**2. A. Distributed Database Design and Implementation**

**Aim:**

To configure MongoDB's replica sets and sharding, then perform CRUD operations to test data distribution, high availability, and fault tolerance in a distributed database environment.

**Procedure:**

const { MongoClient } = require('mongodb');

// Function to set up the database and perform operations async function run() { const uri = "mongodb://localhost:27025"; // Connect to the mongos router instance const client = new MongoClient(uri);

try { await client.connect(); console.log("Connected to MongoDB via mongos router on port 27025");

const adminDb = client.db("admin");

// Step 1: Enable Sharding on Database console.log("Enabling sharding on 'myDatabase'..."); await adminDb.command({ enableSharding: "myDatabase" }); console.log("Sharding enabled on 'myDatabase'.");

// Step 2: Shard the Collection

console.log("Sharding collection 'myDatabase.myCollection' on 'userId'..."); await adminDb.command({ shardCollection: "myDatabase.myCollection", key: { userId: 1 }

});

console.log("Sharding enabled for 'myDatabase.myCollection'.");

// Step 3: Insert Data

const db = client.db("myDatabase"); const collection = db.collection("myCollection"); console.log("Inserting documents..."); await collection.insertMany([

{ userId: 1, name: "Alice", age: 25, location: "Region1" },

{ userId: 2, name: "Bob", age: 30, location: "Region2" },

{ userId: 3, name: "Charlie", age: 28, location: "Region3" }

]);

console.log("Documents inserted.");

// Step 4: Query Data console.log("Querying all documents..."); const docs = await collection.find().toArray(); console.log("Documents in 'myCollection':", docs);

// Step 5: Update Data console.log("Updating document with userId 1..."); await collection.updateOne(

{ userId: 1 },

{ $set: { age: 26 } }

);

console.log("Document updated.");

// Step 6: Delete Data

console.log("Deleting document with userId 2..."); await collection.deleteOne({ userId: 2 }); console.log("Document deleted.");

// Step 7: Check Sharding Status (using `sh.status()` isn't available in Node.js) console.log("Sharding status can be checked in the MongoDB shell using `sh.status()`.");

} finally { await client.close();

}

}

run().catch(console.dir);

**Output:**

- Connection to MongoDB via the mongos router is successful.

- Sharding enabled on the database and collection.

- Documents inserted successfully:

```

[

{ userId: 1, name: "Alice", age: 25, location: "Region1" },

{ userId: 2, name: "Bob", age: 30, location: "Region2" },

{ userId: 3, name: "Charlie", age: 28, location: "Region3" }

]

```

- Document with `userId: 1` updated successfully.

- Document with `userId: 2` deleted successfully.

**Result:**

Thus, the configuration of MongoDB's replica sets and sharding, along with CRUD operations to test data distribution, high availability, and fault tolerance in a distributed database environment, has been successfully completed.

**2.B. Row level and statement level triggers**

**Aim:**

To create row level and statement level triggers in mongodb.

**Procedure:**

// Load environment variables from the .env file (e.g., MongoDB connection URI) require('dotenv').config();

// Import MongoClient from the MongoDB package to interact with the database const { MongoClient } = require('mongodb');

// MongoDB connection URI retrieved from the .env file (you can store it securely here) const uri = process.env.MONGODB\_URI;

async function main() {

// Create a new MongoClient instance using the connection URI and options

const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });

try {

// Connect to MongoDB using the MongoClient instance await client.connect();

console.log("Connected to MongoDB!");

// Access the 'mydb' database const db = client.db("mydb");

// Access the 'users' collection in the 'mydb' database const usersCollection = db.collection("users");

// Insert sample data into the 'users' collection to simulate real-world data await usersCollection.insertMany([

{ name: "John Doe", email: "john@example.com", age: 30 },

{ name: "Jane Smith", email: "jane@example.com", age: 25 },

{ name: "Emily Davis", email: "emily@example.com", age: 35 }

]);

console.log("Sample data inserted.");

// 2. Row-Level Trigger (Document-Level Trigger):

// Create a change stream to listen for updates on documents in the 'users' collection.

// We filter the events to only capture 'update' operations on documents.

const rowLevelChangeStream = usersCollection.watch([

{ $match: { 'operationType': 'update', 'fullDocument.\_id': { $exists: true } } }

]);

// Set up an event listener that reacts to document update changes rowLevelChangeStream.on('change', (next) => {

// This will log the details of the updated document when a row-level update occurs console.log("Row-Level Trigger: Document Updated", next);

});

// 3. Statement-Level Trigger (Collection-Level Trigger):

// Create another change stream to listen for any change in the 'users' collection.

// This captures all operations: insert, update, and delete on any document in the collection.

const statementLevelChangeStream = usersCollection.watch();

// Set up an event listener for all changes in the collection (insert, update, delete) statementLevelChangeStream.on('change', (next) => { // This will log the details of any change in the collection

console.log("Statement-Level Trigger: Change Detected in Collection", next);

});

// Simulating updates to trigger the change streams: setTimeout(async () => {

// Simulate an update operation on a specific document (row-level trigger) console.log("Updating user..."); await usersCollection.updateOne(

{ name: "John Doe" }, // Filter to update document with 'name' equal to 'John Doe' { $set: { age: 31 } } // Update the 'age' field to 31

);

// Simulating another update on a different user to test collection-level trigger console.log("Updating another user..."); await usersCollection.updateOne(

{ name: "Emily Davis" }, // Filter to update document with 'name' equal to 'Emily Davis'

{ $set: { email: "emily.davis@example.com" } } // Update the 'email' field

);

// Simulating document insertion to test collection-level trigger console.log("Inserting new user..."); await usersCollection.insertOne({ name: "Michael Johnson", email: "michael@example.com", age: 40 });

}, 2000); // Wait for 2 seconds before performing the updates and insert

} catch (err) {

// If any error occurs during the operations, log it console.error("Error:", err);

}

}

// Run the 'main' function and handle any potential errors main().catch(console.error);

**Output:**

Connected to MongoDB successfully.

Sample data inserted into the 'users' collection.

Row-level trigger logs the details of updated documents.

Statement-level trigger logs all operations (insert, update, delete) on the collection.

Example operations performed:

Updated document with name "John Doe" (row-level trigger).

Updated document with name "Emily Davis" (collection-level trigger).

Inserted new document for "Michael Johnson" (collection-level trigger).

**Result:**

Thus The creation of row level and statement level triggers in mongodb has been completed successfully.

**2.C.Accessing a Relational Database using PHP, Python and R**

**Aim:**

To Accessing a Relational Database using PHP, Python and R.

**Procedure:**

**1. Accessing a Relational Database using PHP (MySQL with PDO)**

<?php

// Step 1: Define database connection variables

$host = 'localhost'; // The host where the MySQL database is running (usually localhost for local development)

$db = 'mydatabase'; // The name of the database you want to connect to

$user = 'root'; // The username for connecting to the MySQL server

$pass = ''; // The password associated with the user (in this case, empty for local setup)

// Step 2: Set the Data Source Name (DSN) to connect to the database

$dsn = "mysql:host=$host;dbname=$db"; // The DSN contains the MySQL host and database name for the connection

// Step 3: Try to establish a connection to the database try {

// Create a new PDO instance using the DSN, username, and password

$pdo = new PDO($dsn, $user, $pass);

// Set the PDO error mode to exception. This will throw exceptions in case of errors, instead of just warnings.

$pdo->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

// Step 4: Create table if it doesn't exist

$pdo->exec("

CREATE TABLE IF NOT EXISTS mytable ( id INT AUTO\_INCREMENT PRIMARY KEY, column\_name VARCHAR(255) NOT NULL

)

");

// Step 5: Insert sample data into the table

$pdo->exec("INSERT INTO mytable (column\_name) VALUES ('Alice')");

$pdo->exec("INSERT INTO mytable (column\_name) VALUES ('Bob')");

$pdo->exec("INSERT INTO mytable (column\_name) VALUES ('Charlie')");

// Step 6: Query the database to retrieve data from the table

$stmt = $pdo->query("SELECT \* FROM mytable"); // This executes an SQL query to fetch all rows from 'mytable'

// Step 7: Fetch all the results and display them

$rows = $stmt->fetchAll(PDO::FETCH\_ASSOC); // Fetch the results as an associative array (key-value pairs) foreach ($rows as $row) { // Loop through each row returned by the query echo $row['column\_name'] . "<br>"; // Print the value of 'column\_name' from each row

}

} catch (PDOException $e) {

// If there's an error while connecting or querying, catch the exception and display an error message

echo "Connection failed: " . $e->getMessage();

}

?>

**output:**

Alice

Bob

charlie

**2.Accessing a Relational Database using Python (MySQL with mysql-connector)**

import mysql.connector

# Step 1: Establish a connection to the MySQL database conn = mysql.connector.connect(

host='localhost', # The hostname of the MySQL server, typically 'localhost' for local databases

user='root', # The username for connecting to MySQL

password='', # The password for the MySQL user (empty string here for local development)

database='mydatabase' # The name of the database to connect to

)

# Step 2: Create a cursor object to execute SQL queries

cursor = conn.cursor() # A cursor is an object used to interact with the database and execute queries

# Step 3: Create the table `mytable` (if it doesn't already exist) cursor.execute("""

CREATE TABLE IF NOT EXISTS mytable ( id INT AUTO\_INCREMENT PRIMARY KEY, name VARCHAR(255), age INT

)

""")

# Step 4: Insert sample data into `mytable`

cursor.execute("INSERT INTO mytable (name, age) VALUES (%s, %s)", ('Alice', 30)) cursor.execute("INSERT INTO mytable (name, age) VALUES (%s, %s)", ('Bob', 25)) cursor.execute("INSERT INTO mytable (name, age) VALUES (%s, %s)", ('Charlie', 35)) # Commit the changes to the database conn.commit()

# Step 5: Execute an SQL query to retrieve data from the table

cursor.execute("SELECT \* FROM mytable") # This executes the query that selects all rows from 'mytable'

# Step 6: Fetch all rows from the result set

rows = cursor.fetchall() # The `fetchall()` method retrieves all rows returned by the query as a list of tuples

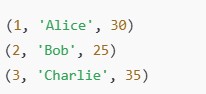
# Step 7: Iterate over the rows and print the data for row in rows: # Loop through the list of rows

print(row) # Print each row (each row is a tuple representing a record)

# Step 8: Close the cursor and connection to free up resources cursor.close() # Close the cursor object

conn.close() # Close the connection to the MySQL server

**output:**



**3.Accessing a Relational Database using R (MySQL with RMySQL)**

# Step 1: Install and load the RMySQL package

install.packages("RMySQL") # Install the RMySQL package (uncomment this line if not installed)

library(RMySQL) # Load the RMySQL package to interact with MySQL databases from R

# Step 2: Establish a connection to the MySQL database

con <- dbConnect(RMySQL::MySQL(), # Create a connection object to the MySQL database

host = "localhost", # The hostname for the MySQL server user = "root", # The username for the MySQL connection

password = "", # The password for the MySQL user (empty here for local development)

dbname = "mydatabase") # The database name to connect to

# Step 3: Create the table if it does not exist dbExecute(con, "

CREATE TABLE IF NOT EXISTS mytable ( id INT AUTO\_INCREMENT PRIMARY KEY, column\_name VARCHAR(255) NOT NULL

)

")

# Step 4: Insert sample data into 'mytable'

dbExecute(con, "INSERT INTO mytable (column\_name) VALUES ('Alice')") dbExecute(con, "INSERT INTO mytable (column\_name) VALUES ('Bob')") dbExecute(con, "INSERT INTO mytable (column\_name) VALUES ('Charlie')")

# Step 5: Execute an SQL query to retrieve data from the table

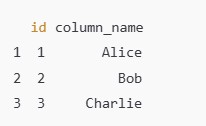
result <- dbGetQuery(con, "SELECT \* FROM mytable") # Execute the SQL query to fetch all rows from 'mytable'

# Step 6: Print the result (the query result is returned as a data frame in R) print(result) # Output the result (a data frame) to the console

# Step 7: Close the connection to the database

dbDisconnect(con) # Close the database connection to free up resources

**Output:**



**Result:**

Thus The Accessing of Relational Database using PHP, Python and R. Has been completed successfully