**SOMAIYA VIDYAVIHAR UNIVERSITY**

**K. J. Somaiya College of Engineering, Mumbai -77**

**(A Constituent College of Somaiya Vidyavihar University**)

RDBMS Report- RDBMS IA-2 23-24

Submitted By **Sanskar Lodha**

16010122097

**Sujal Kumar**

16010122091

**Aditya Patel**

16010122134

**Kashish Mamania**

16010122104



**TOPIC 1**:

**Cost Estimating Algorithms**

**Module No: 4**

# Description

Cost estimation algorithms serve as computational tools utilized to forecast the financial investments necessary for project completion, product manufacturing, or service delivery. These algorithms hold significant importance across various industries such as project management, engineering, construction, manufacturing, etc., where precise cost estimation is critical for informed decision-making and strategic planning purposes.

Cost estimation involves the process of project managers forecasting the financial resources required to finance their projects. This entails considering both direct and indirect costs, encompassing utilities, materials, equipment, vendors, and employee compensation.

During cost estimation, managers take into account various project elements, including:

Duration: The time required to complete the project.

Size: The scale of the project, distinguishing between major endeavors like constructing a community center and smaller tasks like publishing a limited edition book.

Scope: The breadth of the project's impact and participation, identifying beneficiaries and stakeholders.

Complexity: The level of intricacy within the project, which influences the number of steps involved and their associated costs.

By analyzing these factors, project managers can generate accurate cost estimates to facilitate effective project planning and decision-making.

***TYPES***

1. **Analogous estimating**

The analogous estimating method combines historical data and expert judgment to anticipate the costs of a project. Here are its steps:

Identify the project's elements, such as size, cost and duration.

Research similar projects that have used the same elements.

Base the cost estimation for the current project on a budget of past projects.

1. **Parametric Estimating**

Parametric estimating relies on statistical correlations between past project data and specific project characteristics, like size, weight, or complexity, to forecast costs. This method entails constructing mathematical models derived from historical records and applying them to the current project's attributes. While more precise than analogous estimating, parametric estimating necessitates dependable historical data and a comprehensive grasp of the project's parameters for accurate predictions.

1. **Bottom-up estimating**

Bottom-Up Estimating involves breaking down a project into smaller components or work packages and estimating the cost for each component individually. These estimates are then combined to determine the total project cost. While Bottom-Up Estimating offers a detailed and precise cost breakdown, it can be laborious and resource-intensive due to the need to estimate each component separately.

1. **Three-Point Estimating**

Three-Point Estimating entails evaluating the best-case, worst-case, and most probable scenarios for every task or component within a project. These estimates are then utilized to compute a weighted average, considering the probability of each scenario. This method assists in factoring in uncertainty and risk when estimating costs.



***Algorithm***

1. Begin.
2. Initialize three arrays previousSizes, previousCosts, and previousDurations to store the size, cost, and duration of 10 previous projects.
3. Display a prompt to input the size, cost, and duration of 10 previous projects.
4. Iterate 10 times: a. Prompt the user to input the size of the project. b. Store the input size in the previousSizes array. c. Prompt the user to input the cost of the project. d. Store the input cost in the previousCosts array. e. Prompt the user to input the duration of the project. f. Store the input duration in the previousDurations array.
5. Prompt the user to input the size of the current project.
6. Prompt the user to input the duration of the current project.
7. Calculate the total previous size, cost, and duration by summing up the corresponding elements in the previousSizes, previousCosts, and previousDurations arrays, respectively.
8. Calculate the average previous size, cost, and duration by dividing the total previous size, cost, and duration by 10.
9. Calculate the estimated cost of the current project using the formula: estimatedCost = (currentSize / averagePreviousSize) \* averagePreviousCost
10. Calculate the estimated duration of the current project using the formula: estimatedDuration = (currentSize / averagePreviousSize) \* averagePreviousDuration
11. Display the estimated cost and duration of the current project.
12. End.

***Implementation Code***

import java.util.Scanner;

public class Estimating {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        double[] previousSizes = new double[10];

        double[] previousCosts = new double[10];

        double[] previousDurations = new double[10];

        System.out.println("Enter the size, cost, and duration of 10 previous projects:");

        for (int i = 0; i < 10; i++) {

            System.out.print("Enter size of project " + (i + 1) + " (in terms of effort): ");

            previousSizes[i] = scanner.nextDouble();

            System.out.print("Enter cost of project " + (i + 1) + " ₹");

            previousCosts[i] = scanner.nextDouble();

            System.out.print("Enter duration of project " + (i + 1) + " (in months): ");

            previousDurations[i] = scanner.nextDouble();

        }

        System.out.print("Enter the size of the current project (in terms of effort): ");

        double currentSize = scanner.nextDouble();

        System.out.print("Enter the duration of the current project (in months): ");

        double currentDuration = scanner.nextDouble();

        double totalPreviousSize = 0;

        double totalPreviousCost = 0;

        double totalPreviousDuration = 0;

        for (int i = 0; i < 10; i++) {

            totalPreviousSize += previousSizes[i];

            totalPreviousCost += previousCosts[i];

            totalPreviousDuration += previousDurations[i];

        }

        double averagePreviousSize = totalPreviousSize / 10;

        double averagePreviousCost = totalPreviousCost / 10;

        double averagePreviousDuration = totalPreviousDuration / 10;

        double estimatedCost = (currentSize / averagePreviousSize) \* averagePreviousCost;

        double estimatedDuration = (currentSize / averagePreviousSize) \* averagePreviousDuration;

        System.out.println("Estimated cost of the current project: ₹" + estimatedCost);

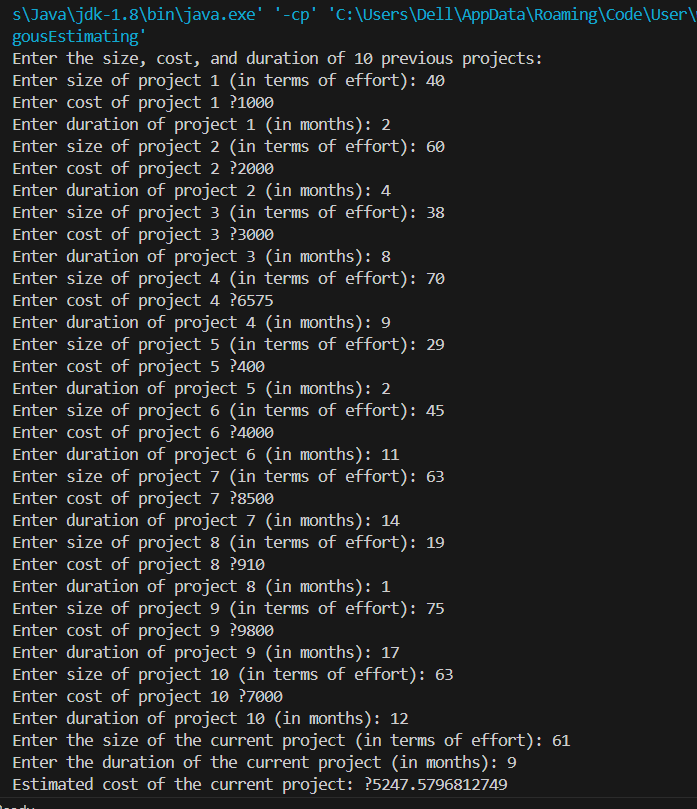
        System.out.println("Estimated duration of the current project: " + estimatedDuration + " months");

        scanner.close();

    }

}

***Output***



***Algorithm***

1. Start
2. Input the best-case estimate, worst-case estimate, and most likely estimate for each project task or component.
3. Calculate the weighted average using the formula:

weightedAverage = (bestCase + (4 \* mostLikely) + worstCase) / 6

1. Output the weighted average estimate.
2. End.

***Implementation Code***

import java.util.Scanner;

public class ThreePointEstimating {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the best-case estimate: ");

        double bestCase = scanner.nextDouble();

        System.out.print("Enter the worst-case estimate: ");

        double worstCase = scanner.nextDouble();

        System.out.print("Enter the most likely estimate: ");

        double mostLikely = scanner.nextDouble();

        double weightedAverage = (bestCase + (4 \* mostLikely) + worstCase) / 6;

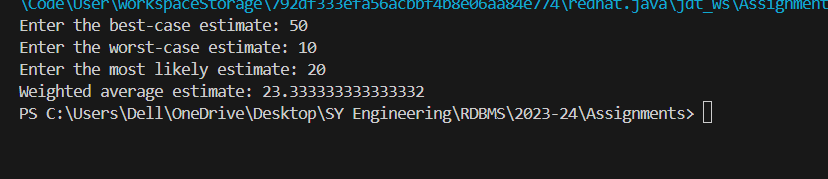
        System.out.println("Weighted average estimate: " + weightedAverage);

        scanner.close();

    }

}

***OUTPUT***



**TOPIC 1**:

**Concurrency Control Algorithms**

**Module No: 5**

# Description

Concurrency control algorithms are crucial in database management systems to ensure data integrity and consistent views of data when multiple transactions are executed concurrently. These algorithms prevent errors such as lost updates, dirty reads, and non-repeatable reads, among other anomalies. Here’s a brief overview of the primary types of concurrency control algorithms:

***TYPES***

1. Lock-Based Protocols

Locks: Transactions lock objects to control access during their execution.

Types of Locks: Shared locks for reading data, exclusive locks for writing data.

Two-Phase Locking (2PL): Ensures serializability by dividing the transaction into a growing phase (acquiring locks) and a shrinking phase (releasing locks).

2. Timestamp-Based Protocols

Timestamp Ordering: Transactions are timestamped, and the system ensures that the chronological order is respected in accessing the database.

Multiversion Concurrency Control (MVCC): Uses timestamps to maintain several versions of data, thus allowing readers and writers to not block each other.

3. Optimistic Concurrency Control

- Assumption of Rarity of Conflict: Lets transactions execute without locking resources but checks for conflict at the end before commit.

- Validation Phase: Determines if the transaction can be safely committed or needs rollback.

4. Snapshot Isolation

Snapshot Creation: Transactions see a consistent snapshot of the database at the start of their execution and operate on this snapshot.

Commit Conditions: Only successful if no conflicting updates were committed during its execution.

5. Hybrid Approaches

Combination of Methods: Modern databases often integrate multiple concurrency control techniques to optimize performance and provide robust data integrity.

In essence, these algorithms manage the access of multiple transactions to the shared database to minimize conflicts and maximize performance, adhering to the database's ACID properties. Each method has its strengths and is suited to different types of database workloads and operational environments.



***Algorithm***

Lock-based protocols manage database access by using locks. Locks can be either:

1. Shared (S): Allows a transaction to read a data item.
2. Exclusive (X): Allows a transaction to read and write a data item.
3. Basic Algorithm:
4. Lock Request: A transaction requests a lock on an object. If the lock is available (i.e., no conflicting locks are held by other transactions), the lock is granted.
5. Lock Release: After a transaction finishes using the data item, it releases the lock, making it available for other transactions.
6. Two-Phase Locking (2PL): This protocol ensures conflict serializability. It has two phases:
7. Growing Phase: Locks are acquired and no locks are released.
8. Shrinking Phase: Locks are released and no new locks are acquired.

***Implementation Code***

import java.util.concurrent.locks.ReentrantLock;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.Scanner;

public class Counter {

    private int count = 0;

    private final ReentrantLock lock = new ReentrantLock();

    // Method to increment the counter

    public void increment() {

        lock.lock();  // Acquire the lock

        try {

            count++;

        } finally {

            lock.unlock();  // Ensure the lock is released even if an exception occurs

        }

    }

    // Getter method for the counter

    public int getCount() {

        return count;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Get the number of threads from user

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

        final int numberOfThreads = scanner.nextInt();

        // Get the number of increments per thread from user

        System.out.print("Enter the number of increments per thread: ");

        final int incrementsPerThread = scanner.nextInt();

        ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);

        Counter counter = new Counter();

        // Create and start multiple threads to increment the counter

        for (int i = 0; i < numberOfThreads; i++) {

            executor.execute(() -> {

                for (int j = 0; j < incrementsPerThread; j++) {

                    counter.increment();

                }

            });

        }

        executor.shutdown();  // Shutdown the executor service

        // Wait until all threads are finished

        while (!executor.isTerminated()) {

            // Wait for all tasks to finish

        }

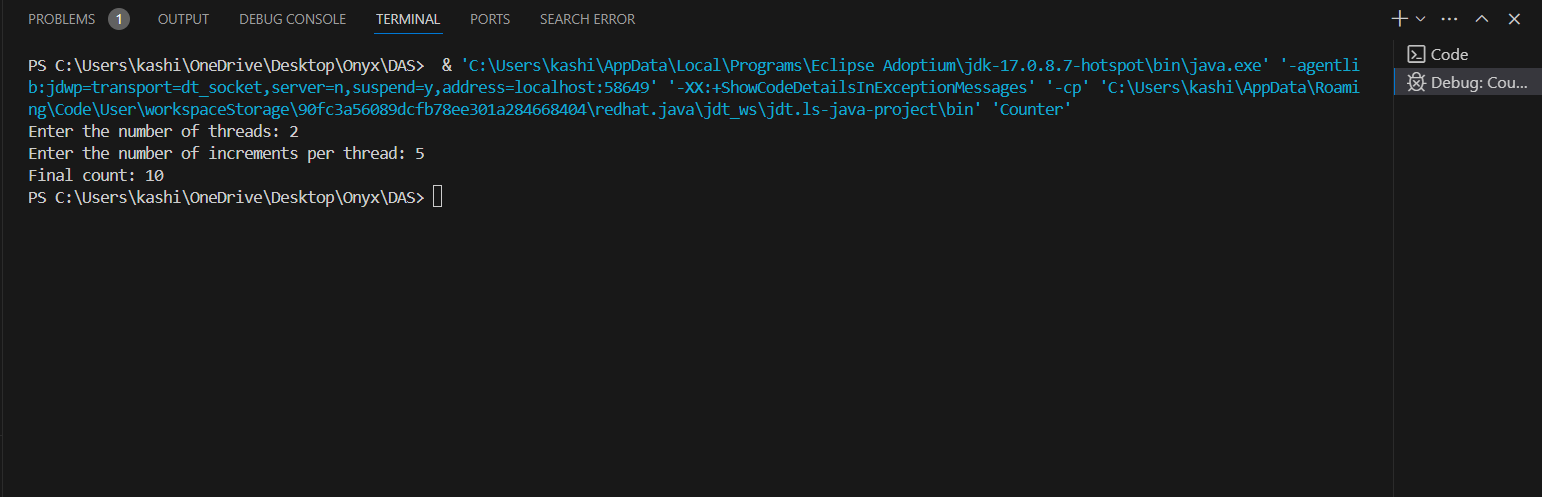
        System.out.println("Final count: " + counter.getCount());  // Print the final count

        scanner.close();  // Close the scanner

    }

}

***Output***

******

***Algorithm***

Timestamp-Based Protocols

These protocols use timestamps to order transactions in a way that avoids conflicts.

1. Assign Timestamp: Each transaction is given a unique timestamp when it starts.
2. Read and Write Rules:

Read Rule: A transaction T can read an object O only if the last write on O was by a transaction with a smaller timestamp than T.

1. Write Rule: A transaction T can write an object O only if the last read and write operations on O were by transactions with timestamps smaller than T. Give me implementation codes for these two algorithms

***Implementation Code***

class Lock:

    def \_\_init\_\_(self):

        self.locked = False

    def acquire(self):

        if not self.locked:

            self.locked = True

            return True

        else:

            return False

    def release(self):

        self.locked = False

class Transaction:

    def \_\_init\_\_(self, tid):

        self.tid = tid

        self.locks = []

    def acquire\_lock(self, resource):

        if resource.acquire():

            self.locks.append(resource)

            return True

        else:

            return False

    def release\_locks(self):

        for resource in self.locks:

            resource.release()

class TwoPhaseLocking:

    def \_\_init\_\_(self):

        self.resources = {}

        self.transactions = {}

    def add\_resource(self, rid):

        self.resources[rid] = Lock()

    def begin\_transaction(self, tid):

        self.transactions[tid] = Transaction(tid)

    def end\_transaction(self, tid):

        if tid in self.transactions:

            self.transactions[tid].release\_locks()

            del self.transactions[tid]

    def read(self, tid, rid):

        if tid in self.transactions and rid in self.resources:

            if self.transactions[tid].acquire\_lock(self.resources[rid]):

                print(f"Transaction {tid} reads resource {rid}")

            else:

                print(f"Transaction {tid} cannot read resource {rid} due to conflicts")

        else:

            print("Invalid transaction ID or resource ID")

    def write(self, tid, rid):

        if tid in self.transactions and rid in self.resources:

            if self.transactions[tid].acquire\_lock(self.resources[rid]):

                print(f"Transaction {tid} writes to resource {rid}")

            else:

                print(f"Transaction {tid} cannot write to resource {rid} due to conflicts")

        else:

            print("Invalid transaction ID or resource ID")

# Example usage

lock\_manager = TwoPhaseLocking()

lock\_manager.add\_resource('A')

lock\_manager.add\_resource('B')

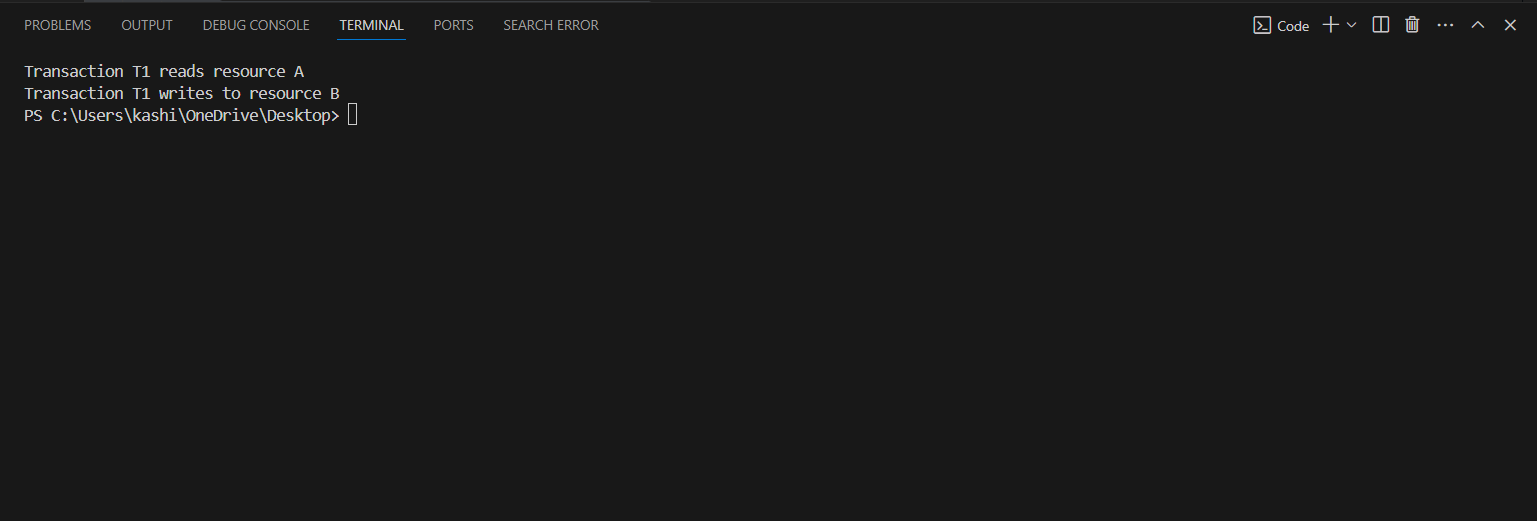
lock\_manager.begin\_transaction('T1')

lock\_manager.read('T1', 'A')

lock\_manager.write('T1', 'B')

lock\_manager.end\_transaction('T1')

***Output***

******

***References***

<https://www.geeksforgeeks.org/cost-based-optimization/>

<https://www.geeksforgeeks.org/concurrency-control-in-dbms/>