07 ADVANCED SQL
CS 340: INTRODUCTION TO DATABASE SYSTEMS

Adapted from: Dr. Omar Alomeir 2023

Course instructor:
Dr. Maram Alajlan.

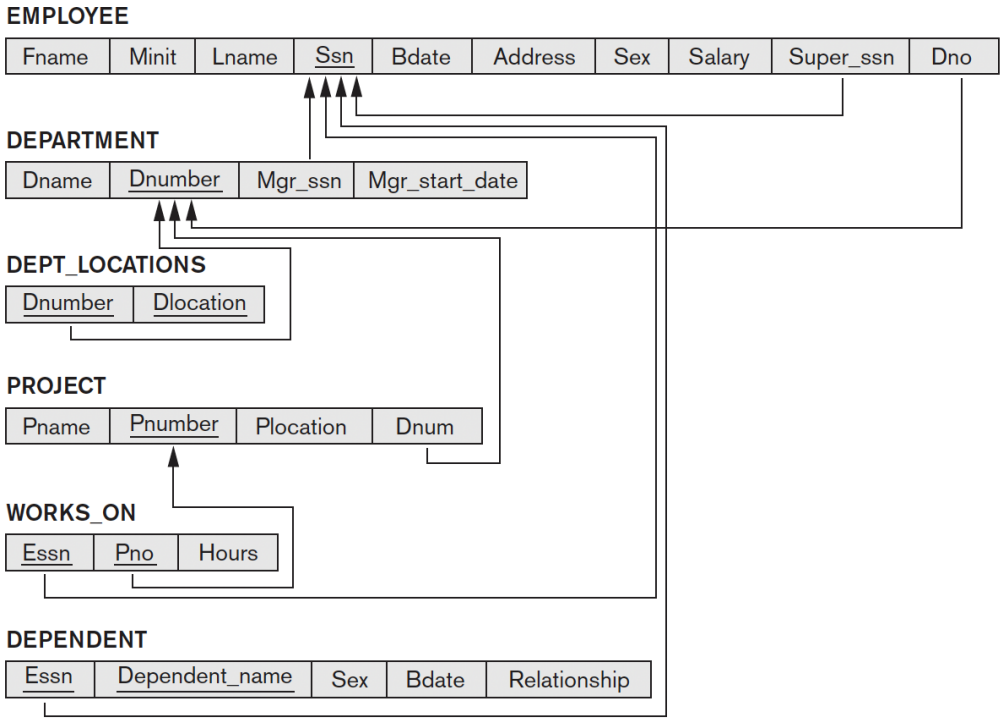
LEARNING GOALS

CLO1: Create a relational database schema in SQL that incorporates key, entity integrity, and referential integrity constraints and uses a declarative query language (SQL) to elicit information from a database. (Skills)

Chapter objectives:

- Write complex SQL queries. Includes: outer-joins, aggregation, sub-queries, and other forms.
- Write triggers and assertions.
- Create views and use them.
- Schema changing statements (DROP and ALTER).

SCHEMA REMINDER: EXAMPLES



NULL VALUES

Recall that NULL has multiple interpretations:

1. Unknown value. A person's date of birth is not known, so it is represented by NULL in the database. Another example: NULL for a person's home phone because it is not known whether or not the person has a home phone.
2. Unavailable or withheld value. A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. Not applicable attribute. An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

```
CREATE TABLE EVENT
( ID INT NOT NULL,
  Name VARCHAR(150),
  StartingDate DATE NOT NULL,
  Confirmed BOOLEAN );
```

3 VALUED LOGIC

SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE.

The values evaluate as follows:

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

NULL IN SQL QUERIES

- SQL allows queries that check whether an attribute value is NULL. It does not use = or <>, SQL uses the comparison operators IS or IS NOT.
- SQL considers each NULL value as being distinct from every other NULL value.

Example: Retrieve the names of all employees who do not have supervisors.

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE Super_ssn IS NULL;
```

is not Null

What if I wanted to find employees who have supervisors?

NESTED QUERIES

Consider the following table:

Workshop

Title	Presenter	hours
Introduction to Robotics	Dr. Maram	20
Robot Operating System	Dr. Maram	25
Introduction to Python	Dr. Maram	16
Writing Research Papers	Dr. Asma	10
Research Methodologies	Dr. Asma	15

We want to find all workshops which offered by the same presenter as Robot Operating System? How can we do that?

We want to find all workshops which offered by the same presenter as Robot Operating System? How can we do that?

One way to do it is by writing TWO queries.

First: find out the name of the presenter which present **Robot Operating System.**

Second: we used the presenter name which we found in the first query.

Workshop

Title	Presenter	hours
Introduction to Robotics	Dr. Maram	20
Robot Operating System	Dr. Maram	25
Introduction to Python	Dr. Maram	16
Writing Research Papers	Dr. Asma	10
Research Methodologies	Dr. Asma	15

Q1:

```
SELECT Presenter
FROM Workshop
WHERE Title = 'Robot Operating System';
```

Result: Dr. Maram

Q2:

```
SELECT *
FROM Workshop
WHERE Presenter = 'Dr. Maram';
```

Result:

Title	Presenter	hours
Introduction to Robotics	Dr. Maram	20
Robot Operating System	Dr. Maram	25
Introduction to Python	Dr. Maram	16

We want to find all workshops which offered by the same presenter as Robot Operating System? How can we do that?

Another way to do it in ONE step (query) with **subqueries** or **Nested queries**.

In Q2 instead of writing the result of Q1 explicitly, we will write Q1.

Workshop

Title	Presenter	hours
Introduction to Robotics	Dr. Maram	20
Robot Operating System	Dr. Maram	25
Introduction to Python	Dr. Maram	16
Writing Research Papers	Dr. Asma	10
Research Methodologies	Dr. Asma	15

Q2:

```
SELECT *
FROM Workshop
WHERE Presenter = 'Dr. Maram';
```

(Q1)

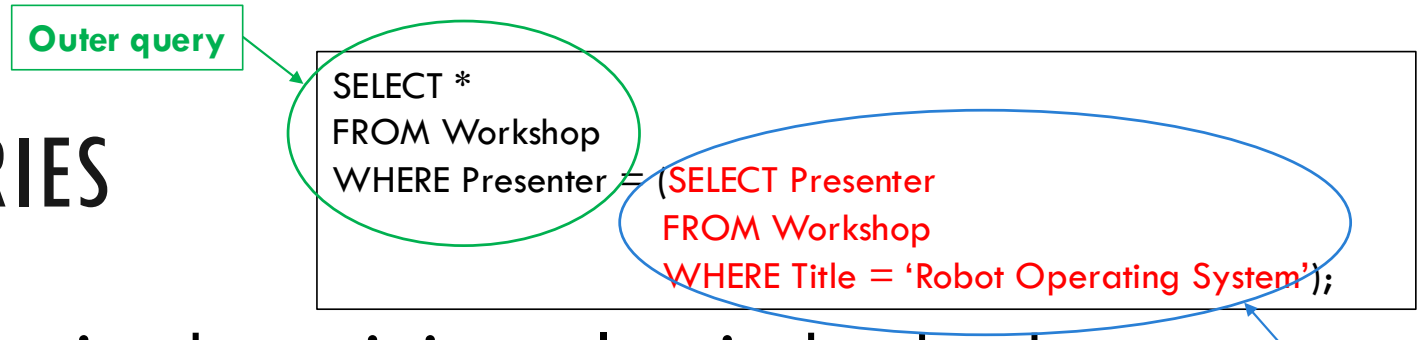
Q1:

```
SELECT Presenter
FROM Workshop
WHERE Title = 'Robot Operating System';
```

Nested query

```
SELECT *
FROM Workshop
WHERE Presenter = (SELECT Presenter
                   FROM Workshop
                   WHERE Title = 'Robot Operating System');
```

NESTED QUERIES



Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within another SQL query.

That other query is called the outer query. These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed.

NESTED QUERIES

Example: Find Essns of all employees who work the same (project, hours) combination that employee 'John Smith' (whose Ssn = '123456789') works on.

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE (Pno, Hours) IN ( SELECT Pno, Hours
                        FROM WORKS_ON
                        WHERE Essn = '123456789' );
```

IN operator compares a value v with a set (or multiset) of values V, IN is TRUE if v is one of the elements in V.

COMPARISON OPERATORS FOR NESTED QUERIES

- The = ANY (SOME) operator returns TRUE if the value v is equal to some value in the set V (equivalent to IN).
- The keyword ALL refers to all elements in the set V.
- Operators: >, >=, <, <=, and <> can be combined with ANY and ALL.

Example: (v > ALL V) returns TRUE if v is greater than all the values in V.

Example: Find names of employees with salary greater than all employees in Department 5.

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ALL ( SELECT Salary
                     FROM EMPLOYEE
                     WHERE Dno = 5 );
```

Any ← أكبر من أي واحد من الجدول

NESTED QUERY EXERCISES

Exercises: Use sub-queries with IN, NOT IN, op ANY, op ALL

Query 1: Find the name and the age of the youngest sailor.

Query 2: Find the names and ratings of sailors whose rating is better than dustin.

```
select names , rating
From sailors
where rating > Any (select rating from sailors
                    where name = 'dustin')
```

NESTED QUERY EXERCISES

select name, age
From sailors
where age <= ALL(Select age
From sailors

Query 1: Find the name and the age of the youngest sailor.

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age <= ALL ( SELECT age
                     FROM Sailors )
```

NESTED QUERY EXERCISES

Query 2: Find the names and ratings of sailors whose rating is better than dustin.

```
SELECT S.sname, S.rating
FROM Sailors S
WHERE S.rating > ANY ( SELECT S2.rating
                       FROM Sailors S2
                       WHERE S2.sname = 'dustin')
```

SQL EXPLICIT SETS

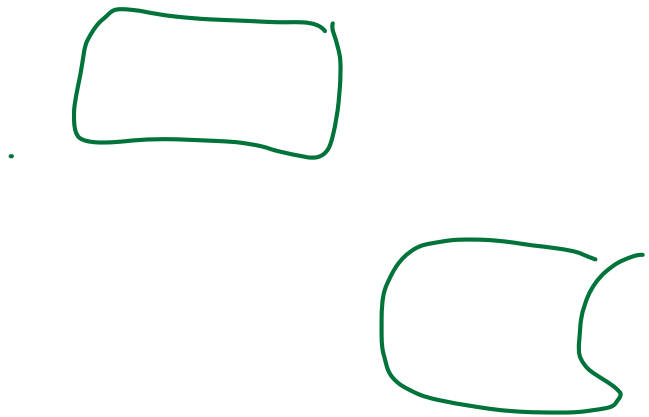
Sets do not have to be the results of sub-queries. They can be explicitly stated.

Example: Find employees who work on projects 1, 2, and 3

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE Pno IN (1, 2, 3);
```

CORRELATED NESTED QUERIES

Correlated queries: When the WHERE clause of a **nested query** references some attribute of a relation declared in the **outer query**, the two queries are said to be correlated.



CORRELATED NESTED QUERIES

country

id	name	population
1	Singapore	5700000
2	Spain	46900000
3	Peru	32000000

city

id	name	population	country_id
1	Singapore	5700000	1
2	Madrid	6600000	2
3	Barcelona	1600000	2
4	Valencia	800000	2
5	Lima	8300000	3

```
SELECT name
FROM country
WHERE id = (
      SELECT country_id
      FROM city
      WHERE city.population = country.population
      AND country.id = city.country_id
);
```

Correlated subqueries are subqueries which refer to the main, outer query and cannot be run as independent queries without the main query.

CORRELATED NESTED QUERIES

Example:

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE E.Ssn IN ( SELECT D.Essn
                 FROM DEPENDENT AS D
                 WHERE E.Fname = D.Dependent_name
                 AND E.Sex = D.Sex );
```

CORRELATED NESTED QUERY EVALUATION

Think about it like a loop:

For each employee, check the qualification by computing the subquery!

if the Ssn value of the EMPLOYEE tuple is in the result of the nested query, then select that EMPLOYEE tuple

CORRELATED NESTED QUERIES

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can always be expressed as a single block query.

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E, DEPENDENT AS D
WHERE E.Ssn = D.Essn
AND E.Sex = D.Sex
AND E.Fname = D.Dependent_name;
```

EXISTS AND UNIQUE

- EXISTS returns TRUE if the nested query result contains at least one tuple, or FALSE if the nested query result contains no tuples.
- UNIQUE returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE.
- EXISTS is used more often and is the focus here.

Rewriting the same dependent query with EXISTS:

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE EXISTS ( SELECT *
                FROM DEPENDENT AS D
                WHERE E.Ssn = D.Essn AND E.Sex = D.Sex
                AND E.Fname = D.Dependent_name);
```

NOT EXISTS

Example: Find employees who have no dependents

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE NOT EXISTS ( SELECT *
                   FROM DEPENDENT
                   WHERE Ssn = Essn );
```

EXERCISES

Exercise 1: Write nested sub-queries to determine the following:

a) Find names of sailors who have reserved a boat?

b) Find names of sailors who have never reserved a boat?

c) Can you express a with a join? How about b?

Exercise 2: Can you write the query to find the youngest sailor as a correlated nested query? (Hint: use NOT EXISTS, compare age with every other sailor (loop logic))

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

MORE JOINS

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

We looked briefly at joins in the last chapter. In this one, we will look at more types of joins.

The basic syntax in the last chapter was to specify tables in FROM and conditions in WHERE.

Look at this query:

```
SELECT Fname, Lname, Address
FROM (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)
WHERE Dname = 'Research';
```

We will look at other types of joins.

Course

id	name	Lecturer_id
1	Introduction to Database	1
2	Operating systems	2
3	Programming 1	2
4	Data structure	Null

Faculty

id	First_name	Last_name
1	Maram	Alajlan
2	Roohi	Jan
3	Khwlah	Alrajhi

Write a query to show each course with the first and last name of its lecturer.

```
SELECT name, First_name, Last_name
FROM Course JOIN Faculty ON Course. Lecturer_id = Faculty.id
```

Join condition

Result

name	First_name	Last_name
Introduction to Database	Maram	Alajlan
Operating systems	Roohi	Jan
Programming 1	Roohi	Jan

Course

id	name	Lecturer_id
1	Introduction to Database	1
2	Operating systems	2
3	Programming 1	2
4	Data structure	Null

Faculty

id	First_name	Last_name
1	Maram	Alajlan
2	Roohi	Jan
3	Khwlah	Alrajhi

Write a query to show each course with the first and last name of its lecturer.

```
SELECT Course.name, Faculty.First_name, Faculty.Last_name
FROM Course JOIN Faculty ON Course. Lecturer_id = Faculty.id
```

If two attributes share the same name, it is mandatory to specify table name, otherwise, it is optional.

Result

name	First_name	Last_name
Introduction to Database	Maram	Alajlan
Operating systems	Roohi	Jan
Programming 1	Roohi	Jan

Why **Data structure** does not appear in the result? Is it possible to show a tuple without a match from the other table?

Course

id	name	Lecturer_id
1	Introduction to Database	1
2	Operating systems	2
3	Programming 1	2
4	Data structure	Null

Faculty

id	First_name	Last_name
1	Maram	Alajlan
2	Roohi	Jan
3	Khwlah	Alrajhi

Write a query to show each course with the first and last name of its lecturer. All courses must appear in the result even if they do not have lecturers.

```
SELECT name, First_name, Last_name
FROM Course LEFT OUTER JOIN Faculty ON Course. Lecturer_id = Faculty.id
```

Result

name	First_name	Last_name
Introduction to Database	Maram	Alajlan
Operating systems	Roohi	Jan
Programming 1	Roohi	Jan
Data structure	Null	Null

Course

id	name	Lecturer_id
1	Introduction to Database	1
2	Operating systems	2
3	Programming 1	2
4	Data structure	Null

Faculty

id	First_name	Last_name
1	Maram	Alajlan
2	Roohi	Jan
3	Khwlah	Alrajhi

Write a query to show each course with the first and last name of its lecturer. All lecturer must appear in the result even if they do not have courses.

```
SELECT name, First_name, Last_name
FROM Course RIGHT OUTER JOIN Faculty ON Course. Lecturer_id = Faculty.id
```

Result

name	First_name	Last_name
Introduction to Database	Maram	Alajlan
Operating systems	Roohi	Jan
Programming 1	Roohi	Jan
Null	Khwlah	Alrajhi

Course

id	name	Lecturer_id
1	Introduction to Database	1
2	Operating systems	2
3	Programming 1	2
4	Data structure	Null

Faculty

id	First_name	Last_name
1	Maram	Alajlan
2	Roohi	Jan
3	Khwlah	Alrajhi

Write a query to show each course with the first and last name of its lecturer. All courses and lecturers must appear in the result even if they do not have matches.

```
SELECT name, First_name, Last_name
FROM Course FULL OUTER JOIN Faculty ON Course. Lecturer_id = Faculty.id
```

Result

name	First_name	Last_name
Introduction to Database	Maram	Alajlan
Operating systems	Roohi	Jan
Programming 1	Roohi	Jan
Data structure	Null	Null
Null	Khwlah	Alrajhi

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

NATURAL JOIN

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

Natural join is a similar join with small differences:

1. No join condition is specified.
2. Implicit condition for each pair of attributes with the same name.
3. Each such pair of attributes is included only once in the result.

```
SELECT Fname, Lname, Address
FROM (EMPLOYEE NATURAL JOIN
(DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))
WHERE Dname = 'Research';
```

What is the join condition here?

INNER JOIN

The **default** type of join is called an inner join, where a tuple is included in the result only if a matching tuple exists in the other relation.

These are the joins we have looked at so far.

```
SELECT Fname, Lname, Address
FROM (EMPLOYEE INNER JOIN DEPARTMENT ON Dno = Dnumber)
WHERE Dname = 'Research';
```

Exercise: Try INNER JOIN, NATURAL JOIN, and JOIN on sailors and reserves. What is different?

```
select name, address
from employee e, department d
where e.dno = d.dno
and d.dname = 'Research';
```

OUTER JOIN

```
SELECT E.Lname AS Employee_name,
S.Lname AS Supervisor_name
FROM (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S
ON E.Super_ssn = S.Ssn);
```

Types of outer join:

- LEFT OUTER JOIN: every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table.
- RIGHT OUTER JOIN: every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table.
- FULL OUTER JOIN.

In all three options, the keyword OUTER may be omitted.

LIST OF IDENTICAL JOINS WITH DIFFERENT NAMES

There is redundancy in the syntax of joins. This means that the pairs of queries in each line below are identical:

LEFT JOIN and LEFT OUTER JOIN

RIGHT JOIN and RIGHT OUTER JOIN

FULL JOIN and FULL OUTER JOIN

INNER JOIN and JOIN.

OTHER TYPES OF JOIN

If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN).

The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation. This one must be used carefully.

MULTIWAY JOIN

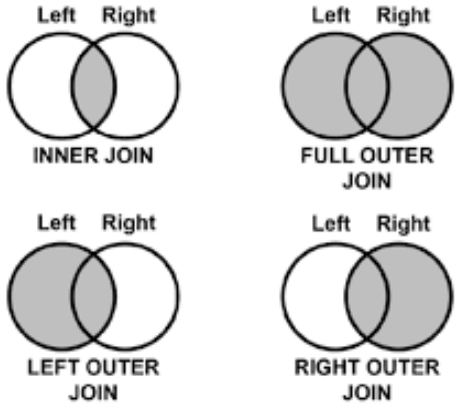
It is also possible to nest join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a multiway join.

Example:

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM ((PROJECT JOIN DEPARTMENT ON Dnum = Dnumber)
JOIN EMPLOYEE ON Mgr_ssn = Ssn)
WHERE Plocation = 'Stafford';
```

SUMMARY

The following figure summarizes joins where left is a relation and right is another relation.



JOIN EXERCISES

Exercise 1: Retrieve all sailors who have made reservations and sailors who haven't.

This query will return all sailors, including those who have not made any reservations (in which case, r.bid will be NULL for those sailors).

Exercise 2: Retrieve all boats that have been reserved and boats that haven't.

This query will return all boats, including those that have never been reserved (in which case, r.sid will be NULL for those boats).

Exercise 3: Retrieve all sailors and all boats, regardless of reservations.

This query will return all sailors and all boats, including sailors who have not made any reservations and boats that have not been reserved.

AGGREGATION

Aggregation functions:

COUNT ([DISTINCT] A): The number of (unique) values in the A column.

SUM ([DISTINCT] A): The sum of all (unique) values in the A column.

AVG ([DISTINCT] A): The average of all (unique) values in the A column.

MAX (A): The maximum value in the A column.

MIN (A): The minimum value in the A column.

AGGREGATION

Find the average of the GPA of all students.

```
SELECT AVG(gpa)
FROM Student;
```

Result

AVG(gpa)
3.242857

Student

student_name	department	gpa
Sarah	CS	3.5
Rahaf	CS	3.9
Alanoud	IS	3.3
Fahdah	CS	2.5
Aseel	SE	2.1
Norah	SE	3.9
Reem	IS	3.5
Sarah	CS	3.5

AGGREGATION

Find the highest and lowest GPA.

```
SELECT MAX(gpa), MIN(gpa)
FROM Student;
```

Result

MAX(gpa)	MIN(gpa)
3.9	2.1

Student

student_name	department	gpa
Sarah	CS	3.5
Rahaf	CS	3.9
Alanoud	IS	3.3
Fahdah	CS	2.5
Aseel	SE	2.1
Norah	SE	3.9
Reem	IS	3.5
Sarah	CS	3.5

AGGREGATION

Find the number of the departments.

```
SELECT COUNT(DISTINCT department)
FROM Student;
```

Result

COUNT(DISTINCT department)
3

Student

student_name	department	gpa
Sarah	CS	3.5
Rahaf	CS	3.9
Alanoud	IS	3.3
Fahdah	CS	2.5
Aseel	SE	2.1
Norah	SE	3.9
Reem	IS	3.5
Sarah	CS	3.5

AGGREGATION

Find the number of the students (total number of the records).

```
SELECT COUNT(*)
FROM Student;
```

Result

COUNT(*)
8

Student

student_name	department	gpa
Sarah	CS	3.5
Rahaf	CS	3.9
Alanoud	IS	3.3
Fahdah	CS	2.5
Aseel	SE	2.1
Norah	SE	3.9
Reem	IS	3.5
Sarah	CS	3.5

AGGREGATION

Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
SELECT SUM(Salary), MAX(Salary), MIN(Salary), AVG(Salary)
FROM EMPLOYEE;
```

Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
SELECT SUM(Salary), MAX(Salary), MIN(Salary), AVG(Salary)
FROM (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)
WHERE Dname = 'Research';
```

select .....
From department d , employee e
Where d.dno = e.dno and

d.dname = 'Research';

AGGREGATION

Find the total number of employees.

```
SELECT COUNT (*)
FROM EMPLOYEE;
```

Find the total number of employees in the research department.

```
SELECT COUNT (*)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER AND DNAME = 'Research';
```

AGGREGATES IN NESTED QUERIES

We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query.

For example, to retrieve the names of all employees who have two or more dependents:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE ( SELECT COUNT (*)
        FROM DEPENDENT
        WHERE Ssn = Essn ) > = 2;
```

EXERCISES

Count the number of reservations made by ‘rusty’.

Count the number of sailors.

Calculate the average, max, and min of sailor ages.

GROUP BY CLAUSE

In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values.

Examples:

- The average salary of employees in each department or
- The number of employees who work on each project

To do this, we partition the data using a GROUP BY clause.

We apply functions (e.g. count, avg) to each group.

Example: For each department, retrieve department number, number of employees, and average salary.

```
SELECT Dno, COUNT (*), AVG (Salary)
FROM EMPLOYEE
GROUP BY Dno;
```

GROUP BY CLAUSE

Find the average of the GPA in each department.

```
SELECT department, AVG(gpa)
FROM Student;
GROUP BY department
```

Result

department	AVG(gpa)
CS	3.3
IS	3.4
SE	3

Student

student_name	department	gpa
Sarah	CS	3.5
Rahaf	CS	3.9
Alanoud	IS	3.3
Fahdah	CS	2.5
Aseel	SE	2.1
Norah	SE	3.9
Reem	IS	3.5
Sarah	CS	3.5

GROUP BY CLAUSE

Aggregation queries have specific restrictions when writing queries. You can only SELECT attributes that are in the GROUP BY clause. You should also make sure your data contains feasible and sensible groupings.

Grouping by multiple attributes will create sub-groups for each attribute.

In class examples:

Q1: SELECT count(*), colour

FROM boats

GROUP BY colour

Q2: SELECT major, standing, count(*), avg(age)

FROM student

GROUP BY major, standing

NULLS IN GROUPS

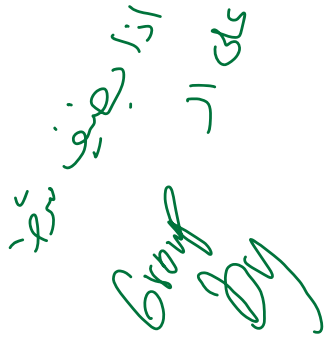
If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a NULL value in the grouping attribute. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples.

Example: For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
SELECT Pnumber, Pname, COUNT (*)
FROM PROJECT, WORKS_ON
WHERE Pnumber = Pno
GROUP BY Pnumber, Pname;
```

Example shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied after the joining of the two relations in the WHERE clause.

HAVING CLAUSE



Sometimes we want to retrieve the values of these functions only for groups that satisfy certain conditions. For example, suppose that we want to modify the previous query so that only projects with more than two employees appear in the result.

```
SELECT Pnumber, Pname, COUNT(*)
FROM PROJECT, WORKS_ON
WHERE Pnumber = Pno
GROUP BY Pnumber, Pname
HAVING COUNT(*) > 2;
```

WHERE applies to tuples, HAVING applies to groups.

SUMMARY OF SQL (NOW THAT WE KNOW MORE)

SELECT <attribute and function list>

FROM <table list>

[WHERE <condition>]

[GROUP BY <grouping attribute(s)>]

[HAVING <group condition>]

[ORDER BY <attribute list>];

VIEWS

A view in a relational database is essentially a virtual table representing a subset of data from one or more tables.

Views are not physically stored data; they are queries that dynamically generate data when accessed.

Views can be used for various purposes, such as simplifying complex queries, ensuring **data security** by restricting access to specific data, or presenting data in a particular format.

VIEWS

```
Employees(EmployeeID, Name, Position, salary, hiring date)
```

A view can be created to display only specific columns from a table.

```
SELECT EmployeeID, Name, Position
FROM Employees;
```

```
SELECT *
FROM EmployeeView
```

What is the output?

A view can join and present data from multiple tables.

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

```
SELECT *
FROM DepartmentEmployeeView
```

MATERIALIZED VS NON-MATERIALIZED VIEWS

Non-Materialized Views:

These are standard views that do not store their data physically.

They run the underlying SQL query every time they are accessed.

Suitable for data that changes frequently, ensuring that the view always presents the most current data.

Inefficient for views defined via complex queries that are time-consuming to execute.

Materialized Views:

These views store the query result as a physical table.

They need to be refreshed periodically to reflect changes in the underlying data.

Useful for complex queries on large tables where performance is a concern, as they can significantly reduce query time.

VIEW FUNCTIONS

View Update

Updating a view means updating the underlying base tables. However, not all views are updatable. For a view to be updatable, it generally needs to map directly to data in a base table without complex transformations, such as joins or aggregations. Some databases have specific restrictions or capabilities regarding view updates.

Views as Access Control Mechanism

Views can serve as a powerful access control mechanism. They allow database administrators to expose only certain parts of the database schema to certain users, enhancing both security and data integrity. For instance, a view can be created to show only non-sensitive columns from a table to a group of users, effectively hiding sensitive data from them.

SCHEMA CHANGES (DROP AND ALTER)

DROP Function in a DBMS:

The DROP function in a Database Management System (DBMS) is used to remove database objects such as tables, views, indexes, or even the entire database itself from the system. When an object is dropped, it is permanently deleted and cannot be recovered.

Examples:

1. **Dropping a Table:** DROP TABLE tableName; - This command deletes the specified table and all its data permanently.
2. **Dropping a Database:** DROP DATABASE databaseName; - This command deletes the entire database, including all its tables and data.

SCHEMA CHANGES (DROP AND ALTER)

ALTER Function in a DBMS:

The ALTER function is used to modify the structure of an existing database object, like a table. It can be used for a variety of changes such as adding, deleting, or modifying columns in a table, changing data types of columns, or modifying constraints.

Examples:

1. **Adding an attribute:** ALTER TABLE tableName ADD columnName dataType;
2. **Modifying an attribute:** ALTER TABLE tableName MODIFY COLUMN columnName newDataType;
3. **Dropping an attribute:** ALTER TABLE tableName DROP COLUMN columnName;

In summary, the DROP function is used for deleting database objects, while the ALTER function is used for modifying the structure of existing objects. Both are critical for database management, allowing for the evolution and maintenance of the database schema over time.

EXECUTING QUERIES IN CODE

We will look at 2 APIs: JDBC for Java and DB-API for Python.

JDBC (Java Database Connectivity) is an application programming interface (API) for the Java programming language that defines how a client may access a database.

DB-API (Database Application Programming Interface). The Python DB-API provides a standard interface for Python applications to interact with relational databases. Each one needs its own DB-API compliant driver.

Each API allows Java/Python applications to connect to a database, execute SQL queries, retrieve results, handle transactions, and deal with exceptions.

EXAMPLE (JAVA)

Assuming JDBC for MySQL driver and MySQL server running locally. This code connects to the database, and executes a query.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class MySQLAccess {
    public static void main(String[] args) {
        // Database credentials
        String url = "jdbc:mysql://localhost:3306/myDatabase"; // Replace 'myDatabase' with your database name
        String user = "username"; // Replace with your MySQL username
        String password = "password"; // Replace with your MySQL password

        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;

        try {
            // Register JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Open a connection
            System.out.println("Connecting to database...");
            connection = DriverManager.getConnection(url, user, password);

            // Execute a query
            System.out.println("Creating statement...");
            statement = connection.createStatement();
            String sql = "SELECT id, name FROM employees"; // Replace with your SQL query
            resultSet = statement.executeQuery(sql);

            // Extract data from result set
            while (resultSet.next()) {
                // Retrieve by column name
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");

                // Display values
                System.out.print("ID: " + id);
                System.out.println(", Name: " + name);
            }
        } catch (SQLException se) {
            // Handle errors for JDBC
            se.printStackTrace();
        } catch (Exception e) {
            // Handle errors for Class.forName
            e.printStackTrace();
        } finally {
            // Finally block used to close resources
            try {
                if (resultSet != null) resultSet.close();
                if (statement != null) statement.close();
                if (connection != null) connection.close();
            } catch (SQLException se) {
                se.printStackTrace();
            }
        }

        System.out.println("Goodbye!");
    }
}
```

PROJECT CODE

The expectation for the project is that you execute your data retrieval (SELECT)and data manipulation (UPDATE, DELETE or INSERT) queries in code and write some basic application logic. Example: Allow users to register, book sessions, inspect their data, and make changes.

If you fully implement a graphical user interface, you get max +1 to a quiz grade.

If you cannot implement a GUI, you can show me some advanced function related to :

- Database security
- Triggers or assertions
- Deployment

SUMMARY

We have looked at more advanced SQL code: Nested queries, joins, and aggregation.

Views are another way to perform multiple operations and allow for different types of authorized access.

We have also seen schema altering operations: Drop and alter.

With that, we are done with relational content for this course. Next NoSQL!