KNOWLEDGE INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION)

Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai

Kakapalayam (Po), Salem – 637 504

DEPARTMENT OF INFORMATION TECHNOLOGY

BE23CS405 – DATABASE MANAGEMENT SYSTEM LABORATORY MANUAL

Prepared by **Approved by**

REGULATION 2023

	KNOWLEDGE INSTITUTE OF TECHNOLOGY, SALEM **(An Autonomous Institution)**	
	Approved by AICTE, Affiliated to Anna University, Accredited by NAAC and NBA (B.E: Mech., ECE, EEE & CSE)	
	Kakapalayam (PO), Salem – 637 504	www.kiot.ac.in

Vision of the Information Technology Department	
To inculcate students with cutting-edge information technologies and enhance them into globally recognized engineers who are socially responsible citizen.	
Mission of the Information Technology Department	
M1	To deliver reliable education with innovative techniques, software updates and programming languages to the students.
M2	To nurture students to work seamlessly with changing industry requirements.
M3	To impart skills to meet the growing demands in industry.
M4	To shape the students as successful professionals with resilient ethics and society consciousness.
The **Program Educational Objectives (PEOs)** of the department represent major accomplishments that the graduates are expected to achieve after graduation. **The Graduates of Computer Science and Engineering will be able to**	
PEO1	Enable graduates to pursue higher education and research, or have a successful career in industries associated with Computer Science and Engineering, or as entrepreneurs.
PEO2	Ensure that graduates will have the ability and attitude to adapt to emerging technological changes
PEO3	Acquire leadership skills to perform professional activities with social consciousness
Program Specific Outcomes (PSOs)	
PSO 1	Analyse large volume of data and make business decisions to improve efficiency with different algorithms and tools
PSO 2	Have the capacity to develop web and mobile applications for real time scenarios
PSO 3	Provide automation and smart solutions in various forms to the society with Internet of Things

HOD/IT

COURSE CODE	BE23CS405	COURSE NAME	DATABASE MANAGEMENT SYSTEM		
REGULATION	R2023	SEMESTER	III	YEAR	II

COURSE OUTCOMES

Upon completion of this course, students will be able to		BLOOM'S TAXONOMY
CO1	Apply SQL for data management tasks.	L3 – Apply
CO2	Design a database using ER model and normalize the designed database.	L3 – Apply
CO3	Construct queries to handle transaction processing and maintain consistency of the database.	L3 – Apply
CO4	Identify the file organization techniques for an application.	L3 – Apply
CO5	Classify the advanced databases and find a suitable database for the given requirement.	L2 – Understand

CO-PO-PSO MAPPING

COs	POs												PSOs		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
CO1	2	2	3	1	3				2	1		1	3	2	
CO2	2	2	3	1	3				2	1		1	3	2	
CO3	2	2	3	1	3				2	1		1	3	2	
CO4	2	2	3	1	3				2	1		1	3	2	
CO5	2	2	3	1	3				2	1		2	3	2	
Avg	2.0	2.0	3.0	1.0	3.0				2.0	1.0		1.2	3.0	2.0	
1 – Low, 2 – Medium, 3 – High															

TABLE OF CONTENTS

Ex. No. 1 Date:	**Design a database and create the required tables**

AIM

To create a database for an online game store with the required tables.

STEPS

1. Identify the required tables and attributes

For an online game store where users can purchase games, we will need the following tables

- A **users** table that stores each user's ID, username, email, password, date of birth, and location
- A **games** table to store each game's ID, title, release date, developer ID, publisher ID, game genre *(i.e., category, e.g., adventure, action, puzzle, horror, etc)*, description, price, and rating
- Separate tables for the **developers** and **publishers** respectively, storing their ID, name, and location
- A **shopping cart** table that stores the games each user has added to the cart *(hint: you just need to store the user ID and the game ID)*

2. Write SQL commands to create the above tables

The SQL syntax to create a table is as follows

```
CREATE TABLE TableName (
      Attribute1 datatype,
      Attribute2 datatype,
      …
);
```

Replace the table name with the names of the tables identified in step 1 and the attribute names and datatypes with the attributes and suitable types identified in step 1 *(hint: you need to look up the datatypes available in MySQL and choose the appropriate types for each attribute)*.

3. Check whether the tables have been created correctly

To check whether a table has been created with the correct attributes, we can use the **desc** command. The SQL syntax is

```
desc TableName;
```

Replace the TableName with the names of the tables you created in step 2.

SAMPLE OUTPUT

Output:

```
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| UserID      | int         | YES  |     | NULL    |       |
| Username    | varchar(191)| YES  |     | NULL    |       |
| Email       | varchar(191)| YES  |     | NULL    |       |
| Password    | varchar(255)| YES  |     | NULL    |       |
| DateOfBirth | date        | YES  |     | NULL    |       |
| Location    | varchar(191)| YES  |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
```

RESULT

The required tables for the online games store were successfully created.

Ex. No. 2	Create database tables, add constraints and insert, update, and delete rows using DDL and DML
Date:	

AIM

- To create additional tables with constraints for the online game store database
- To add constraints to the existing tables
- To use DML commands to insert, update, and delete rows in the created tables

STEPS

1. **Create two new tables with primary key constraints defined during table creation**

Create the following tables with appropriate primary keys

- A transactions table with a unique ID for each transaction *(hint: this will be the primary key)*, the date and time, and the ID of the user that initiated the transaction
- A purchase history table that stores the games that are part of each transaction and the purchase price *(hint: you only need to store the transaction ID, game ID, and the purchase price, and the combination needs to be unique since multiple games can be bought in a single transaction and each game can be bought in multiple transactions)*

To set an attribute as a primary key during table creation, we use the following SQL syntax

```
CREATE TABLE TableName (
      Attribute1 datatype PRIMARY KEY,
      Attribute2 datatype,
      …
);
```

Note that the above syntax only works where the primary key consists of only one attribute. For composite primary keys, we use the following syntax:

```
CREATE TABLE TableName (
      Attribute1 datatype,
      Attribute2 datatype,
      …,
      PRIMARY KEY (Attribute1, Attribute2)
);
```

Following the syntax above, write SQL queries to create the transactions and purchase history tables with the appropriate primary keys.

2. **Add primary keys to existing tables**

While we've added primary keys to the new tables we've just created, we didn't add any constraints to the tables we created in exercise 1. Thankfully, there is a way to add constraints to tables after creating a table using the ALTER TABLE DDL command.

The SQL syntax to add a primary key constraint after table creation is as follows:

```
ALTER TABLE TableName ADD CONSTRAINT ConstraintName PRIMARY
KEY(AttributeName);
```

Note that you can add multiple attributes within the parentheses for composite keys.

As an exercise, add primary keys to all the tables we created in exercise 1 *(Users, Games, Developers, Publishers, Cart)*.

3. Add other constraints

3.1 UNIQUE and NOT NULL

While having a primary key constraint helps to uniquely identify each row in a table, we usually need other constraints as well to have a robust database design. For example, the username and email fields in the Users table must be unique. We can add it using the same syntax as above, just replacing PRIMARY KEY with the corresponding keyword – UNIQUE.

```
ALTER TABLE TableName ADD CONSTRAINT ConstraintName
UNIQUE(AttributeName);
```

Some column values should also never be empty, i.e., NOT NULL. To set a column as NOT NULL, we use the following syntax:

```
ALTER TABLE TableName MODIFY COLUMN ColumnName datatype NOT
NULL;
```

Use the syntax above to make the following fields in the Users table unique and not null

- Username
- Email

Also use the syntax to make the password field not null.

3.2 CHECK

For some columns, we need the values to satisfy some business constraints. E.g., in our games database, the price of a game should never be less than 0. To implement such a constraint, we use CHECK.

```
ALTER TABLE TableName ADD CONSTRAINT ConstraintName
CHECK(ColumnName > Value);
```

Instead of >, we can also use < or = depending on our need.

Using the syntax above, write a query to check whether the price of the game is greater than 0.

4. Insert rows into our database tables

Now that we have the tables set up with some basic constraints, it's time to finally start adding data to our database. In order to insert values into a table, we use the following syntax:

```
INSERT INTO TableName VALUES(Val1, Val2, Val3…);
```

The above is used if we want to provide values for all the columns in a table. If we only want to insert values into specific columns and let the other column values be NULL, then we use

```
INSERT INTO TableName(Col1, Col2, Col3…)
VALUES(Val1, Val2, Val3…);
```

Use the above syntax to insert a few rows into the tables we've created *(for this exercise, only insert into users, games, developers, and publishers)*.

To check whether the rows have been inserted correctly, use `SELECT * FROM TableName;`

5. **Update values already inserted**

In order to update some values that we've already inserted into the database, we use UPDATE queries with the following syntax:

```
UPDATE TableName SET ColName = Val;
```

Note that we usually combine the UPDATE query with a WHERE clause since most of the time, we don't want to update all the rows in our table, only some rows that satisfy a condition, so, we'll use it like this

```
UPDATE TableName SET ColName = Val WHERE Condition;
```

Use the above syntax to do the following

- Change the location of different users based on usernames or user IDs in your table
- Change the price of a game based on its game ID

To check whether the updates are successful, you can use SELECT queries.

6. **Delete rows from tables**

In order to delete rows, we use the following syntax:

```
DELETE FROM TableName WHERE Condition;
```

It is very important in a DELETE query to add the WHERE clause since without it, all the rows will be deleted (which is something we 99.99% of the time don't want to happen!).

Use the above syntax to delete an entry from the Developers table *(hint: first add a few extra rows and then delete a row based on the developer ID).*

Use SELECT queries to check whether the entry is deleted.

SAMPLE OUTPUT

1. **Table creation with primary key**

```
+-----------------+------+------+-----+---------+-------+
| Field           | Type | Null | Key | Default | Extra |
+-----------------+------+------+-----+---------+-------+
| TransactionID   | int  | NO   | PRI | NULL    |       |
| TransactionDate | date | YES  |     | NULL    |       |
| UserID          | int  | YES  |     | NULL    |       |
+-----------------+------+------+-----+---------+-------+
```

2. **Adding primary key to existing table**

```
+-------------------+-------------+------+-----+---------+-------+
| Field             | Type        | Null | Key | Default | Extra |
+-------------------+-------------+------+-----+---------+-------+
| DeveloperID       | int         | NO   | PRI | NULL    |       |
| DeveloperName     | varchar(191) | YES  |     | NULL    |       |
| DeveloperLocation | varchar(191) | YES  |     | NULL    |       |
+-------------------+-------------+------+-----+---------+-------+
```

3. **Inserting values**

```
+--------+----------+---------------+----------+-------------+----------+
| UserID | Username | Email         | Password | DateOfBirth | Location |
+--------+----------+---------------+----------+-------------+----------+
|      1 | test     | test@test.com | 123456   | 2000-01-01  | India    |
+--------+----------+---------------+----------+-------------+----------+
```

4. **Update values**

```
+--------+----------+---------------+----------+-------------+-----------+
| UserID | Username | Email         | Password | DateOfBirth | Location  |
+--------+----------+---------------+----------+-------------+-----------+
|      1 | test     | test@test.com | 123456   | 2000-01-01  | Australia |
+--------+----------+---------------+----------+-------------+-----------+
```

5. **Delete values**

DeveloperID	DeveloperName	DeveloperLocation
500	Nintendo	Japan
501	Test	USA

DeveloperID	DeveloperName	DeveloperLocation
500	Nintendo	Japan

RESULT

The tables were successfully created, the constraints implemented, and the insert, update, and delete operations were performed successfully.

Ex. No. 3	**Create a set of tables, add foreign key constraints and incorporate referential integrity**
Date:	

AIM

To add foreign keys and enforce referential integrity in the database.

STEPS

1. **Identify required foreign keys**

The foreign key constraint means that an attribute that is set as a foreign key in one table can only have values that are present for the same attribute in another table. E.g., a game can only have developers that are present in the developer table.

For our games database, we need at least the following foreign keys

- The developer ID and publisher ID in the Games table should refer to the corresponding IDs in the Developers and Publishers tables respectively
- The UserID and GameID in the Cart table should reference the UserID from the Users table and the GameID from the Games table
- The UserID in the transactions table should reference the UserID in the users table
- The TransactionID and GamesID in the PurchaseHistory table should refer to the corresponding IDs in their respective tables

2. **Decide what should happen when data that is referenced by another table via foreign keys is deleted**

The two options that we have are to either delete all the data that makes reference to the original data (CASCADE), or simply mark the foreign key values as NULL (SET NULL). E.g., when a developer is deleted, should all the games developed by them also be deleted? Or, should the value of developer ID in the games table simply be set to NULL? What happens

when a game is deleted, it should obviously be deleted from the carts of anyone who currently has it in their cart, but what about from purchase history? These are decisions we have to make when designing a database.

Generally, we need to be careful when using ON DELETE CASCADE since it could result in data loss.

3. Write SQL queries to add foreign keys and enforce referential integrity

After the above steps, we can write queries to implement foreign keys using the syntax

```
ALTER TABLE TableName ADD CONSTRAINT ConstraintName FOREIGN KEY(ColName) REFERENCES SourceTable(ColName) ON DELETE CASCADE (or SET NULL);
```

Write queries based on the above syntax to implement the foreign keys identified in steps 1 & 2.

4. Test the foreign keys

Try to insert values into foreign key columns that don't have a matching value in the table they reference, you should get an error. Next, insert correct values and then deleting them to see the functioning of the ON DELETE referential integrity constraints.

SAMPLE OUTPUT

1. Trying to insert incorrect values

```
ERROR 1452 (23000) at line 116: Cannot add or update a child row: a foreign key constraint fails (`sandbox_db`.`games`, CONSTRAINT `Games_FK_1` FOREIGN KEY (`DeveloperID`) REFERENCES `developers` (`Deve
```

2. ON DELETE SET NULL in action

```
+-------------+---------------+-------------------+
| DeveloperID | DeveloperName | DeveloperLocation |
+-------------+---------------+-------------------+
|         500 | Nintendo      | Japan             |
|         501 | GameFreak     | Japan             |
+-------------+---------------+-------------------+
+--------+-----------------+-------------+-------------+-------------+-----------+-----------------------------------------------------+-----------+--------+
| GameID | Title           | ReleaseDate | DeveloperID | PublisherID | Genre     | Description                                         | BasePrice | Rating |
+--------+-----------------+-------------+-------------+-------------+-----------+-----------------------------------------------------+-----------+--------+
|    101 | Super Mario Bros | 1980-01-01  |         500 |         700 | Platformer | Play as Mario and rescue Princess Peach from Bowser |     40.00 |  10.00 |
|    102 | Pokemon Black    | 2008-01-01  |         501 |         700 | RPG       | Best game in the Pokemon franchise                  |     50.00 |  10.00 |
+--------+-----------------+-------------+-------------+-------------+-----------+-----------------------------------------------------+-----------+--------+
+-------------+---------------+-------------------+
| DeveloperID | DeveloperName | DeveloperLocation |
+-------------+---------------+-------------------+
|         500 | Nintendo      | Japan             |
+-------------+---------------+-------------------+
+--------+-----------------+-------------+-------------+-------------+-----------+-----------------------------------------------------+-----------+--------+
| GameID | Title           | ReleaseDate | DeveloperID | PublisherID | Genre     | Description                                         | BasePrice | Rating |
+--------+-----------------+-------------+-------------+-------------+-----------+-----------------------------------------------------+-----------+--------+
|    101 | Super Mario Bros | 1980-01-01  |         500 |         700 | Platformer | Play as Mario and rescue Princess Peach from Bowser |     40.00 |  10.00 |
|    102 | Pokemon Black    | 2008-01-01  |        NULL |         700 | RPG       | Best game in the Pokemon franchise                  |     50.00 |  10.00 |
+--------+-----------------+-------------+-------------+-------------+-----------+-----------------------------------------------------+-----------+--------+
```

RESULT

The foreign key constrains were successfully implemented and referential integrity was incorporated into the database.

Ex. No. 4	Query the database tables using different 'WHERE' clauses and implement aggregate functions
Date:	

AIM

- To query the database using different WHERE clauses
- To use aggregate functions

STEPS

1. **Use WHERE clauses to get a subset of rows matching the condition provided from a table**

We use the following syntax to get specific rows from a table that match our condition

```
SELECT * FROM TableName WHERE Condition;
```

The condition can be anything – a column value being less than, greater than, or equal to a certain value.

Write SQL queries with WHERE clauses to do the following:

- Get the username and location of all users from Australia
- Get a list of action games from the games table
- Get a list of games where the price is greater than Rs. 500
- Get a list of games with "Resident Evil" in their title (Hint: you should make sure you have a few by using INSERT queries and use LIKE to check the condition)
- Get the title, rating, and price of adventure games that have a rating of at least 7 (to combine conditions, you can use AND, e.g., SELECT * FROM Games WHERE Genre = 'Strategy' AND Price < 300; (this query selects strategy games that are cheaper than Rs. 300)

2. **Use aggregate functions and GROUP BY to retrieve data from the database**

We can use aggregate functions to get computed values from the database.

To get the number of rows, we can use

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

We can also combine the above with a WHERE clause to find the number of rows that satisfy a certain condition.

```
SELECT COUNT(*) FROM TableName WHERE Condition;
```

We can get the minimum and maximum values of a column and the average of a column as follows

```
SELECT MIN(ColName) FROM TableName;
SELECT MAX(ColName) FROM TableName;
SELECT AVG(ColName) FROM TableName;
```

All the above can be combined with a WHERE clause as well.

We can use GROUP BY to get aggregated data.

```
SELECT COUNT(*) FROM TableName GROUP BY ColName;
```

The above will give the number of rows matching each distinct value in the column name provided.

Write queries to do the following

- Find the total number of games in the database
- Find the total number of games in each genre available in the database *(Hint: you have to use GROUP BY Genre at the end of your select statement)*
- Find the lowest rating and the highest rating of games in the database *(Hint: use MIN and MAX)*
- Find the average price of action games in the database

SAMPLE OUTPUT

1. **WHERE Clauses**

```
+----------+-----------+
| Username | Location  |
+----------+-----------+
| test     | Australia |
+----------+-----------+
+-----------------------------+------------------+
| Title                       | Genre            |
+-----------------------------+------------------+
| Grand Theft Auto V          | Action-Adventure |
| Assassin's Creed Valhalla   | Action RPG       |
| Resident Evil 4             | Action Horror    |
+-----------------------------+------------------+
```

2. **Aggregate Functions**

```
| No. of Games |
+--------------+
|           11 |
+--------------+
+-----------------+----------+
| Genre           | COUNT(*) |
+-----------------+----------+
| RPG             |        2 |
| Survival Horror |        4 |
+-----------------+----------+
+-------------+
| MIN(Rating) |
+-------------+
|        8.80 |
```

RESULT

The database was queried successfully using WHERE clauses and aggregate functions.

Ex. No. 5	Query the database tables and explore sub-queries and simple join operations
Date:	

AIM

To query the database using sub-queries and joins.

STEPS

1. **Using sub-queries**

In the previous exercise, we used aggregate functions to find the lowest and highest ratings available in our database. But, we weren't able to get the titles of the games with those ratings. In order to do that, we make user of sub-queries.

```
SELECT * FROM TableName WHERE ColName = (SELECT MIN(ColName)
FROM
TableName);
```

The above is just an example of how sub-queries can be used. They can also be used to query data from multiple tables, e.g.,

```
SELECT * FROM Table1 WHERE ColName = (SELECT ColName FROM
Table2
WHERE Condition);
```

Using the syntax above, write queries to do the following:

- Get the title of the game with the lowest and highest ratings
- Get the titles of the games published by Nintendo *(hint: the publisher name is in the publishers table and the ID corresponding to it is also in the same table, but referenced as a foreign key in the games table)*
- **[Hard]** Get the title of the game with the maximum number of sales *(hint: this requires multiple sub-queries, the first to get the sales of each game from the purchase history table using COUNT(*) and GROUP BY, next to select the MAX of that from the previous query, next to get the GameID from the purchase history table where the sales matches the sales, and finally to get the game title from the games table based on the ID)*

2. **Using joins to retrieve data from multiple tables**

Since our data is split across multiple tables, we need to use joins to retrieve data from multiple tables. The general syntax is as follows

```
SELECT Col1, Col2, … FROM Table1 INNER (or LEFT or RIGHT) JOIN
Table2
ON Condition;
```

The condition is usually a foreign key value in Table2 matching the primary key value of Table1. INNER JOIN only joins tables where there is a matching entry for each value in both

tables, LEFT JOIN keeps all the values of the left table and if there is no corresponding value on the right table, it sets it to NULL (and RIGHT JOIN does the reverse).

Using the syntax above, write queries to do the following:

- Get a list of game titles along with the publisher names *(hint: you have to join the games table and the publishers table)*
- Get a list of all publishers and their games, even if they haven't published any games *(hint: Just insert some new publishers into the publishers table, and use left join)*
- Get the title and sales of each game in the database that has at least one sale *(hint: here, instead of joining two tables, you'll join the games table with the result of a sub-query to retrieve the sales of each game based on their ID)*

SAMPLE OUTPUT

1. **Sub-queries**

```
+-----------------------------+--------+
| Title                       | Rating |
+-----------------------------+--------+
| Assassin's Creed Valhalla   |   8.80 |
+-----------------------------+--------+
+------------------+--------+
| Title            | Rating |
+------------------+--------+
| Super Mario Bros |  10.00 |
| Pokemon Black    |  10.00 |
+------------------+--------+
+------------------+
| Title            |
+------------------+
| Super Mario Bros |
| Pokemon Black    |
+------------------+
+------------------+
| Title            |
+------------------+
| Super Mario Bros |
+------------------+
```

2. **Joins**

```
+---------------------+---------------------------+
| PublisherName       | Title                     |
+---------------------+---------------------------+
| Nintendo            | Pokemon Black             |
| Nintendo            | Super Mario Bros          |
| Electronic Arts     | NULL                      |
| Valve Corporation   | Half-Life 2               |
| CD Projekt          | The Witcher 3: Wild Hunt  |
| Rockstar Games      | Grand Theft Auto V        |
| Ubisoft Entertainment | Assassin's Creed Valhalla |
| Capcom              | Resident Evil 7: Biohazard |
| Capcom              | Resident Evil 4           |
| Capcom              | Resident Evil 3: Nemesis  |
| Capcom              | Resident Evil 2           |
| Capcom              | Resident Evil             |
+---------------------+---------------------------+
+---------------------------+-------+
| Title                     | Sales |
+---------------------------+-------+
| Super Mario Bros          |     3 |
| Pokemon Black             |     1 |
| Half-Life 2               |     1 |
| The Witcher 3: Wild Hunt  |     1 |
| Grand Theft Auto V        |     1 |
| Assassin's Creed Valhalla |     1 |
+---------------------------+-------+
```

RESULT

The sub-queries and joins were written successfully to retrieve complex data from multiple tables in the database.

Ex. No. 6	**Write SQL triggers for insert, update, and delete operations**
Date:	

AIM

To write triggers to update or log information in the database

- When a row is inserted
- Before a row is deleted
- When a row is updated

STEPS

1. **Initial setup**

Add a new column to the Games table called "CurrentPrice" using the query `ALTER TABLE Games ADD COLUMN CurrentPrice decimal(10, 2) AFTER Price;`

Set the value of current price to be the same as the value of price using the query `UPDATE Games SET CurrentPrice = Price;`

Create a new table called Offers with the GameID as a primary key and the percentage off as another column using

```
CREATE TABLE Offers (
      GameID int PRIMARY KEY,
      PercentOff int NOT NULL,
      FOREIGN KEY (GameID) REFERENCES Games(GameID)
);
```

Use DELIMITER $$ before the triggers and $$ at the end of each trigger, and finally, set the delimiter back to ; (Ensure that you don't run any other queries in between your triggers)

2. **Create the required triggers**

Triggers are used to automatically run some queries when a different query is run. They are used to update other tables or log information usually upon INSERT, UPDATE, or DELETE operations on a table. The triggers can be set to run BEFORE or AFTER an operation on a table. The general syntax is as follows:

```
DELIMITER $$
CREATE TRIGGER TriggerName
AFTER (or BEFORE) UPDATE (or INSERT or DELETE) ON TableName
FOR EACH ROW
BEGIN
      Query1
      Query2
```

```
    ...
END;
$$
DELIMITER ;
```

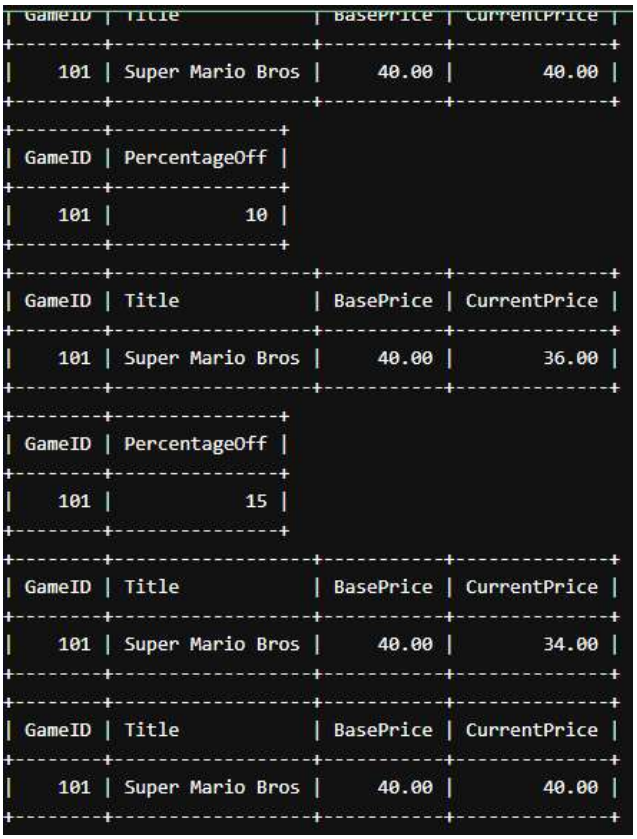
Write triggers to do the following

- When a new offer is inserted into the Offers table, the current price of the game should be updated based on the percentage off
- When an offer is updated, the current price should be updated accordingly
- When an offer is deleted, the current price should be set to the price of the game

3. Test the triggers

Write some INSERT, UPDATE, and DELETE queries on the Offers table to test whether the triggers are working (you also need to write SELECT queries on the Games table to verify).

SAMPLE OUTPUT



```
| GameID | Title           | BasePrice | CurrentPrice |
+--------+-----------------+-----------+--------------+
|    101 | Super Mario Bros |     40.00 |        40.00 |
+--------+-----------------+-----------+--------------+
+--------+---------------+
| GameID | PercentageOff |
+--------+---------------+
|    101 |            10 |
+--------+---------------+
+--------+-----------------+-----------+--------------+
| GameID | Title           | BasePrice | CurrentPrice |
+--------+-----------------+-----------+--------------+
|    101 | Super Mario Bros |     40.00 |        36.00 |
+--------+-----------------+-----------+--------------+
+--------+---------------+
| GameID | PercentageOff |
+--------+---------------+
|    101 |            15 |
+--------+---------------+
+--------+-----------------+-----------+--------------+
| GameID | Title           | BasePrice | CurrentPrice |
+--------+-----------------+-----------+--------------+
|    101 | Super Mario Bros |     40.00 |        34.00 |
+--------+-----------------+-----------+--------------+
+--------+-----------------+-----------+--------------+
| GameID | Title           | BasePrice | CurrentPrice |
+--------+-----------------+-----------+--------------+
|    101 | Super Mario Bros |     40.00 |        40.00 |
+--------+-----------------+-----------+--------------+
```

RESULT

The triggers were successfully written and executed.

Ex. No. 7	Create views and indexes for database tables
Date:	

AIM

- To create a view to simplify data retrieval from the games store database
- To create indexes on frequently searched columns to improve query performance

STEPS

1. **Creating a view**

The syntax to create a view for frequently run queries is as follows

```
CREATE VIEW ViewName AS
SELECT Query
```

The SELECT query can be any of the queries we've already written.

Using the syntax above, create views for the following

- List of developers and the games that they've developed
- List of games and their sales *(hint: you can reuse the query you wrote for this in exercise 5)*

Test the above views by using a simple SELECT query (instead of SELECT * FROM TableName; you'll use `SELECT * FROM ViewName;` since the view is treated as a virtual table).

2. **Indexing columns to improve query performance**

Indexes are used to improve the performance of queries that operate on the columns that are indexed. The syntax to add an INDEX on a column is as follows:

```
CREATE INDEX IndexName ON TableName(ColName);
```

Note that for text columns, you also need to specify a size alongside the column name within parentheses. We can show the list of indexes present on a table using the following syntax

```
SHOW INDEXES FROM TableName;
```

which will provide a list of all indexes including the primary keys and foreign keys.

Using the syntax above, write queries to do the following

- Index the genre column since a common way of searching for games is by genre
- Create indexes on the names of the developers and publishers in their respective tables
- View the indexes on each table using the `SHOW INDEXES` statement
- SELECT rows on the games, developers, and publishers tables with one of the indexed columns in the WHERE clause

SAMPLE OUTPUT

1. Views

DeveloperName	Title
Nintendo	Super Mario Bros
Valve	Half-Life 2
CD Projekt Red	The Witcher 3: Wild Hunt
Rockstar Games	Grand Theft Auto V
Ubisoft	Assassin's Creed Valhalla
Capcom	Resident Evil
Capcom	Resident Evil 2
Capcom	Resident Evil 3: Nemesis
Capcom	Resident Evil 4
Capcom	Resident Evil 7: Biohazard

Title	Sales
Super Mario Bros	3
Pokemon Black	1
Half-Life 2	1
The Witcher 3: Wild Hunt	1
Grand Theft Auto V	1
Assassin's Creed Valhalla	1

2. Indexes

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Car
games	0	PRIMARY	1	GameID	A	
games	1	Games_FK_1	1	DeveloperID	A	
games	1	Games_FK_2	1	PublisherID	A	
games	1	Games_IDX_1	1	Genre	A	

Table	Non_unique	Key_name	Seq_in_index	Column_name	Col
developers	0	PRIMARY	1	DeveloperID	A
developers	1	Developers_IDX_1	1	DeveloperName	A

RESULT

Ease of data retrieval and query performance were successfully improved by using views and indexes.

Ex. No. 8	Execute complex database transactions and realize DCL and TCL commands
Date:	

AIM

- To use database transactions along with TCL to handle the purchase of a game
- To use DCL commands to control access to the database

STEPS

1. **Initial setup and prerequisites**

Before following the steps below, ensure that the following tables exist on your database and that the Cart table has some entries.

- Transactions*(TransactionID, DateAndTime, UserID) [Set TransactionID to AUTO_INCREMENT]
- PurchaseHistory(TransactionID, GameID, SalePrice)
- Cart(UserID, GameID)

**Note: Please note that the table with the name "Transactions" is different from the DBMS concept of transactions that we are learning in this exercise.*

2. **Creating a transaction for the process of purchasing a game**

The overall process we need to follow for this transaction is as follows

- Start the transaction *(DBMS concept)*
- Insert a new row in the transactions table along with the transaction date and user ID of the user who made the transaction *(assume the UserID is 1 for our exercise)*
- Get the last inserted transaction ID (this is needed because transaction ID is automatically generated due to AUTO INCREMENT
- Insert new rows into the PurchaseHistory table for each game in the user's cart along with the transaction ID and the current price of each game (retrieved from the games table)
- Delete the above entries from the Cart table
- Commit the transaction *(DBMS concept)*

In order to start a transaction, we need to use the SQL command START TRANSACTION;

This ensures that if any of the queries in-between the START TRANSACTION; and the eventual COMMIT; command fail, then the database is rolled back to a previous consistent state.

The last inserted ID can be retrieved and stored in a variable using the syntax

```
SET @ID = 0;
SELECT LAST_INSERT_ID() INTO @ID;
```

The @ID variable can then be passed as a value into other queries.

In order to insert rows into one table based on values read by a SELECT statement in another table, we use the following syntax

```
INSERT INTO Table1(Col1, Col2, …) SELECT Col1, Col2, … FROM Table2 WHERE Condition;
```

We can also add a variable to the query – it'll simply attach the same value to each row.

```
INSERT INTO Table1(Col1, Col2, …) SELECT @Variable1, Col1, Col2… FROM Table2 WHERE Condition;
```

The above syntax will be needed to copy the data from the cart to the purchase history table while also attaching the transaction ID. Note that the SELECT sub-query can also include its own sub-queries and joins *(hint: you actually need to join both the cart table and the games table in order to get the sale price, which will be current price of the game at the time of the transaction)*.

In order to get the current date, instead of manually typing a date like 'YYYY-MM-DD', you can simply use the NOW() function.

Don't forget to add a COMMIT; statement at the end of the transaction.

Put all the above steps together to create a transaction for the cart checkout process. Test whether the transaction worked successfully by using appropriate SELECT statements on the relevant tables before and after the transaction.

3. Using DCL commands to control access to the database

In order to create a new user on the database, we use

```
CREATE USER 'user'@'localhost' IDENTIFIED BY 'password';
```

Then, we can use GRANT and REVOKE statements in combination with common query operations like SELECT, INSERT, UPDATE, and DELETE to grant or revoke permissions to execute those queries to the different users.

```
GRANT SELECT (or INSERT or UPDATE or DELETE) ON TableName TO 'user'@'localhost';
```

```
REVOKE SELECT (or INSERT or UPDATE or DELETE) ON TableName FROM 'user'@'localhost';
```

Using the syntax above, write queries to perform the following operations

- Create a new user by the name of 'customer' on localhost and set the password to just 'password
- Grant only SELECT privileges to this new user and revoke all other permissions (INSERT, UPDATE, DELETE) on the Games table

SAMPLE OUTPUT

RESULT

The complex transaction for the game purchase operation and the DCL commands were executed successfully.

Ex. No. 9	Write a program to implement B+ tree
Date:	

AIM

To write a C program to implement B+ Tree.

STEPS

Step 1: Create a Node

Sequence

1. Allocate memory for a new node.
2. Set whether it's a leaf.
3. Set number of keys to 0.
4. Initialize child pointers to NULL.
5. Initialize next pointer to NULL.

PSEUDO CODE

```
FUNCTION create_node(is_leaf):
    node ← allocate memory for BPlusTreeNode
    node.is_leaf ← is_leaf
    node.num_keys ← 0
    node.next ← NULL
    FOR i FROM 0 TO ORDER-1:
        node.children[i] ← NULL

    RETURN node
```

Step 2: Split a Child

Sequence

1. Find midpoint mid of the full child.
2. Create a new node (new_child) with the same leaf status.
3. Move the last mid keys from child to new_child.
4. If not a leaf, move corresponding children pointers too.
5. Reduce the original child's key count to mid.
6. Shift parent's children and keys to make room for new_child.
7. Insert new_child into parent's children array.
8. Copy middle key from child to parent's keys.

PSEUDO CODE

```
FUNCTION split_child(parent, index, child):
    mid ← (ORDER - 1) / 2
    new_child ← create_node(child.is_leaf)
    new_child.num_keys ← mid

    FOR i FROM 0 TO mid-1:
        new_child.keys[i] ← child.keys[i + mid + 1]
```

```
    IF child.is_leaf == FALSE:
        FOR i FROM 0 TO mid:
            new_child.children[i] ← child.children[i + mid + 1]
    child.num_keys ← mid

    FOR i FROM parent.num_keys DOWNTO index+1:
        parent.children[i+1] ← parent.children[i]
        parent.keys[i] ← parent.keys[i-1]

    parent.children[index+1] ← new_child
    parent.keys[index] ← child.keys[mid]
    parent.num_keys ← parent.num_keys + 1
```

Step 3: Insert into Non-Full Node

Sequence

1. Find position i to insert the key (search backward from the last key).
2. If the node is a leaf:
    - Shift larger keys to the right.
    - Insert the new key at position i+1.
3. If the node is not a leaf:
    - Find the child i+1 to insert into.
    - If that child is full:
        - Split it.
        - If the key is greater than the promoted key, move to next child.
    - Recursively insert into that child.

PSEUDO CODE

```
FUNCTION insert_non_full(node, key):
    i ← node.num_keys - 1

    IF node.is_leaf:
        WHILE i >= 0 AND key < node.keys[i]:
            node.keys[i+1] ← node.keys[i]
            i ← i - 1
        node.keys[i+1] ← key
        node.num_keys ← node.num_keys + 1

    ELSE:
        WHILE i >= 0 AND key < node.keys[i]:
            i ← i - 1
        i ← i + 1

        IF node.children[i].num_keys == ORDER - 1:
            split_child(node, i, node.children[i])
            IF key > node.keys[i]:
                i ← i + 1
        insert_non_full(node.children[i], key)
```

Step 4: Insert into Tree

Sequence

1. If root is full:
    - Create new node s (not a leaf).
    - Set its first child to root.
    - Split root.
    - Insert into the correct child of s.
    - Update root to be s.
2. Else, insert directly into root.

```
FUNCTION insert(root_ref, key):
    r ← root_ref
    IF r.num_keys == ORDER - 1:
        s ← create_node(FALSE)
        root_ref ← s
        s.children[0] ← r
        split_child(s, 0, r)
        insert_non_full(s, key)
    ELSE:
        insert_non_full(r, key)
```

Step 5: Search in B+ Tree

Sequence

1. Find smallest index i where key ≤ node.keys[i].
2. If key == node.keys[i], return FOUND.
3. If node is a leaf, return NOT FOUND.
4. Else, recurse into the i-th child.

PSEUDO CODE

```
FUNCTION search(node, key):
    i ← 0
    WHILE i < node.num_keys AND key > node.keys[i]:
        i ← i + 1

    IF i < node.num_keys AND key == node.keys[i]:
        RETURN TRUE

    IF node.is_leaf:
        RETURN FALSE
    ELSE:
        RETURN search(node.children[i], key)
```

Step 6: Traverse Tree (Inorder)

Sequence

1. For each key in the node:
    - If not a leaf, recurse into the left child before printing the key.
    - Print the key.
2. After last key, recurse into rightmost child.

PSEUDO CODE

```
FUNCTION traverse(node):
    FOR i FROM 0 TO node.num_keys-1:
        IF node.is_leaf == FALSE:
            traverse(node.children[i])
        PRINT node.keys[i]

    IF node.is_leaf == FALSE:
        traverse(node.children[node.num_keys])
```

SAMPLE OUTPUT

Traversal: 5 6 7 10 12 17 20 30

Search for 6: Found

Search for 15: Not Found

RESULT

The C program to implement B+ tree was executed successfully.

Ex. No. 10 Date:	Create a document database using NoSQL

AIM

To perform basic CRUD operations on a simple document (No SQL) database using MongoDB.

STEPS

1. **Setup MongoDB**

   - Install MongoDB or use a cloud-based MongoDB instance such as MongoDB Atlas.
   - Install a MongoDB client (like mongo shell, Compass, or any supported driver, e.g., Node.js or Python driver).

2. **Connect to MongoDB**

Using the MongoDB client, connect to the server using:

```
mongo
```

For MongoDB Atlas or a remote server, connect with:

```
mongo "mongodb+srv://<username>:<password>@cluster-url.mongodb.net/test"
```

3. **Create a Database**

MongoDB automatically creates a database if it doesn't exist.

```
use DatabaseName;
```

4. **Create a Collection**

Collections are like tables in a relational database.

```
db.createCollection("CollectionName");
```

5. **Insert Data (Create Operation)**

Insert a document (record) into the collection using the following syntax.

```
db.CollectionName.insertOne({
      "Attribute1": "Value1",
      "Attribute2": "Value2",
      …
});
```

6. **Retrieve Data (Read Operation)**

Fetch a document from the collection.

```
db.CollectionName.findOne({ "Attribute": "Value" });
```

7. **Update Data (Update Operation)**

Update a field in the document.

```
db.CollectionName.updateOne(
      { "ConditionAttribute": "ConditionValue" },
      { $set: { "UpdatedAttribute": "UpdatedValue" } }
);
```

8. **Delete Data (Delete Operation)**

Remove a document from the collection.

```
db.CollectionName.deleteOne({ "Attribute": "Value" });
```

SAMPLE OUTPUT

```
{ "ok" : 1 }
{
        "acknowledged" : true,
        "insertedId" : ObjectId("689d7e3de7a2d314d7d91c81")
}
{
        "_id" : ObjectId("689d7e3de7a2d314d7d91c81"),
        "gameID" : 101,
        "title" : "Control",
        "genre" : "Action Adventure",
        "price" : "80"
}
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
{
        "_id" : ObjectId("689d7e3de7a2d314d7d91c81"),
        "gameID" : 101,
        "title" : "Control",
        "genre" : "Action Adventure",
        "price" : 70
}
```

RESULT

The CRUD operations were performed successfully on a simple NoSQL (MongoDB) database.