INTRODUCTION TO DATABASE

SQL and NoSQL Database

INTRO TO DB

- A database is an organized collection of structured information, or data, typically stored electronically in a computer system.
- A database is usually controlled by a database management system (DBMS).
- The data and the DBMS, along with the applications associated with them, are referred to as a database system, simply said database.

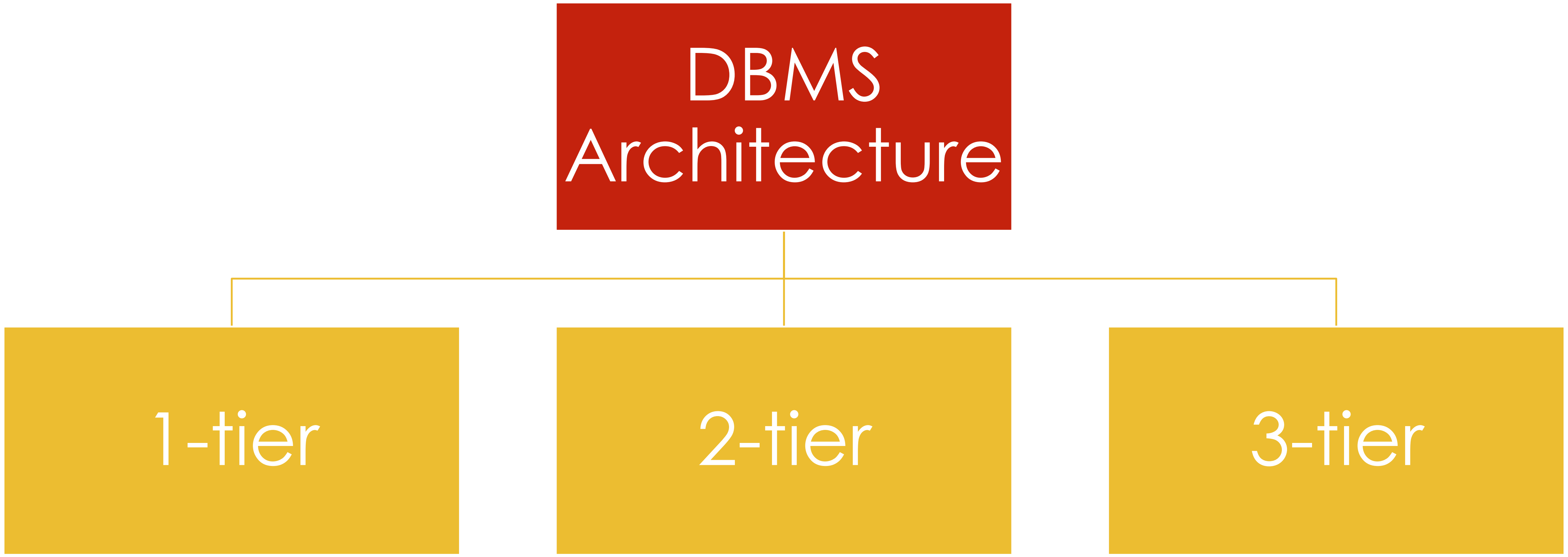
IMPORTANCE OF DB

- Data Management:
- Data Integrity
- Data Security
- Data Sharing
- Backup and Recovery
- Scalability
- Performance
- Automation

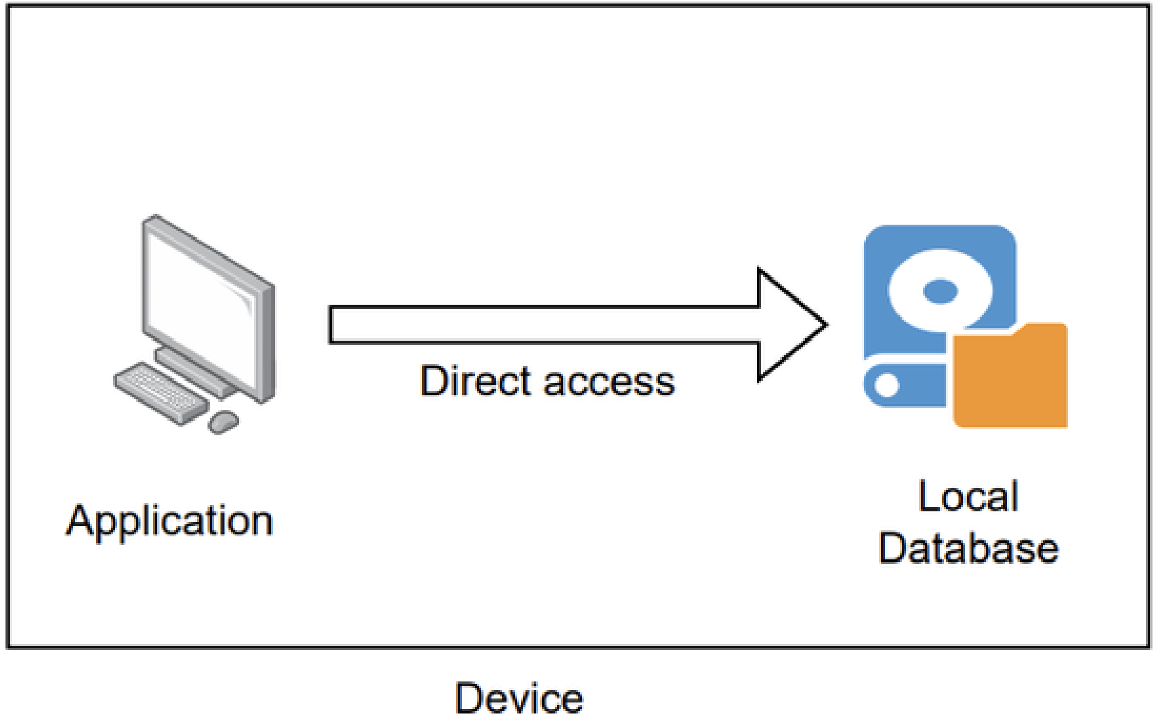
DB TYPES

- Relational
  - MYSQL
  - PostgreSQL
  - Oracle
  - MS SQL Server
- NO SQL
  - MongoDB
  - Cassandra
  - CouchDB
- In-Memory
  - MemCached
  - Redis
- Graph
  - Neo4j
  - OrientDB
- Time-Series
  - db4O
  - ObjectDB
- NewSQL
  - GoogleSpanner
  - VoltDB

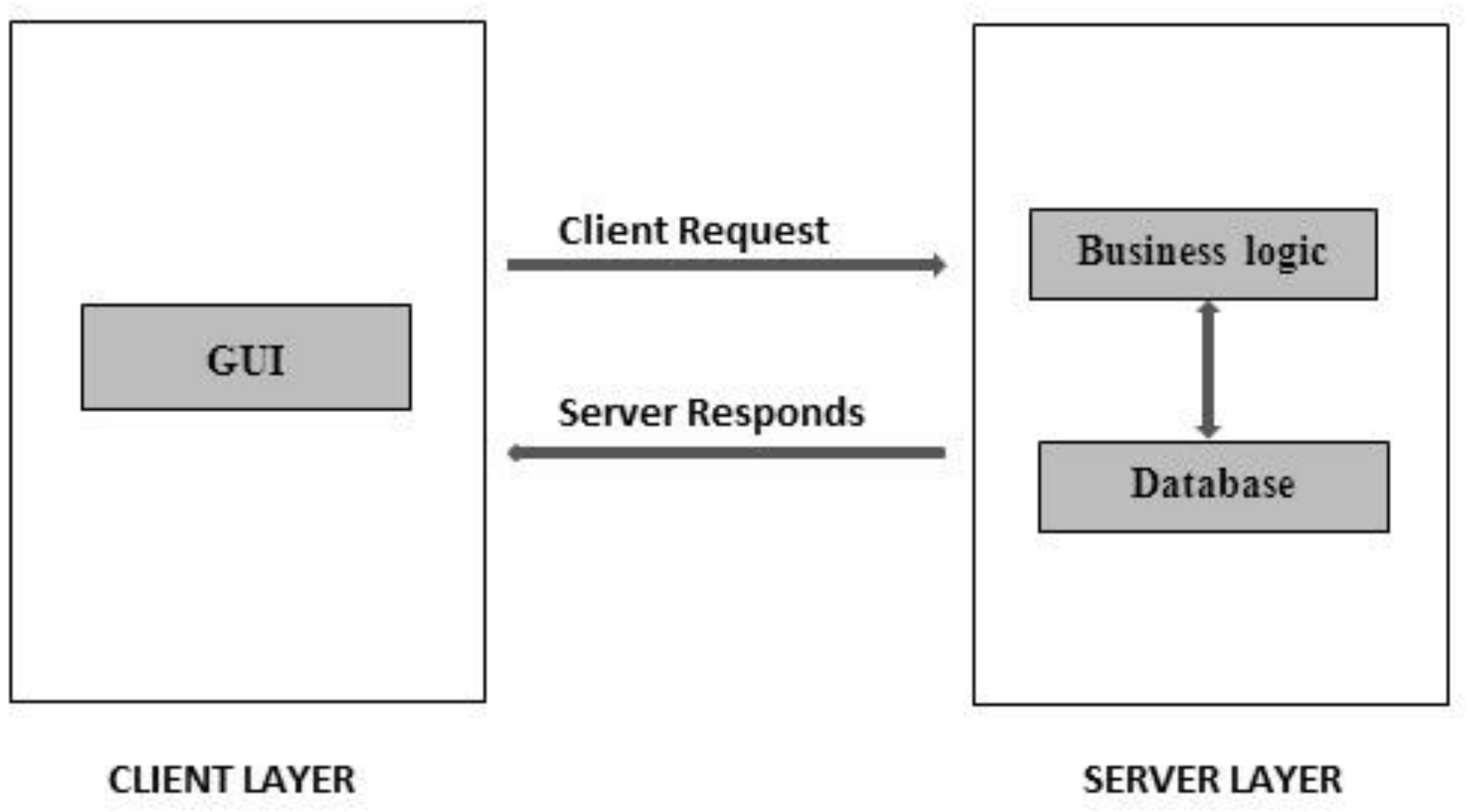
DBMS ARCHITECTURE



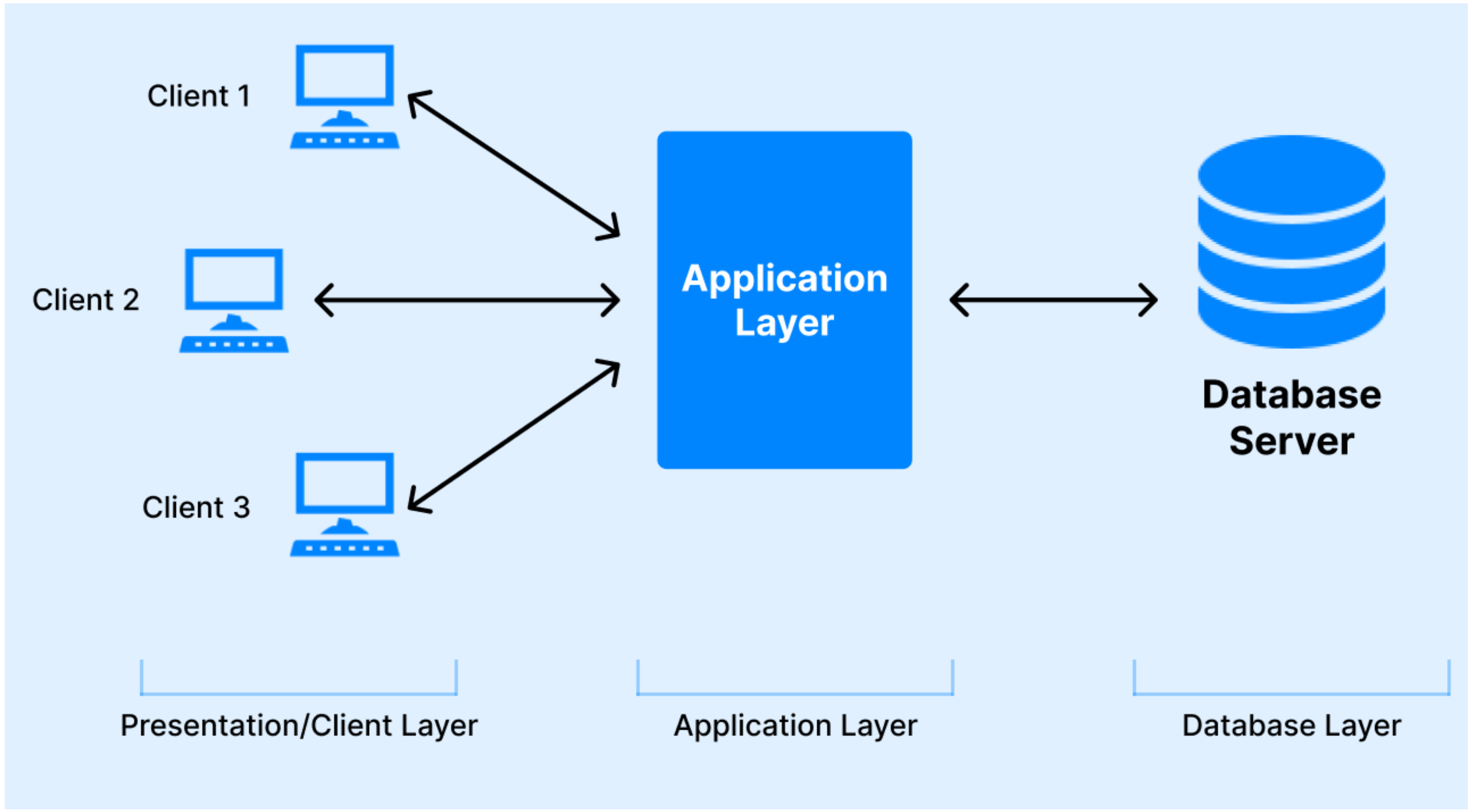
1 TIER ARCHITECTURE



2-TIER ARCHITECTURE



3-TIER ARCHITECTURE



WHAT ARE DATA MODELS?

- Data models are abstract representations of the structures and relationships within a database.
- They define how data is organized, stored, and manipulated, providing a blueprint for designing and creating databases.
- Data models help in understanding the nature of the data and the relationships between different data elements, which is crucial for ensuring data integrity, consistency, and efficiency in database management.

TYPES OF MODELS

- Hierarchical Data Model
- Network Data Model
- Relational Data Model
- Entity-Relationship Model (ER Model)
- Object-Oriented Data Model

- Document Data Model
- Columnar Data Model
- Graph Data Model
- Key-Value Data Model
- Time-Series Data Model

ENTITY-RELATIONSHIP (ER) MODEL

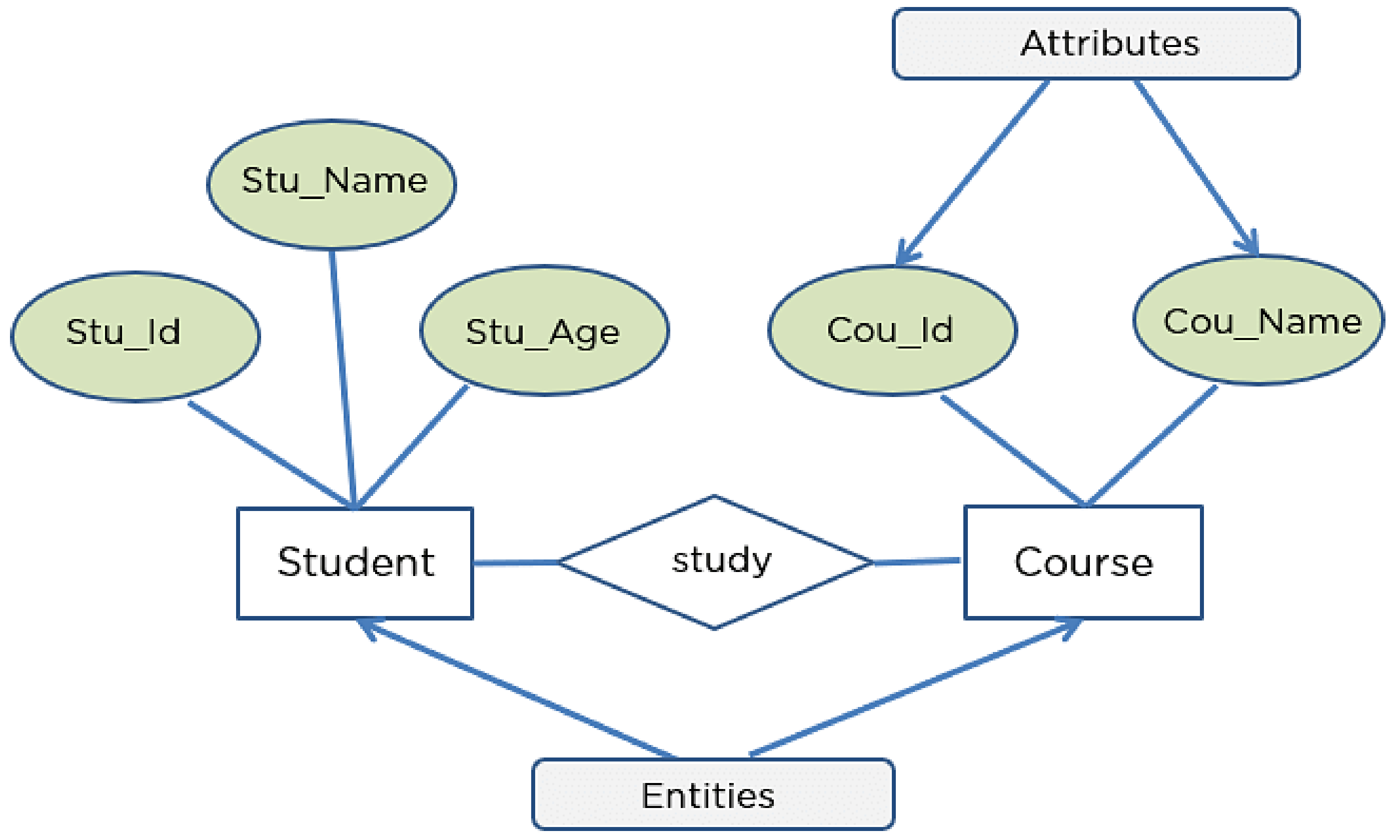
- The Entity-Relationship (ER) model is a conceptual framework used to describe and design the structure of a database.
- It provides a high-level view of data, focusing on the entities (objects) and the relationships between them.
- The ER model is typically represented by an ER diagram, which visually depicts the entities, their attributes, and the relationships between them.

COMPONENTS OF AN ER MODEL

- **Entities**:
  - An entity is a real-world object or concept that can have data stored about it in a database.
  - Examples: Customer, Product, Employee, Order.
  - Representation: represented by rectangles.
- **Attributes**:
  - Attributes are properties or characteristics of an entity.
  - Examples: For a Customer entity, attributes might include CustomerID, Name, Email, Phone.
  - Types of Attributes:
    - Simple: Cannot be divided further (e.g., Name).
    - Composite: Can be divided into smaller parts (e.g., Address can be divided into Street, City, Zip Code).
    - Derived: Calculated from other attributes (e.g., Age from Date of Birth).
    - Multi-valued: Can have multiple values (e.g., Phone Numbers).Representation:
  - Representation: represented by ovals connected to their entity.

- **Relationships**:
- Relationships describe how entities interact with each other.
- Examples: A Customer places an Order, an Employee works in a Department.
- **Types of Relationships:**
- One-to-One (1:1): Each employee is assigned one office
- One-to-Many :A customer can place multiple orders
- Many-to-Many: Students enroll in multiple courses, and courses have multiple students.
- Representation: represented by diamonds connecting entities

ER DIAGRAM


Attributes
Stu_Name
Stu_Id
Stu_Age
Cou_Id
Cou_Name
Student
study
Course
Entities

ACTIVITY: LIBRARY MANAGEMENT SYSTEM

- Entities and Attributes:
- Book
  - BookID (Primary Key)
  - Title
  - Author
- Member
  - MemberID (Primary Key)
  - Name
  - Email
- Loan
  - LoanLoanID (Primary Key)
  - LoanDate
  - ReturnDate
  - BookID (Foreign Key)
  - MemberID (Foreign Key)

- Relationships:
  - Member borrows Book:
  - One-to-Many relationship (A member can borrow multiple books, but each loan record is for one book and one member).

ONLINE STORE

Entities and Attributes:

Customer

- CustomerID (Primary Key)
- Name
- Email

Product

- ProductID (Primary Key)
- Name
- Price

Order

- OrderID (Primary Key)
- OrderDate
- CustomerID (Foreign Key)

OrderItem

- OrderItemID (Primary Key)
- Quantity
- OrderID (Foreign Key)
- ProductID (Foreign Key)

Relationships:

Customer places Order:

- One-to-Many relationship
- A customer can place multiple orders, but each order is placed by one customer

Order contains Product:

- Many-to-Many relationship resolved with OrderItem
- An order can contain multiple products, and a product can appear in multiple orders.

KEYS IN DATABASE

keys are crucial elements used to uniquely identify records, enforce data integrity, and establish relationships between tables.

4 Types of Keys:

- Primary Key
- Foreign Key
- Candidate Key
- Super Key

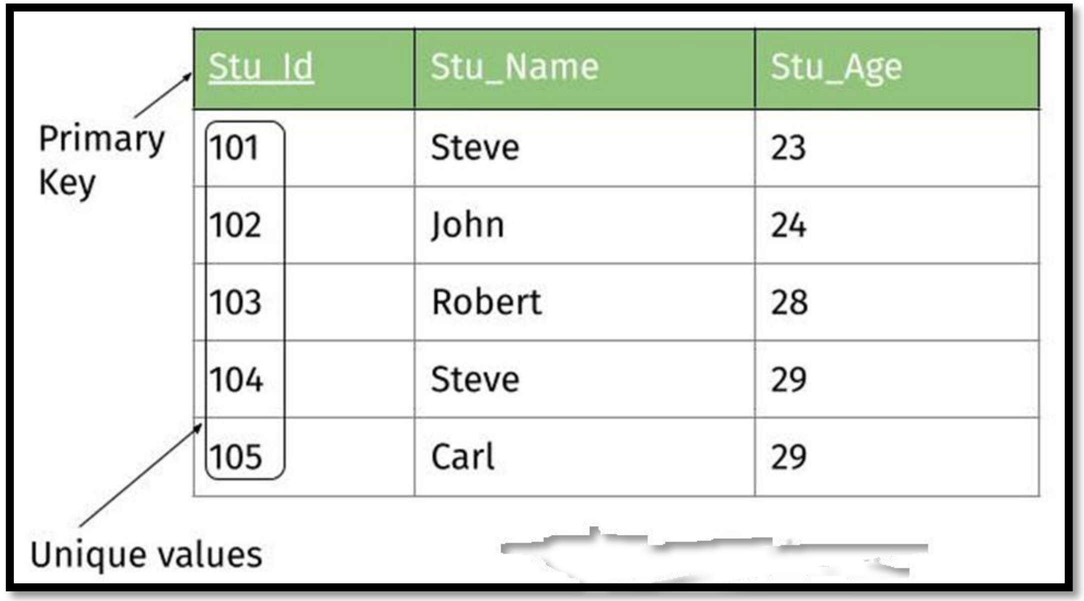
PRIMARY KEY

A primary key is a column (or a set of columns) in a table that uniquely identifies each row in that table.

Each value in the primary key column(s) must be unique.

Primary key columns cannot contain null values.

Each table can have only one primary key.



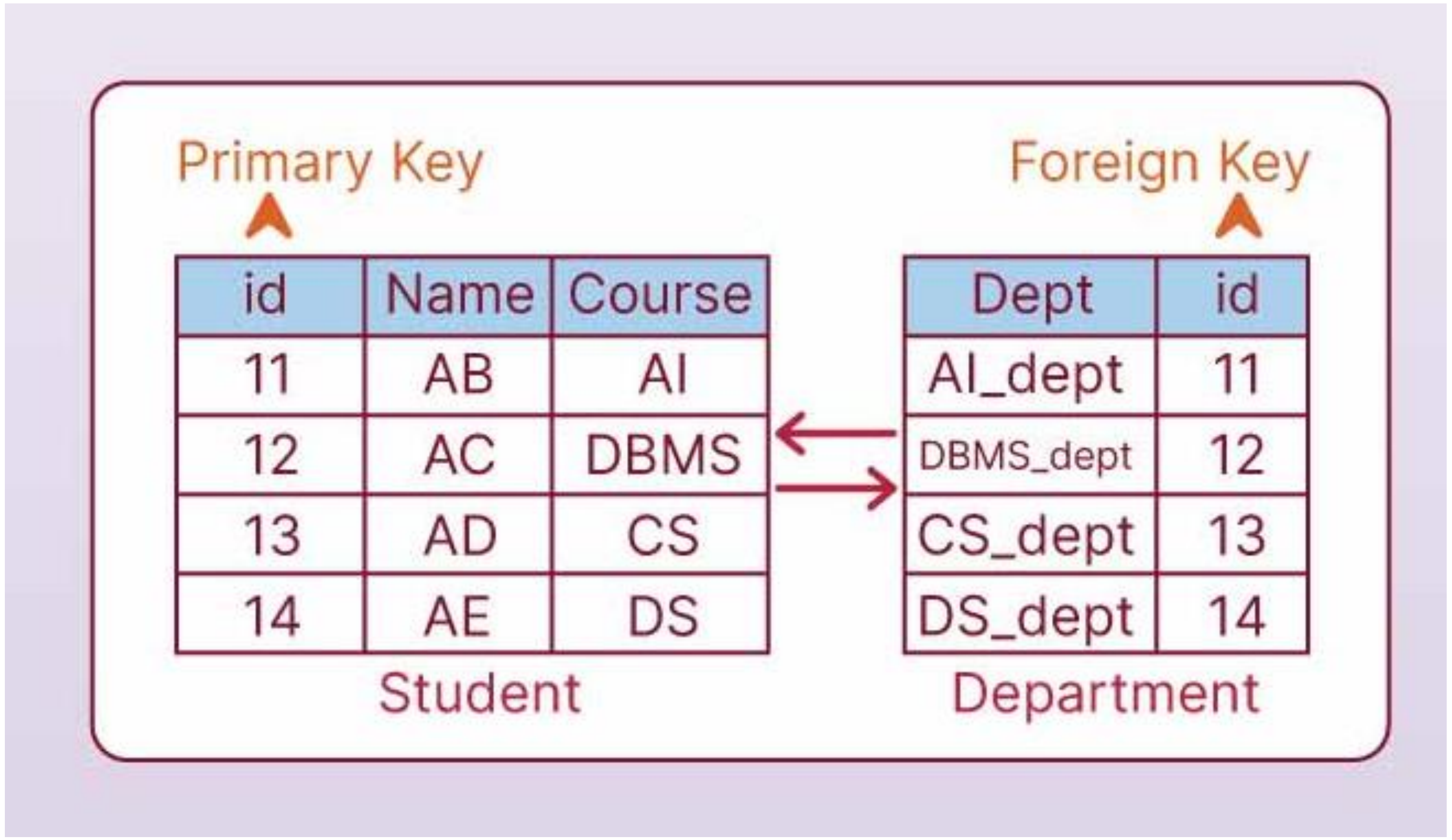
Primary Key
Unique values

Stu_Id	Stu_Name	Stu_Age
101	Steve	23
102	John	24
103	Robert	28
104	Steve	29
105	Carl	29

FOREIGN KEY

- A foreign key is a column (or a set of columns) in a table that creates a link between the data in two tables.
- It enforces referential integrity by ensuring that the value in the foreign key column(s) corresponds to a valid value in the referenced primary key column(s) of another
- It ensures that a value in the foreign key column must match a value in the primary key column of the referenced table.
- Relationships: Establishes relationships between tables (One-to-One, One-to-Many, Many-to-Many).

FOREIGN KEY

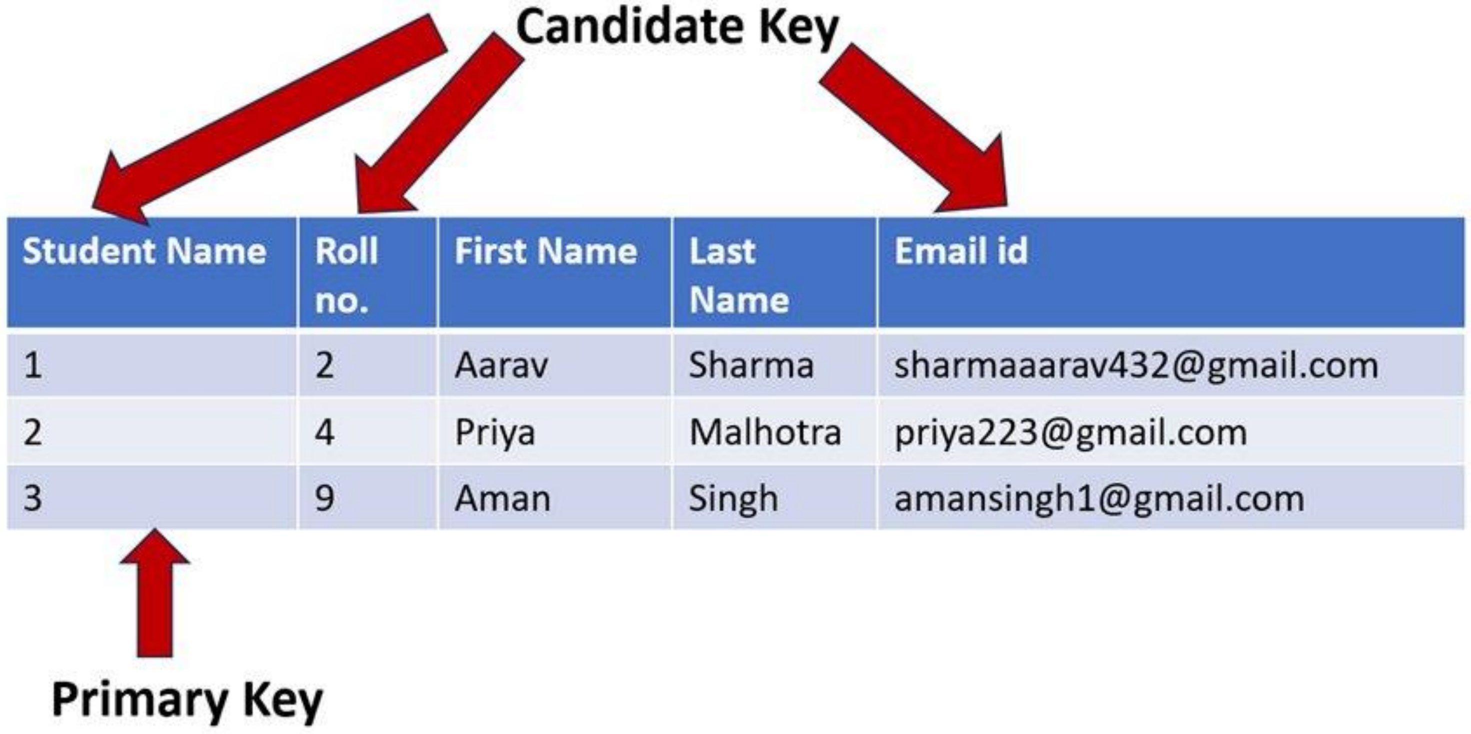


id	Name	Course
11	AB	AI
12	AC	DBMS
13	AD	CS
14	AE	DS

Dept	id
AI_dept	11
DBMS_dept	12
CS_dept	13
DS_dept	14

CANDIDATE KEY

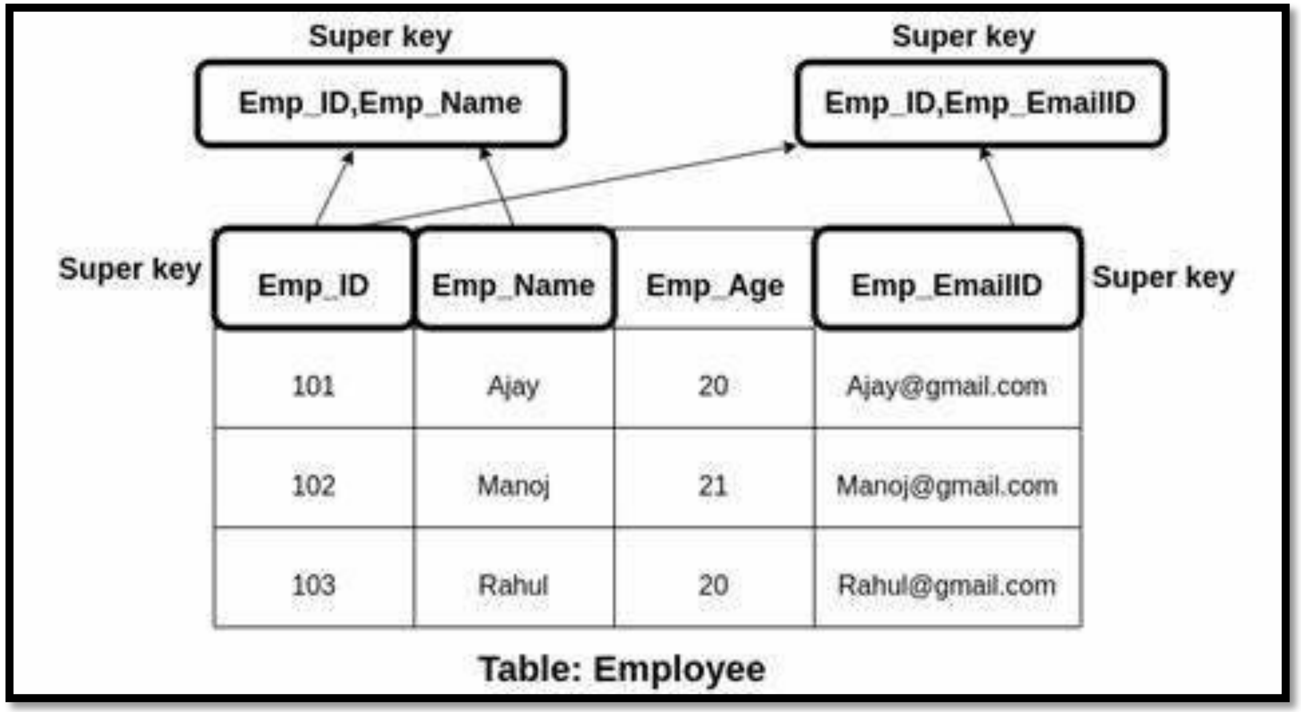
- A candidate key is a column (or a set of columns) that can uniquely identify any record in a table.
- It is a candidate for becoming the primary key.
- Each value in the candidate key column(s) must be unique.
- Candidate key columns cannot contain null values.
- A table can have multiple candidate keys.


Candidate Key
Primary Key

Student Name	Roll no.	First Name	Last Name	Email id
1	2	Aarav	Sharma	sharmaaarav432@gmail.com
2	4	Priya	Malhotra	priya223@gmail.com
3	9	Aman	Singh	amansingh1@gmail.com

SUPER KEY

- A super key is a column that uniquely identifies any record in a table.
- It is a superset of a candidate key.
- It ensures unique identification of each record.
- It can include additional attributes that are not necessary for unique identification.



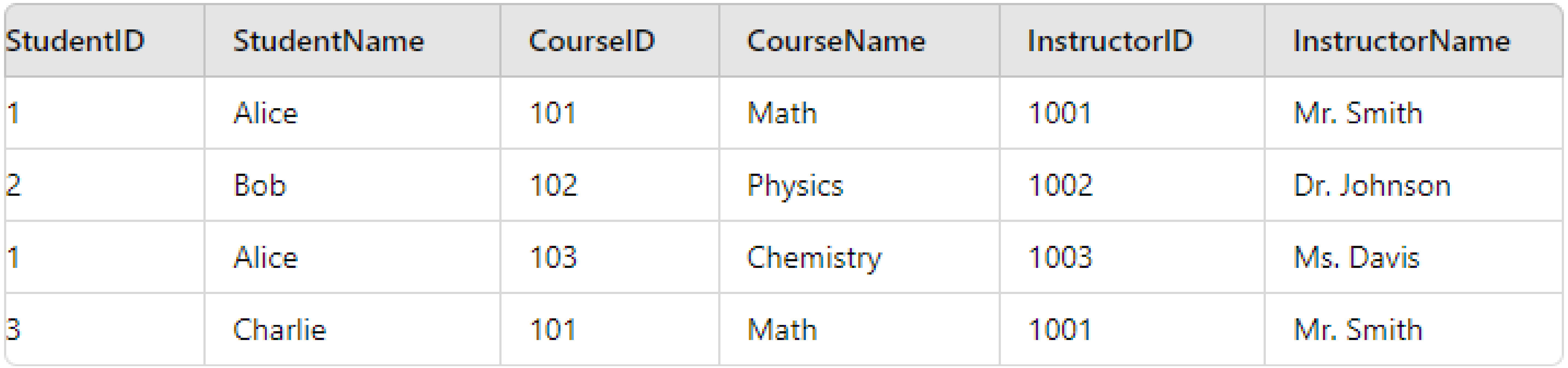
Emp_ID	Emp_Name	Emp_Age	Emp_EmailID
101	Ajay	20	Ajay@gmail.com
102	Manoj	21	Manoj@gmail.com
103	Rahul	20	Rahul@gmail.com

Table: Employee

NORMALIZATION

- Normalization is a process in database design that organizes columns and tables of a database to reduce data redundancy and improve data integrity.
- We have multiple Normal form available to normalize data
- 1 NF
  2 NF
- 3 NF
- BCNF
- 4 NF
- 5 NF

SCENARIO



StudentID	StudentName	CourseID	CourseName	InstructorID	InstructorName
1	Alice	101	Math	1001	Mr. Smith
2	Bob	102	Physics	1002	Dr. Johnson
1	Alice	103	Chemistry	1003	Ms. Davis
3	Charlie	101	Math	1001	Mr. Smith

1 NF

- 1NF requires that the table is free from repeating groups
- each cell contains atomic (indivisible) values.
- In our example, the table is already in 1NF because each cell contains atomic values and there are no repeating groups.
- To work with 2 NF the data must be in 1NF.

2 NF

- 2NF
  - Requires that the table is in 1NF
  - all non-key attributes are fully functional dependent on the primary key.
  - This usually involves removing partial dependencies.
- To achieve 2NF, we need to remove partial dependencies by splitting the table into two or more tables.
- We can create separate tables for Students, Courses, and Instructors.

StudentId	StudentName
1	Alice
2	Bob
3	Charlie

courseID	CourseName	InstructorID
101	Math	1001
102	Physics	1002
103	Chemistry	1003

SInstructorID	InstructorName
1	Mr. Smith
2	Dr. Johnson
3	Ms. Davis

CREATE ENROLMENTS TABLE

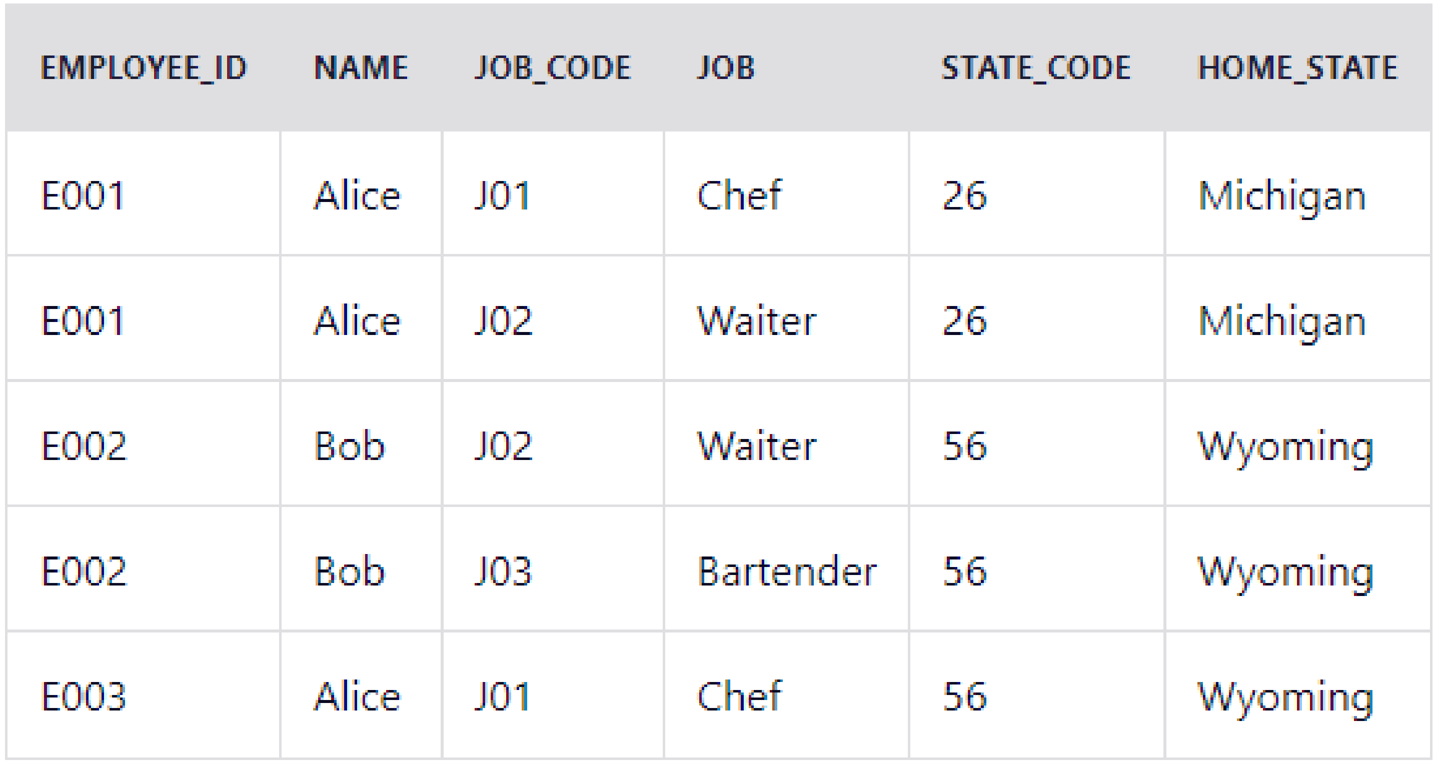
- This table links students to their courses, resolving the many-to-many relationship between Students and Courses.

StudentId	CourseID
1	101
1	103
2	102
3	101

THIRD NORMAL FORM (3NF)

- 3NF requires that the table is in 2NF and all the attributes are only dependent on the primary key, meaning there are no transitive dependencies.
- In our current structure, we need to ensure that all non-key attributes are dependent only on the primary key and not on other non-key attributes.
- In our case, the tables are already in 3NF since:
- Students table has StudentID as the primary key, and StudentName is dependent only on StudentID.
- Courses table has CourseID as the primary key, and CourseName and InstructorID are dependent only on CourseID.
- Instructors table has InstructorID as the primary key, and InstructorName is dependent only on InstructorID.
- Enrollments table has a composite primary key consisting of StudentID and CourseID, and there are no other non-key attributes.

ACTIVITY: NORMALIZE BELOW TABLE



EMPLOYEE_ID	NAME	JOB_CODE	JOB	STATE_CODE	HOME_STATE
E001	Alice	J01	Chef	26	Michigan
E001	Alice	J02	Waiter	26	Michigan
E002	Bob	J02	Waiter	56	Wyoming
E002	Bob	J03	Bartender	56	Wyoming
E003	Alice	J01	Chef	56	Wyoming

RELATIONAL ALGEBRA: BASIC OPERATIONS

- Relational algebra is a procedural query language that works on relations, which are sets of tuples.
- It provides a formal foundation for relational databases and SQL.
  - **Selection**: Selects rows from a relation that satisfy a given predicate
  - **Projection**: Selects specific columns from a relation
  - **Union**: Combines the tuples of two relations, eliminating duplicate tuples.
  - **Set Difference**: Returns the tuples that are in one relation but not in the other.
  - **Cartesian Product**: Combines tuples from two relations in all possible ways.
  - **Rename**: Renames the attributes of a relation.

INTRODUCTION TO SQL

- SQL (Structured Query Language) is a standard programming language specifically designed for managing and manipulating databases.
- It is used to query, update, and manage data stored in relational database management systems (RDBMS).
- SQL allows users to perform various operations on data, including querying, inserting, updating, and deleting records.
- SQL syntax is composed of commands that follow a specific structure to perform different operations.

SQL SUBCATEGORIES

- Data Definition Language (DDL) : CREATE,ALTER, DROP, TRUNCATE
- Data Manipulation Language (DML): INSERT, UPDATE, DELETE
- Data Control Language (DCL): GRANT, REVOKE
- Data Query Language (DQL): SELECT
- Transaction Control Language (TCL): COMMIT, ROLLBACK, SAVEPOINT

DATA TYPES IN SQL

- **Numeric Data Types:**
  - INT: Integer Value
  - FLOAT: Floating-point number
  - DECIMAL(p, s): Fixed-point number with precision (p) and scale (s)
- **String Data Types:**
  - CHAR(n): Fixed-length string with length n
  - VARCHAR(n): Variable-length string with maximum length n
  - TEXT: Large variable-length string
- **Date and Time Data Types:**
  - DATE: Date value (YYYY-MM-DD)
  - TIME: Time value (HH:MI).
  - DATETIME: Date and time value (YYYY-MM-DD HH:MI)
  - TIMESTAMP: Similar to DATETIME but can be used to automatically track changes in data
- **Boolean Data Type:**
  - BOOLEAN: Represents a boolean value (TRUE or FALSE)

WORKING WITH TABLES

- The CREATE TABLE statement allows you to create a new table in a database.
- Syntax:

```
CREATE TABLE [IF NOT EXISTS] table_name(
   column1 datatype constraints,
   column2 datatype constraints,
   ...
) ENGINE=storage_engine;
```

CREATE TABLE

```
CREATE TABLE employees (
        id INT PRIMARY KEY,
        name VARCHAR(100),
        position VARCHAR(100),
        salary DECIMAL(10, 2)
);
```

```
ALTER TABLE employees
        ADD COLUMN
        department VARCHAR(100);
```

INSERT VALUES

```
INSERT INTO employees
(id, name, position, salary)
VALUES
(1, 'John Doe', 'Manager',
75000.00),
```

Single Data Insertion

```
INSERT INTO employees
(id, name, position, salary) VALUES
(2, 'Jane Smith', 'Developer', 65000.00),
(3, 'Emily Johnson', 'Designer', 60000.00),
(4, Sonam Soni', 'Manager', 95000.00);
```

Multiple Data Insertion

DROP & TRUNCATE

DROP: To delete an existing database object.

```
DROP TABLE employees;
```

TRUNCATE: To remove all records from a table without deleting the table itself.

```
TRUNCATE TABLE employees;
```

RECREATE TABLE

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    position VARCHAR(100),
    salary DECIMAL(10, 2),
    department VARCHAR(100)
);
```

```
INSERT INTO employees (id, name, position, salary,
department) VALUES (1, 'John Doe', 'Manager',
75000.00, 'Sales');
```

```
INSERT INTO employees (id, name, position, salary, department)
VALUES
(2, 'Jane Smith', 'Developer', 65000.00, 'IT'),
(3, 'Emily Johnson', 'Designer', 60000.00, 'Marketing'),
(4, 'Michael Brown', 'Analyst', 70000.00, 'Finance');
```

UPDATE & DELETE

Updating a Single Employee's Salary	• UPDATE employees SET salary = 80000.00 WHERE id = 1;
Changing the Department for Multiple Employees	• UPDATE employees SET department = 'R&D' WHERE department = 'IT';
Deleting a Single Employee:	• DELETE FROM employees WHERE id = 4;
Deleting Employees from a Specific Department	• DELETE FROM employees WHERE department = 'Marketing';

INSERT SOME MORE DATA

- Recreate Table with Department Column

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    position VARCHAR(100),
    salary DECIMAL(10, 2),
    department VARCHAR(100)
);
```

- Insert Some Data

```
INSERT INTO employees (id, name, position, salary, department) VALUES
(1, 'John Doe', 'Manager', 75000.00, 'Sales'),
(2, 'Jane Smith', 'Developer', 65000.00, 'R&D'),
(3, 'Emily Johnson', 'Designer', 60000.00, 'Marketing'),
(4, 'Michael Brown', 'Analyst', 70000.00, 'Finance'),
(5, 'Linda Green', 'Manager', 75000.00, 'Sales'),
(6, 'James White', 'Developer', 65000.00, 'R&D');
```

QUERYING DATA

- Select All Data:
  - Select * from table_name
- retrieve data from a single column:
  - SELECT name FROM employees;
- query data from multiple columns:
  - SELECT name, salary, department FROM employees;

SELECT STATEMENT

- In MySQL, the SELECT statement doesn’t require the FROM clause.
- It means that you can have a SELECT statement without the FROM clause like this:
- SELECT 1 + 1;
- SELECT NOW();
- SELECT CONCAT('John',' ','Doe');
- Aliases: SELECT CONCAT('John',' ','Doe') AS name;
- Alias with space: SELECT CONCAT('Jane',' ','Doe') AS 'Full name';

ORDER BY

To sort the rows in the result set, you add the ORDER BY clause to the SELECT statement.

Order by salary in Ascending Order:	• SELECT * FROM employees ORDER BY salary ASC
Order by name in Descending Order:	• SELECT * FROM employees ORDER BY name DESC;
Order by department Ascending and salary Descending:	• SELECT * FROM employees ORDER BY department ASC, salary DESC;
Order by position Ascending and name Ascending:	• SELECT * FROM employees ORDER BY position ASC, name ASC;
Order by salary Descending, department Ascending, and name Descending	• SELECT * FROM employees ORDER BY salary DESC, department ASC, name DESC;

FILTERING DATA

- **Where Clause:**
- Find employees in the 'Sales' department
  - SELECT * FROM employees WHERE department = 'Sales';
- Find employees with a salary greater than 65000
  - SELECT * FROM employees WHERE salary > 65000;
- **DISTINCT:** (unique value)
- Get distinct positions in the company
  - SELECT DISTINCT position FROM employees;
- **AND, OR:**
- Find employees in 'Sales' department with a salary greater than 70000
  - SELECT * FROM employees WHERE department = 'Sales' AND salary > 70000;
- Find employees in 'Sales' or 'R&D' department
  - SELECT * FROM employees WHERE department = 'Sales' OR department = 'R&D';

FILTERING DATA

- **IN, NOT IN:**
- Find employees in specific departments
    - SELECT * FROM employees WHERE department IN ('Sales', 'Finance');
- Find employees not in specific departments
    - SELECT * FROM employees WHERE department NOT IN ('Sales', 'R&D');
- **BETWEEN**:
- Find employees with a salary between 60000 and 70000
    - SELECT * FROM employees WHERE salary BETWEEN 60000 AND 70000;
- **LIKE**
- Find employees whose names start with 'J'
    - SELECT * FROM employees WHERE name LIKE 'J%';
- Find employees whose position contains 'Dev'
    - SELECT * FROM employees WHERE position LIKE '%Dev%';

FILTERING DATA

- **LIMIT**
- Get the first 3 employees
  - SELECT * FROM employees LIMIT 3;
- **IS NULL**
- Find employees with a NULL salary
  - SELECT * FROM employees WHERE salary IS NULL;

ACTIVITY

- Create Students table:
- **Students**
  - student_id (INT, Primary Key)
  - name (VARCHAR)
  - age (INT)
  - major (VARCHAR)
  - gpa (DECIMAL)
  - enrollment_date (DATE)

```
INSERT INTO Students (student_id, name, age, major, gpa,
enrollment_date)

VALUES

(101, 'Alice Johnson', 20, 'Computer Science', 3.8, '2023-01-15'),

(102, 'Bob Smith', 22, 'Mathematics', 3.4, '2023-03-22'),

(103, 'Carol Lee', 19, 'Biology', 3.2, '2023-04-10'),

(104, 'David Brown', 21, 'Physics', 2.9, '2022-11-05'),

(105, 'Eve Davis', 23, 'Computer Science', 3.6, '2022-08-20'),

(106, 'Frank Miller', 20, 'Mathematics', 3.1, '2023-02-28');
```

QUERIES

1. Select all students who are majoring in "Computer Science“
2. Select students with a GPA greater than 3.5 or who are majoring in "Mathematics“
3. Select students who are older than 20 and have a GPA less than 3.0
4. Select students who enrolled between January 1, 2023 and December 31, 2023
5. Select distinct majors from the Students table
6. Select students with IDs in the list (101, 102, 103)
7. Select students where the GPA is NULL (if applicable):
8. Select students where the name is not NULL
9. Select students whose age is exactly 18 or 22
10. Select students who are either less than 19 years old or have a GPA greater than 3.8
11. Select students who have a GPA between 2.5 and 3.5 and are majoring in "Biology“
12. Select students and order them by name in ascending order and then by GPA in descending order

LET'S CREATE PK AND FK

- Create departments table

```
CREATE TABLE departments (
	id INT PRIMARY KEY,
	name VARCHAR(100)
	NOT NULL);
```

```
INSERT INTO departments (id, name)
VALUES
(1, 'Sales'),
(2, 'R&D'),
(3, 'Marketing'),
(4, 'Finance'),
(5, 'Human Resources');
```

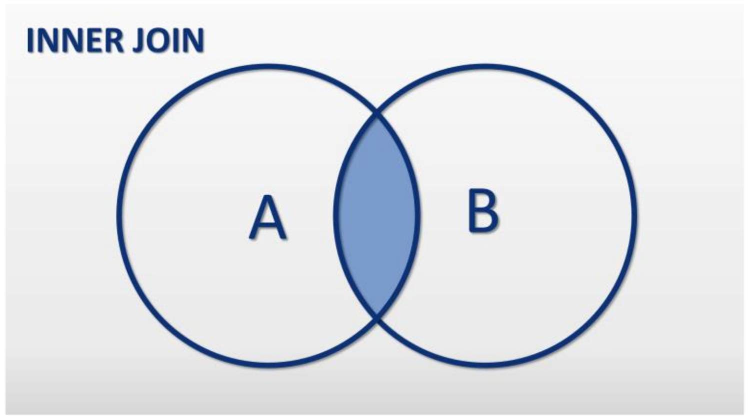
LET’S CREATE PK AND FK

```
CREATE TABLE employees (
        id INT PRIMARY KEY,
        name VARCHAR(100) NOT NULL,
        position VARCHAR(100) NOT NULL,
        salary DECIMAL(10, 2),
        department_id INT,
        FOREIGN KEY (department_id)
        REFERENCES departments(id) );
```

```
INSERT INTO employees
(id, name, position, salary, department_id)
VALUES
(1, 'John Doe', 'Manager', 75000.00, 1),
(2, 'Jane Smith', 'Developer', 65000.00, 2),
(3, 'Emily Johnson', 'Designer', 60000.00, 3),
(4, 'Michael Brown', 'Analyst', 70000.00, 4),
(5, 'Linda Green', 'Manager', 75000.00, 1),
(6, 'James White', 'Developer', 65000.00, 2),
(7, 'William Black', 'Developer', NULL, 2),
(8, 'Mary Blue', 'HR', 50000.00, 5);
```

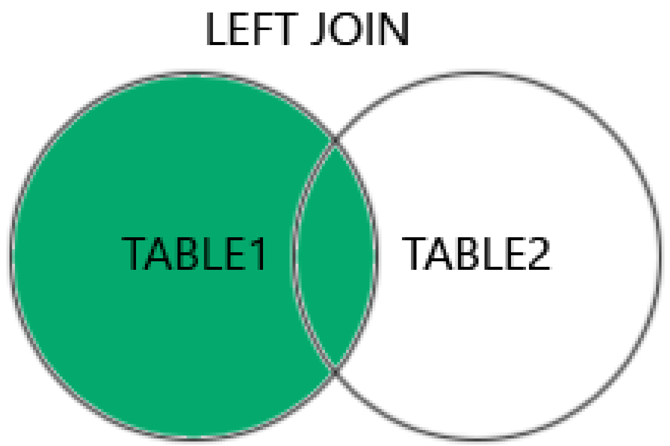
INNER JOIN

```
SELECT employees.id, employees.name, employees.position,
employees.salary, departments.name AS department
FROM employees
INNER JOIN departments ON employees.department_id = departments.id;
```



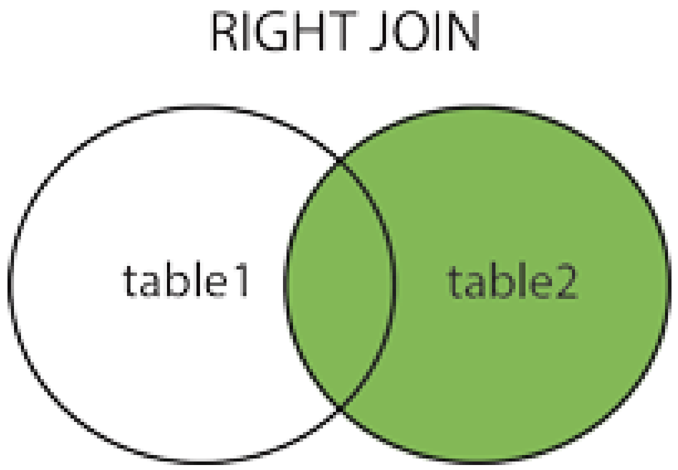
LEFT JOIN

```
SELECT employees.id, employees.name, employees.position,
employees.salary, departments.name AS department
FROM employees
LEFT JOIN departments ON employees.department_id = departments.id;
```



RIGHT JOIN

```
SELECT employees.id, employees.name, employees.position,
employees.salary, departments.name AS department
FROM employees
RIGHT JOIN departments ON employees.department_id = departments.id;
```

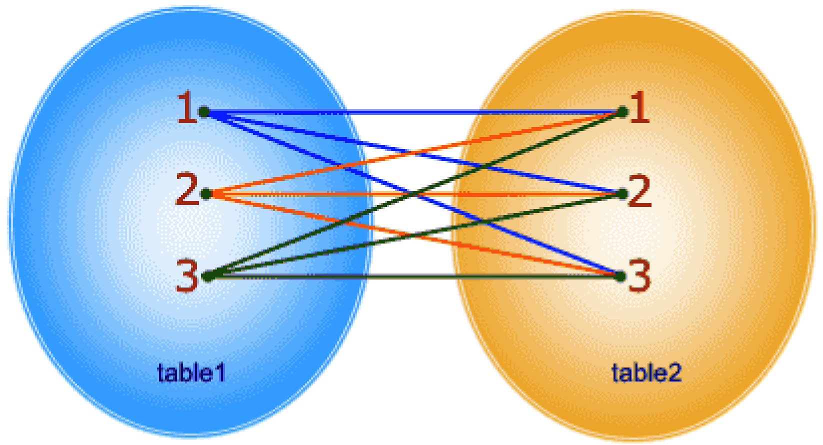


FULL OUTER JOIN (USING UNION OF LEFT JOIN AND RIGHT JOIN)

```
SELECT employees.id, employees.name, employees.position,
employees.salary, departments.name AS department
FROM employees
LEFT JOIN departments ON employees.department_id = departments.id
UNION
SELECT employees.id, employees.name, employees.position,
employees.salary, departments.name AS department
FROM employees
RIGHT JOIN departments ON employees.department_id = departments.id;
```

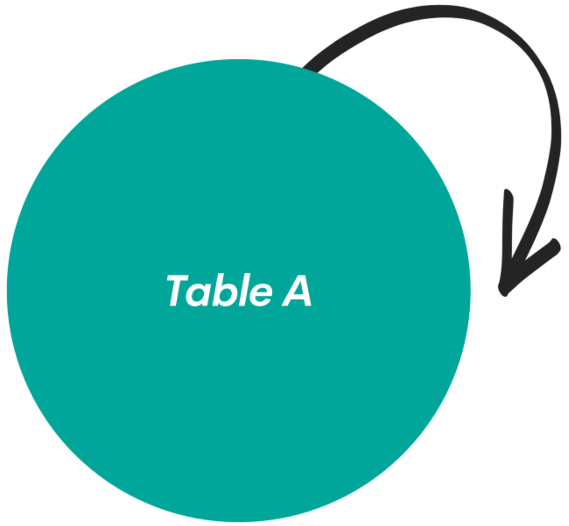
CROSS JOIN

```
SELECT employees.id AS employee_id, employees.name AS employee_name,
departments.id AS department_id, departments.name AS department_name

FROM employees

CROSS JOIN departments;
```



SELF JOIN

```
SELECT e1.name AS employee1, e2.name AS employee2, e1.position
FROM employees e1
INNER JOIN employees e2 ON e1.position = e2.position AND e1.id <> e2.id;
```



GROUP BY

- Grouping Data
- Calculate the total salary for each department

```
SELECT departments.name AS department, SUM(employees.salary) AS total_salary
FROM employees
JOIN departments ON employees.department_id = departments.id
GROUP BY departments.name;
```

HAVING

- Calculate the total salary for each department and only show departments with a total salary greater than 100,000

```
SELECT departments.name AS department, SUM(employees.salary) AS total_salary
FROM employees
JOIN departments ON employees.department_id = departments.id
GROUP BY departments.name
HAVING SUM(employees.salary) > 100000;
```

HAVING COUNT

- List departments with more than 2 employees

```
SELECT departments.name AS department, COUNT(employees.id) AS
employee_count
FROM employees
JOIN departments ON employees.department_id = departments.id
GROUP BY departments.name
HAVING COUNT(employees.id) > 2;
```

ROLLUP

- A grouping set is a set of columns to which you want to group.
- Calculate the total salary for each department, including a grand total

```
SELECT departments.name AS department, SUM(employees.salary) AS total_salary
FROM employees
JOIN departments ON employees.department_id = departments.id
GROUP BY ROLLUP(departments.name);
```

SET OPERATION: UNION

- The UNION operation combines the results of two queries and removes duplicates.
- List all unique employees from both employees and employees_archive tables

```
SELECT id, name, position, salary, department_id
FROM employees
UNION
SELECT id, name, position, salary, department_id
FROM employees_archive;
```

SET OPERATION: UNION ALL

- The UNION ALL operation combines the results of two queries and includes duplicates.
- List all employees from both employees and employees_archive tables, including duplicates

```
SELECT id, name, position, salary, department_id
FROM employees
UNION ALL
SELECT id, name, position, salary, department_id
FROM employees_archive;
```

SET OPERATION: EXCEPT (OR MINUS)

- The EXCEPT operation returns the results of the first query that are not in the results of the second query. Note that some databases, like Oracle, use MINUS instead of EXCEPT.
- List employees in the employees table who are not in the employees_archive table

```
SELECT id, name, position, salary, department_id
FROM employees
EXCEPT
SELECT id, name, position, salary, department_id
FROM employees_archive;
```

SET OPERATION: INTERSECT

- The INTERSECT operation returns the results that are common to both queries.
- List employees who are present in both employees and employees_archive tables

```
SELECT id, name, position, salary, department_id
FROM employees
INTERSECT
SELECT id, name, position, salary, department_id
FROM employees_archive;
```

RELATIONSHIP CREATION

- Create 3 tables
- Students:
  - student_id (INT, Primary Key)
  - name (VARCHAR)
  - age (INT)
  - major (VARCHAR)
  - gpa (DECIMAL)
  - enrollment_date (DATE)
- Courses:
  - course_id (INT, Primary Key)
  - course_name (VARCHAR)
  - credits (INT)

- Enrollment:
  - enrollment_id (INT, Primary Key)
  - student_id (INT, Foreign Key referencing Students)
  - course_id (INT, Foreign Key referencing Courses)
  - enrollment_date (DATE)
- Student to Enrollment (One-to-Many)
- Course to Enrollment (One-to-Many)

SAMPLE DATA

```
INSERT INTO Courses (course_id,
course_name, credits)
VALUES
(201, 'Introduction to Programming', 3),
(202, 'Calculus I', 4),
(203, 'Biology 101', 3),
(204, 'Physics Fundamentals', 4),
(205, 'Data Structures', 3),
(206, 'Advanced Mathematics', 3);
```

```
INSERT INTO Enrollments (enrollment_id,
student_id, course_id, enrollment_date)
VALUES
(301, 101, 201, '2023-01-20'),
(302, 102, 202, '2023-03-25'),
(303, 103, 203, '2023-04-15'),
(304, 104, 204, '2022-11-10'),
(305, 105, 205, '2022-08-25'),
(306, 106, 202, '2023-03-01');
```

QUERIES

- Find the names of students and the courses they are enrolled in
- List all courses with the number of students enrolled in each course
- Find students who are enrolled in more than 2 courses
- Find the course with the highest number of students enrolled
- Retrieve the average GPA of students for each major
- Get the details of students who are enrolled in 'Introduction to Programming' course
- List all students who have a GPA higher than the average GPA of their major
- Show the total credits of courses each student is enrolled in
- Find the students who are enrolled in courses but have not enrolled in any course in 2023
- List all majors and the number of students with a GPA less than 3.0 in each major
- Find students who have enrolled in the most number of courses
- Retrieve the details of students enrolled in more than 1 course and their average GPA
- List the students and their total GPA for the courses they are enrolled in
- Get the number of students enrolled in each course, sorted by the number of students in descending order

SUBQUERY

- Subqueries and nested queries are powerful tools in SQL that allow you to perform more complex queries by embedding one query inside another.

StudentId	Name
1	Alice
2	Bob
3	Charlie

CourseId	CourseName
101	Mathematics
102	History
103	Science

EnrollmentID	StudentId	CourseID	Grade
1	1	101	A
2	1	102	B
3	2	101	B
4	3	103	C
5	2	103	A

USING SELECT CLAUSE

- Find the highest grade for each student.

```
SELECT
    Name,
    (SELECT MAX(Grade)
     FROM Enrollments
     WHERE Enrollments.StudentID = Students.StudentID) AS HighestGrade
FROM
    Students;
```

USING FROM CLAUSE

- Find the average grade for each course.

```
SELECT
   CourseName,
   AvgGrade
FROM
   (SELECT
      CourseID,
      AVG(Grade) AS AvgGrade
    FROM
      Enrollments
GROUP BY
       CourseID) AS
CourseAverages
JOIN
   Courses
ON
   Courses.CourseID =
CourseAverages.CourseID;
```

USING WHERE CLAUSE

- Find students who are enrolled in the Mathematics course.

```
SELECT
    Name
FROM
    Students
WHERE
    StudentID IN (SELECT StudentID
                  FROM Enrollments
                  WHERE CourseID = 101);
```

CORRELATED SUBQUERIES

- A correlated subquery is a subquery that references columns from the outer query. It is executed once for each row in the outer query.
- Find students who have enrolled in more than one course.

```
SELECT
    Name
FROM
    Students
WHERE
    (SELECT COUNT(*)
     FROM Enrollments
     WHERE Enrollments.StudentID = Students.StudentID) > 1;
```

COMBINED DEMO

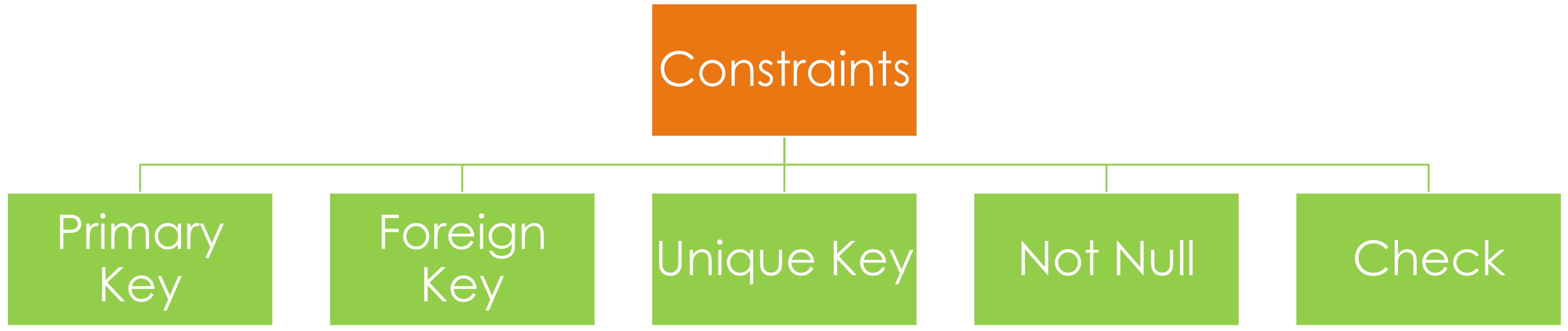
- Find the names of students who have achieved the highest grade in any course.

```
SELECT
    Name
FROM
    Students
WHERE
    StudentID IN (SELECT
                      StudentID
                  FROM
                      Enrollments
                  WHERE
                      Grade = (SELECT
                                   MAX(Grade)
                               FROM
                                   Enrollments
                               WHERE
                                   CourseID = Enrollments.CourseID));
```

ACTIVITY

1. Find the total number of courses each student is enrolled in.
2. Find the average grade of students in each course.
3. Find the names of students who have received an 'A' grade in any course.
4. Find the names of students who have taken the Science course.
5. Find the names of students who have taken more than one course and received at least one 'A' grade.

INTEGRITY CONSTRAINTS



PRIMARY KEY

- Ensures that each row in a table can be uniquely identified.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100)
);
```

FOREIGN KEY

- Ensures that a value in one table matches a value in another table, enforcing referential integrity.

```
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    CourseID INT,
    Grade CHAR(1),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

UNIQUE CONSTRAINT

- Ensures that all values in a column are unique.

```
CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    CourseName VARCHAR(100) UNIQUE
);
```

NOT NULL CONSTRAINT

- Ensures that a column cannot have a NULL value.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL
);
```

CHECK CONSTRAINT

- Ensures that all values in a column satisfy a specific condition.
- Ensure that the Grade column in the Enrollments table only contains the values 'A', 'B', 'C', 'D', or 'F'.

```
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    CourseID INT,
    Grade CHAR(1),
    CHECK (Grade IN ('A', 'B', 'C', 'D', 'F')),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

INDEXES

- Indexes are used to speed up the retrieval of data from a database table. They create a separate data structure that allows for faster searching.
- **Creating an Index**:
- Create an index on the Name column in the Students table to speed up search queries.
  - CREATE INDEX idx_student_name ON Students(Name);
- **Composite Index**: An index on multiple columns.
- Create a composite index on the StudentID and CourseID columns in the Enrollments table.
  - CREATE INDEX idx_student_course ON Enrollments(StudentID, CourseID);

INDEXES

- **Unique Index**: Ensures that the indexed columns do not contain duplicate values.
- Create a unique index on the CourseName column in the Courses table.
- CREATE UNIQUE INDEX idx_course_name ON Courses(CourseName);

- You can insert some records in your tables and try to fetch data without adding any index.
- EXPLAIN ANALYZE SELECT * FROM Students WHERE Name = 'Alice';
- CREATE INDEX idx_student_name ON Students(Name);
- Now once the index created and check the same query and see the performance.

ACID PROPERTIES

- **Atomicity**
  - Ensures that all operations within a transaction are completed successfully. If any operation fails, the entire transaction fails, and the database remains unchanged.
  - Transferring money from one account to another. Both debit and credit operations must succeed; otherwise, both are rolled back.
- **Consistency:**
  - Ensures that a transaction brings the database from one valid state to another, maintaining database rules such as constraints, cascades, and triggers.
  - After a transaction, all primary keys, foreign keys, and unique constraints are still valid.

ACID PROPERTIES

- **Isolation**
  - Ensures that transactions are executed in isolation, meaning that concurrent transactions do not affect each other. The outcome should be the same as if the transactions were executed sequentially.
  - While one transaction is updating a record, another transaction cannot read that record until the first transaction is complete.
- **Durability:**
  - Ensures that once a transaction has been committed, it remains so, even in the event of a system failure. This is typically achieved through logging and recovery mechanisms.
  - After a transaction commits, changes made to the database persist even if there is a power failure immediately afterward.

TRANSACTION DEMO

```
CREATE TABLE Accounts (
    account_id VARCHAR(10) PRIMARY KEY,
    account_name VARCHAR(100),
    balance DECIMAL(10, 2)
);
```

```
INSERT INTO Accounts (account_id, account_name, balance)
VALUES
('A001', 'Alice', 1000.00),
('A002', 'Bob', 1500.00),
('A003', 'Charlie', 2000.00);
```

COMMIT DEMO

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE Accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

ROLLBACK

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE account_id = 1;

ROLLBACK;
```

SAVEPOINT

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE account_id = 1;
SAVEPOINT sp1;
UPDATE Accounts SET balance = balance + 100 WHERE account_id = 2;

ROLLBACK TO sp1;
COMMIT;
```

STORED PROCEDURES

- Stored procedures are precompiled collections of SQL statements that can be executed as a single unit.
- They can accept parameters and are used to encapsulate logic that can be reused.

```
DELIMITER //

CREATE PROCEDURE GetAllAccounts()
BEGIN
   SELECT * FROM Accounts;
END //

DELIMITER ;
```

Calling stored procedure

```
CALL GetAllAccounts();
```

FUNCTIONS

- A function is a database object that performs a calculation or operation and returns a single value.
- Functions can be used in SQL statements wherever an expression is allowed.

Use the function:
SELECT GetBalance('A001')
AS balance;

```
DELIMITER //

CREATE FUNCTION GetBalance(account_id
VARCHAR(10))
RETURNS DECIMAL(10, 2)
DETERMINISTIC
BEGIN
    DECLARE account_balance DECIMAL(10,
2);

    SELECT balance INTO account_balance
    FROM Accounts
    WHERE account_id = account_id;

    RETURN account_balance;
END //

DELIMITER ;
```

TRIGGER

- A trigger is a special type of stored procedure that is automatically executed or fired when certain events occur in the database.
- Triggers can be set to execute in response to events such as INSERT, UPDATE, or DELETE on a specific table or view.
- Let’s Implement the same with one example:
- Create account table:

```
CREATE TABLE Accounts (
    account_id VARCHAR(10) PRIMARY KEY,
    account_name VARCHAR(100),
    balance DECIMAL(10, 2)
);
```

- Create Log Table:

```
CREATE TABLE AccountChanges (
    change_id INT AUTO_INCREMENT PRIMARY KEY,
    account_id VARCHAR(10),
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    old_balance DECIMAL(10, 2),
    new_balance DECIMAL(10, 2)
);
```

- Some Sample Data:

```
INSERT INTO Accounts (account_id, account_name, balance) VALUES
('A001', 'Alice', 1000.00),
('A002', 'Bob', 1500.00),
('A003', 'Charlie', 2000.00);
```

- Let’s Create Trigger:

```
DELIMITER //

CREATE TRIGGER AfterAccountUpdate
AFTER UPDATE ON Accounts
FOR EACH ROW
BEGIN
   INSERT INTO AccountChanges (account_id, old_balance, new_balance)
   VALUES (OLD.account_id, OLD.balance, NEW.balance);
END //

DELIMITER ;
```

- Update an Account and Observe the Trigger Action:

- Update Alice's balance

```
UPDATE Accounts SET balance = balance - 200.00 WHERE account_id = 'A001';
```

- Check the log table to see the change

```
SELECT * FROM AccountChanges;
```

USE MYSQL SAMPLE DB

- Let's take one sample Database:
- https://github.com/datacharmer/test_db
- download zip from here
- Unzip the data
- Move to the folder and execute below command
- mysql -u root -p -t < employees.sql
- Enter your password and done..
- Verify the installation using below command:
  - mysql -u root -p -t < test_employees_md5.sql

QUERIES

- Connect with MYSQL
  - use employees;
  - show tables;
  - select * from employees limit 5;

1. Retrieve the first name, last name, and job title of all employees.
2. Find all employees who work in the Sales department.
3. List all products sorted by product name in ascending order.
4. Calculate the average salary of all employees.
5. Find the total number of employees in each department.
6. list departments with more than 5 employees.
7. Retrieve a list of orders along with the customer names who placed the orders.

8. Find all customers and their orders, including customers who have not placed any orders.
9. List all employees and the names of their managers.
10. Find all products that have never been ordered.
11. Find all employees who work in the 'Marketing' or 'Finance' departments.
12. List all orders placed between January 1, 2020, and December 31, 2020.
13. Find all customers whose names start with the letter 'A'.
14. Combine the first names of employees and customers into a single list.
15. Find the names of employees who have placed the highest number of orders.

HOW TO USE SAMPLE DB

15. Find the number of Male and Female employees in the database and the order count in descending order.
16. Find the Average salary by employee title and order by descending order.
17. List first 5 employees (id,fname,lname,department name) alog with their department names.
18. Display firstname, lastname,salary of the highest payed employee.
19. second highest payed employee