20)

NOT BETWEEN

```
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

(gets Products outside 10-20 range)

With IN

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20
AND CategoryID NOT IN (1, 2, 3);
```

(gets products with price 10-20 but category not 1,2,3)

Text Values:

```
SELECT * FROM Products
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND
    'Mozzarella di Giovanni'
ORDER BY ProductName;
```

(gets products alphabetically b/w those names)

Date: __/__/20__ Day: _____

NOT BETWEEN Text

SELECT * FROM Products

WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'

ORDER BY ProductName;

Dates:

SELECT * FROM Orders

WHERE OrderDATE BETWEEN '1997-07-1" AND '1996-07-1';

(gets order b/w July 1 & July 31, 1996)

Aliases

Aliases are temporary names for tables, or columns, used to make queries shorter and easier to read.

- Created using AS keyword
- Valid only for that query

Column Alias

Syntax

SELECT column_name AS alias_name

FROM table_name;

Example

SELECT CustomerID AS ID, CustomerName AS Customer

FROM Customers;

SELECT CustomerName AS Customer, ContactName AS "Contact Person"

FROM Customers;

Date: __/__/20__ Day: _____

```
SELECT CustomerName, CONCAT_WS(', ', address, Postalcode,
        city, Country) AS Address
FROM Customers;
```

(combines multiple columns into one "Address" column)

Table Alias

Syntax

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

Examples

Using aliases

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.customerName = 'Around the Horn' AND c.customeID
        = o.customerID;
```

Without aliases

```
SELECT orders.OrderID, Orders.orderDate, Customers.Cust-
        omerName
FROM Customer, Orders
WHERE Customers.CustomerName = 'Around the Horn'
AND customer ID = Orders.customerID;
```

When to Use Aliases?

- When multiple tables are used
- When column names are long or unclear
- When Functions are used
- When combining multiple columns

Date: __/__/20 Day: _____

MySQL Joins

A join is used to combine rows from two or more tables, based on a related column (usually a foreign key)

INNER JOIN

Returns only rows that have matching values in both tables

Syntax

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

Example:

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Custom-
ers.CustomerID;
```

∴ Shows orders only where a customer exists.

LEFT JOIN (LEFT OUTER JOIN)

Result Returns all records from the left table, and matching records from the right table. If no match → NULL

Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Date: __/__/20 Day: _____

Example

```
SELECT c.CustomerName, o.OrderID
FROM Customers AS c
LEFT JOIN Orders AS o
    ON c.customerID = o.customerID
ORDER BY c.customerName;
```

∴ Customers with no orders still appear; OrderID will be NULL

RIGHT JOIN

Keep all rows from the right table, match from left if possible (else NULL)

Syntax

```
SELECT t1.col, t2.col
FROM table1 AS t1
RIGHT JOIN table2 AS t2
    ON t1.key = t2.key;
```

Example

```
SELECT o.OrderID, e.LastName, e.FirstName
FROM Orders AS o
RIGHT JOIN Employees AS e
    ON o.EmployeeID = e.EmployeeID
ORDER BY o.OrderID;
```

∴ All employees show up; employees with no orders have OrderID = NULL

CROSS JOIN

Cartesian product — every row of left x every row of right

Syntax

```
SELECT t1.col, t2.col
```

Date: __/__/20__ Day: _____

FROM table1 AS t1

CROSS JOIN table2 AS t2;

Example

SELECT c.CustomerName, o.OrderID

FROM Customers AS c

CROSS JOIN Orders AS o;

∴ Can be huge; rows(table1) * rows(table2)

Note: Adding a WHERE to relate Keys makes it behave like an INNER JOIN

SELECT c.CustomerName, o.OrderID

FROM Customers AS c

CROSS JOIN Orders AS o

WHERE c.CustomerID = o.CustomerID;

SELF JOIN

Join a table to itself (use aliases). Useful for comparisons within one table

Syntax (explicit JOIN style)

SELECT A.col, B.col

FROM table1 AS A

JOIN table1 AS B

ON <relation between A and B>;

Example (customers from the same city)

SELECT A.CustomerName AS Customer1,

B.CustomerName AS Customer2,

A.City

FROM Customers AS A

JOIN Customers AS B

ON A.City = B.City

Date: __/__/20__ Day: _____

AND A.CustomerID <> B.CustomerID

ORDER BY A.City;

∴ A.CustomerID < > B.CustomerID avoids pairing the same row with itself

MySQL UNION vs UNION ALL

What is Union?

- Combines result sets of two or more SELECT statements
- Important
  - Each SELECT must have the same no. of columns
  - Columns must have similar data types
  - Columns must be in the same order
- Special : Removes duplicates rows automatically

Syntax

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

Example with Customers & Suppliers table

1. UNION Example

Get all cities (no duplicates):

```
SELECT City FROM Customers
UNIDN
SELECT City FROM Suppliers
ORDER BY City;
```

2- UNION with WHERE

Get German cities (no duplicates):

```
SELECT City, Country FROM Customers
```

Date: ./__/20__ Day: _____

```
WHERE Country = 'Germany'
UNION
SELECT City, Country FROM Suppliers
WHERE Country = 'Germany'
ORDER BY City
```

UNION with Aliases

Show all customers and suppliers with type label:

```
SELECT 'Customer' AS Type, ContactName, City, Country
FROM Customers
UNION
SELECT 'Supplier', ContactName, City, Country
FROM Suppliers;
```

∴ Here, "Type" is a temporary column (alias)

What is UNION ALL?

- Same as UNION, but keeps duplicates
- When you want all rows including duplicates

Syntax

```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

Example with UNION ALL

- Union ALL Example

Get all cities (with duplicates)

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY city;
```

Date: __/__/20 Day: _____

- UNION ALL with WHERE

Get German cities (with duplicates)

```
SELECT City, Country FROM Customers
WHERE Country = 'Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country = 'Germany'
ORDER BY City;
```

MySQL GROUP BY / HAVING

GROUP BY

The Group BY statement groups rows that have the same values into summary rows.

∴ Mostly used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()).

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Examples (using Customers Table)

1- Number of customers in each country

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

2- Number of customers in each country (sorted high → low)

Date: __/__/20__ Day: _____

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

GROUP BY with JOIN

Example: Number of orders sent by each shipper

```
SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOf
FORDER
FROM Orders
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID
GROUP BY ShipperName;
```

HAVING

The HAVING clause is used to filter groups (like WHERE, but works with aggregate functions)

- WHERE works on rows
- HAVING works on grouped data

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

Examples (using Customers Table)

- Only include countries with more than 5 customers

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

Date: __/__/20__ Day: _____

• Same as above but sorted

```
SELECT COUNT (CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT (CustomerID) > 5
ORDER BY COUNT (CustomerID) DESC;
```

HAVING with Employees (Orders Table)

• Employees with more than 10 orders

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfFolders
FROM Orders
INNER JOIN Employees ON Orders.EmployeeID = Employee.EmployeID
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 10;
```

• Check if "Davolio" or "Fuller" have more than 25 orders

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
WHERE LastName = 'Davolio' OR LastName = 'Fuller'
GROUP BY LastName
HAVING COUNT (Orders.OrderID) > 25;
```

MySQL Exists, ANY & ALL Operators

Exists Operator

EXISTS checks if a subquerry returns any rows

• Returns TRUE if subquery has 1 or more rows

• Returns FALSE if subquery has no rows

∴ Often used with correlated subqueries (subquery depends on outer query)

Date: __/__/20__ Day: _____

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS (SELECT column_name FROM table_name WHERE
                condition);
```

Examples

- Suppliers with a product price < 20

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS(
    SELECT ProductName
    FROM Products
    WHERE Products.SupplierID = Suppliers.SupplierID AND Price < 20
);
```

- Suppliers with a product price = 22

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS(
    SELECT ProductName
    FROM Products
    WHERE Products.SupplierID = Suppliers.SupplierID AND Price = 22
);
```

ANY Operator

ANY compares a value with any value in a subquery result.

- Return TRUE if condition is true for atleast one value
- work with comparison operators =, !=, <, >, <=, >=.

Date: __/__/20 Day: _____

Syntax

```
SELECT column_name(s)
FROM tablename
WHEER column_name operator ANY (SELECT column_name FROM
                                tablename WHERE condition);
```

Examples

- Products where Quantity = 10 in ANY Order

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY(
    SELECT ProductID FROM OrderDetails WHERE Quantity = 10
);
```

- Products where Quantity > 99 In ANY order

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY(
    SELECT ProductID FROM OrderDetails WHERE Quantity
                                > 99
);
```

- Quantity > 1000 (False because none exist)

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY(
    SELECT ProductID FROM OrderDetails WHERE Quantity
                                > 1000
);
```

ALL Operator

All compares a value with all values in a subquery result.

Date: _/__/20__ Day: _____

- Returns TRUE only if the condition is true for every value.
- Also works with comparison operators.

Syntax

```
-- with SELECT
SELECT ALL column_name(s)
FROM table_name
WHERE condition;
-- with Where
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL(SELECT column_name
                    FROM table_name WHERE condition);
```

Examples

- List all product names

```
SELECT ALL ProductName
FROM Products
WHERE TRUE;
```

- Products where ALL orders have Quantity = 10 (will be Fake because not all orders have 10)

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL(
    SELECT ProductID FROM OrderDetails WHERE Quantity = 10
);
```

MySQL Insert Select

INSERT INTO SELECT is used to copy data from

Date: __/__/20 Day: _____

one table into another table

- The columns and data types in both tables must match
- Existing records in the target table are not affected.(they stay as they are)

Syntax.

Columns (copy all)

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

Copy Specific columns:

```
INSERT INTO table2 (col1, col2, col3)
SELECT col1, col2, col3
FROM table1
WHERE condition;
```

Example

Copy some columns

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country
FROM Suppliers;
```

Summary

- Use INSERT INTO SELECT to copy data b/w tables.
- You can copy all columns or specific columns
- Use WHERE to filter which rows get copied.

MySQL Case Statement

CASE evaluates conditions and returns a value for

Date: _/__/:0__ Day: _____

the first true condition (like if-then-else), if no WHEN matches and there is no ELSE, it returns NULL

Two forms (syntax)

1- Searched CASE (use condition)

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ....
  ELSE default_result
END
```

2- Simple CASE (compare one expression to values)

```
CASE expr
  WHEN value1 THEN result1
  WHEN value2 THEN result2
  ...
  ELSE default_result
END
```

Example

- Label quantities

```
SELECT OrderID, Quantity,
  CASE
    WHEN Quantity > 30 THEN 'greater than 30'
    WHEN Quantity = 30 THEN 'equal 30'
    ELSE 'under 30'
  END AS QuantityText
FROM OrderDetails;
```

Summary

- Use IS NULL to check out NULL (not =)
- CASE stops at the first matching WHEN
- You can nest CASEs, but keep it readable.

Date: __/__/20__ Day: _____

MySQL Null Functions

Arthetic or concatenation with NULL → result becomes NULL. Use these functions to supply fallback values.

IFNULL(expr, alt)

If expr is NULL, return alt; otherwise expr. MySQL specific (two arguments)

Syntax

IFNULL (expr, alt_value)

Example

```
SELECT ProductName, UniPrice *(UnitsInStock + IFNULL(UnitsOnOrder,0))
                    AS InventoryValue
FROM Products;
```

If UnitsOnOrder is NULL, IFNULL(...., 0) treats it as 0

COALESCE(expr1, expr2, ..., exprN)

Returns the first non-NULL expression in the list. Standard SQL can take many arguments

Syntax

COALESCE (expr1, expr2, ..., exprN)

Notes / Differences

- COALESCE is standard SQL and accepts multiple fallbacks.
- IFNULL(a,b) ≈ COALESCE(a,b) but COALESCE is prefered for portability
- Use these to prevent NULL from propagating in calculations or strings.

Date: _/__/20__ Day: _____

MySQL Comments

Comments in MySQL are used to explain SQL code or disable parts of a query (ignored by MySQL)

- Single-line Comments
  - Start with -- (double dash)
  - Everything after -- until the end of line is ignored

Examples

```
-- Select all customers
SELECT * FROM Customers;
SELECT * FROM Customers -- Where city = 'Berlin';

--select * FROM Customers; (this line Ignored)
SELECT * FROM Products;
```

Multi-line Comments

- Start with /* and end with */.
- Can span across multiple lines or used inline

Examples

```
/* Select all columns
from Customers table */
SELECT * FROM Customers;

/* Select * FROM Customers;
SELECT * FROM Products;
SELECT * FROM Orders; */
SELECT * FROM Suppliers;
-- Ignore part of a column
SELECT CustomerName, /* City, */ Country FROM Customers;
```

Date: __/__/20__ Day: _____

MySQL Operators

MySQL Arithematic Operators

Operator	Description
+	Add
-	Substract
*	Multiply
/	Divide
%	Module

MySQL Bitwise Operator

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR

MySQL Comparison Operators

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	less than or equal to
<>	Not equal to

MySQL Compand Operators

Operator	Description
+=	Add equals

Date: _/__/20__ Day: _____

-=	Add. Subsbract equals
*=	Multiply equals
/=	Divide equals
%=	Module equals
&=	Bitwise AND equals
^-=	Bitwise exclusive equals
|*=	Bitwise OR equals

MySQL Logical Operators.

Operator	Description
ALL	TRUE if all subquery values meet condition
AND	TRUE if all condition separate by AND is True
ANY	TRUE if all subquery values meet condition
BETWEEN	TRUE if operand is within range of comparisons
EXISTS	TRUE if subquery returns one or more records
IN	TRUE if operand is equal to one of list of expression
LIKE	True if operand matches a patten
NOT	Displays a record if condition is NOT TRUE
OR	TRUE if any of condition separated by OR is True
SOME	TRUE if any of subquery value meet condition

MySQL (1) Data Base

MySQL, Create DB, Drop DB

CREATE DataBase

Create a new a database (a container for tables, views)

Syntax

```
CREATE DATABASE databasename;
-- avoid error if already exists:
CREATE DATABASE IF NOT EXISTS databasename;
```

Date: __/__/20__ Day: _____

Example

```
CREATE DATABASE testDB;

CREATE DATABASE IF NOT EXISTS testDB;
```

Important Points

- You need the CREATE privilege (usually admin)
- To start using the new DB: Use testDB;
- To view databases: SHOW DATABASES;
- To see how DB was created (MySQL): SHOW CREATE DATABASE testDB;

DROP DATABASE

Delete (remove) an existing databases and all its contents (table, data, indexes). This is destructive and irreversible

Syntax

```
DROP DATABASE databasename;

-- safer: only drop if it exists:
DROP DATABASE IF EXISTS databasename;
```

Example

```
DROP DATABASE testDB;

DROP DATABASE IF EXISTS testDB;
```

Warning / Tips

- You need the DROP privilege
- All data in the database is lost after DROP. Back up if needed.
- After Droping, check SHOW DATABASES; to confirm removal.

Date: _/__/20__ Day: _____

MySQL Table Statements

CREATE TABLE

Used to create a new table in the database

Syntax

```
CREATE TABLE table_name (
    column1 data type,
    column2 datatype,
    ....
);
```

- Column → name of the column
- datatype → type of data (int, varchar, data etc)

Example

```
CREATE TABLE Persons(
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Addregs varchar(255),
    City varchar(255)
);
```

CREAT TABLE Using Another Table

Create a new table as a copy of an existing one

Syntax

```
CREATE TABLE new_table AS
SELECT column1, column2
FROM existing_table
WHERE condition;
```

Example

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM Customers;
```

Date: __/__/20__ Day: _____

DROP TABLE

Permanently deletes a table and all its data

Syntax

PROP TABLE table_name;

Example

DROP TABLE Shippers;

∴ Be careful - this deletes everything

TRUNCATE TABLE

Deletes all rows (data) from a table but keeps the table structure

Syntax

TRUNCATE TABLE table_name;

ALTER TABLE

Used to add, delete or modify columns/constraints in an existing table.

- ADD Column

ALTER Column TABLE table_name

ADD column_name datatype;

Example

ALTER TABLE Customers

ADD Email varchar (255);

- DROP Column

ALTER TABLE table_name

DROP COLUMN column_name;

Example

ALTER TABLE Customers

DROP COLUMN Email;

Date: _/_/20 Day: _____

MODIFY Column (change Data type)

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

Example

```
ALTER TABLE Persons
MODIFY COLUMN DateOF BIRTh year;
```

Example- Add then Drop Column

- Add Column

```
ALTER TABLE Persons
ADD DateOFBirth date;
```

- Drop column:

```
ALTER TABLE Persons
DROP COLUMN DateOFBirth;
```

Summary

- CREATE TABLE → make a new table
- CRAETE AS SELECT → copy an existing table
- DROP TABLE → delete table completely
- TRUNCATE → clear data, keep table
- ATTER TABLE → add/remove/change columns

MySQL Constraints

What are Constraints?

Rules applied to table columns to ensure accuracy & reliability of data.

Can be applied when creating a table (CREATE TABLE) or later (ALTER TABLE)

Column-level: applied to single column

Table-level: applied to whole table

Date: __/__/20__ Day: _____

Common MySQL Constraints

NOT NULL → No empty values

UNIQUE → No Duplicates

PRIMARY KEY → Unique row identifier

FOREIGN KEY → Link between tables

CHECK → Must need a condition

DEFAULT → Auto value if not provided

CREATE INDEX → Faster Search

MySQL NOT NULL

Ensure column cannot store NULL (empty) values

- ON CREATE

```
CREATE TABLE Persons(
    ID int NOT NULL,
    LostName varchar(255) NOT NULL
);
```

- ON ALTER

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

MySQL Unique

Ensures all values in a column are different

- Multiply UNIQUE allowed, but only one PRIMARY KEY
- ON CREATE

```
CREATE TABLE Persons(
    ID int UNIQUE,
    LASTName varchar(255)
);
```

- Multiple columns

```
CONSTRAINTS UC_Person UNIQUE (ID, LastName);
```

Date: __/__/20__ Day: _____

- On ALTER

```
ALTER TABLE Persons ADD UNIQUE (ID);
ALTER TABLE Persons ADD CONSTRINT UC_Person UNIQUE (ID, LastName);
```

- Drop UNIQUE

```
ALTER TABLE Persons DROP INDEX UC_Person;
```

MySQL Primary Key

Uniquely identifies each row (NOT NULL + UNIQUE)

- Only one PRIMARY KEY per table (can be multiple columns)
- On CREATE

```
CREATE TABLE Persons(
    ID int NOT NULL,
    PRIMARY Key (ID)
);
```

- Composite Key

```
CONSTRAINT PK_Person PRIMARY KEY (ID, LastName);
```

- On ALTER

```
ALTER TABLE Persons ADD PRIMARY KEY (ID);
```

- DROP PK

```
ALTER TABLE Persons DROP PRIMARY KEY;
```

MySQL FOREIGN KEY

Creates a link between two tables (child → parent)

- Must match a PRIMARY KEY in another table
- On CREATE

```
CREATE TABLE Orders(
    OrderID int PRIMARY KEY,
    PersonID. Int,
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

Date: __/__/20__ Day: _____

- with name

CONSTRAINT PK_PersonOrder FOREIGN KEY (PersonID) REFERENCES Persons (PersonID);

- On ALTER

ALTER TABLE Orders ADD FOREIGN KEY (PersonID) REFERENCE Persons (PersonID);

- DROP FK

ALTER TABLE Orders DROP FOREIGN KEY Fk_PersonOrder;

MySQL Check

Restricts values with a condition

- On CREATE

```
CREATE TABLE Persons (
    ID Int,
    Age int,
    CHECK (Age >= 18)
);
```

- Multiple conditions:

CONSTRAINT CHK_Person CHECK (Age >=18 AND City "Sandes");

- On ALTER

ALTER TABLE Persons ADD CHECK (Age >= 18);

- DROP CHECK

ALTER TABLE Persons DROP CHECK CHK_PersonAge;

MySQL Default

Sets a default value when no value is given

- On CREATE

```
CREATE TABLE Persons (
    City varchar (255) DEFAULT 'Sandnes'
);
```

Date: __/__/20__ Day: _____

• with System values:

OrderDate date DEFAULT CURRENT_DATE()

• On ALTER

ALTER TABLE Persons ALTER City SET DEFAULT 'Sandes';

• Drop DEFAULT

ALTER TABLE Persons ALTER City DROP DEFAULT;

MySQL Index

Speeds up searches (but slows down inserts/updates)

• Normal Index (duplicates allowed):

CREATE INDEX idx_lastname ON Persons (LastName);

• Unique Index (no duplicates):

CREATE UNIQUE INDEX idx_pname ON Persons (LastName, FirstName);

• Diop Index

ALTER TABLE Persons DROP INDEX idx_lastName;

MySQL AUTO_INCREMENT

What is AUTO_INCREMENT?

• It automatically generates a unique number for each new row

• Commonly used for the primary key column.

AUTO_INCREMENT Keyword

• Used in MySQL to enable auto_numbering

• By default: starts at 1, increases by 1 each time

Example: Create Table with AUTO_INCREMENT

```
CREATE TABLE Persons (
    PersonID Int INT NOT NULL AUTO_INCREMENT,
    LastName VARCHAR (255) NOT NULL,
    FirstName VARCHAR (255),
```

Date: __/__/20 Day: _____

```
    Age INT,
    PRIMARY KEY (PersonId)
);
```

∴ Here, PersonId is the auto_increment primary key

Start from another value

```
ALTER TABLE Persons AUTO_INCREMENT = 100;
```

∴ The next inserted record will start from 100 instead of 1

Insert Data (no need to mention auto column)

```
INSERT INTO Persons (FirstName, LastName)
VALUES ('Lars', 'Monsen');
```

∴ MySQL automatically assigns a unique PersonId.

MySQL Working with Dates

Dates in MySQL

- Important to match the format of the data when inserting into a table
- If only date is stored → easy
- If time is included → queries get more complex

MySQL Date Data Types

- DATE → YYYY-MM-DD (only date)
- DATETIME → YYYY-MM-DD HH:MI:SS (date + time)
- TIMESTAMP → " " " (date+time, also stores time zone info)
- YEAR → YYYY or YY (only Year)

Example Table

OrderId	ProductName	OrderDate
1	Geitost	2008-11-11
2	Camembert Pierrot	2008-11-09

Date: __/__/20__ Day: _____

Selecting by date (only date stored)

SELECT * FROM Orders WHERE OrderDate = '2008-11-11';

∴ Output will return rows 1 and 3

→ Easy when no time is stored.

Tip

- Use Plain DATE type if you don't need time
- If using DATETIME / TIMESTAMP, You may need functions like DATE(orderDate) to compare only the date

MySQL Views

- A view is a virtual table created from the result of any SQL query
- It looks like a real table (with rows & columns) but does not store data itself
- Always shows up-to-date data since it fetches from the original tables.

Create a View

Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ....
FROM table_name
WHERE condition;
```

Update a View

Use CREATE OR REPLACE VIEW to modify an existing view

Syntax

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Date: __/__/20__ Day: _____

Drop a View

Use Drop VIEW to delete a view

Syntax

DROP VIEW view_name;

Summary

- View = virtual table (based on query)
- CREATE VIEW → make a new view
- CREATE OR REPLACE VIEW → update view
- DROP VIEW → delete views
- Always feteches latest data from underlying tables

MongoDB (NoSQL)

What is MongoDB?

- MongoDB is a NoSQL document database
- It stores data is BSON (Binary JSON), which is similar to JSON but supports more data types,

Document in MongoDB

- A document = record in MongoDB
- Structure: key-value pairs (like JSON objects)
- Values can be: numbers, strings, booleans, arrays, or even other documents.

Example Document

```
{
  title: "Post Title 1",
  body: "Body of Post.",
  category: "News",
  likes: 1,
  tags: ["news", "events"],
  date: Date()
}
```

Date: __/__/20__ Day: _____

Query Example

Find all posts with category = "News":

db.posts.find({ category: "News" })

SQL vs Document Databases

- SQL (Relational): Data is stored in multiple tables, linked with relationships. Example: a "Users" table and an "Orders" table joined together.
- MongoDB (Document/Non-Relational): Data is stored in collections (like tables) and inside them as documents (like rows but more flexible)
  - You can keep all related data inside one document.
  - Faster to read because less joining is needed.
- Collection: A group of documents (like a table in SQL)
- Document: A record with key-value pairs (like JSON)

Local Vs Cloud Database

- Local MongoDB: Installed on your computer. You manage setup, updates, and maintenance. Free with community Server.
- Cloud MongoDB (Atlas): Hosted online by MongoDB, Eaiser, no server management needed.

∴ We'll use MongoDB Atlas (cloud)

Setting Up MongoDB Atlas

- Create a Free Account → Go to MongoDB Atlas.
- Create Cluster → choose free shared Cluster.
- Add Access
  - Database User → Make a username + password.
  - Network Access → Add your IP address so your computer can connect.

Date: __/__/20__ Day: _____

MongoDB Shell (mongosh)

- mongosh: Command-line tool to talk to MongoDB
- Install: Follow official guide for your OS.
- Check Version

  mongosh --version

Connecting to Your Database

- In Atlas, go to Databases → connect → connect with mongodB shell :
- Copy the connection string. Example

  mongosh "mongodb+srv://cluster0.mongodb.net/myFirstDatabase" --apiversion 1 --username Your_User_Name

- Paste it into terminal press Enter
- Enter Your database password.

∴ You can now connected to MongoDB

Next Steps

You will learn CRUD operations (create, Read, Update, Delete) using mongosh.

MongoDB Query API

What is the Query API?

The mongodB Query API is how you interact with your data. It lets you find, insert, update, delete and analyze documents.

- You can use it in two main types
  - CRUD Operation: Basic actions (Create, Read, Update, Delete)
  - Aggregation Pipelines: Advanced data processing & transformation

Where Can You Use it?

- mongosh → Command line tool

Date: __/__/20__ Day: _____

- Compass → MongoDB's GUI tool
- Vs Code → with MongoDB extension
- Drivers → Code libraries (Python, Node.js, java etc)

What Can the Query API Do?

- CRUD Operations

Basic actions to work with data.

- Create → Add new documents
- Read → Find documents
- Update → Modify documents
- Delete → Remove documents.

- Aggregation Pipelines

Process and transform data step by step (like SQL, GROUP BY, SUM, AVG)

Useful for analytics and reports

- Document Join

Combine data from different collections (like SQL JOIN)

- Graph & Geospatial Queries
  - Graph queries: Work with connected data (like friend in network)
  - Geospatial queries: Work with maps/ location (e.g Restaurant near me)
- Full-text Search

Search documents by text (like searching articles or product names)

- Indexing

Special data structures that make queries faster

Example: Searching by "username" is quicker if that field is indexed

- Time Series Analysis

Handle time-based data (e.g.; stock prices, IoT sensor data)

Date: __/__/20 Day: _____

MongoDB Create DB

What is mongosh?

mongosh = MongoDB Shell (command-line tool) used to interact with MongoDB

- Check Current DataBase

```
db
```

∴ Shows the database you are currently using

Example: If you connected with Atlas, you will see myFirstDocuments.

- Show All databases

```
show dbs
```

∴ Lists all existing databases.

Note, Empty databases are not shown (MongoDB only creates a database when it has data)

- Create or Switch Database

```
use databaseName
```

∴ If the database exists → it switches to it
If not, → it creates a new empty database (but not visible until data is added)

Example

```
use blog
```

∴ Now you are inside the blog database
But it won't appear in show dbs until you insert some documents.

Key Point: In MongoDB, a database is created only after it gets content.

Date: __/__/20__ Day: _____

MongoDB mongosh Create Collection

What is a Collection?

- A collection in MongoDB = like a table in SQL
- It stores multiple documents (records)

Ways to Create a Collection

- Method 1: using createCollection()

```
db.createCollection("posts")
```

∴ Creates an empty collection called posts.

- Method 2: Automatic Creation on Insert

```
db.posts.insertOne({title: "First Post", body: "Hello MongoDB"})
```

∴ If the post collection does not exists, MongoDB creates it automatically when you insert data.

Main Point

Just like databases, a collection is not considered created until it has content.

MongoDB mongosh Insert Documents

What is a documents?

- A documents = one record in MongoDB (like a row in SQL)
- Stored as BSON (JSON-like format)

Methods to Insert

1- insertOne()

- Inserts one document into a collection
- If the collection doesn't exist → MongoDB creates it automatically

Date: __/__/20 Day: _____

Example:

```
db.posts.insertOne({
    title: "Post Title 1",
    body: "Body of Post.",
    category: "News",
    likes: 1,
    tags: ["news", "events"],
    date: Date()
})
```

2- insertMany()

- Inserts multiple documents at once (arrays of objects)

Example:

```
db.posts.insertMany([
  {
   title: "Post Title 2",
   body: "Body of Post.",
   category: "Event",
   likes: 2,
   tags: ["news", "events"],
   date: Date()
  },
  {
   title: "Post Title 3",
   body: "Body of Post.",
   category: "Technology",
   likes: 3,
   tags: ["tech", "update"],
   date: Date()
  },
```

Date: __/__/20__ Day: _____

```
{
    title : Post Title 4",
    body : "Body of Post.",
    category : "Events",
    likes : 4,
    tags : ["event", "Fun"]
    date : Date()
}
])
```

Summary

- insertOne() → add one document
- insertMany() → add multiple documents
- if collection doesn't exist → MongoDB auto-creates it
- Documents are JSON-like but stored as BSON internally

MongoDB mongosh Find Data

- Methods to Retrieve Data

Find()

- Returns all documents that match a query
- if query is empty → returns everything

Example:

```
db.posts.find()
db.posts.find( {category: "News"})
```

FindOne()

- Returns only the first matching document
- if query is empty → returns the first document in collection

Example:

```
db.posts.findOne()
```

- Querying Data
  - Add a query object inside find() / findOne to filter results.

Example

db.posts.find({ category: "News" })

- Projection (select fields)
  - Second Parameter = choose which field to show or hide
  - 1 = include field
  - 0 = exclude field

Examples

→ Show only title ad data.

db.posts.find({}, { title: 1, date: 1 })

→ Exclude _id:

db.posts.find({}, { _id: 0, title: 1, date: 1 })

→ Exclude category:

db.posts.find({}, { category: 0 })

Rules: You can't mix 0 and 1 in the same projection object (except _id)

Invalid

db.posts.find({}, { title: 1, date: 0 }) ← error

Summary

- find() → many results
- findOne() → first result only
- Query object → filter documents
- Projection → select which fields to return
- Rule → use only include (1) or exclude (0), not both

Date: __/__/20__ Day: _____

MongoDB mongosh Update Documents

- Updating Documents
  - Use updateOne() → updates the first matching document
  - Use updateMany() → updates all matching documents.

Syntax

db.collection.updateOne(query, update, options)
db.collection.updateMany(query, update, options)

- Parameters
  - query → find which docs to update
  - update → define changes ($set, $inc, etc)
  - options → extra settings (like upsert)

- updateOne()

updates the first document that matches the query
Example: check likes of a post
db.posts.find({ title: "Post Title 1" })

- Update likes to 2

```
db.posts.updateOne(
  { title: "Post Title 1" },
  { $set: { likes: 2 } }
)
```

- Insert if Not Found → upsert

If no document matches, create it automatically
Example: Insert / update Post Title 5

```
db.posts.updateOne(
  { title: "Post Title 5" },
  { $set: {
      title: "Post Title 5",
```

Date: __/__/20__ Day: _____

```
        body: "Body of Posts.",
        category: "Event",
        likes: 5,
        tags: ["new", "events"],
        date: Date()
      }
    },
    { upsert: true }
)
```

- Update Many ()

updates all matching documents.

Example: Increment likes by 1 for all posts.

db.posts.updateMany({}, $inc: {like: 1} })

Summary

- updateOne() → first match only
- updateMany() → all matches
- $set → replace field value
- $inc → increase/decrease numeric field
- upsert: true → insert if not found.

MongoDB mongosh Delete

In MongoDB, you can delete documents using

- deleteOne() → Deletes the first matching document
- deleteMany() → Deletes all matching documents.

Both take a query object to find which documents to remove

delete One

- Deletes the first document that matches the query

Date: __/__/20__ Day: _____

Example

db.posts.deleteOne({ title: "Post Title 5" })

deleteMany()

Deletes all documents that match the query

Example

db.posts.deleteMany({ category: "Technology" })

Tip: Always be careful with deleteMany() as it can remove multiple document at once.

MongoDB Query Operators

Query Operators let you compare, filter and search documents. They are grouped into: Comparsions, Logical, and Evaluation

Comparison Operators

Used to compare values

- $eq → Equal to
- $ne → Not equal to
- $gt → Greater than
- $gte → Greater than or equal
- $lt → less than
- $lte → less than or equal to
- $in → Matches any value in an array

Example:

db.users.find({ age: { $gte: 18 } })

Logical Operator

Used to combine queries

- $and → Match all conditions

Date: __/__/20 Day: _____

- $or → Match any Condition
- $nor → Match if all conditions fail
- $not → Match if condition is not true

Example:

db.users.find({ $or: [{ age: 18 }, { city: "Lahore" }] })

Evaluation Operators

Used to evaluate Values

- $regex → Match with a pattern (regular expression)
- $text → Perform a text search
- $where → Use a Javascript expression

Example

db.users.find({ name: { $regex: "^A" } })

MongoDB Update Operators

Update operators are used with updateOne() or updateMany() to modify document fields or array

Field Operators

- $currentDate → Sets a field to the current day/time
- $inc → Increases or decreases a number value
- $rename → Changes the name of a field
- $set → Sets or replaces a field's value
- $unset → Deletes a field from the document

Array Operators

- $addToSet → Adds a value to an array if not exists
- $pop → Removes the first (-1) or last (1) item from array
- $pull → Removes all array elements that match a condition
- $push → Adds a new element to an array

Date: __/__/20__ Day: _____

MongoDB Aggregation Pipelines

Aggregation is used to analyze, tranform and summarize data (like group, sort or calculate)

- Pipeline: An aggregation is made of stages
  - Each stage processes documents & passes the results to the next stage
  - The order of stages matters

Example

```
db.posts.aggregate([
  // Stage 1: Find post with more than 1 like
  { $match: { likes: { $gt: 1 } } },
  // Stage 2: Group by category and sum likes
  { $group: { _id: "$category", totallikes: { $sum: "$like" } } }
])
```

Common Stages (Basics)

- $ match → Filter documents (like WHERE in SQL)
- $ group → Group by a field and apply aggregation (sum)
- $ sort → Sort documents (ascending / decending)
- $ project → Select specific fields or create new ones
- $ limit → Limit the number of documents returned
- $ skip → Skip a number of documents

MongoDB Indexing & Search

Indexing in MongoDB makes searching faster

MongoDB Atlas has a built-in full-text search engine powered by Apache Lucene

Creating a Search Index (Atlas UI)

- Go to Atlas Dashboard → cluster → search tab
- Click creates Search Index

Date: __/__/20__ Day: _____

- Use Visual Editor → click Next.
- Give your index a name (If you use "default", you don't need to mention it in a query)
- Choose Database & Collection (e.g. sample_mflix.movies)
- Click Create and wait for it to finish

Running a Search Query

Use the $search operator inside an aggregation pipeline

Example

```
db.movies.aggregate([
    {
      $search: {
        index: "default",
        text: {
          query: "Star Wars",
          path: "title"
        }
      }
    },
    {
      $project: { title: 1, year: 1 }
    }
])
```

∴ This will return movies where title contains "Star" or "Wars".

Summary

- Indexing = make search faster
- Atlas Search = full-text search engine
- $search = operator to search inside text fields.
- Use $project to display only the needed fields.

Date: _/__/'0__ Day: _____

MongoDB Schema Validation

MongoDB is schema-less by default (flexible structure)
But you can add rules so all documents in a collection follow the same structure

JSON Schema Validation

MongoDB uses the $jsonSchema operator to define rules

Example

Create a collection with validation rules

```
db.createCollection("Post", {
   validator: {
      $jsonSchema: {
         bsonType = "object",
         required = ["title", "body"],
         properties: {
            title: {bsonType: "string", decription: "post title"},
            body: { bsonType: "string", description: "Post body"},
            category: {bsonType: "string", description: "category"},
            likes: {bsonType: "int", description: "like count},
            tags {bsonType: "array", items: {bsonType: "string"},
            date: {bsonType: "date", description: "Must date"}
         }
      }
   }
})
```

Key Points

- bsonType → defines the data type (string, int, date, array)
- required → fields that must exist
- properties → rules for each field
- If a document doesn't match the schema, it will be rejected

Date: __/__/20__ Day: _____

MongoDB Data Api

The mongoDB Data API lets you query and update data in MongoDB Atlas using HTTPS requests (no drivers needed)

Useful when drivers are unavailable or unnecessary

Features

- Pre-configured HTTPS endpoints
- Supports CRUD (Create, Read, Update, Delete) + Aggregation
- Works directly with MongoDB Atlas clusters

Setup

1- Enable Data Api → Form Atlas dashboard → Data Api → enable

2 - Access level → choose
   - No Access
   - Read Only
   - Read & write
   - Custom Access

3- API key → create and copy it (you only see it once)

4- App ID → find in the Data API UI (needed in request URL)

Sending Requests

Requests are made via HTTPS (e.g using curl or any HTTP client)

Example → find one document.

```
curl --location --request POST \
'http ---- / / /findOne' \
--header 'Content-Type: application/json' \
--header 'api-key: <API_KEY>' \
--data-raw '{
    "dataSource": "<CLUSTER>",
```

Date: _/__/20__ Day: _____

"database": "sample_mflix",

"collection": "movies",

"Projection": { "title": 1 }

}'

Data API Endpoints

All endpoints use:

https://data.mongodb-api.com/app/<APP_ID>/endpoint/data/

Find One Document.

- Endpoint: Post .../findOne
- Find one document.

{ "dataSource": "<cluster>", "database": "<db>", "collection": "<col>",

"filter": {}, "projection": {} }

Find Many Documents

- Endpoint: POST .../find
- Finds multiple documents with option: filter, projection, sort, limit, skip

Insert One Document

- Endpoint: POST .../insertOne

{ "document": { "title": "Movie 1" } }

Insert Many Documents

- Endpoint: POST .../insertMany

{ "documents": [{...}, {...}] }

Update One Document

- Endpoint: POST .../updateOne
- Options: filter, update, upsert

Date: __/__/20__ Day: _____

Update Many Documents

- Endpoint: POST ... /updateMany

Delete One Document.

- Endpoint: POST ... /deleteOne
- Deletes the first matching document

Delete Many Documents

- Endpoint: POST ... /deleteMany
- Deletes all matching documents

Aggregation Documents.

- Endpoint: POST ... /aggregate
- Runs aggregation pipeline

{"pipeline": [{"$match": {"Year": 2020}}]}

Summary

- Data API = use HTTPS requests instead of drivers.
- Supports CRUD + Aggregation
- Needs API ID + API key + Cluster name
- Endpoints: findOne, find, insertOne, insertMany, updateOne, updateMany, deleteOne, deleteMany, aggregate,

MongoDB Drivers

MongoDB drivers let you connect and interact with MongoDB from your programming language or application (not just in the shell)
They provide the same methods you use in mongosh (like find(), insertOne(),) but directly in code.

Date: _/__/20__ Day: _____

Officially Supported Drivers

MongoDB provides official drivers for:

- C
- C++
- C# / .NET
- Go
- Java
- Node.js
- PHP
- Python
- Ruby
- Rust
- Scala
- Swift

& There are also community-supported libraries for other languages

Why Use Drivers?

- Run MongoDB queries directly from your app code
- Use the same syntax/operations as in mongosh
- Integrate MongoDB with different programming lang.

Summary

- mongosh = shell for manual queries
- Drivers = use MongoDB inside your applications
- Official drivers exist for most major languages

MongoDB Charts

MongoDB Charts is a visualization tool that lets you create charts and dashboards directly from

Date: __/__/20__ Day: _____

MongoDB data (without needing extra BI tools)

MongoDB Charts Setup

- Open MongoDB Atlas Dashboard → go to Charts tab
- If first time → click Activate Now (takes ~ 1 minute)
- After setup → you'll see a Charts dashbaard.
- Click a dashboard name to open it.

Creating a Chart

- Click Add Chart
- Choose your Data Source (collection from Database)
- Drag and drop fields to define charts

Example (Movies dataset → sample_mflix)

- Data Source → Movies collection
- Goal → Show how many movies released each year.

Steps:

- Drag Year → Y Axis, set Bin Size = 1
- Drag _id → X Axis, set Aggregate = COUNT
- Result → Bar chart of movies released per year