# CSCI 3308
# Software Dev Methodologies and Tools

Lab 5: February 12, 2025

# First, lets know the difference!

## What is SQL?

Structured Query Language (SQL) is a standardized programming language that is used to **manage relational databases** and **perform various operations on the data** in them.

## What is PostgreSQL?

PostgreSQL is an open-source, highly stable **database system** that provides support to different functions of SQL. It’s used to store data.

# Data types

1. Boolean

1. Character Types
    a. CHAR(n)
    b. VARCHAR(n)
    c. TEXT

1. Numeric Types
    b. INT,  SMALLINT,  BIGINT
    c. SERIAL
    d. numeric(precision, scale)

1. Temporal Types
    b. DATE
    c. TIME
    d. TIMESTAMP
    e. CURRENT_DATE

# SQL Commands

1. Data Definition Language ( DDL ):

   To make changes to the physical structure of any table residing inside a database

   - **CREATE, ALTER, DROP**

1. Data Manipulation Language ( DML ):

   For manipulation of data present in the table

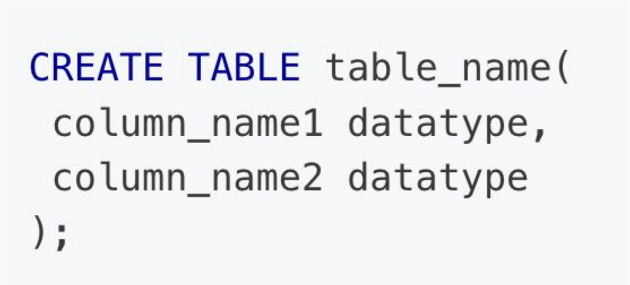
   - **INSERT, UPDATE, DELETE**

1. Data Query Language ( DQL ):

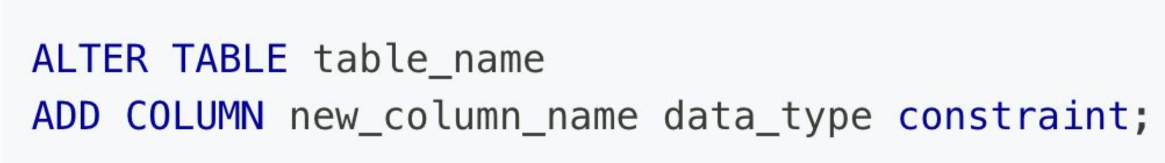
   Used to retrieve and fetch data from databases/tables based on certain conditions

   - **SELECT**

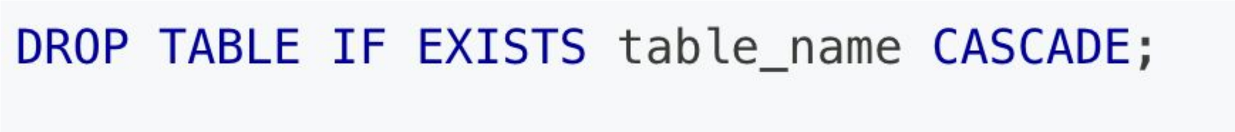
# Data Definition Language (DDL)



```
CREATE TABLE table_name(
 column_name1 datatype,
 column_name2 datatype
);
```

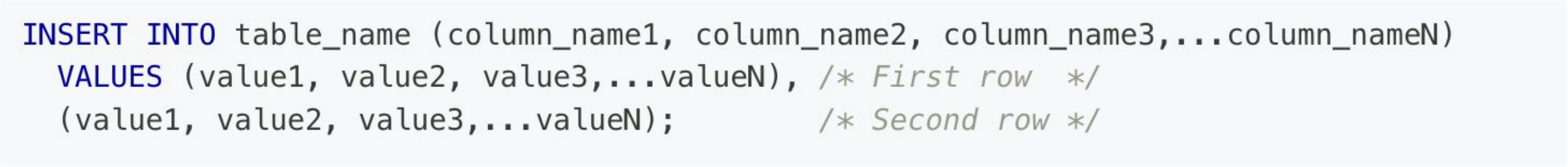


```
ALTER TABLE table_name
ADD COLUMN new_column_name data_type constraint;
```

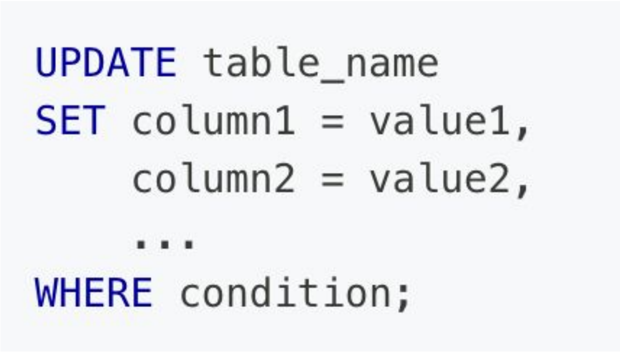


```
DROP TABLE IF EXISTS table_name CASCADE;
```

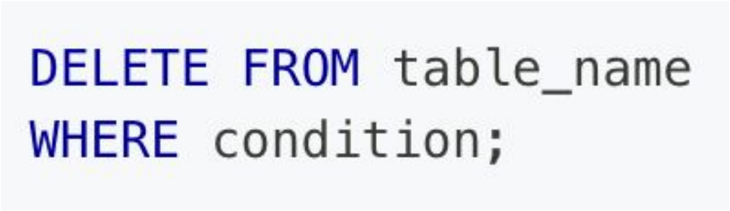
# Data Manipulation Language ( DML )



```
INSERT INTO table_name (column_name1, column_name2, column_name3,...column_nameN)
  VALUES (value1, value2, value3,...valueN), /* First row  */
  (value1, value2, value3,...valueN);        /* Second row */
```

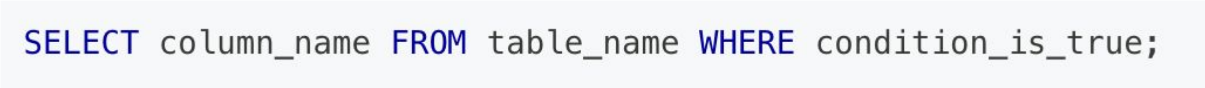


```
UPDATE table_name
SET column1 = value1,
    column2 = value2,
    ...
WHERE condition;
```

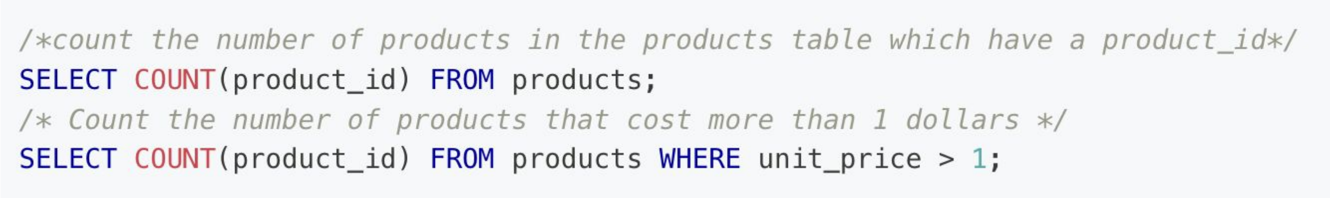


```
DELETE FROM table_name
WHERE condition;
```

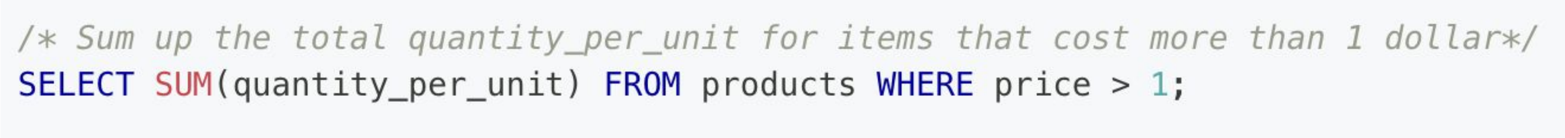
# Data Query Language ( DQL )



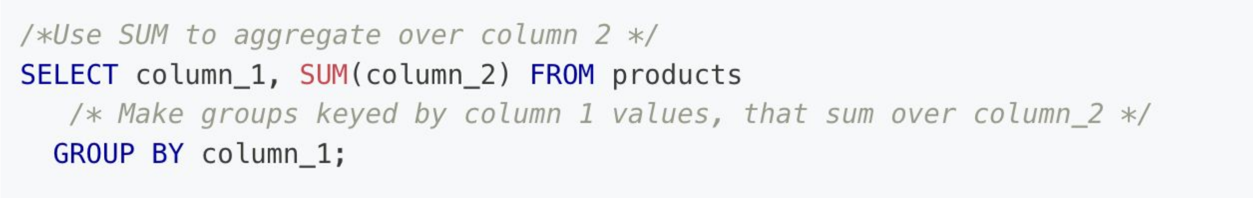
```sql
SELECT column_name FROM table_name WHERE condition_is_true;
```



```sql
/*count the number of products in the products table which have a product_id*/
SELECT COUNT(product_id) FROM products;
/* Count the number of products that cost more than 1 dollars */
SELECT COUNT(product_id) FROM products WHERE unit_price > 1;
```



```sql
/* Sum up the total quantity_per_unit for items that cost more than 1 dollar*/
SELECT SUM(quantity_per_unit) FROM products WHERE price > 1;
```



```sql
/*Use SUM to aggregate over column 2 */
SELECT column_1, SUM(column_2) FROM products
   /* Make groups keyed by column 1 values, that sum over column_2 */
  GROUP BY column_1;
```

# Sample Table Extended:

| student_id | full_name | sat_score | record_updated |
|---|---|---|---|
| 1 | John Maximo | 1200 | 2023-08-10 |
| 2 | Jane Smith | 1100 | 2023-09-05 |
| 3 | Maximo Rodriguez | 1250 | 2023-08-15 |
| 4 | Emily Maximo | 1300 | 2023-08-20 |
| 5 | Michael Johnson | 1150 | 2023-09-01 |
| 6 | Alex Maximo | 1350 | 2023-08-25 |
| 7 | Sarah Lee | 1400 | 2023-09-10 |
| 8 | Maximo Chen | 1450 | 2023-09-15 |
| 9 | David Kim | 1250 | 2023-09-20 |
| 10 | Samantha Maximo | 1200 | 2023-09-25 |

# Let’s test what we learnt so far!

**1.**

```
SELECT student_id,  full_name,  sat_score,
record_updated
FROM student
WHERE
 (
    student_id BETWEEN 1 and 5
    OR student_id = 8
    OR full_name LIKE '%Maximo%'
  )
AND sat_score NOT IN (1000, 1400);
```

| student_id | full_name | sat_score | record_updated |
|---|---|---|---|
| 1 | John Maximo | 1250 | 2023-08-20 |
| 2 | Maria Maximo | 1100 | 2023-09-05 |
| 3 | Michael Smith | 1200 | 2023-09-10 |
| 4 | Maximo Rodriguez | 1150 | 2023-09-15 |
| 8 | Samantha Maximo | 1300 | 2023-10-01 |

# 2.

```
SELECT COUNT(EmployeeID), City
FROM Employee_Info
GROUP BY City
HAVING COUNT(EmployeeID) > 2
ORDER BY COUNT(EmployeeID)
DESC;
```

| EmployeeID | City |
|---|---|
| 1 | New York |
| 2 | Los Angeles |
| 3 | Chicago |
| 4 | New York |
| 5 | New York |
| 6 | Los Angeles |

# Some guidelines for creating efficient database designs

1. Always add an **ID** column to each table and make that as primary key.

1. Each table cell should contain only a **single value**, and each column should have a **unique name**. This helps to eliminate duplicate data and simplify queries. ( 1NF )

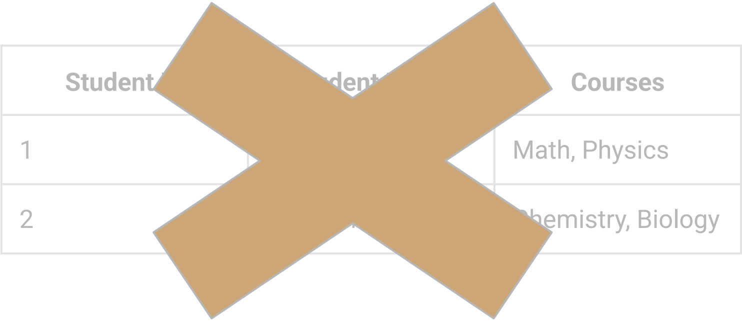
1. To eliminate redundant data, make sure that each non-key attribute be **dependent on the primary key**, and not on other columns. ( 2NF )

1. All non-key attributes are **independent** of each other (3NF)

For more info: https://www.geeksforgeeks.org/normal-forms-in-dbms/

# 1NF (First Normal Form)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attributes.
- The first normal form disallows the multi-valued attribute, composite attribute, and their combinations.

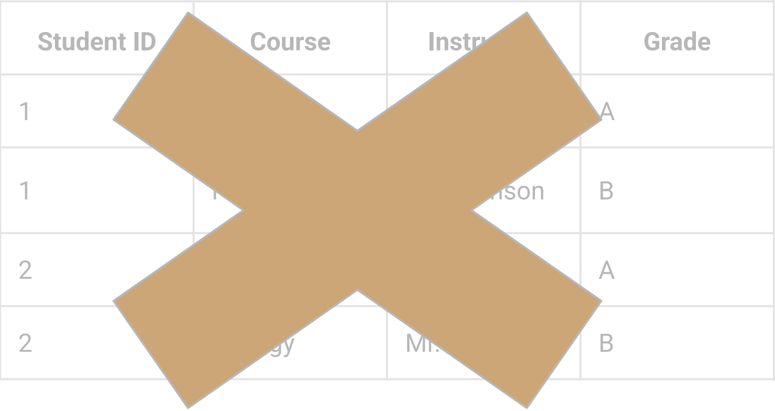


| Student I | dent | Courses |
|---|---|---|
| 1 | | Math, Physics |
| 2 | | hemistry, Biology |

| Student ID | Student Name | Course |
|---|---|---|
| 1 | John Doe | Math |
| 1 | John Doe | Physics |
| 2 | Jane Smith | Chemistry |
| 2 | Jane Smith | Biology |

# 2NF

- In the 2NF, relational DB must be in 1NF.
- In the second normal form, all non-key attributes are fully functional dependent on the primary key



| Student ID | Course | Instr | Grade |
|---|---|---|---|
| 1 | [illegible] | [illegible] | A |
| 1 | [illegible] | nson | B |
| 2 | [illegible] | [illegible] | A |
| 2 | gy | Mr. | B |

**Table:** student_courses

| Student ID | Course |
|---|---|
| 1 | Math |
| 1 | Physics |
| 2 | Chemistry |
| 2 | Biology |

**Table:** course_grades

| Course | Instructor | Grade |
|---|---|---|
| Math | Mr. Smith | A |
| Physics | Mr. Johnson | B |
| Chemistry | Mrs. Lee | A |
| Biology | Mr. Patel | B |

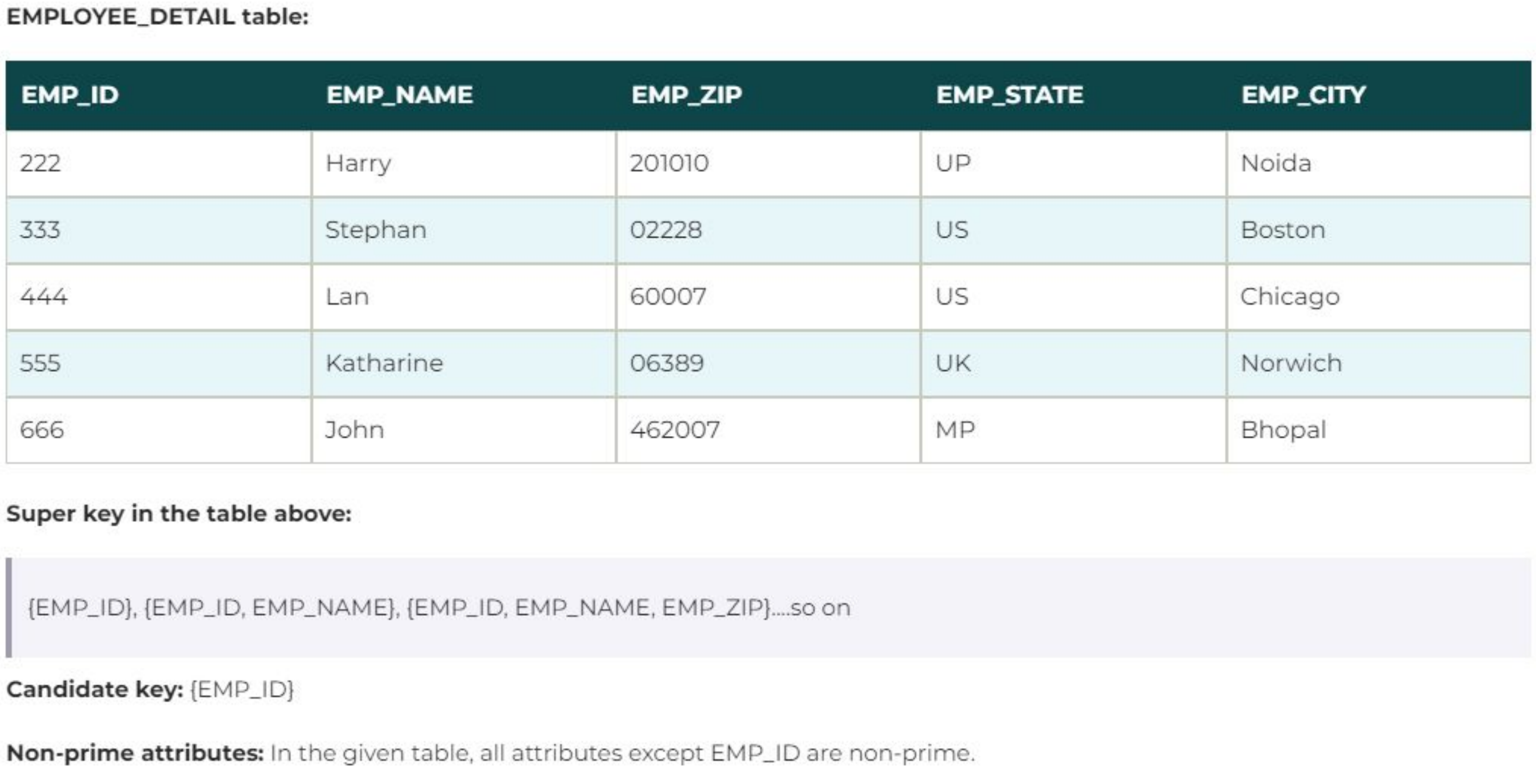
# 3NF

- ○ A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
- ○ 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
- ○ If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

# 3NF



**EMPLOYEE_DETAIL table:**

| EMP_ID | EMP_NAME | EMP_ZIP | EMP_STATE | EMP_CITY |
|---|---|---|---|---|
| 222 | Harry | 201010 | UP | Noida |
| 333 | Stephan | 02228 | US | Boston |
| 444 | Lan | 60007 | US | Chicago |
| 555 | Katharine | 06389 | UK | Norwich |
| 666 | John | 462007 | MP | Bhopal |

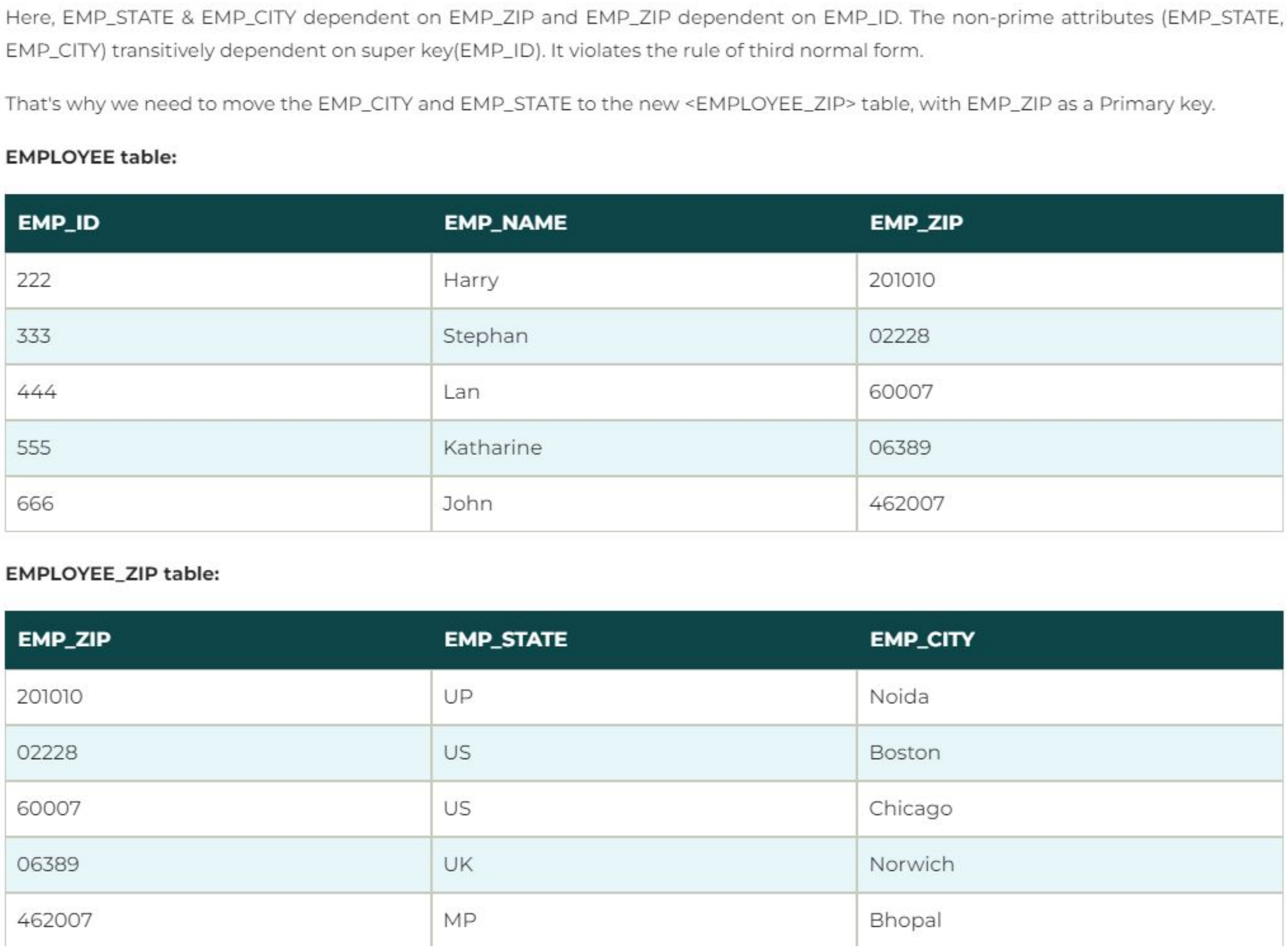
**Super key in the table above:**

> {EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so on

**Candidate key:** {EMP_ID}

**Non-prime attributes:** In the given table, all attributes except EMP_ID are non-prime.

# 3NF



Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

**EMPLOYEE table:**

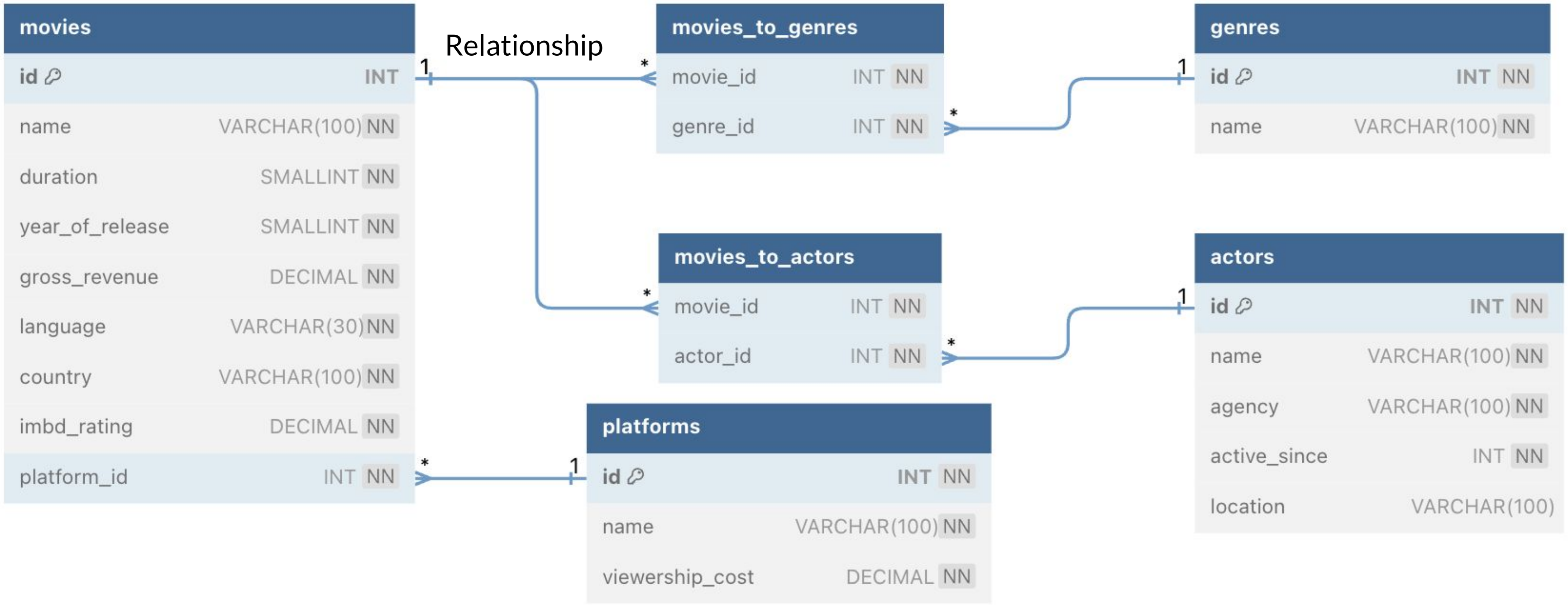
| EMP_ID | EMP_NAME | EMP_ZIP |
|---|---|---|
| 222 | Harry | 201010 |
| 333 | Stephan | 02228 |
| 444 | Lan | 60007 |
| 555 | Katharine | 06389 |
| 666 | John | 462007 |

**EMPLOYEE_ZIP table:**

| EMP_ZIP | EMP_STATE | EMP_CITY |
|---|---|---|
| 201010 | UP | Noida |
| 02228 | US | Boston |
| 60007 | US | Chicago |
| 06389 | UK | Norwich |
| 462007 | MP | Bhopal |

# ER Diagram ( Entity Relationship diagram )

Entity



Relationship

| movies | | |
|---|---|---|
| id | INT | |
| name | VARCHAR(100) | NN |
| duration | SMALLINT | NN |
| year_of_release | SMALLINT | NN |
| gross_revenue | DECIMAL | NN |
| language | VARCHAR(30) | NN |
| country | VARCHAR(100) | NN |
| imbd_rating | DECIMAL | NN |
| platform_id | INT | NN |

| movies_to_genres | | |
|---|---|---|
| movie_id | INT | NN |
| genre_id | INT | NN |

| genres | | |
|---|---|---|
| id | INT | NN |
| name | VARCHAR(100) | NN |

| movies_to_actors | | |
|---|---|---|
| movie_id | INT | NN |
| actor_id | INT | NN |

| actors | | |
|---|---|---|
| id | INT | NN |
| name | VARCHAR(100) | NN |
| agency | VARCHAR(100) | NN |
| active_since | INT | NN |
| location | VARCHAR(100) | |

| platforms | | |
|---|---|---|
| id | INT | NN |
| name | VARCHAR(100) | NN |
| viewership_cost | DECIMAL | NN |

# How do we write queries to fetch data from these tables?

Subqueries and Joins

# Lab Overview

**Part A**

1. Create tables
2. Alter a column in *movies* table
3. Insert data
   - Insert the data provided in the repository
   - Insert 2 additional records in each table
4. Write 3 SQL queries based on the questions

**Part B**

1. Write 7 more queries based on the questions