SQL

Advanced

What are we learning today?

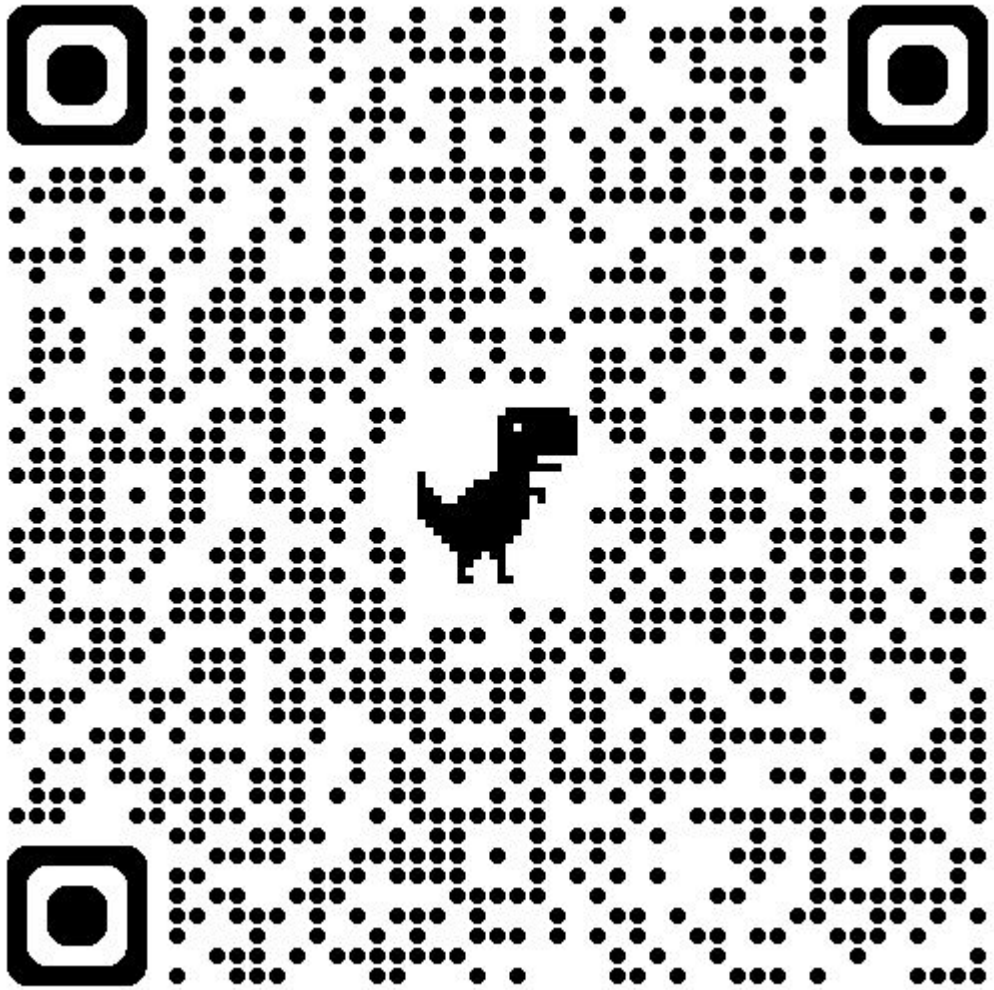
SQL syntax + Examples!

For references -

https://www.w3schools.com/sql/de

To run syntax -

https://sqliteonline.com/



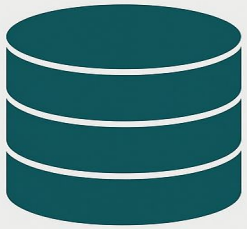
SQL for beginners

What is SQL?

WHAT IS SQL?

DEFINITION

SQL, or Structured Query Language, is the standard language for interacting with databases.



EXAMPLE QUERY

```
SELECT FirstName,
LastName
FROM STUDENT
```

FirstName	LastName

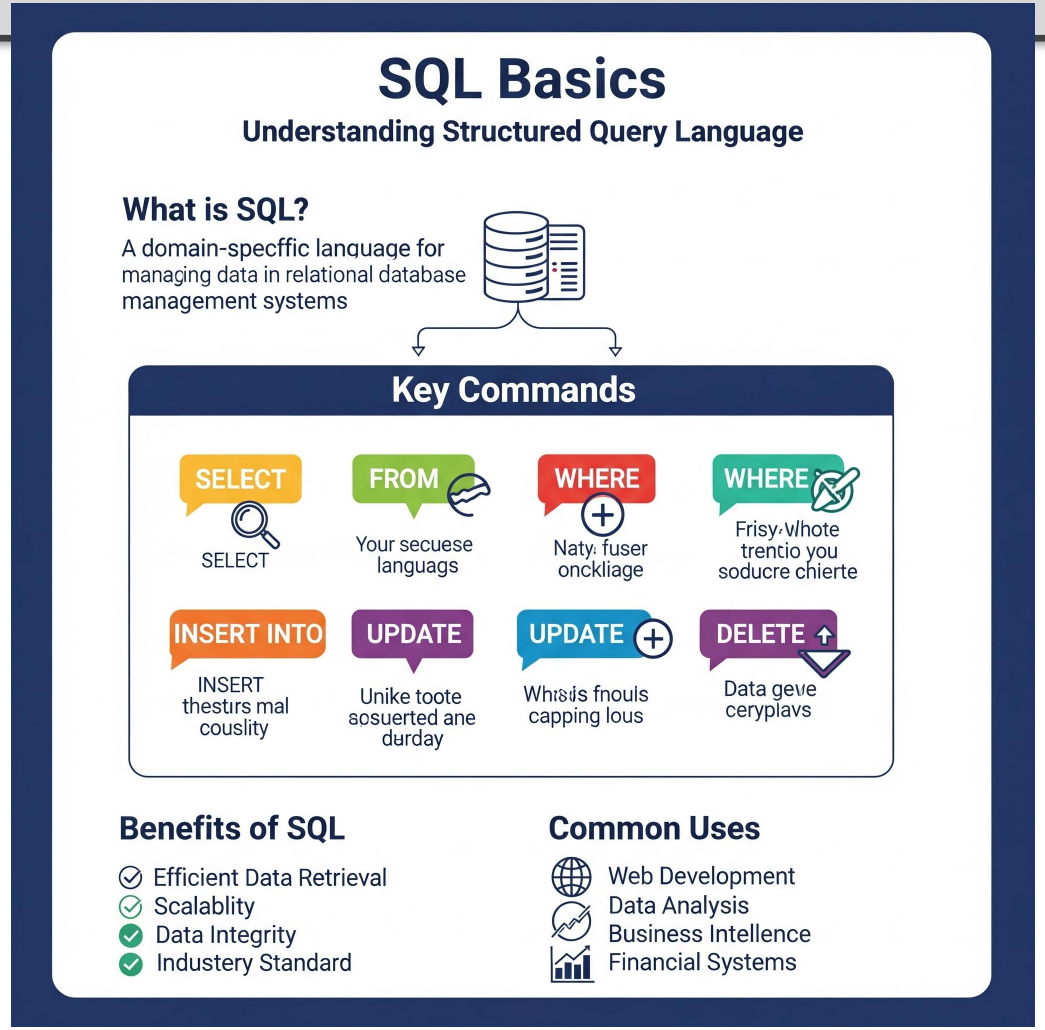
HOW SQL WORKS

SQL statements are executed → Database processes the request → Results are returned

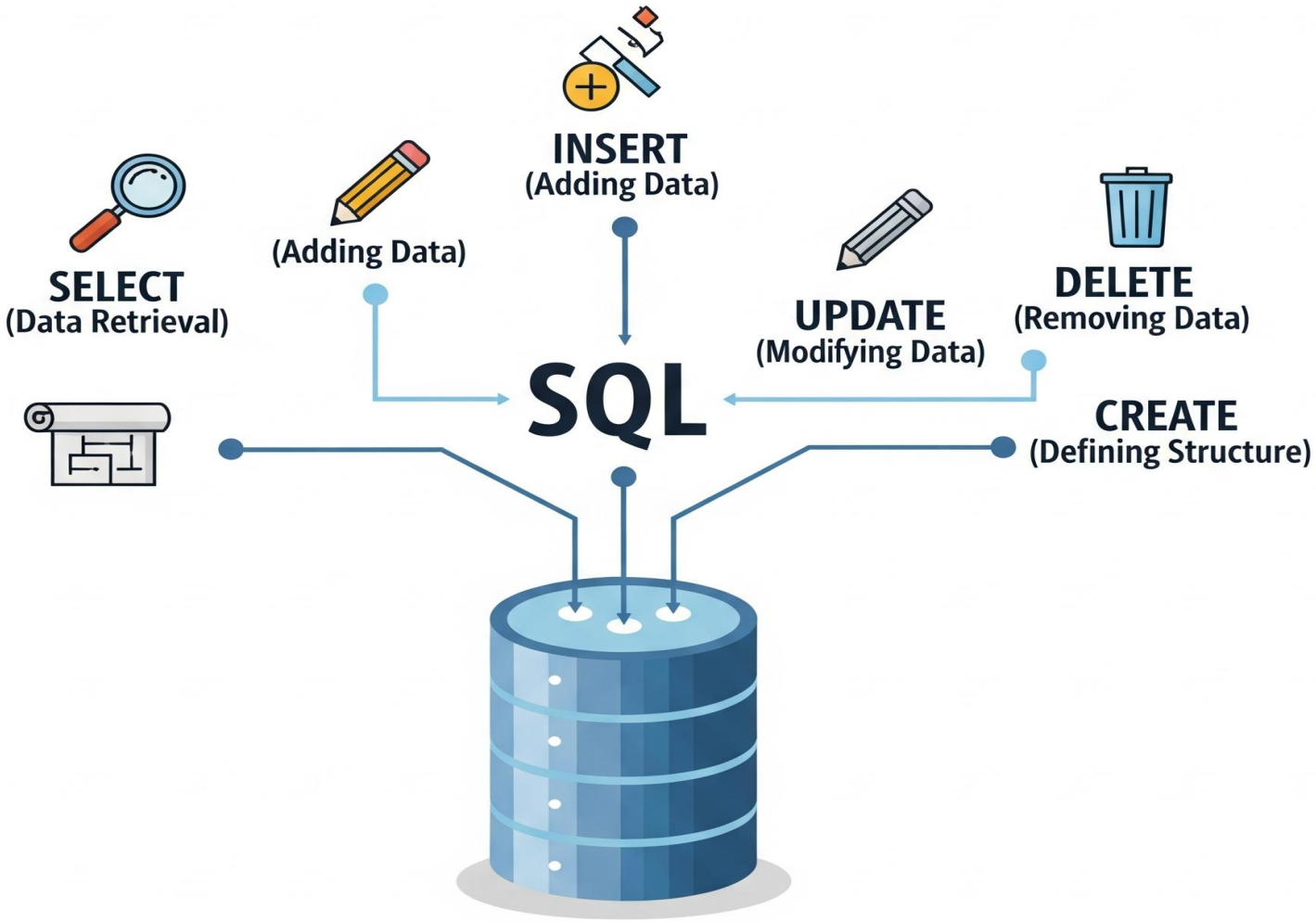
COMMON SQL COMMANDS

- SELECT
- INSERT
- UPDATE
- DELETE
- INSERT
- UPDATE
- DELETE
- CREATE TABLE

What is SQL?


SQL Basics
Understanding Structured Query Language
What is SQL?
A domain-specffic language for managing data in relational database management systems
Key Commands
SELECT
SELECT
FROM
Your secuese languags
WHERE
Naty: fuser onckliage
WHERE
Frisy·vlhote trentio you soducre chierte
INSERT INTO
INSERT thestirs mal couslity
UPDATE
Unike toote sosuerted ane durday
UPDATE
Whisis fnouls capping lous
DELETE
Data geve ceryplavs
Benefits of SQL
Efficient Data Retrieval
Scalablity
Data Integrity
Industery Standard
Common Uses
Web Development
Data Analysis
Business Intellence
Financial Systems

SQL CONCEPTS



Database Schema (Conceptual)

We are working with the following tables:

1. STUDENT
   `(StudentID, FirstName, LastName, Email, Birthday, Grade)`
2. ASSIGNMENT
   `(Type, Quantity, Points, Points_per_one)`
3. SCHEDULE
   `(ClassNum, Week, Date, Day, Topic, Format)`
4. ATTENDANCE
   `(AttendanceID, ClassNum, StudentID, Attended)`
5. DELIVERABLE
   `(DeliverableID, Type, DeliverableNumber, DueDate, Topic)`
6. STUDENT_GRADE
   `(GradeID, StudentID, DeliverableID, Score)`
7. `GRADE_SCALE`
8. `(LetterGrade,MinScore,MaxScore);`

How the Tables Relate

1. STUDENT connects to ATTENDANCE and STUDENT_GRADE via `StudentID`.
2. SCHEDULE connects to ATTENDANCE via `ClassNum`.
3. DELIVERABLE connects to STUDENT_GRADE via `DeliverableID`.
4. ASSIGNMENT provides metadata for deliverables (like type and points).

👉 This forms a relational model where we can track students, their attendance, assignments, deliverables, and performance.

Creating Tables — Sy

```
CREATE TABLE TableName (
  ColumnName DataType Constraints,
  ColumnName DataType Constraints,
  ...
);
```

TableName = name of your table (e.g., STUDENT).

ColumnName = name of each field.

DataType = kind of data (TEXT, INT, DATE, etc.).

Constraints = rules (PRIMARY KEY, NOT NULL, etc.)

CREATING TABLES IN SQL

Syntax

```
CREATE TABLE< CTITE AME>,
  <COLUMN_NAME1  <DATA_TYPE1,
  <COLUMN_NAME2  <DATA_TYPE2,
  <COLUMN_NAME3  <DATA_TYPE3,
  <COLUMN_NAME4  <DATA_TYPE4;
S;
```

Example

```
CREATE TABLE
  STUDENT
StudentID INTEGER
FirstName TEXT
LastName TEXT
Grade TEXT
s;
```

STUDENT		
StudentID	FirstName	Grade

Nim Dvir | State University of New York at Albany

Creating Tables — Example

```
CREATE TABLE STUDENT (
   StudentID INTEGER PRIMARY KEY,
   FirstName TEXT NOT NULL,
   LastName TEXT NOT NULL,
   Email TEXT UNIQUE,
   Birthday DATE,
   Grade REAL
);
```

Creating Tables — Example

```
CREATE TABLE DELIVERABLE (
   DeliverableID INTEGER PRIMARY KEY,
   Type TEXT NOT NULL,
   DeliverableNumber INTEGER,
   DueDate DATE,
   Topic TEXT,
   StudentID INTEGER,
   Score REAL,
   FOREIGN KEY (StudentID) REFERENCES STUDENT(StudentID)
);
```

Creating Tables — Example

```
CREATE TABLE Assignment (
    Type VARCHAR(50),
    Quantity INT,
    Points INT,
    Points_per_one AS (Points / Quantity)   -- Computed Column
```

Define Table Columns:

- `Type` → text (VARCHAR)
- `Quantity` → integer
- `Points` → integer
- `Points_per_one` → **calculated as (Points ÷ Quantity)**

Creating Tables — Example

```
ALTER TABLE Assignment
ADD COLUMN AssignmentID INTEGER;


);
```

Define Table Columns:

- `Type` → text (VARCHAR)
- `Quantity` → integer
- `Points` → integer
- `Points_per_one` → **calculated as (Points ÷ Quantity)**

```
CREATE TABLE SCHEDULE (
   ClassNum INTEGER PRIMARY KEY,
   Week INTEGER,
   Date DATE,
   Day TEXT,
   Topic TEXT,
   Format TEXT
);
```

```
CREATE TABLE GRADE_SCALE (
   LetterGrade TEXT PRIMARY KEY,
   MinScore INT,
   MaxScore INT
);
```

```
CREATE TABLE ATTENDANCE (
   AttendanceID INTEGER PRIMARY KEY,
   ClassNum INTEGER,
   StudentID INTEGER,
   Attended INTEGER,
   FOREIGN KEY (ClassNum) REFERENCES SCHEDULE(ClassNum),
   FOREIGN KEY (StudentID) REFERENCES STUDENT(StudentID)
);
```

```
CREATE TABLE STUDENT_GRADE (
   GradeID INTEGER PRIMARY KEY,
   StudentID INTEGER NOT NULL,
   DeliverableID INTEGER NOT NULL,
   Score INTEGER NOT NULL,
   FOREIGN KEY (StudentID) REFERENCES STUDENT(StudentID),
   FOREIGN KEY (DeliverableID) REFERENCES DELIVERABLE(DeliverableID)
);
```

Insert Values

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

You must list the values in the same order as the

1. Text values go in single quotes '...'.
2. Dates also use quotes: 'YYYY-MM
3. If you omit a column, it must

Vestibulum congue tempus

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor. Donec facilisis lacus eget mauris.

Vestibulum congue tempus

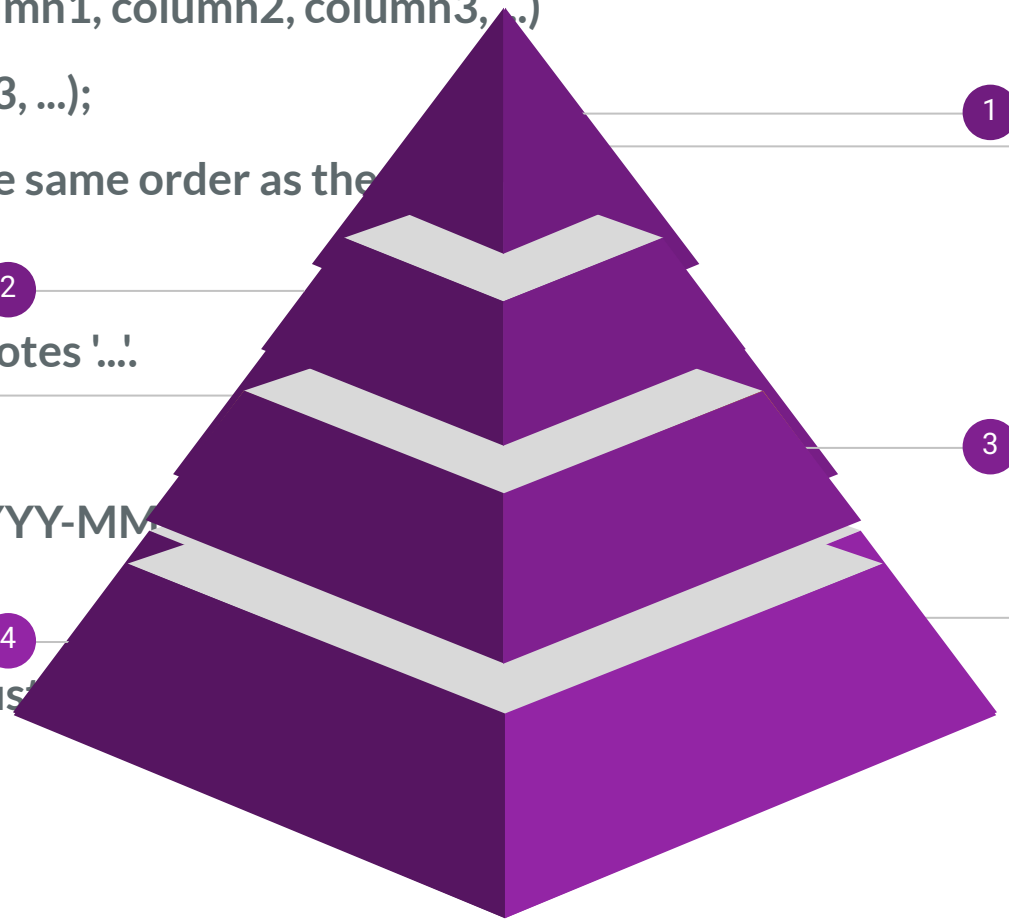
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor. Donec facilisis lacus eget mauris.

Vestibulum congue tempus

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor. Donec facilisis lacus eget mauris.

Vestibulum congue tempus

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor. Donec facilisis lacus eget mauris.



Vestibulum congue tempus

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor. Donec facilisis lacus eget mauris.

Vestibulum congue tempus

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor. Donec facilisis lacus eget mauris.

Vestibulum congue tempus

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor. Donec facilisis lacus eget mauris.

Insert Values

```
INSERT INTO STUDENT (StudentID, FirstName, LastName, Email, Birthday, Grade)
VALUES
(1, 'Alice', 'Johnson', 'alice.johnson@university.edu', '2001-05-14', 'A'),
(2, 'Bob', 'Smith', 'bob.smith@university.edu', '2000-11-22', 'B+'),
(3, 'Carol', 'Davis', 'carol.davis@university.edu', '2002-02-08', 'A-'),
(4, 'David', 'Lee', 'david.lee@university.edu', '2001-07-19', 'B'),
(5, 'Eve', 'Martinez', 'eve.martinez@university.edu', '1999-09-30', 'C+');
```

Insert Values

```
INSERT INTO DELIVERABLE (DeliverableID, Type, DeliverableNumber, DueDate, Topic, StudentID, Score)
VALUES (1, 'Quiz', 1, '2025-09-10', 'Database Basics', 1, 95);
INSERT INTO DELIVERABLE (DeliverableID, Type, DeliverableNumber, DueDate, Topic, StudentID, Score)
VALUES (2, 'Quiz', 1, '2025-09-10', 'Database Basics', 2, 88);
INSERT INTO DELIVERABLE (DeliverableID, Type, DeliverableNumber, DueDate, Topic, StudentID, Score)
VALUES (3, 'Quiz', 1, '2025-09-10', 'Database Basics', 3, 100);
INSERT INTO DELIVERABLE (DeliverableID, Type, DeliverableNumber, DueDate, Topic, StudentID, Score)
VALUES (4, 'Quiz', 2, '2025-09-20', 'Relational Model', 1, 92);
INSERT INTO DELIVERABLE (DeliverableID, Type, DeliverableNumber, DueDate, Topic, StudentID, Score)
VALUES (5, 'Quiz', 2, '2025-09-20', 'Relational Model', 2, 75);
INSERT INTO DELIVERABLE (DeliverableID, Type, DeliverableNumber, DueDate, Topic, StudentID, Score)
VALUES (6, 'Quiz', 2, '2025-09-20', 'Relational Model', 3, 85);
```

Insert Values

Why This Setup Works

Each row = one student's submission.

Easy to query for:

A student's average score across deliverables.

The class average for a specific deliverable.

Performance per topic (e.g., all "Database Basics" quizzes).

Insert Values

```
INSERT INTO Assignment (Type, Quantity, Points)
VALUES
('Quiz', 4, 80),
('Exam', 2, 100),
('Project', 1, 50);
```

Populate with unique IDs

If the table already has data, you'll need to fill `AssignmentID` with unique values. For example (SQLite):

```
UPDATE Assignment
SET AssignmentID = rowid;
```

- `rowid` is a built-in unique identifier in SQLite.
- In MS Access, you can either manually update or add an **AutoNumber** field through design view.

Insert Values

```
INSERT INTO SCHEDULE (ClassNum, Week, Date, Day, Topic, Format)
VALUES (1, 1, '2025-09-01', 'Monday', 'Introduction to Databases', 'Lecture');

INSERT INTO SCHEDULE (ClassNum, Week, Date, Day, Topic, Format)
VALUES (2, 1, '2025-09-03', 'Wednesday', 'Relational Model', 'Lecture');
INSERT INTO SCHEDULE (ClassNum, Week, Date, Day, Topic, Format)
VALUES (3, 2, '2025-09-08', 'Monday', 'Entity Relationship Modeling', 'Lecture');

INSERT INTO SCHEDULE (ClassNum, Week, Date, Day, Topic, Format)
VALUES (4, 2, '2025-09-10', 'Wednesday', 'Hands-on: ER Diagrams in Access', 'Lab');
```

Insert Values

```
INSERT INTO ATTENDANCE (AttendanceID, ClassNum, StudentID, Attended)
VALUES (1, 1, 1, 1);

INSERT INTO ATTENDANCE (AttendanceID, ClassNum, StudentID, Attended)
VALUES (2, 1, 2, 0);

INSERT INTO ATTENDANCE (AttendanceID, ClassNum, StudentID, Attended)
VALUES (3, 2, 3, 1);

-- multiply at once (doesn;t work for MS Access)
INSERT INTO ATTENDANCE (AttendanceID, ClassNum, StudentID, Attended)
VALUES
(1, 1, 1, 1),
(2, 1, 2, 0),
(3, 2, 1, 1),
(4, 2, 2, 1)
```

Insert Values

```
INSERT INTO GRADE_SCALE VALUES
('A', 94, 100),
('A-', 89, 93),
('B+', 85, 88),
('B', 82, 84),
('B-', 79, 81);
```

Insert Values

```
INSERT INTO STUDENT_GRADE (GradeID, StudentID, DeliverableID, Score)

VALUES

-- Deliverable 1

(1, 1, 1, 95),

(2, 2, 1, 88),

(3, 3, 1, 100),

(4, 4, 1, 76),

(5, 5, 1, 82),


-- Deliverable 2

(6, 1, 2, 90),

(7, 2, 2, 84),

(8, 3, 2, 92),

(9, 4, 2, 70),

(10, 5, 2, 85),


-- Deliverable 3

(11, 1, 3, 88),

(12, 2, 3, 91),
```

Insert Values

```
INSERT INTO STUDENT_GRADE (GradeID, StudentID, DeliverableID, Score)

VALUES

-- Deliverable 4

(16, 1, 4, 100),

(17, 2, 4, 86),

(18, 3, 4, 98),

(19, 4, 4, 75),

(20, 5, 4, 83),


-- Deliverable 5

(21, 1, 5, 87),

(22, 2, 5, 82),

(23, 3, 5, 90),

(24, 4, 5, 78),

(25, 5, 5, 88),


-- Deliverable 6

(26, 1, 6, 93),

(27, 2, 6, 85),
```

Basic SELECT Syntax

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

SELECT * retrieves all columns.

You can filter with WHERE.

You can rename columns with AS.

Example: Select Specific Columns

```
SELECT FirstName, LastName, Grade
FROM STUDENT;
```

Result: Only shows first name, last name, and grade.

Filtering with WHERE – Syntax

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

Filtering with WHERE –

```
SELECT S.FirstName, S.LastName, S.Grade

FROM STUDENT S

JOIN GRADE_SCALE G ON S.Grade = G.LetterGrade

WHERE G.MinScore >= 90;
```

Result: Shows students with grades 90 and above.

- problem -- are grades are letters! Need to change

Ordering Results – Syntax

```
SELECT column1, column2

FROM table_name

ORDER BY column_name ASC|DESC;
```

Ordering Results – Example:

```
SELECT FirstName, LastName, Grade
FROM STUDENT
ORDER BY Grade DESC;
```

Result: Students sorted by grade, highest first.

Using Aliases - Syntax

```
SELECT column_name AS alias
FROM table_name;
```

Using Aliases – Example:

```
SELECT FirstName || ' ' || LastName AS FullName, Grade
FROM STUDENT;
```

Result: Combines first and last names into one column

Select with Conditions

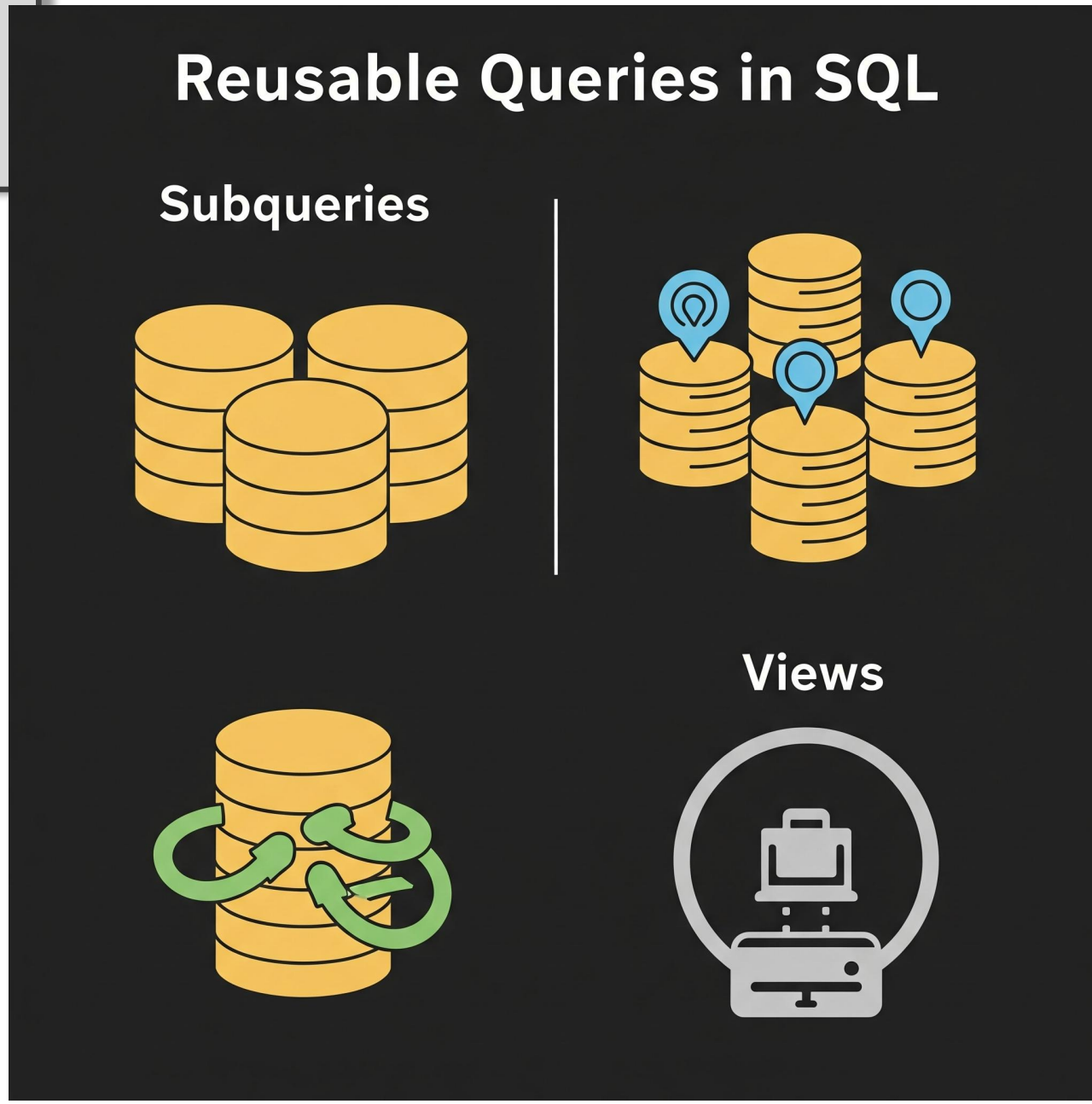
```
SELECT FirstName, LastName, Grade
FROM STUDENT
WHERE Birthday < '2000-01-01';
```

Result: Students born before the year 2000

Reusable Queries (Subqueries, Views)

Queries we can refer back to

Doesn't work in all databases



Subqueries (Inline SELECT) – Syntax

```
SELECT column1, column2
FROM (SELECT ... FROM table WHERE ...) AS temp_table;
```

Subqueries (Inline SELECT)

```
SELECT FirstName, LastName
FROM STUDENT
WHERE StudentID IN (
   SELECT StudentID
   FROM STUDENT_GRADE
   WHERE Score > 90
);
```

Result: Returns names of students who scored above 90.

Views (Permanent, Saved Query)

What is a View?

- **A View is a saved query that acts like a virtual table.**
- **It does not store data but runs the query each time you call it.**
- **Useful for:**
  - Reusing complex queries
  - Hiding complexity from end users
  - Providing consistent reports

Views (Permanent, Saved Query)

Syntax:

```
CREATE VIEW view_name AS
SELECT ...
FROM ...
WHERE ...;
```

Views (Permanent, Saved Query)

Example:

```
CREATE VIEW HighPerformers AS

SELECT s.FirstName, s.LastName, g.Score

FROM STUDENT s

JOIN STUDENT_GRADE g ON s.StudentID = g.StudentID

WHERE g.Score >= 90;
```

Then you can use it like a table:

```
SELECT * FROM HighPerformers;
```

Example: View of All Students with Their Deliverables

```
CREATE VIEW StudentDeliverables AS

SELECT s.StudentID, s.FirstName, s.LastName,

    d.Type, d.DeliverableNumber, d.DueDate, sg.Score

FROM STUDENT s

JOIN STUDENT_GRADE sg ON s.StudentID = sg.StudentID

JOIN DELIVERABLE d ON sg.DeliverableID = d.DeliverableID;
```

👉 Now you can run:

```
SELECT * FROM StudentDeliverables;
```

Example: View for Upcoming Deliverables

```
CREATE VIEW UpcomingDeliverables AS
SELECT DeliverableID, Type, DueDate, Topic
FROM DELIVERABLE
WHERE DueDate > DATE('now');
```

Example: View for Average Scores per Deliverable

```
CREATE VIEW DeliverableAverages AS
SELECT d.Type, d.DeliverableNumber,
    AVG(sg.Score) AS AvgScore
FROM DELIVERABLE d
JOIN STUDENT_GRADE sg ON d.DeliverableID =
sg.DeliverableID
GROUP BY d.Type, d.DeliverableNumber;
```

CTE (Common Table Expression)

Syntax:

```
WITH cte_name AS (
    SELECT ...
    FROM ...
)
SELECT ...
FROM cte_name;
```

CTE (Common Table Expression)

Example:

```
WITH AvgGrades AS (
    SELECT StudentID, AVG(GradeValue) AS AvgGrade
    FROM STUDENT_GRADE
    GROUP BY StudentID
)
SELECT s.FirstName, s.LastName, a.AvgGrade
FROM STUDENT s
JOIN AvgGrades a ON s.StudentID = a.StudentID
WHERE a.AvgGrade > 85;
```

Result: Students with an average grade higher than 85.

GROUP BY Basics - Syntax

```
SELECT column, AGGREGATE_FUNCTION(column2)
FROM table
GROUP BY column;
```

- `AGGREGATE_FUNCTION` = `COUNT()`, `AVG()`, `SUM()`, `MIN()`, `MAX()`
- Groups rows that have the same values in one or more columns.

Example: Count Students Per Course

```
SELECT c.CourseName, COUNT(sc.StudentID) AS TotalStudents

FROM COURSE c

JOIN STUDENT_COURSE sc ON c.CourseID = sc.CourseID

GROUP BY c.CourseName;
```

Result: Returns the number of students enrolled in each course.

Example: Average Grade Per Student

Code (SQLite):

```
SELECT s.FirstName, s.LastName, AVG(g.GradeValue) AS AverageGrade
FROM STUDENT s
JOIN STUDENT_GRADE g ON s.StudentID = g.StudentID
GROUP BY s.StudentID, s.FirstName, s.LastName;
```

Result: Each student's average grade across all their assignments/courses.

Example: Highest Grade Per Course

```
SELECT c.CourseName, MAX(g.GradeValue) AS HighestGrade

FROM COURSE c

JOIN STUDENT_GRADE g ON c.CourseID = g.CourseID

GROUP BY c.CourseName;
```

Result: The maximum grade achieved in each course.

HAVING Basics - Syntax

```
SELECT column, AGGREGATE_FUNCTION(column2)
FROM table
GROUP BY column
HAVING condition;
```

- **HAVING** works like **WHERE** but for aggregated results.
- Example: "Only show students with average grade > 85."

Example: Students with Average > 85

```
SELECT s.FirstName, s.LastName, AVG(g.GradeValue) AS AverageGrade
FROM STUDENT s
JOIN STUDENT_GRADE g ON s.StudentID = g.StudentID
GROUP BY s.StudentID, s.FirstName, s.LastName
HAVING AVG(g.GradeValue) > 85;
```

- *Result:* Only students whose average grade is above 85.

Example: Courses with More Than 3 Students

```
SELECT c.CourseName, COUNT(sc.StudentID) AS TotalStudents
FROM COURSE c
JOIN STUDENT_COURSE sc ON c.CourseID = sc.CourseID
GROUP BY c.CourseName
HAVING COUNT(sc.StudentID) > 3;
```

Result: Only courses with more than 3 enrolled students.

Example: Courses with Maximum Grade Below 70

Code (SQLite):

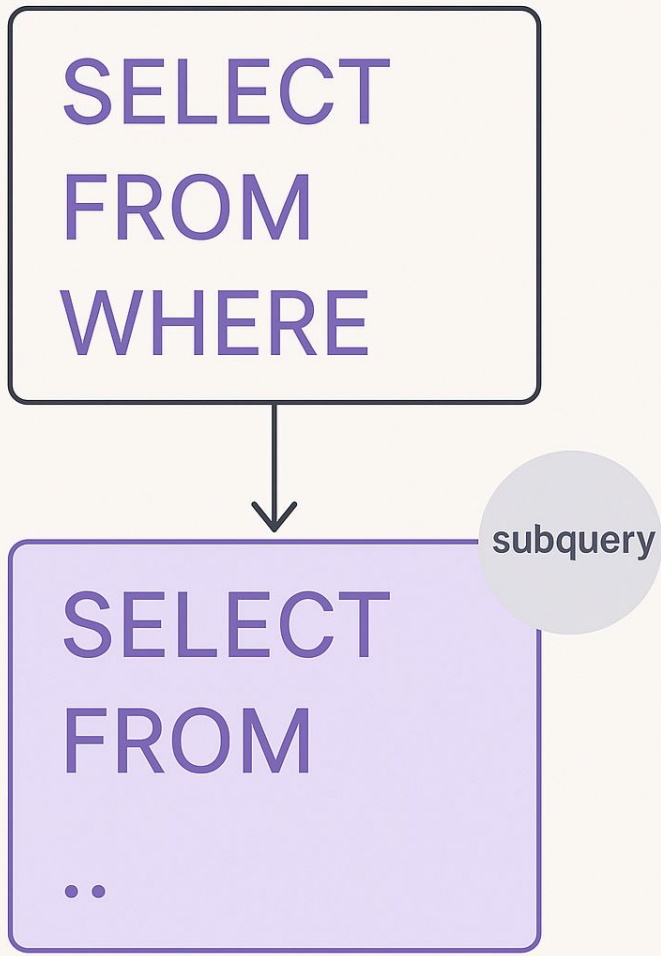
```
SELECT c.CourseName, MAX(g.GradeValue) AS HighestGrade
FROM COURSE c
JOIN STUDENT_GRADE g ON c.CourseID = g.CourseID
GROUP BY c.CourseName
HAVING MAX(g.GradeValue) < 70;
```

Result: Lists courses where no student scored above 70.

Subqueries

What is a Subquery?

- A subquery is a query nested inside another SQL statement.
- It can return a single value, a row, or even an entire table.
- Common use cases:
  - Filtering (`WHERE`)
  - Calculated fields (`SELECT`)
  - Derived tables (`FROM`)



Subqueries

Syntax:

```
SELECT column1
FROM table
WHERE column2 = (SELECT column2 FROM table WHERE condition);
```

Example: Find Students Above the Average

```
SELECT FirstName, LastName, GradeValue
FROM STUDENT s
JOIN STUDENT_GRADE g ON s.StudentID = g.StudentID
WHERE g.GradeValue > (
  SELECT AVG(GradeValue)
  FROM STUDENT_GRADE
);
```

Result: Only students whose grade is higher than the average grade across all students.

Example: Students Who Submitted Deliverable #1

```
SELECT FirstName, LastName
FROM STUDENT
WHERE StudentID IN (
    SELECT StudentID
    FROM STUDENT_GRADE
    WHERE DeliverableID = 1
);
```

Result: Lists names of students who submitted the first deliverable.

Example: Students Who Submitted Deliverable #1

```
SELECT FirstName, LastName
FROM STUDENT
WHERE StudentID IN (
    SELECT StudentID
    FROM STUDENT_GRADE
    WHERE DeliverableID = 1
);
```

Result: Lists names of students who submitted the first deliverable.

Example: Find Assignments Worth More Than the Average Points

```
SELECT Type, Points
FROM ASSIGNMENT
WHERE Points > (
    SELECT AVG(Points)
    FROM ASSIGNMENT
);
```

Result: Returns only assignments whose `Points` value is above the average across all assignments.

Example: Select Assignments Matching Maximum Points

Code (SQLite):

```
SELECT Type, Points
FROM ASSIGNMENT
WHERE Points = (
    SELECT MAX(Points)
    FROM ASSIGNMENT
);
```

Result: Finds the assignment(s) with the highest point value.

Example: Subquery in FROM (Derived Table)

```
SELECT Type, Quantity, Points
FROM (
  SELECT Type, Quantity, Points
  FROM ASSIGNMENT
  WHERE Quantity > 1
) AS MultiAssignments;
```

Result: Only assignments that require more than one submission, treated as a temporary table.

Example: Subquery in SELECT

```
SELECT Type, Quantity, Points
FROM (
   SELECT Type, Quantity, Points
   FROM ASSIGNMENT
   WHERE Quantity > 1
) AS MultiAssignments;
```

Result: Only assignments that require more than one submission, treated as a temporary table.

Example: Subquery in SELECT

```
SELECT Type, Points,
  (SELECT AVG(Points) FROM ASSIGNMENT) AS AvgPoints
FROM ASSIGNMENT;
```

Result: Shows each assignment with the class-wide average points appended.

Introduction to JOINs

Relational databases are powerful because they allow us to combine information from multiple tables.

JOINs are used to retrieve data spread across related tables.

Introduction to JOINs

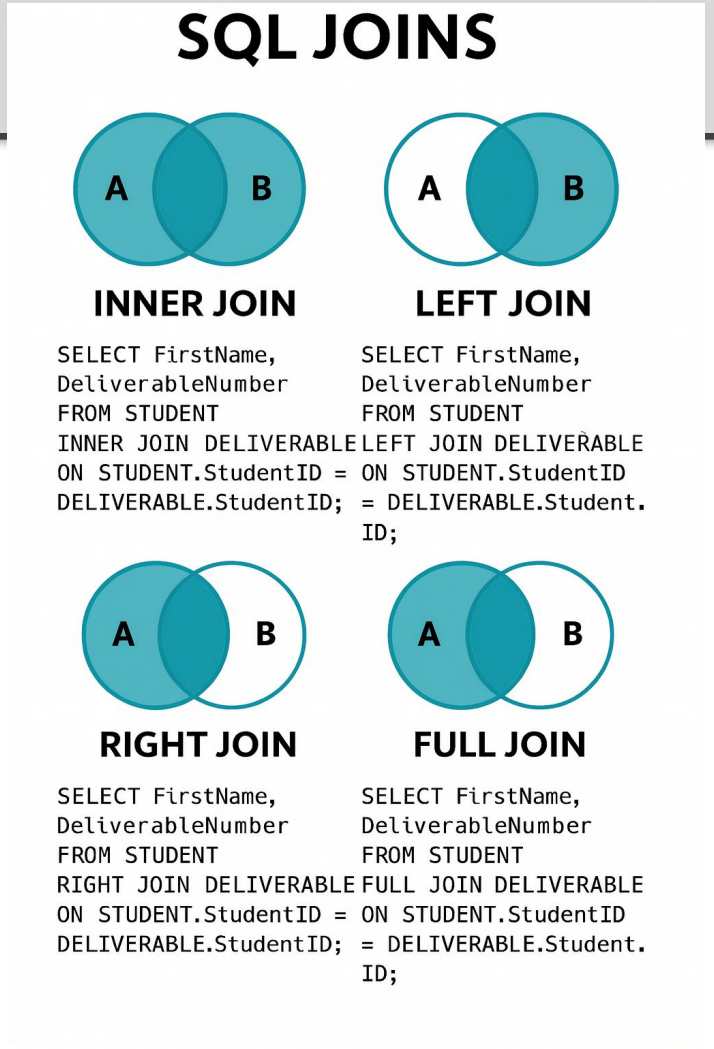
Main types of JOINs:

INNER JOIN – returns only matching rows

LEFT JOIN – returns all rows from the left table + matching rows from the right

RIGHT JOIN – not supported in SQLite (can simulate with LEFT JOIN by swapping tables)

FULL OUTER JOIN – also not directly supported in SQLite (workaround with UNION)



INNER JOIN Syntax

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

Example: Students and Their Deliverables

```
SELECT s.FirstName, s.LastName, d.Type, d.DueDate
FROM STUDENT s
INNER JOIN STUDENT_GRADE sg ON s.StudentID = sg.StudentID
INNER JOIN DELIVERABLE d ON sg.DeliverableID = d.DeliverableID;
```

Explanation:

Combines STUDENT, STUDENT_GRADE, and DELIVERABLE.

Shows which student has which deliverable and when it is due.

LEFT JOIN Syntax

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

Example: Students Without Grades Yet

```
SELECT s.FirstName, s.LastName, sg.Score

FROM STUDENT s

LEFT JOIN STUDENT_GRADE sg ON s.StudentID = sg.StudentID;
```

Explanation:

Returns all students, even those with no grades yet (NULL in Score).

Useful for tracking missing submissions.

Example: Students Without Grades Yet

```
SELECT s.FirstName, s.LastName, sg.Score
FROM STUDENT s
LEFT JOIN STUDENT_GRADE sg ON s.StudentID = sg.StudentID;
```

Explanation:

Returns all students, even those with no grades yet (NULL in Score).

Useful for tracking missing submissions.

UNION in SQL

Syntax

```
SELECT column_list FROM table1
UNION [ALL]
SELECT column_list FROM table2;
```

- `UNION` → combines results and removes duplicates.
- `UNION ALL` → combines results without removing duplicates (faster, but may repeat rows).
- Each `SELECT` must have the same number of columns and compatible data types.

Example with Our Schema

Suppose we want to see a list of all student names who either submitted a deliverable or attended a class.

Using **UNION**:

```
-- Students from DELIVERABLE table
SELECT StudentID
FROM DELIVERABLE
UNION
-- Students from ATTENDANCE table
SELECT StudentID
FROM ATTENDANCE;
```

✅ This query gives us a unique list of students who either:

- have a submission in **DELIVERABLE** OR
- have attendance in **ATTENDANCE**.

Adding More Columns

```
-- Names of students from Deliverables

SELECT s.FirstName || ' ' || s.LastName AS StudentName, 'Deliverable' AS Source

FROM STUDENT s

JOIN DELIVERABLE d ON s.StudentID = d.StudentID


UNION


-- Names of students from Attendance

SELECT s.FirstName || ' ' || s.LastName AS StudentName, 'Attendance' AS Source

FROM STUDENT s

JOIN ATTENDANCE a ON s.StudentID = a.StudentID;
```

Using `UNION ALL` (keeping duplicates)

```
SELECT StudentID
FROM DELIVERABLE

UNION ALL

SELECT StudentID
FROM ATTENDANCE;
```

Now, if a student has both attendance and a deliverable, their ID will appear twice.

Practical Example: Who Engages More?

```
SELECT StudentID, COUNT(*) AS EngagementCount
FROM (
   SELECT StudentID FROM DELIVERABLE
   UNION ALL
   SELECT StudentID FROM ATTENDANCE
)
GROUP BY StudentID;
```

```
SELECT column1 + column2 AS
alias_name

FROM table_name;
```

You can use: **+**, **-**, *****, **/**, **%** (modulus).

ARITHMETIC EXPRESSIONS IN SQL

In SQL, arithmetic expressions perform calculations on numeric data.

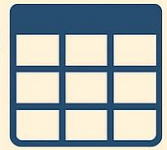
USES

OPERATORS

+	Addition	-	Subtraction
*	Multiplication	/	Division

EXAMPLE

```
SELECT first_name, last_name,
salary * 1.1
FROM employee;
```



Example (Deliverable Scores)

Suppose we want to add 5 bonus points to each deliverable score:

```
SELECT DeliverableID, StudentID, Score,
    Score + 5 AS BonusScore
FROM DELIVERABLE;
```

Example String Expressions

Syntax

```
SELECT column1 || ' ' || column2 AS alias_name
FROM table_name;
```

- `||` → concatenation operator in SQLite.

Example (Full Student Names)

```
SELECT FirstName || ' ' || LastName AS FullName, Email
FROM STUDENT;
```

Date and Time Calculations

SQLite has `DATE()`, `DATETIME()`, `JULIANDAY()`.

Example (Age from Birthday)

```
SELECT FirstName, LastName,
       (strftime('%Y', 'now') - strftime('%Y', Birthday)) AS
Age
FROM STUDENT;
```

✅ This calculates the student's age based on their birthday.

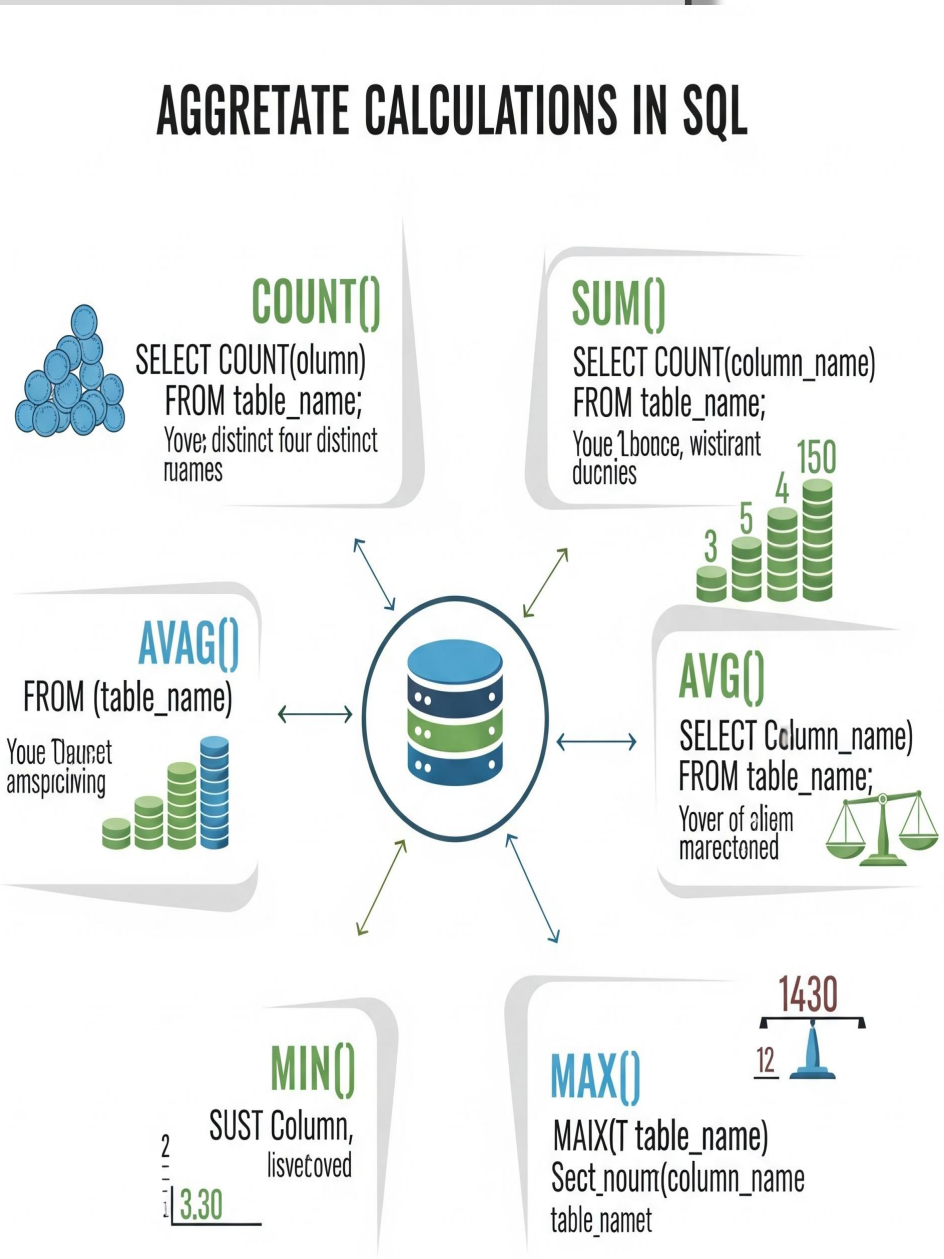
Aggregate Calculations

Example (Average Score)

```
SELECT AVG(Score) AS AverageScore
FROM DELIVERABLE;
```

Example (Highest & Lowest Score)

```
SELECT MAX(Score) AS MaxScore,
MIN(Score) AS MinScore
FROM DELIVERABLE;
```



AGGREGATE CALCULATIONS IN SQL

SUM	AVG	COUNT	MIN	MAX
calculates the total	calculates the average	counts rows	finds the smallest	finds largest

Example Queries:

```
SELECT SUM(Score) FROM DELIVERABBLE;
SELECT AVG(Score) FROM DELIVERABBLE;
SELECT COUNT(Score) FROM DELIVERABLE;
SELECT MIN(Score) FROM DELIVERABLE;
SELECT MAX(Score) FROM DELIVERABLE;
```

Outputs:

```
SUM: 283          AVG: 94.33
COUNT: 3        COUNT: 88
MIN: 88           MAX: 100
```

Conditional Expressions - Example (Letter Grade Assignment)

```
SELECT StudentID, Score,
       CASE
         WHEN Score >= 90 THEN 'A'
         WHEN Score >= 80 THEN 'B'
         WHEN Score >= 70 THEN 'C'
         WHEN Score >= 60 THEN 'D'
         ELSE 'F'
       END AS LetterGrade
FROM DELIVERABLE;
```

Combined Example

Imagine we want a report of each student with their name, average score, and assigned grade:

```
SELECT s.FirstName || ' ' || s.LastName AS StudentName,
       ROUND(AVG(d.Score), 2) AS AvgScore,
       CASE
         WHEN AVG(d.Score) >= 90 THEN 'A'
         WHEN AVG(d.Score) >= 80 THEN 'B'
         WHEN AVG(d.Score) >= 70 THEN 'C'
         WHEN AVG(d.Score) >= 60 THEN 'D'
         ELSE 'F'
       END AS FinalGrade
  FROM STUDENT s
```

Beyond Expressions — Commands We Haven't Covered Yet

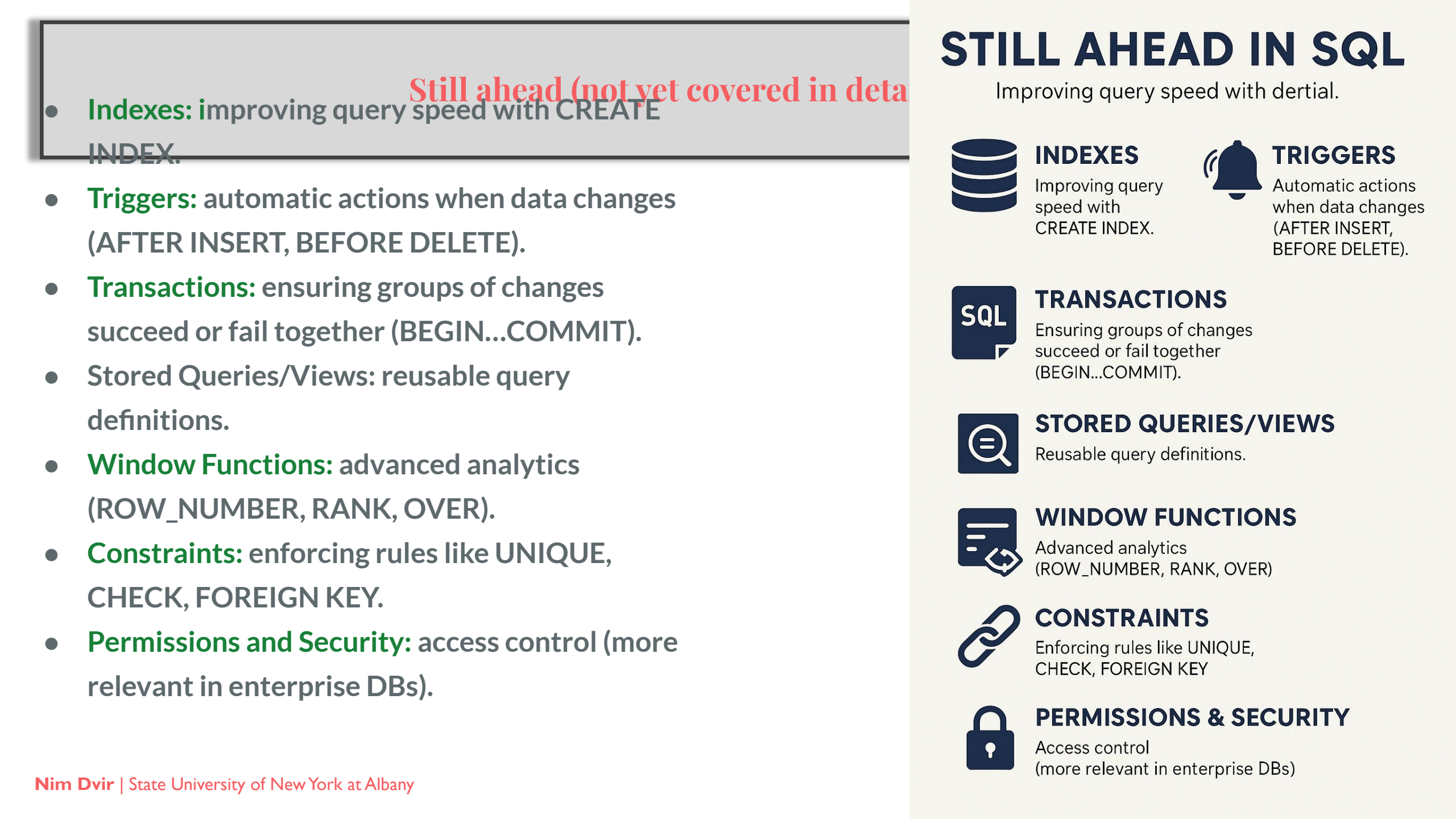
We have focused on:

- Arithmetic operations (`+`, `-`, `*`, `/`)
- String concatenation (`||`)
- Date/time functions (`strftime`, `DATE`)
- Aggregates (`AVG`, `MAX`, `MIN`)
- Conditional logic (`CASE`)

Still ahead (not yet covered in deta

- **Indexes:** improving query speed with CREATE INDEX.
- **Triggers:** automatic actions when data changes (AFTER INSERT, BEFORE DELETE).
- **Transactions:** ensuring groups of changes succeed or fail together (BEGIN…COMMIT).
- **Stored Queries/Views:** reusable query definitions.
- **Window Functions:** advanced analytics (ROW_NUMBER, RANK, OVER).
- **Constraints:** enforcing rules like UNIQUE, CHECK, FOREIGN KEY.
- **Permissions and Security:** access control (more relevant in enterprise DBs).

Nim Dvir | State University of New York at Albany



W3Schools SQL Tutorial

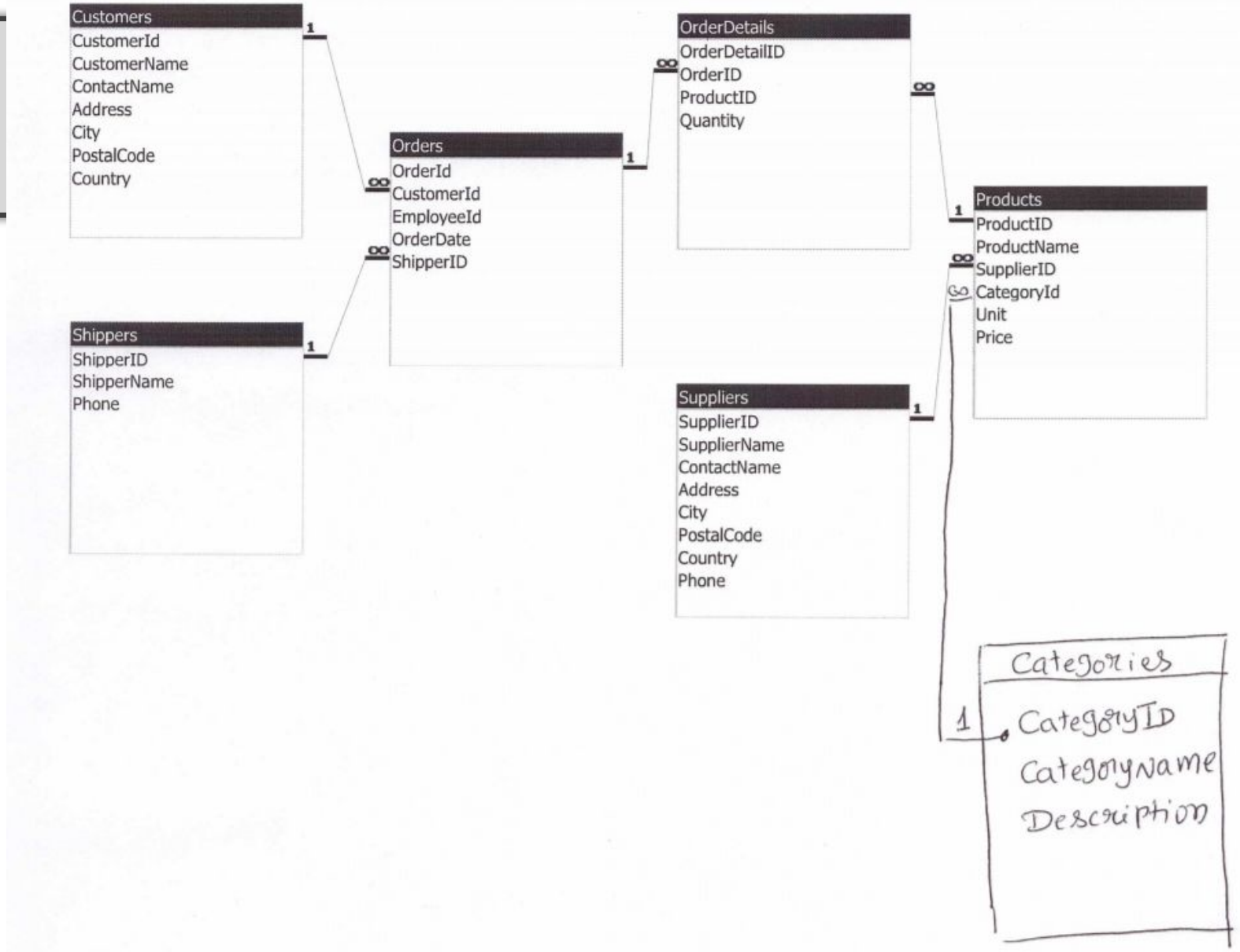
f you are new to SQL or would like a clear, accessible reference, I strongly recommend exploring the W3Schools SQL Tutorial.

- Clarity and Accessibility: The tutorial is written in simple, straightforward language with many examples you can run directly in their interactive editor ("Try it Yourself" feature). This makes it particularly suitable for beginners and practitioners who want quick reinforcement.

https://www.w3schools.com/sql/default.asp

While W3Schools is excellent for syntax and quick examples, **it does not always go into theoretical depth (like relational algebra or database design theory).**

You should use it in combination with your course materials and textbooks for a **well-rounded understanding.**



Attendance question: How would you like to learn SQL?