1. **Implement Parallel Hash Join**

**Theory:**

Parallel hash join is a distributed computing technique employed in databases to efficiently merge two large datasets. Initially, both datasets are divided into smaller partitions using a hash function applied to a join key, with each partition assigned to a processing unit. These units independently hash their local data, creating hash tables where each entry represents a bucket containing tuples sharing the same hash value. Following this, a data shuffling phase occurs, redistributing hash tables across units based on their hash values, ensuring tuples with the same hash value are located on the same unit. Once data is shuffled, each unit locally executes the join operation by matching tuples with identical hash values from the two datasets, typically employing algorithms like nested loop join or hash join. Finally, results are aggregated across units to produce the final output. Parallel hash join offers scalability, enabling processing of datasets too large for a single machine, and performance benefits through parallel execution, load balancing, and fault tolerance, making it a crucial tool in distributed database systems.

**Code:**

import threading

from prettytable import PrettyTable

EMPLOYEE\_LIST = [

{"E.Id": 101, "Employee Name": "Arjun Mehta", "Department": "Engineering"},

{"E.Id": 102, "Employee Name": "Neha Patel", "Department": "Marketing"},

{"E.Id": 103, "Employee Name": "Rajesh Singh", "Department": "Sales"},

{"E.Id": 104, "Employee Name": "Kavya Kapoor", "Department": "Engineering"},

{"E.Id": 105, "Employee Name": "Suresh Kumar", "Department": "Engineering"},

{"E.Id": 106, "Employee Name": "Priya Sharma", "Department": "HR"},

{"E.Id": 107, "Employee Name": "Vikas Gupta", "Department": "Finance"},

{"E.Id": 108, "Employee Name": "Anjali Chawla", "Department": "Marketing"},

{"E.Id": 109, "Employee Name": "Rohit Verma", "Department": "Engineering"},

{"E.Id": 110, "Employee Name": "Pooja Joshi", "Department": "HR"},

{"E.Id": 111, "Employee Name": "Manoj Rao", "Department": "Sales"},

{"E.Id": 112, "Employee Name": "Sneha Desai", "Department": "Finance"},

]

PROJECT\_LIST = [

{"E.Id": 101, "Project": "Project One"},

{"E.Id": 102, "Project": "Project Two"},

{"E.Id": 103, "Project": "Project Three"},

{"E.Id": 104, "Project": "Project Four"},

{"E.Id": 105, "Project": "Project Five"},

]

def partition\_data(data, num\_partitions):

partitions = [[] for \_ in range(num\_partitions)]

for item in data:

partition\_index = hash(item["E.Id"]) % num\_partitions

partitions[partition\_index].append(item)

return partitions

def local\_join(employee\_partition, project\_partition, result):

project\_hash = {item["E.Id"]: item["Project"] for item in project\_partition}

for employee in employee\_partition:

emp\_id = employee["E.Id"]

project = project\_hash.get(emp\_id)

result.append({\*\*employee, "Project": project})

def parallel\_hash\_join(EMPLOYEE\_LIST, PROJECT\_LIST, num\_partitions):

employee\_partitions = partition\_data(EMPLOYEE\_LIST, num\_partitions)

project\_partitions = partition\_data(PROJECT\_LIST, num\_partitions)

threads = []

result = []

for emp\_partition, proj\_partition in zip(employee\_partitions, project\_partitions):

thread = threading.Thread(target=local\_join, args=(emp\_partition, proj\_partition, result))

thread.start()

threads.append(thread)

for thread in threads:

thread.join()

return result

# Perform the parallel hash join

result = parallel\_hash\_join(EMPLOYEE\_LIST, PROJECT\_LIST, num\_partitions=4)

# Create a table to display the result

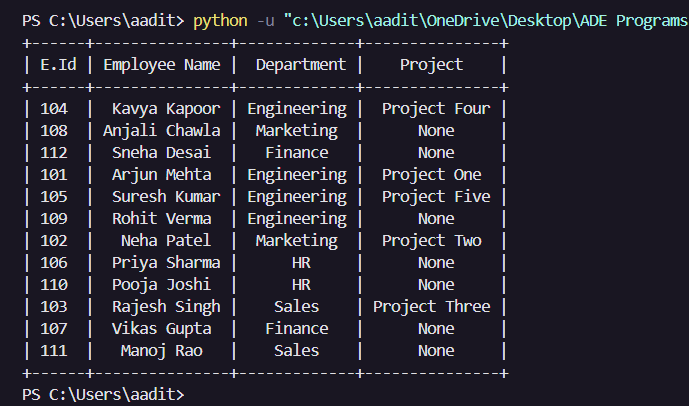
table = PrettyTable(["E.Id", "Employee Name", "Department", "Project"])

for item in result:

table.add\_row([item["E.Id"], item["Employee Name"], item["Department"], item["Project"]])

print(table)

**Output:**



1. **Implement two phase commit protocol in distributed DBMS**

**Theory:**

The two-phase commit protocol (2PC) is a distributed algorithm used in distributed database management systems (DBMS) to ensure atomicity and consistency across multiple nodes during transaction commits. In the first phase, the coordinator node sends a prepare message to all participant nodes, asking them to prepare for the commit. Upon receiving this message, each participant node checks if it can successfully commit the transaction based on its local state. If a participant is ready to commit, it replies with a "prepared" message; otherwise, it responds with a "abort" message. Once the coordinator receives "prepared" messages from all participants, it proceeds to the second phase. In this phase, the coordinator sends a commit message to all participants, instructing them to commit the transaction. Upon receiving this message, each participant performs the actual commit operation and acknowledges the coordinator. If any participant encounters an error during the commit phase or does not respond, the coordinator sends an abort message to all participants, instructing them to roll back the transaction. The two-phase commit protocol ensures that either all participants commit the transaction successfully or none of them commit it, thereby maintaining the atomicity and consistency of distributed transactions. However, 2PC suffers from some limitations, such as blocking issues in case of failures and the need for synchronous communication, which can impact performance in large-scale distributed systems.

**Code:**

Coordinator side:

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class Coordinator {

private static ServerSocket serverSocket;

private static List<Socket> participants = new ArrayList<>();

private static List<String> participantResponses = new ArrayList<>();

private static int expectedParticipants;

private static boolean preparePhase = false;

public static void main(String[] args) {

try {

serverSocket = new ServerSocket(12345); // Server socket listening on port 12345

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of participants: ");

expectedParticipants = scanner.nextInt();

// Accept participants until the expected number is reached

System.out.println("\nWaiting for the participants...");

while (participants.size() < expectedParticipants) {

Socket participant = serverSocket.accept();

participants.add(participant);

System.out.println("Participant " + (participants.size()) + " connected from: " + participant.getRemoteSocketAddress());

// If expected number of participants is reached, start the prepare phase

if (participants.size() == expectedParticipants && !preparePhase) {

preparePhase = true;

sendPrepareMessage();

}

}

// Receive responses from participants

receiveResponsesFromParticipants();

// Make a decision based on received responses

makeDecision();

// Calculate the total sum of numbers from participants

calculateTotalSum();

// Close all connections

closeConnections();

} catch (IOException e) {

e.printStackTrace();

}

}

private static void sendPrepareMessage() {

System.out.println("\nSending PREPARE message to all participants...");

sendMessageToAll("PREPARE");

}

private static void sendMessageToAll(String message) {

for (Socket participant : participants) {

try {

PrintWriter out = new PrintWriter(participant.getOutputStream(), true);

out.println(message);

} catch (IOException e) {

e.printStackTrace();

}

}

}

private static void receiveResponsesFromParticipants() throws IOException {

System.out.println();

for (Socket participant : participants) {

BufferedReader in = new BufferedReader(new InputStreamReader(participant.getInputStream()));

String response = in.readLine();

participantResponses.add(response);

System.out.println("Received " + response + " response from participant: " + participant.getRemoteSocketAddress());

}

}

private static void makeDecision() {

if (participantResponses.contains("NO")) {

System.out.println("\nTransaction aborting...");

sendMessageToAll("ABORT");

} else {

System.out.println("\nTransaction committing...");

sendMessageToAll("COMMIT");

}

}

private static void calculateTotalSum() throws IOException {

int totalSum = 0;

// Receive numbers from each participant and calculate the total sum

for (Socket participant : participants) {

BufferedReader in = new BufferedReader(new InputStreamReader(participant.getInputStream()));

int number = Integer.parseInt(in.readLine());

totalSum += number;

System.out.println("Received number " + number + " from participant: " + participant.getRemoteSocketAddress());

}

// Display total sum

System.out.println("\nTotal sum of all numbers received from participants: " + totalSum);

}

private static void closeConnections() {

try {

serverSocket.close();

for (Socket participant : participants) {

participant.close();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

Subordinate side:

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class Subordinate {

public static void main(String[] args) {

try {

// Connect to Coordinator (localhost) on port 12345

Socket socket = new Socket("localhost", 12345);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

// Display the local port address

int port = socket.getLocalPort();

System.out.println("Local port address: " + port);

// Step 1: Receive PREPARE message from Coordinator

System.out.println("\nWaiting for PREPARE message from Coordinator...");

String prepareMessage = in.readLine();

if ("PREPARE".equals(prepareMessage)) {

System.out.println("\nReceived PREPARE from Coordinator.");

Scanner scanner = new Scanner(System.in);

System.out.print("\nAre you ready to commit? (Enter YES or NO): ");

String response = scanner.nextLine();

out.println(response); // Send response to Coordinator

System.out.println("\nSending " + response + " to the Coordinator.");

}

// Step 3: Receive decision from Coordinator

String decision = in.readLine();

System.out.println("\nReceived decision from Coordinator: " + decision);

if (decision.equals("COMMIT")) {

// Step 4: If the decision is COMMIT, enter a number to send to the Coordinator

System.out.print("Enter a number to send to the Coordinator: ");

Scanner scanner = new Scanner(System.in);

int number = scanner.nextInt();

out.println(number); // Send the number to the Coordinator

System.out.println("Sent number " + number + " to the Coordinator.");

}

// Close the connection

socket.close();

} catch (IOException e) {

e.printStackTrace();

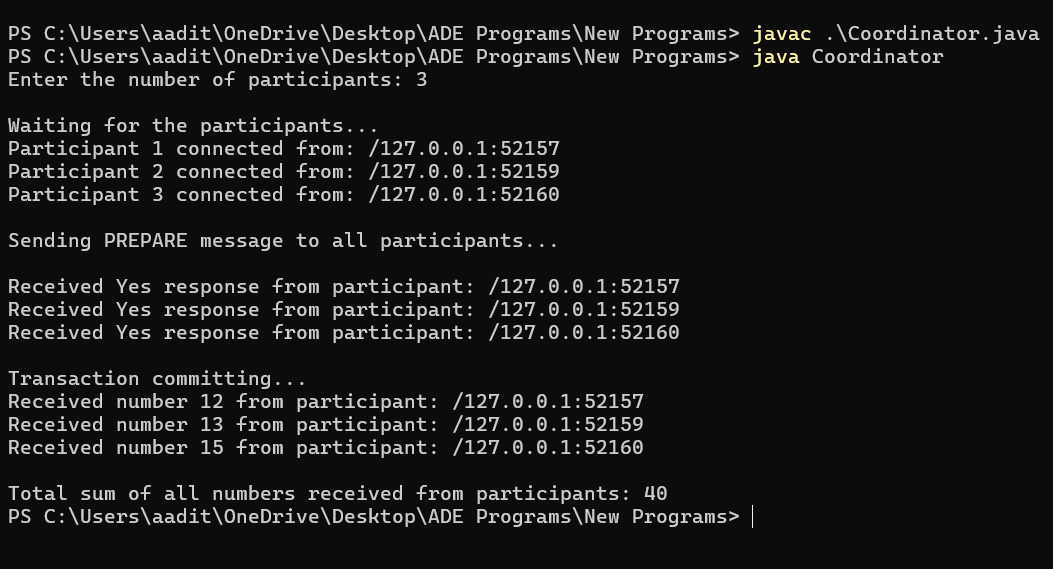
}

}

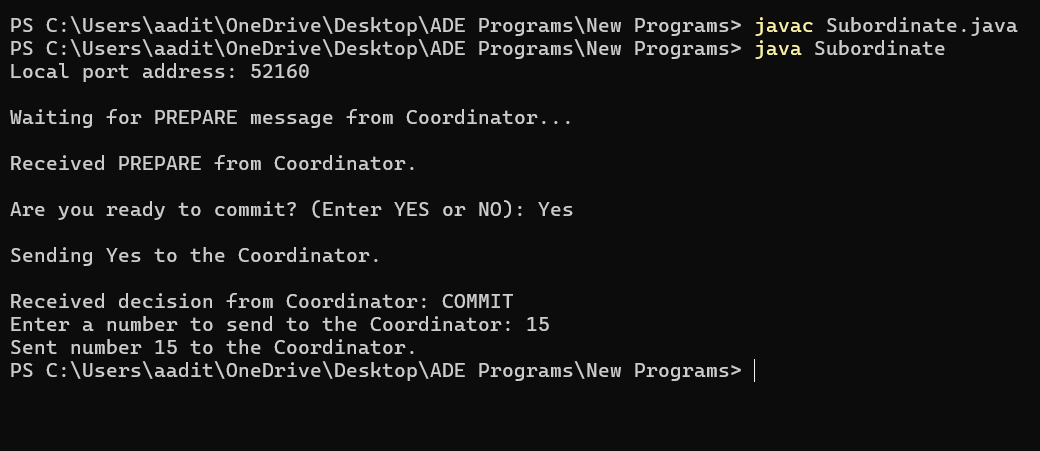
}

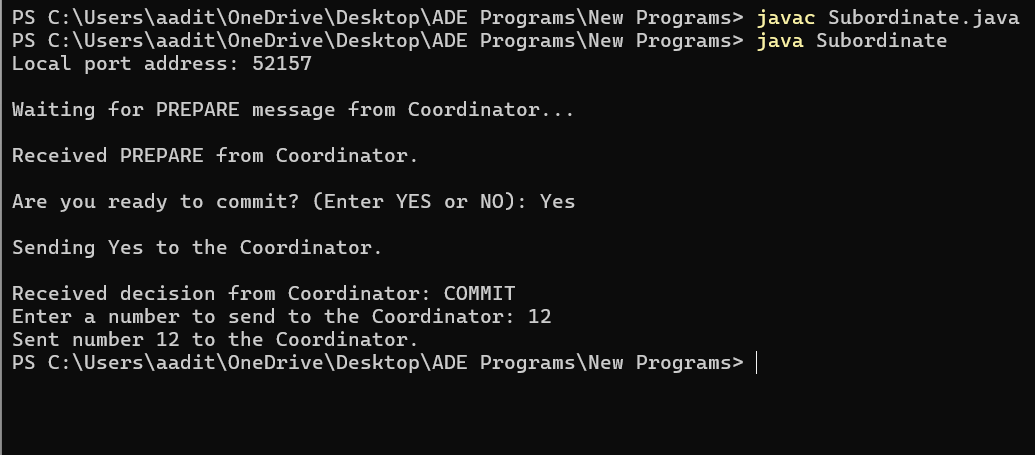
**Output:**

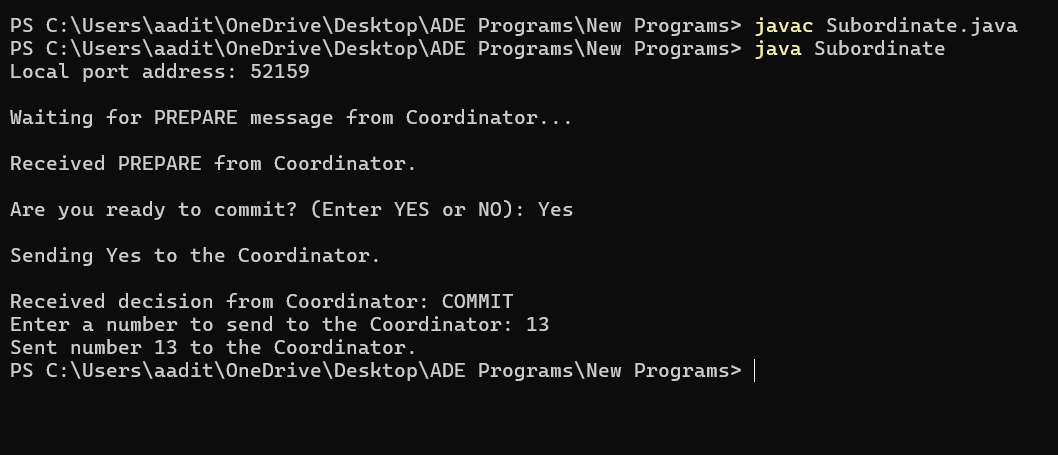
Coordinator Side



Subordinate Side







1. **Implement ADT concept**

**Theory:**

Abstract Data Types (ADTs) are fundamental concepts in computer science and software engineering that define a logical model for data and operations on that data. ADTs encapsulate data structures and their associated operations, allowing users to manipulate data without needing to know the underlying implementation details. The theory behind ADTs revolves around the concept of abstraction, which separates the interface (operations) from the implementation (data structure). This separation enables modularity, as users can interact with ADTs through a well-defined set of operations without needing to understand how those operations are implemented. Common examples of ADTs include lists, stacks, queues, trees, graphs, and sets, each with its own set of operations such as insertion, deletion, traversal, and searching. ADTs serve as building blocks for designing complex systems, facilitating code reuse, modularity, and maintainability. They also provide a means for specifying and reasoning about the behavior of abstract data structures independently of any particular programming language or implementation. In programming languages such as Java, C++, and Python, ADTs are often implemented using classes or structures, where the data fields represent the state of the abstract data structure and methods represent the operations that can be performed on it. By adhering to the principles of encapsulation, abstraction, and information hiding, ADTs enable developers to write more robust, scalable, and maintainable software systems.

**Code:**

CREATE TYPE prn\_type AS (

registration\_year CHAR(4), -- Year part of the PRN (4 characters)

degree CHAR(3), -- Degree part of the PRN (e.g., BTE for B. Tech, 3 characters)

branch CHAR(4), -- Branch part of the PRN

permanent\_registration\_number CHAR(8) -- Permanent registration number (8 characters)

);

CREATE TABLE students (

student\_id SERIAL PRIMARY KEY,

name VARCHAR(255),

prn prn\_type -- Using the composite type as a column type

);

INSERT INTO students (name, prn)

VALUES

('Aaditya Khot', ROW('2021', 'BTE', 'IT', '21610051')),

('Tanish Gadvir', ROW('2021', 'BTE', 'ELN', '21310022')),

('Shrenik Jadhav', ROW('2020', 'BTE', 'IT', '20610055')),

('Tushar Jadhav', ROW('2019', 'BTE', 'CIV', '19110018')),

('Aryan Magdum', ROW('2021', 'BTE', 'CSE', '21510070')),

('Ashwin Madane', ROW('2020', 'BTE', 'MECH', '20210038'));

SELECT

student\_id,

name,

(prn).registration\_year || (prn).degree || (prn).branch || (prn).permanent\_registration\_number AS prn\_string

FROM

students;

**Output:**

