**EXP.No : 2**

**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:**

javascript

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

console.log("Sharding status can be checked in the MongoDB shell using `sh.status()`.");

} finally {

await client.close();

}

}

run().catch(console.dir);

**Output:**

-The connection to MongoDB through the \*mongos router\* was established successfully.

- 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:**

The configuration of MongoDB's replica sets and sharding, along with the execution of CRUD operations to validate 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:**

javascript

// 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.");

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

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

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) => {

// Log the details of the updated document when a row-level update occurs

console.log("Row-Level Trigger: Document Updated", next);

});

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

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

const statementLevelChangeStream = usersCollection.watch();

// Set up an event listener for all changes in the collection (insert, update, delete)

statementLevelChangeStream.on('change', (next) => {

// 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

);

// Simulate 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

);

// Simulate 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:**

**-** Successfully connected to MongoDB.

- Sample data was inserted into the 'users'collection.

- Row-level trigger logs details of updated documents.

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

1. Updated the document with the name "John Doe" (\*row-level trigger\*).

2. Updated the document with the name "Emily Davis" (\*collection-level trigger\*).

3. Inserted a new document for "Michael Johnson" (\*collection-level trigger\*).

**Result:**

Thus, the creation of row-level and statement-level triggers in MongoDB has been successfully completed.

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

**Aim:**

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

**Procedure:**

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

`php

<?php

// Step 1: Define database connection variables

$host = 'localhost';

$db = 'mydatabase';

$user = 'root';

$pass = '';

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

$dsn = "mysql:host=$host;dbname=$db";

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

try {

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

$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");

$rows = $stmt->fetchAll(PDO::FETCH\_ASSOC);

// Step 7: Display the retrieved data

foreach ($rows as $row) {

echo $row['column\_name'] . "<br>";

}

} catch (PDOException $e) {

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

}

?>

**Output:**

Alice

Bob

charlie

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

python

import mysql.connector

Step 1: Establish a connection to the MySQL database

conn = mysql.connector.connect(

host='localhost',

user='root',

password='',

database='mydatabase'

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

cursor = conn.cursor()

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")

Step 6: Fetch all rows from the result set

rows = cursor.fetchall()

Step 7: Display the retrieved data

for row in rows:

print(row)

Step 8: Close the cursor and connection

cursor.close()

conn.close()

**Output:**

1. ‘Alice’, 30)

2. ‘Bob’,25)

3.`charlie’,35)

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

R

Step 1: Install and load the RMySQL package

install.packages("RMySQL")

library(RMySQL)

Step 2: Establish a connection to the MySQL database

con <- dbConnect(RMySQL::MySQL(),

host = "localhost",

user = "root",

password = "",

dbname = "mydatabase")

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")

Step 6: Display the retrieved data

print(result)

Step 7: Close the connection to the database

dbDisconnect(con)

**Output**:

id coloumn - name

1 1. Alice

2 2. Bob

3 3. Charlie

**Result:**

Thus, accessing a Relational Database using PHP, Python, and R has been completed successfully.