2. Database Schema Design

Object of Java = Entity in RDBMS

Relationships

ER diagram (entity relationship diagram)

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

Types of relationship are as follows:

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

a**. One-to-One relationship**

For example, A female can marry to one male, and a male can marry to one female.

b. One-to-many relationship

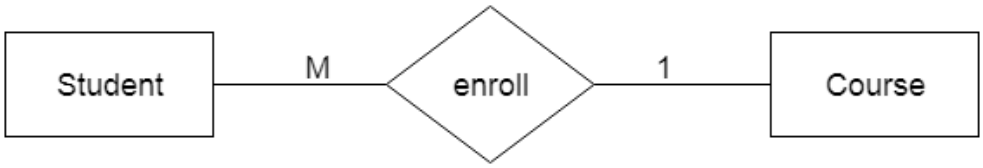
When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.

c. Many-to-one relationship

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

For example, Student enrolls for only one course, but a course can have many students.



d. Many-to-many relationship

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

For example, Employee can assign by many projects and project can have many employees.

Schema = Blueprint, entities, relationships, tables, indexes, joins

Why DB Schema Design?

Poorly designed databases can cause many problems, including a waste of resources, difficult maintenance, and faulty performance. That's why having a great database schema design is a crucial part of effective data management.

"Great database schema design" is certainly easier said than done, though. Since a given domain or business problem can be modeled in infinitely different ways, it stands to reason that there are comparatively fewer ways of doing it correctly.

We'll start with some basics, explaining what a database schema is and why it's important. Then, we'll talk about how you create schemas in practice. Last but not least, we'll see an example of database schema design.

Database schema: The fundamentals

What is a database schema?

A database schema, in a nutshell, is the way a given database is organized or structured.

Metadata = Data about data. E.g. describe table command.

Why is a database schema important?

Relational database systems rely on having a schema in place so you can work with them. So that's the first reason a database schema is important: you literally can't have a database without one.

If your schema isn't structured in an optimal way—e.g., it doesn't follow normalization—that might result in duplicated or inconsistent data. On the other hand, it's possible to get to the other extreme where your queries become slow.

To sum it up, database schema design is essential because you want your databases to work as efficiently as possible.

SLA?

What is essential to a good database design?

- **Reduces redundancy**: divide information carefully among tables to eliminate data redundancy. Duplicate data wastes space and can lead to inconsistency.
- **Provides access** with information a user needs to join tables together. (Use good primary keys, while creating proper relationships between tables.)
- Ensures **data accuracy** and **integrity**.
- Accommodates your data processing and reporting needs.

Database schema design: Real world examples

I Problem

The demo database schema we'll design will be for a fictional bookstore.

Let's start by creating a schema called **bookstore** and a table called **category**:

```
CREATE TABLE bookstore.category
( id int PRIMARY KEY,
name varchar(255) NOT NULL UNIQUE,
description varchar(255) NOT NULL )
```

The snippet of code above, besides defining a schema, also creates a table in it, which we call `category`. The `category` table has three columns:

- `id`, which is a unique identifier for each `category`

- `name`, which is a unique name for each `category`
- and `description`, which is a brief, optional description

A bookstore certainly needs information about authors, so let's create a table for that as well.

Creating a table for authors

```
CREATE TABLE bookstore.authors
( id int PRIMARY KEY,
 name varchar(255) NOT NULL,
 bio varchar(500) NOT NULL )
```

Learn More

With the `authors` table, we can manage data about authors, using three columns:

- id
- a `name`—this time without a "unique" constraint, since it's perfectly fine for two authors to have the same name
- and a `biography`

Creating a table for books

We're now getting closer to being able to store the books themselves. Let's create a table for that:

```
CREATE TABLE bookstore.books
( id int PRIMARY KEY,
title varchar(255) NOT NULL,
description varchar(255) NOT NULL,
ISBN char(13) NOT NULL,
category_id INT NOT NULL,

FOREIGN KEY(category_id)
REFERENCES bookstore.category(id) )
```

The table `books` is more involved than the previous ones. Besides the usual unique numerical identifier, a display column—in this case, "title" instead of "name"—and a description, this table also contains a column called `ISBN` (an International Standard Book Number) and, most interestingly, a foreign key constraint.

In a nutshell, **foreign keys are what allow us to connect tables together.** In this case, the column `category_id` in the table `books` references the columns `id` from the table `category`. That way, it's possible to express the relationship between those two entities. To use the jargon, we've defined a one-to-many relationship. In other words, a single `category` can have one or more books assigned to it. On the other hand, there's no way to assign more than one `category` to a book since the `category_id` column can only point to a single row at a time.

Connecting books and authors

You might have noticed that our table `books` doesn't have any relationship to authors. Couldn't we just include another foreign key in the table, connecting it to authors, the same as we did for `category`? Sure, that would be possible. However, remember that our requirements state that books can have more than one author.

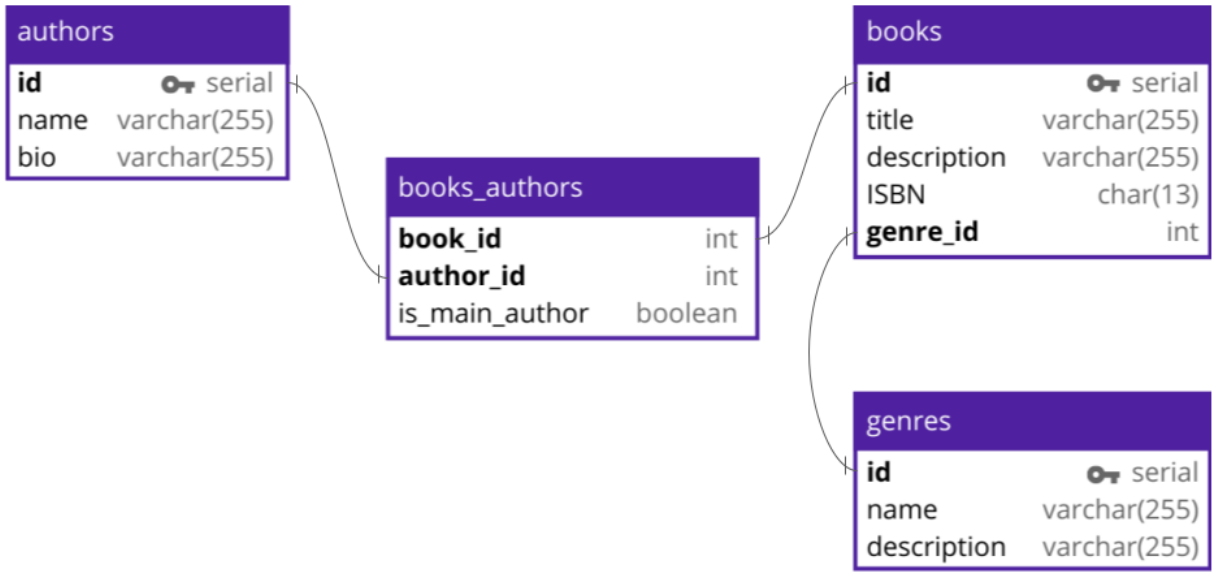
So an author might be associated with many books, while a book might have more than one author. That's what we call a many-to-many relationship. There's no way we can design that relationship by adding a column to the `authors` or to the `books` table.

Instead, our solution here is to create a third table, which will hold foreign keys to both authors and books:

```
CREATE TABLE bookstore.books_authors ( book_id  int REFERENCES bookstore.books (id),
  author_id int REFERENCES bookstore.authors (id),
  is_main_author BOOLEAN NOT NULL DEFAULT false,
  CONSTRAINT bill_product_pkey PRIMARY KEY (book_id, author_id) );
```

In this third table, we reference both `author`_id and `book_id`. We also have an extra column, `is_main_author`, of type boolean—that is, it can be true or false and indicates whether a given author is the main author of the book specified in the relationship.

That's a lot to keep track of (and we've just scratched the surface!), so here's how all those tables fit together:



With the schema we designed so far, we're able to store and manage information about a simple bookstore. Keep in mind that a real-world schema for a bookstore would probably be much more complex. However, our example should be enough to give you an idea of what is involved in creating a database schema.

```
drop database bookstore;
create database bookstore;
use bookstore;
CREATE TABLE category
( id int PRIMARY KEY,
name varchar(255) NOT NULL UNIQUE,
description varchar(255) NOT NULL );

CREATE TABLE authors
( id int PRIMARY KEY,
name varchar(255) NOT NULL,
bio varchar(500) NOT NULL );

CREATE TABLE bookstore.books
( id int PRIMARY KEY,
title varchar(255) NOT NULL,
description varchar(255) NOT NULL,
ISBN char(13) NOT NULL,
category_id INT NOT NULL,
FOREIGN KEY (category_id) REFERENCES category(id)
);

CREATE TABLE books_authors ( book_id int REFERENCES books(id),
  author_id int REFERENCES authors(id),
  is_main_author BOOLEAN NOT NULL DEFAULT false,
  CONSTRAINT bill_product_pkey PRIMARY KEY (book_id, author_id) );
```

We Problem

Step 1: Handle Ambiguity
Database questions often have some ambiguity, intentionally or unintentionally. Before you proceed with your design, you must understand exactly what you need to design.

Imagine you are asked to design a system to represent an apartment rental agency. You will need to know whether this agency has multiple locations or just one. You should also discuss with your interviewer how general you should be. For example, it would be extremely rare for a person to rent two apartments in the same building. But does that mean you shouldn't be able to handle that? Maybe, maybe not. Some very rare conditions might be best handled through a work around (like duplicating the person's contact information in the database).

1. There are many complexes.
2. Each Complex has many buildings.
3. Each building has multiple apartments.
4. An apartment can have mulitple tenants.
5. A tenant can occupy multiple apartments.

Step 2: Define the Core Objects
Next, we should look at the core objects of our system. Each of these core objects typi- cally

translates into a table. In this case, our core objects might be Property, Building, Apartment, Tenant and Manager.
Step3: Analyze Relationships
Outlining the core objects should give us a good sense of what the tables should be. Are they many-to-many? One-to-many?
If buildings has a one-to-many relationship with Apartments (one B u ild in g has many Apartments), then we might represent this as follows:

Buildings(BuildingID, BuildingName, BuildingAddress)

Apartments(ApartmentID, ApartmentAddress, BuildingID)

If we want to allow for the possibility that one person rents more than one apartment,
we might want to implement a many-to-many relationship asfollows:

Tenants		
TenantID	TenantName	TenantAddress

Apartments		
ApartmentID	ApartmentAddress	BuildingID

TenantApartments	
TenantID	ApartmentID

▼ Practice Questions

Questions 1 through 3 (refer to the below database schema):

Apartments	**Buildings**	**Tenants**	**Requests**	**Complexes**	**AptTenants**
AptID int	BuildingID int	TenantID int	ReqeuestID int	ComplexID int	ID
UnitNumber varchar	ComplexID int	TenantName varchar	status varchar()	ComplexName varchar	AptID
BuildingID int	BuildingName varchar		AptID int		TenantID

	Address varchar		Description varchar		

1. Write a SQL query to get a list of tenants who are renting more than one apartment.
2. Write a SQL query to get a list of all buildings and the number of open requests (Requests in which status equals 'Open').
3. Building #11 is undergoing a major renovation. Implement a query to close all requests from apartments in this building.

Solutions to Practice Questions

```
1.
SELECT TenantName FROM Tenants INNER DOIN
(SELECT TenantID
FROM AptTenants
GROUP BY TenantID HAVING count(*) > 1) C
ON Tenants.TenantID = C.TenantID

2.
SELECT BuildingName, ISNULL(Count, 0) as 'Count' 2 FROM Buildings
LEFT JOIN
(SELECT Apartments.BuildingID, count(*) as 'Count' FROM Requests INNER DOIN Apartments
ON Requests.AptID = Apartments.AptID
WHERE Requests.Status = 'Open'
GROUP BY Apartments.BuildingID) ReqCounts ON ReqCounts.BuildingID = Buildings.BuildingID;

3.
 UPDATE Requests
SET Status='Closed'  WHERE AptID IN
(SELECT AptID
FROM Apartments
WHERE BuildingID = 11);
```

You Problem:

Imagine a simple database storing information for students' grades. Design what this database might look like and provide a SQL query to return a list of the honor roll students (top 10%), sorted by their grade point average.

Transactions & ACID Properties

A transaction is a sequence of operations performed (using one or more SQL statements) on a database as a single logical unit of work.

Properties of a Transaction

The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database). A database transaction must be atomic, consistent, isolated and durable.

A database transaction must be atomic, consistent, isolated and durable. Bellow we have discussed these four points.

Atomic : A transaction is a logical unit of work which must be either completed with all of its data modifications, or none of them is performed.

Consistent : At the end of the transaction, all data must be left in a consistent state.

Isolated : Modifications of data performed by a transaction must be independent of another transaction. Unless this happens, the outcome of a transaction may be erroneous.

Durable : When the transaction is completed, effects of the modifications performed by the transaction must be permanent in the system.

Often these four properties of a transaction are acronymed as ACID.

To assure the ACID properties of a transaction, any changes made to data in the course of a transaction must be **committed or rolled back.**

When a transaction completes normally, a transaction processing system commits the changes made to the data; that is, it makes them permanent and visible to other transactions.

When a transaction does not complete normally, the system rolls back (or backs out) the changes; that is, it restores the data to its last consistent state.

Resources that can be rolled back to their state at the start of a transaction are known as recoverable resources: resources that cannot be rolled back are nonrecoverable.

Transfer INR 100 from mine to your bank account:

Initial state

My account balance 1000

Your account balance 500

Total Balance 1500

```
-- Transaction

check if both the accounts are active.

check if I have sufficient balance

My balance - 100

Your balance + 100
```

Final State

My account balance 900

Your account balance 600

Total Balance 1500

MySQL Transaction Statement

MySQL control transactions with the help of the following statement:

MySQL provides a START TRANSACTION statement to begin the transaction. It also offers a "BEGIN" and "BEGIN WORK" as an alias of the START TRANSACTION.

We will use a COMMIT statement to commit the current transaction. It allows the database to make changes permanently.

We will use a ROLLBACK statement to roll back the current transaction. It allows the database to cancel all changes and goes into their previous state.

```
start transaction; delete from Persons where age=30; rollback;

start transaction; delete from Persons where age=30; commit;
```

In real world applications, mostly it is handled by Application layer by using exception handling.

More Example on DB Schema Design:

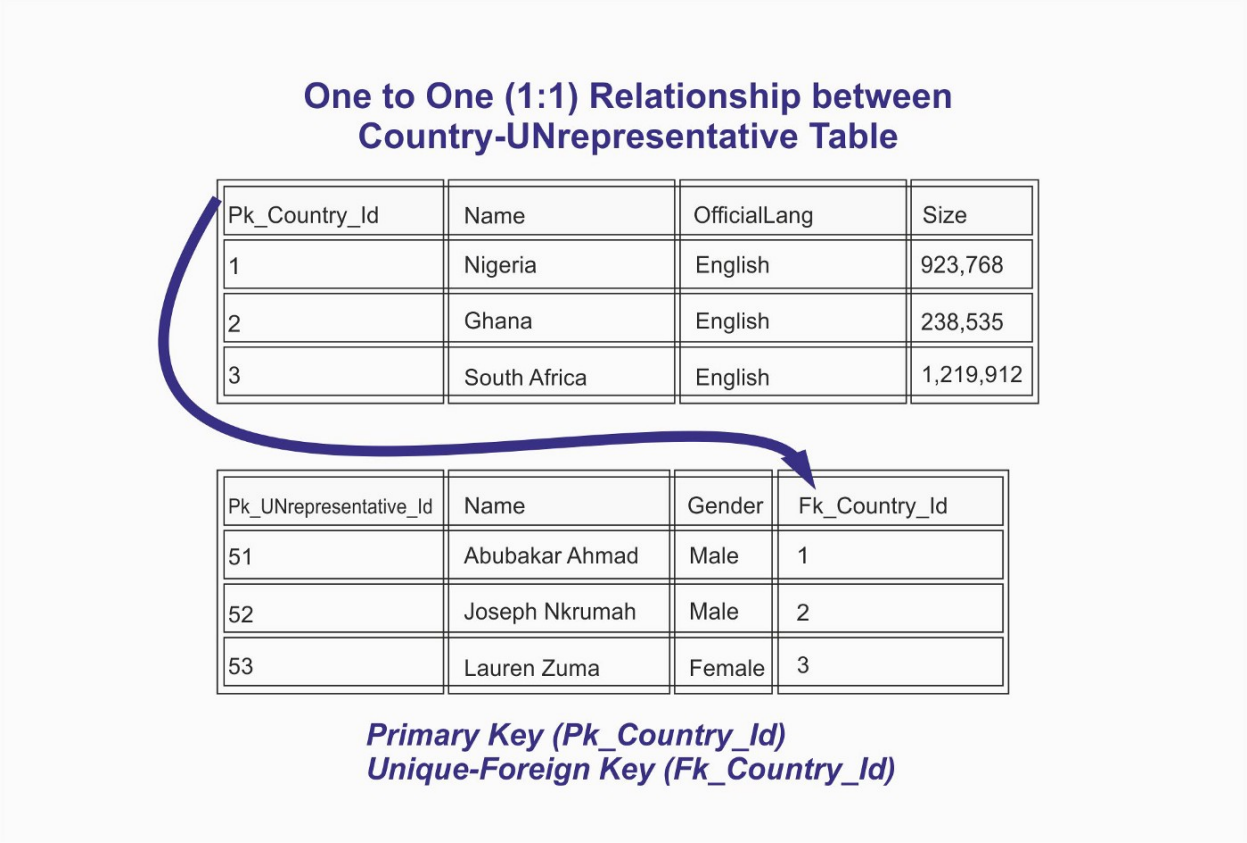
One to One Relationship (1:1)

When a row in a table is related to only one role in another table and vice versa,we say that is a **one to one relationship**. This relationship can be created using *Primary key-Unique foreign key constraints*.

For instance a Country can only have one UN Representative, and also a UN Representative can only represent one Country.

Let's try out the implement and see for ourselves.

```
CREATE TABLE Country
(
Pk_Country_Id INT IDENTITY PRIMARY KEY,
Name VARCHAR(100),
Officiallang VARCHAR(100),
Size INT(11),
);
CREATE TABLE UNrepresentative
(
Pk_UNrepresentative_Id INT PRIMARY KEY,
Name VARCHAR(100),
Gender VARCHAR(100),
Fk_Country_Id INT UNIQUE FOREIGN KEY REFERENCES Country(Pk_Country_Id)
);
INSERT INTO Country ('Name','Officiallang',’Size’)
VALUES ('Nigeria','English',923,768);INSERT INTO Country ('Name','Officiallang',’Size’)
VALUES ('Ghana','English',238,535);INSERT INTO Country ('Name','Officiallang',’Size’)
VALUES ('South Africa','English',1,219,912);INSERT INTO UNrepresentative ('Pk_Unrepresentative_Id','Nam
e','Gender','Fk_Country_Id')
VALUES (51,'Abubakar Ahmad','Male',1);INSERT INTO UNrepresentative ('Pk_Unrepresentative_Id','Name','Gen
der','Fk_Country_Id')
VALUES (52,'Joseph Nkrumah','Male',2);INSERTINTOUNrepresentative ('Pk_Unrepresentative_Id','Name','Gende
r','Fk_Country_Id')
VALUES (53,'Lauren Zuma,'Female',3);
SELECT * FROM Country
SELECT * FROM UNrepresentative;
```



One-to-One (1:1) Relationship between Country-UNrepresentative Table

With Country(Pk_Country_Id) serving as the Primary key and *UNrepresentative(fk_Country_id)* as (Unique Key Constraint-Foreign Key), you can implement One-to-One Relationship.

One to Many Relationship (1:M)

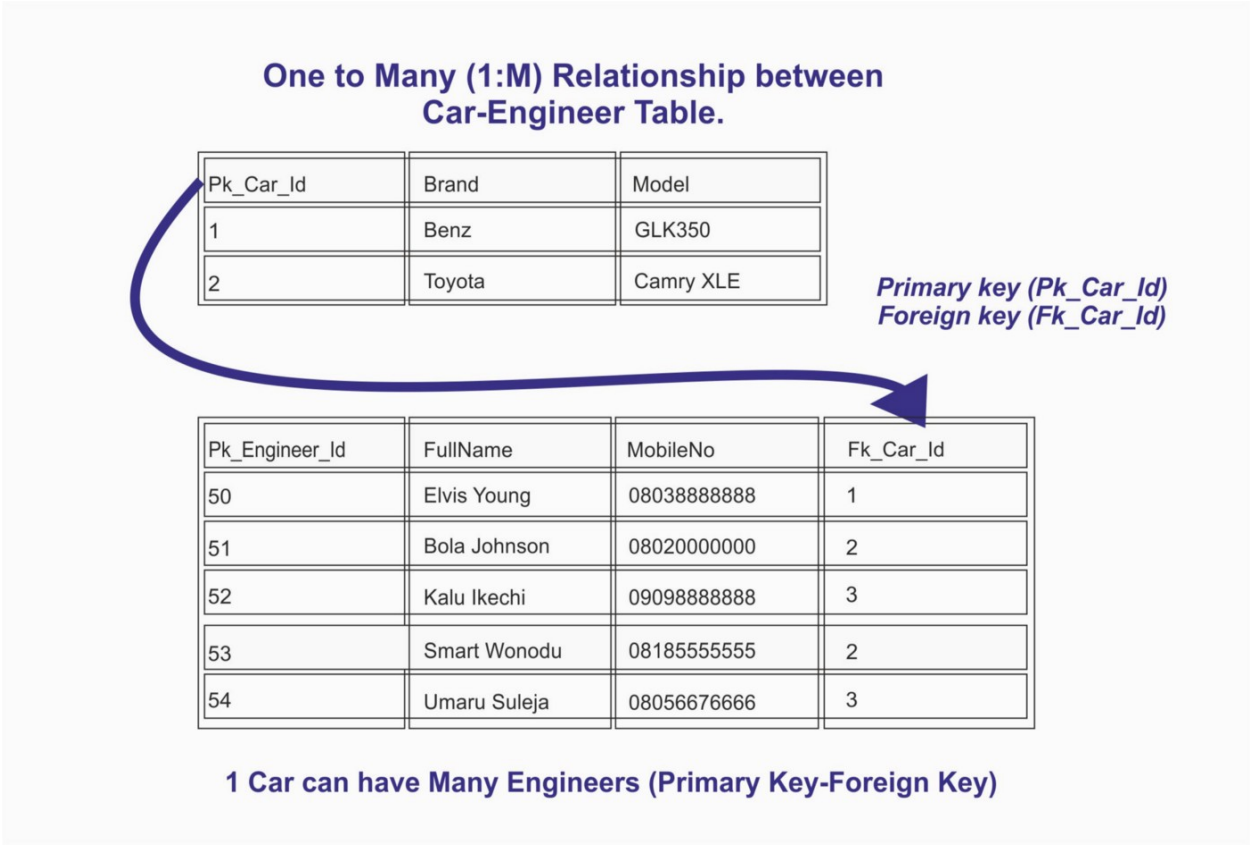
This is where a row from one table can have multiple matching rows in another table this relationship is defined as a **one to many relationship**. This type of relationship can be created using *Primary key-Foreign key relationship.*

This kind of Relationship, allows a Car to have multiple Engineers.

Let's try out the implementation and see for ourselves.

```
CREATE TABLE Car
(
Pk_Car_Id INT PRIMARY KEY,
Brand VARCHAR(100),
Model VARCHAR(100)
);CREATE TABLE Engineer
(
Pk_Engineer_Id INT PRIMARY KEY,
FullName VARCHAR(100),
MobileNo CHAR(11),
Fk_Car_Id INT FOREIGN KEY REFERENCES Car(Pk_Car_Id)
);
```

```
INSERT INTO Car ('Brand','Model')
VALUES ('Benz','GLK350');INSERT INTO Car ('Brand','Model')
VALUES ('Toyota','Camry XLE');INSERT INTO Engineer ('Pk_Engineer_Id','FullName','MobileNo','Fk_Car_Id')
VALUES(50,'Elvis Young','08038888888',2);INSERT INTO Engineer ('Pk_Engineer_Id','FullName','MobileNo','F
k_Car_Id')
VALUES(51,'Bola Johnson','08020000000',1);INSERT INTO Engineer ('Pk_Engineer_Id','FullName','MobileN
o','Fk_Car_Id')
VALUES(52,'Kalu Ikechi','09098888888',1);INSERT INTO Engineer ('Pk_Engineer_Id','FullName','MobileNo','F
k_Car_Id')
VALUES(53,'Smart Wonodu','08185555555',1);INSERT INTO Engineer ('Pk_Engineer_Id','FullName','MobileN
o','Fk_Car_Id')
VALUES(54,Umaru Suleja','08056676666',1);SELECT * FROM Car;
SELECT * FROM Engineer;
```



One-to-Many (1:M) Relationship between Car-Engineer Table

Car(Pk_Car_Id) serving as *the Primary Key* and *Engineer(Fk_Car_Id) as (Foreign Key)*.

Based on the Car (Pk_Car_Id)-Engineer(Fk_Car_Id) relationship, we now have a design for our database tables that consolidates the One-to-Many relationship using a foreign key!

Many to Many Relationship (M:M)

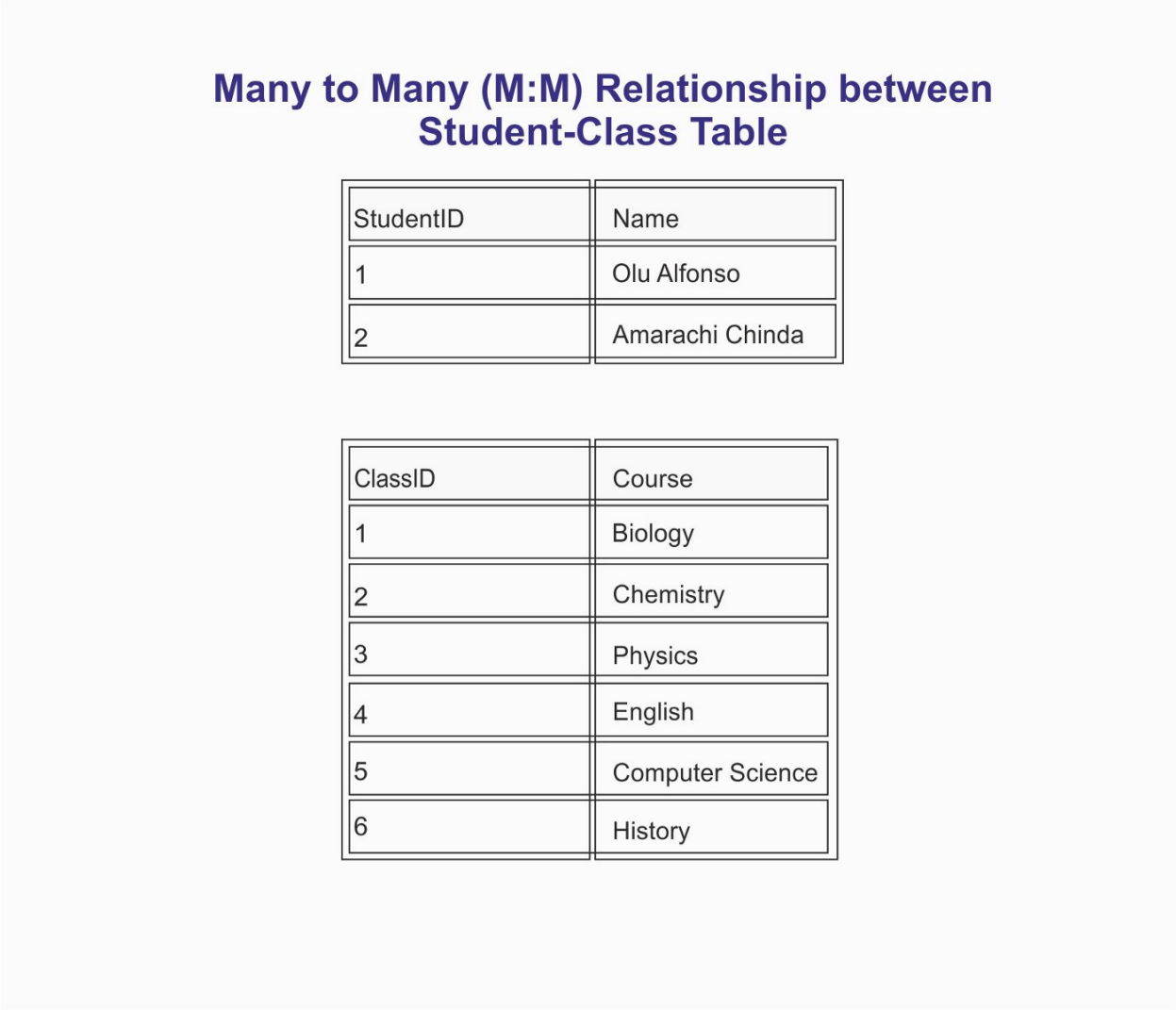
A row from one table can have multiple matching rows in another table, and a row in the other table can also have multiple matching rows in the first table this relationship is defined as a **many to many relationship**. This type of relationship can be created using a third table called "*Junction table*" or

“*Bridging table*”. This Junction or Bridging table can be assumed as a place where attributes of the relationships between two lists of entities are stored.

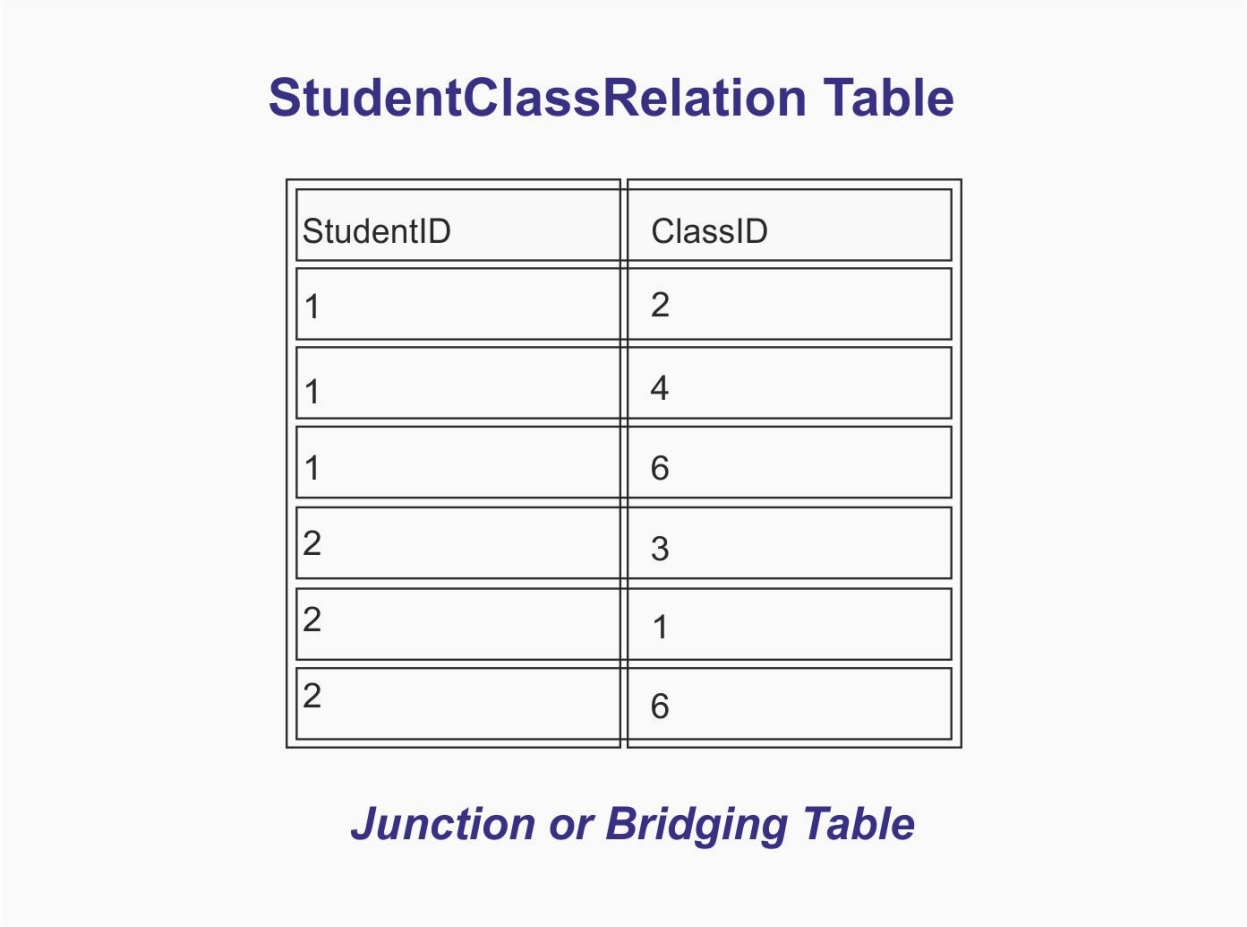
This kind of Relationship, allows a junction or bridging table as a connection for the two tables.

Let’s try out the implementation and see for ourselves.

```
CREATE TABLE Student(
StudentID INT(10) PRIMARY KEY,
Name VARCHAR(100),
);CREATE TABLE Class(
ClassID INT(10) PRIMARY KEY,
Course VARCHAR(100),
);CREATE TABLE StudentClassRelation(
StudentID INT(15) NOT NULL,
ClassID INT(14) NOT NULL,FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
FOREIGN KEY (ClassID) REFERENCES Class(ClassID),
UNIQUE (StudentID, ClassID)
);INSERT INTO Student ('Name')
VALUES ('Olu Alfonso');INSERT INTO Student ('Name')
VALUES ('Amarachi Chinda');
INSERT INTO Class ('Course')
VALUES ('Biology');INSERT INTO Class ('Course')
VALUES ('Chemistry');INSERT INTO Class ('Type')
VALUES ('Physics');INSERT INTO Class ('Type')
VALUES ('English');INSERT INTO Class ('Type')
VALUES ('Computer Science');INSERT INTO Class ('Type')
VALUES ('History');INSERT INTO StudentClassRelation ('StudentID','ClassID')
VALUES (1,2);INSERT INTO StudentClassRelation ('StudentID','ClassID')
VALUES (1,4);INSERT INTO StudentClassRelation ('StudentID','ClassID')
VALUES (1,6);INSERT INTO StudentClassRelation ('StudentID','ClassID')
VALUES (2,3);INSERT INTO StudentClassRelation ('StudentID','ClassID')
VALUES (2,1);INSERT INTO StudentClassRelation ('StudentID','ClassID')
VALUES (2,6);
INSERT INTO StudentClassRelation ('StudentID','ClassID')
VALUES (2,1);
```



Many to Many (M:M) Relationship between Student-Class Table



StudentClassRelation Table

Two columns are stored in the *Junction or Bridging Table* one for each of the primary keys from the other tables, therefore combination of student and class records are stored here, while our student and class tables remains unaltered

This data structure looking this way, is easier to add more relationships between tables and also allow us update our students and Classes without influencing the relationships between them.

With our data structure in this way, it is easier to add more relationships between tables and to update our employees and departments without influencing the relationships between them.

References:

[https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions

https://www.w3resource.com/sql/controlling-transactions.php](https://www.ibm.com/docs/en/cics

ts/5.4?topic=processing-acid-properties-transactions

https://www.w3resource.com/sql/controlling-transactions.php)

3. Transactions, ACID