Index

Experiment 1

Title: Implement Bloom Join

Theory:

A Bloom Join, also known as Bloomjoin, is a distributed join processing technique used in parallel and distributed databases. It is an optimization technique used to reduce the amount of data that needs to be transferred over the network during a join operation between two relations residing on different nodes or sites.

The Bloomjoin technique works as follows:

Bit-Vector Creation: One of the relations, typically the smaller one (e.g., Sailors), is used to create a bit-vector of a fixed size (say, k bits). Each tuple in this relation is hashed to a value in the range of 0 to k-1 using a hash function. The corresponding bit in the bit-vector is set to 1 if at least one tuple hashes to that value.
Relation Reduction: The other relation (e.g., Reserves) is reduced by hashing each of its tuples using the same hash function as in step 1. If the bit corresponding to the hashed value is 0 in the bit-vector, that tuple is discarded from the relation. The remaining tuples form the reduced relation.
Data Transfer: The bit-vector and the reduced relation are transferred to the site where the join operation needs to be performed.
Join Execution: At the final site, the join is executed between the original relation (e.g., Sailors) and the reduced relation received from the other site.

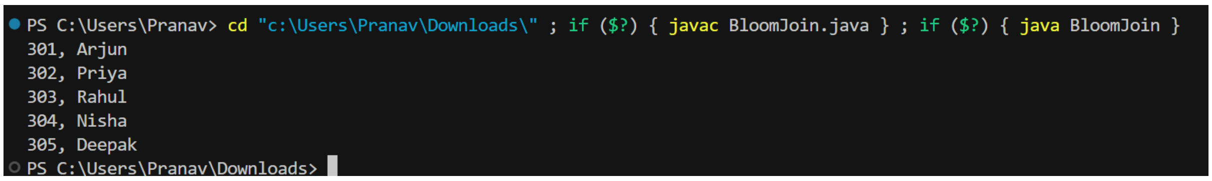
Example:

Consider the strategy of shipping Reserves to London and computing the join at London. Some tuples in (the current instance of) Reserves do not join with any tuple in (the current instance of) Sailors. If we could somehow identify Reserves tuples that are guaranteed not to join with any Sailors tuples, we could avoid shipping them.

A bit-vector of (some chosen) size k is computed by hashing each tuple of Sailors into the range 0 to $k - 1$ and setting bit i to 1 if some tuple hashes to i, and 0 otherwise. In the second step, the reduction of Reserves is computed by hashing each tuple of Reserves (using the sid field) into the range 0 to $k - 1$, using the same hash function used to construct the bit-vector, and discarding tuples whose hash value i corresponds to a 0 bit. Because no Sailors tuples hash to such an i, no Sailors tuple can join with any Reserves tuples that are not in the reduction.

Implementation:

```
import java.util.ArrayList;
import java.util.BitSet;
import java.util.List;
public class BloomJoin {
// Bloom filter size
private static final int FILTER_SIZE = 10;
public static void main(String[] args) {
// Sample relations (tables)
List<String> relation1 = new ArrayList<>();
List<String> relation2 = new ArrayList<>();
// Fill relations with sample data
// Format: "id, name"
relation1.add("201, John");
relation1.add("202, Alice");
relation1.add("203, Bob");
relation1.add("204, Charlie");
relation1.add("205, David");
// Format: "id, branch"
relation2.add("201, Engineering");
relation2.add("202, Sales");
relation2.add("204, Marketing");
relation2.add("206, HR");
relation2.add("207, Finance");
// Perform Bloom join
List<String> result = bloomJoin(relation1, relation2);
// Print result
for (String row : result) {
System.out.println(row);
}
}
public static List<String> bloomJoin(List<String> relation1, List<String> relation2) {
// Create Bloom filter for relation 2
BloomFilter bloomFilter = new BloomFilter(FILTER_SIZE);
for (String row : relation2) {
String[] parts = row.split(",");
String key = parts[0];
bloomFilter.add(key);
}
// Perform join
List<String> result = new ArrayList<>();
for (String row : relation1) {
String[] parts = row.split(",");
String key = parts[0];
if (bloomFilter.contains(key)) {
result.add(row);
}
}
return result;
}
static class BloomFilter {
private final BitSet bitSet;
private final int filterSize;
public BloomFilter(int filterSize) {
this.filterSize = filterSize;
this.bitSet = new BitSet(filterSize);
}
public void add(String key) {
int hash1 = hashFunction1(key);
int hash2 = hashFunction2(key);
int hash3 = hashFunction3(key);
bitSet.set(hash1);
bitSet.set(hash2);
bitSet.set(hash3);
}
public boolean contains(String key) {
int hash1 = hashFunction1(key);
int hash2 = hashFunction2(key);
int hash3 = hashFunction3(key);
return bitSet.get(hash1) && bitSet.get(hash2) && bitSet.get(hash3);
}
private int hashFunction1(String key) {
int x = Integer.parseInt(key);
return (3 * x + 1) % filterSize;
}
private int hashFunction2(String key) {
int x = Integer.parseInt(key);
return (3 * x + 5) % (filterSize - 1); // Use filterSize - 1 as the di visor
}
private int hashFunction3(String key) {
int x = Integer.parseInt(key);
return (3 * x + 7) % (filterSize - 2); // Use filterSize - 2 as the divisor
}
}
}
```
Output:



```
PS C:\Users\Pranav> cd "c:\Users\Pranav\Downloads\" ; if ($?) { javac BloomJoin.java } ; if ($?) { java BloomJoin }
301, Arjun
302, Priya
303, Rahul
304, Nisha
305, Deepak
PS C:\Users\Pranav\Downloads>
```

Experiment 2

Title: Implementation of cube operator in OLAP queries in data Warehousing.

Theory:

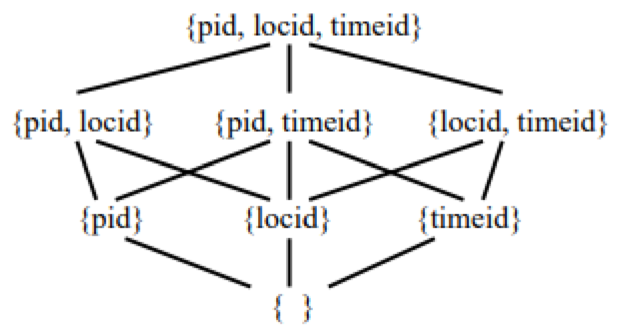
CUBE operator allows generating multiple GROUP BY queries with a single statement, where each query corresponds to a different subset of dimensions for performing roll-up operations.
The CUBE operator is used in the context of multi-dimensional data analysis, where data is organized in a multi-dimensional model with measures and dimensions. In such a scenario, if there are k dimensions associated with a measure, there can be 2^k different combinations of dimensions for which roll-up (aggregation) operations can be performed.
The CUBE operator generates a single result set that contains the results of all possible GROUP BY queries for every subset of the specified dimensions. It is equivalent to generating a lattice of GROUP BY queries, where each node in the lattice corresponds to a GROUP BY query on a specific subset of dimensions.
The CUBE operator is useful for generating cross-tabulations, pivot tables, and other multi-dimensional data analysis tasks, as it allows computing aggregations at multiple levels of detail in a single query.

Example:

For example, consider the following query:

CUBE pid, locid, timeid BY SUM Sales
This query will generate a result set containing the aggregations (SUM of Sales) for all 8 possible subsets of the dimensions {pid, locid, timeid}, which are:

{pid, locid, timeid}
{pid, locid}
{pid, timeid}
{locid, timeid}
{pid}
{locid}
{timeid}
{}

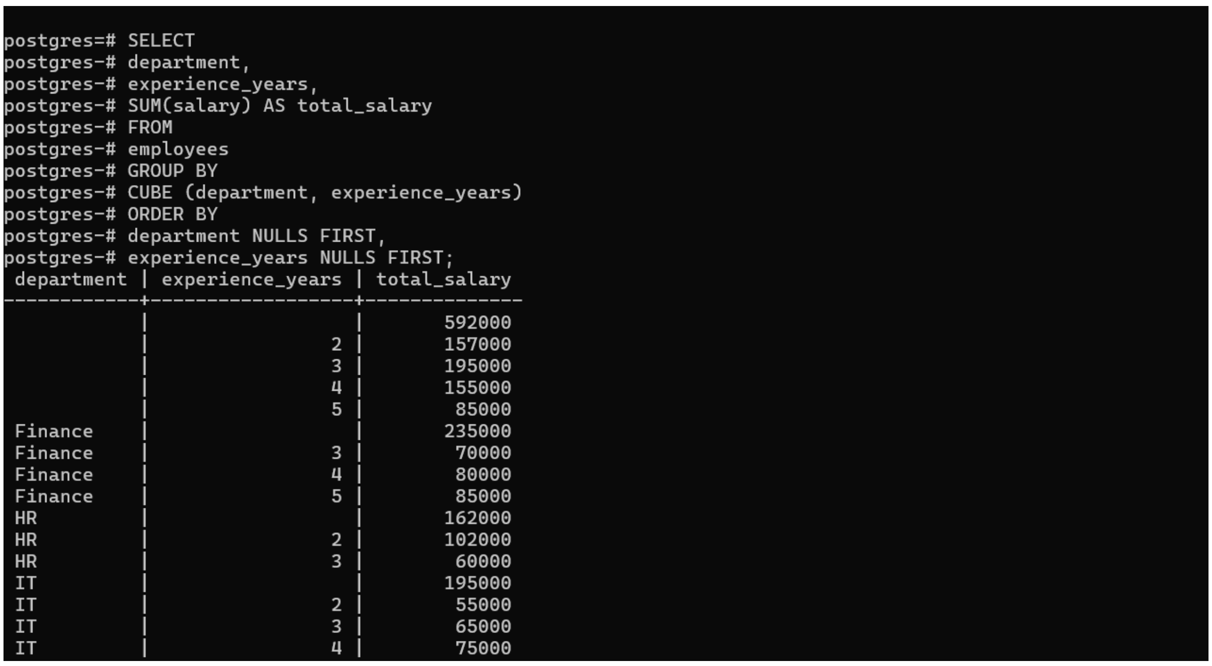


Implementation:

```
-- Create a sample table
CREATE TABLE employees (
department VARCHAR(50),
experience_years INT,
salary NUMERIC
);
-- Insert some sample data
INSERT INTO employees (department, experience_years, salary) VALUES
('HR', 2, 50000),
('HR', 3, 60000),
('HR', 2, 52000),
('Finance', 3, 70000),
('Finance', 4, 80000),
('Finance', 5, 85000),
('IT', 2, 55000),
('IT', 3, 65000),
('IT', 4, 75000);
-- Query to simulate roll up using GROUP BY ROLLUP
SELECT
department,
experience_years,
SUM(salary) AS total_salary
FROM
employees
GROUP BY
ROLLUP (department, experience_years)
ORDER BY
department NULLS FIRST,
experience_years NULLS FIRST;
-- Query to simulate roll down using GROUP BY CUBE
SELECT
```

department,
experience_years,
SUM(salary) AS total_salary
FROM
employees
GROUP BY
CUBE (department, experience_years)
ORDER BY
department NULLS FIRST,
experience_years NULLS FIRST;

Output:



```
postgres=# SELECT
postgres-# department,
postgres-# experience_years,
postgres-# SUM(salary) AS total_salary
postgres-# FROM
postgres-# employees
postgres-# GROUP BY
postgres-# CUBE (department, experience_years)
postgres-# ORDER BY
postgres-# department NULLS FIRST,
postgres-# experience_years NULLS FIRST;
 department | experience_years | total_salary
------------+------------------+--------------
            |                  |       592000
            |                2 |       157000
            |                3 |       195000
            |                4 |       155000
            |                5 |        85000
 Finance    |                  |       235000
 Finance    |                3 |        70000
 Finance    |                4 |        80000
 Finance    |                5 |        85000
 HR         |                  |       162000
 HR         |                2 |       102000
 HR         |                3 |        60000
 IT         |                  |       195000
 IT         |                2 |        55000
 IT         |                3 |        65000
 IT         |                4 |        75000
```

Experiment 3

Title: Implement Abstract Data Type concept.

Theory:

Abstract Data Type (ADT) is a conceptual model that defines the behavior and semantics of a data type, without specifying the implementation details. It encapsulates the internal representation and operations of the data type, allowing users to interact with the data type through a well-defined interface.ADTs consist of two parts: the data type itself and the associated operations or methods defined for that data type. ADTs hide the internal implementation details of the data type and its operations. The users of the ADT do not need to know how the data is stored or how the operations work internally. This concept is known as encapsulation. ADTs provide an abstract interface for interacting with the data type, exposing only the necessary operations and behavior to the users.Object-relational database management systems (ORDBMSs) allow users to define their own custom ADTs, in addition to the built-in ADTs. The "abstract" term in ADT refers to the fact that the database system does not need to know the internal details of how the data is stored or how the operations are implemented. It only needs to know the available methods, their input and output types, and how to invoke them. Encapsulation of ADT internals simplifies the implementation of ORDBMSs, as they can treat different data types and their operations uniformly, without anticipating the specific types and methods that users might want to add.

Example:

Suppose Dinky needs to store and manage the video footage of Herbert the Worm films in their database. They can define a custom ADT called VideoClip to represent a video clip or segment from a film.

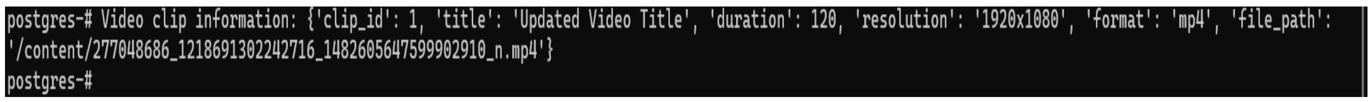
The VideoClip ADT could have the following components:

Data Representation: The internal representation of a video clip could be a binary large object (BLOB) or a file path pointing to the actual video file on the file system.
Associated Methods:
play(start_time, end_time): Plays the video clip between the specified start and end times.
trim(start_time, end_time): Creates a new VideoClip object containing the trimmed portion of the original clip.
concat(other_clip): Concatenates the current clip with another VideoClip and returns a new VideoClip object.
export(format, quality): Exports the video clip to a specified format (e.g., MP4, AVI) and quality setting.
add_subtitle(language, subtitle_file): Adds subtitles in the specified language to the video clip.
With this VideoClip ADT, Dinky can store video clips in their database and perform various operations on them. For example, they could create a table called FilmClips with columns like clip_id, film_title, scene_description, and video_data (of type VideoClip).

Implementation:

```
class VideoClip:
    def __init__(self, clip_id, title, duration, resolution, format, file_path):
        self.clip_id = clip_id  # Unique identifier for the video clip
        self.title = title  # Title of the video clip
        self.duration = duration  # Duration of the video clip (in seconds)
        self.resolution = resolution  # Resolution of the video clip (e.g., "1920x1080")
        self.format = format  # Format of the video clip (e.g., "mp4", "avi")
        self.file_path = file_path  # File path where the video clip is stored

    def play(self):
        # Method to play the video clip
        print("Playing video:", self.title)

    def pause(self):
        # Method to pause the video clip
        print("Pausing video:", self.title)

    def stop(self):
        # Method to stop the video clip
        print("Stopping video:", self.title)

    def get_info(self):
        # Method to get information about the video clip
        info = {
            "clip_id": self.clip_id,
            "title": self.title,
            "duration": self.duration,
            "resolution": self.resolution,
            "format": self.format,
            "file_path": self.file_path
        }
        return info

    def update_title(self, new_title):
        # Method to update the title of the video clip
        self.title = new_title

    def update_duration(self, new_duration):
        # Method to update the duration of the video clip
        self.duration = new_duration

    def update_resolution(self, new_resolution):
        # Method to update the resolution of the video clip
        self.resolution = new_resolution

    def update_format(self, new_format):
        # Method to update the format of the video clip
        self.format = new_format

    def update_file_path(self, new_file_path):
        # Method to update the file path of the video clip
        self.file_path = new_file_path

# Example usage:
if __name__ == "__main__":
   # Create a video clip object
  video_clip = VideoClip(1, "Sample Video", 120, "1920x1080", "mp4",
"/content/277048686_1218691302242716_1482605647599902910_n.mp4")

# Play the video clip
  print("Playing video:", video_clip.title)

# Update the title of the video clip
  video_clip.update_title("Updated Video Title")

# Get information about the video clip
  clip_info = video_clip.get_info()
  print("Video clip information:", clip_info)
```

Output:



```
postgres-# Video clip information: {'clip_id': 1, 'title': 'Updated Video Title', 'duration': 120, 'resolution': '1920x1080', 'format': 'mp4', 'file_path':
'/content/277048686_1218691302242716_1482605647599902910_n.mp4'}
postgres-#
```

Experiment 4

Title: Implement Path expression for XML data.

Theory:

Path expressions are used to navigate XML input documents to select elements and attributes of interest. This chapter explains how to use path expressions to select elements and attributes from an input document and apply predicates to filter those results. It also covers the different methods of accessing input documents.
A path expression is made up of one or more steps that are separated by a slash (/) or double slashes (//).

Path Expressions and Context
A path expression is always evaluated relative to a particular context item, which serves as the starting point for the relative path. Some path expressions start with an initial step that sets the context item, as in:

A path expression can also be relative.
This means that the path expression will be evaluated relative to the current context node, which must have been previously determined outside the expression. It may have been set by the processor outside the scope of the query, or in an outer expression.

Steps and changing context
The context item changes with each step. A step returns a sequence of zero, one, or more nodes that serve as the context items for evaluating the next step.
The final step, number, is evaluated in turn for each product child in this sequence. During this process, the processor keeps track of three things:
The context node itself—for example, the product element that is currently being processed
The context sequence, which is the sequence of items currently being processed—for example, all the product elements
The position of the context node within the context sequence, which can be used to retrieve nodes based on their position

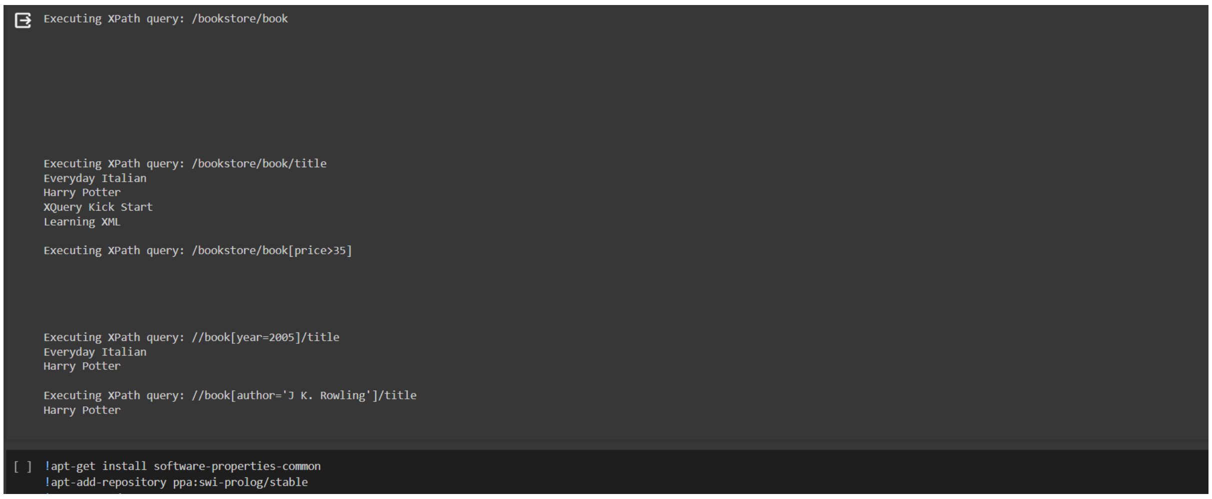
Example:

doc("catalog.xml")/catalog	The catalog element that is the outermost element of the document
doc("catalog.xml")//product	All product elements anywhere in the document
doc("catalog.xml")//product/@dept	All dept attributes of product elements in the document
doc("catalog.xml")/catalog/*	All child elements of catalog
doc("catalog.xml")/catalog/*/number	All number elements that are grandchildren of catalog

Implementation:

```
from lxml import etree

# Parse the XML file
tree = etree.parse("doc.xml")

# Define a function to execute XPath queries
def execute_xpath(query):
    results = tree.xpath(query)
    for result in results:
        print(result.text)

# Example XPath queries
queries = [
    "/bookstore/book",                  # Select all books
    "/bookstore/book/title | /bookstore/book/author",           # Select titles of all books
    "/bookstore/book[price>35]",       # Select books with a price greater than 35
    "//book[year=2005]/title",         # Select titles of books published in 2005
    "//book[author='J K. Rowling']/title" # Select titles of books by J K. Rowling
]

# Execute the XPath queries
for query in queries:
    print("Executing XPath query:", query)
    execute_xpath(query)
    print()
```

Output:



```
Executing XPath query: /bookstore/book







Executing XPath query: /bookstore/book/title
Everyday Italian
Harry Potter
XQuery Kick Start
Learning XML

Executing XPath query: /bookstore/book[price>35]





Executing XPath query: //book[year=2005]/title
Everyday Italian
Harry Potter

Executing XPath query: //book[author='J K. Rowling']/title
Harry Potter
```

```
[ ]  !apt-get install software-properties-common
     !apt-add-repository ppa:swi-prolog/stable
```

Experiment 5

Title: Implement Region Quad trees

Theory:

A Region Quad tree is a hierarchical data structure used for spatial indexing and organizing multidimensional data, commonly applied in Geographic Information Systems (GIS) and spatial databases. It's a variation of the Quadtree data structure, tailored for managing regions or spatial objects within a two-dimensional space.

Recursive Decomposition of Data Space: The Region Quad tree decomposes the data space recursively into smaller square-shaped regions. Each node in the tree represents a region of the data space, with the root node representing the entire space.
Node Structure: Each internal node of the tree has four children, corresponding to the four quadrants into which the node's space is partitioned. The quadrants are identified as 00 (bottom left), 01 (top left), 10 (bottom right), and 11 (top right).
Point Representation: Leaf nodes of the tree correspond to individual points in the data space. The Z-value (or Z-ordering value) of a point is obtained by traversing the path from the root to the leaf node for that point and concatenating all the edge labels encountered on the path.
Storage and Retrieval: Regions or objects within the data space are stored in the database management system (DBMS) using unique identifiers. These regions can be represented using a set of records, each containing a Z-value and other relevant information about the region. The records are clustered and indexed using a B+ tree based on the Z-values for efficient retrieval.
Generalization to Higher Dimensions: The concept of Region Quad trees can be extended to higher dimensions (k dimensions). In k dimensions, each node partitions the space into 2^k subregions. For instance, in two dimensions (k = 2), the space is divided into four equal quadrants, while in three dimensions (k = 3), it would be divided into eight octants.

Example:

Suppose we have a 2-dimensional space represented by a square area with coordinates ranging from (0, 0) to (10, 10). We want to organize and manage points within this space using a Region Quad tree.

Initialization: Start with the root node representing the entire space, labeled as '00' to indicate it covers the entire area.
Insertion of Points: Let's say we have four points in our space:
Point A: Coordinates (2, 3)
Point B: Coordinates (8, 5)
Point C: Coordinates (6, 9)
Point D: Coordinates (4, 2)

We insert these points into the Region Quad tree. Starting from the root node, we recursively partition the space and insert each point into the appropriate leaf node based on its coordinates.
Partitioning the Space: The space is divided into four quadrants: bottom-left (00), top-left (01), bottom-right (10), and top-right (11).
Point A (2, 3) goes into the bottom-left quadrant (00).
Point B (8, 5) goes into the top-right quadrant (11).
Point C (6, 9) goes into the top-right quadrant (11).
Point D (4, 2) goes into the bottom-left quadrant (00).
Each quadrant may further divide if necessary to accommodate additional points.

Traversal and Retrieval: To retrieve points within a specific region, we traverse the Region Quad tree starting from the root node and descend into the appropriate child nodes based on the region of interest. For example:
If we want to retrieve points within the area bounded by (0, 0) and (5, 5), we would traverse the tree starting from the root and descend into the bottom-left and top-left quadrants.

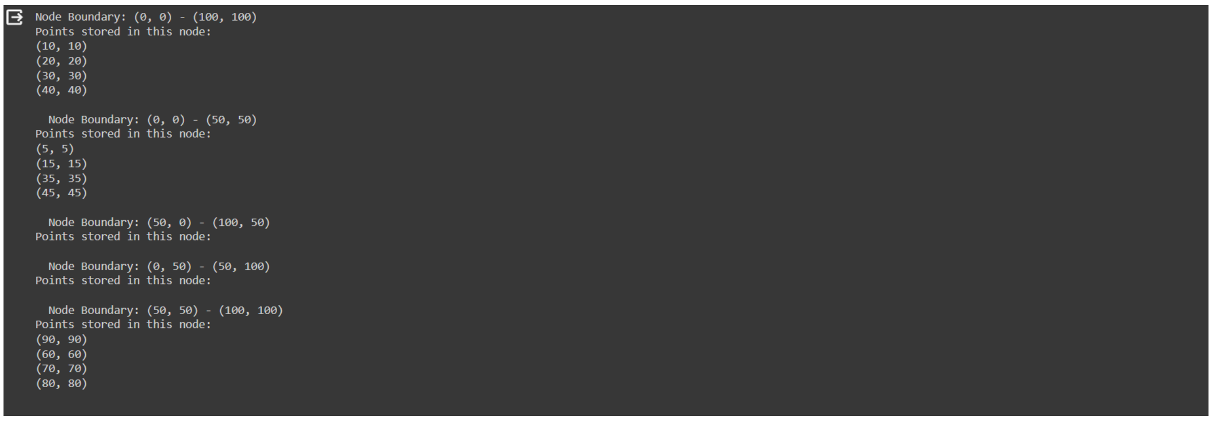
Implementation:

```
#include <iostream>
#include <vector>
#include <memory>

// Define a Point structure to represent a point in 2D space
struct Point {
   double x;
   double y;
   Point(double x, double y) : x(x), y(y) {}
};

// Define a Rectangle structure to represent a rectangular region
struct Rectangle {
   double xMin, xMax, yMin, yMax;
   Rectangle(double xmin, double xmax, double ymin, double ymax) : xMin(xmin),
xMax(xmax), yMin(ymin), yMax(ymax) {}

   // Function to check if a point is contained within the rectangle
   bool contains(const Point& point) const {
      return (point.x >= xMin && point.x <= xMax && point.y >= yMin && point.y <=
yMax);
   }
};

// Define the QuadTree Node structure
```

```
struct QuadTreeNode {
   Rectangle boundary;
   std::vector<Point> points;
   std::shared_ptr<QuadTreeNode> children[4]; // 0: bottom left, 1: bottom right, 2: top left,
3: top right

   QuadTreeNode(const Rectangle& boundary) : boundary(boundary) {
      for (int i = 0; i < 4; ++i) {
         children[i] = nullptr;
      }
   }
};

// QuadTree class
class QuadTree {
private:
   std::shared_ptr<QuadTreeNode> root;

   void insertHelper(const Point& point, std::shared_ptr<QuadTreeNode>& node) {
      if (!node->boundary.contains(point)) {
         return; // Point does not belong to this node's boundary
      }

      if (node->points.size() < 4) { // Maximum points per node
         node->points.push_back(point);
      } else {
         if (!node->children[0]) { // If children not yet created, create them
            double xMid = (node->boundary.xMin + node->boundary.xMax) / 2;
            double yMid = (node->boundary.yMin + node->boundary.yMax) / 2;

            node->children[0] =
std::make_shared<QuadTreeNode>(Rectangle(node->boundary.xMin, xMid,
node->boundary.yMin, yMid));
            node->children[1] = std::make_shared<QuadTreeNode>(Rectangle(xMid,
node->boundary.xMax, node->boundary.yMin, yMid));
            node->children[2] =
std::make_shared<QuadTreeNode>(Rectangle(node->boundary.xMin, xMid, yMid,
node->boundary.yMax));
            node->children[3] = std::make_shared<QuadTreeNode>(Rectangle(xMid,
node->boundary.xMax, yMid, node->boundary.yMax));
         }

         // Recursively insert into appropriate child node
         for (int i = 0; i < 4; ++i) {
                insertHelper(point, node->children[i]);
            }
        }
    }

    void printPointsHelper(const std::shared_ptr<QuadTreeNode>& node, int depth) const {
        if (!node) {
            return;
        }

        // Print indentation based on depth
        for (int i = 0; i < depth; ++i) {
            std::cout << "  ";
        }

        std::cout << "Node Boundary: (" << node->boundary.xMin << ", " <<
node->boundary.yMin << ") - (" << node->boundary.xMax << ", " << node->boundary.yMax
<< ")\n";
        std::cout << "Points stored in this node:\n";
        for (const auto& point : node->points) {
            std::cout << "(" << point.x << ", " << point.y << ")\n";
        }
        std::cout << "\n";

        // Recursively print points stored in child nodes
        for (int i = 0; i < 4; ++i) {
            printPointsHelper(node->children[i], depth + 1);
        }
    }

public:
    QuadTree(const Rectangle& boundary) {
        root = std::make_shared<QuadTreeNode>(boundary);
    }

    void insert(const Point& point) {
        insertHelper(point, root);
    }

    void printPoints() const {
        printPointsHelper(root, 0);
    }
};
```

```
int main() {
    Rectangle boundary(0, 100, 0, 100); // Define the boundary of the quad tree
    QuadTree quadTree(boundary);

    // Insert points into the quad tree
    quadTree.insert(Point(10, 10));
    quadTree.insert(Point(20, 20));
    quadTree.insert(Point(30, 30));
    quadTree.insert(Point(40, 40));
    quadTree.insert(Point(90, 90));
    quadTree.insert(Point(5, 5));
    quadTree.insert(Point(15, 15));
    quadTree.insert(Point(35, 35));
    quadTree.insert(Point(45, 45));
    quadTree.insert(Point(60, 60));
    quadTree.insert(Point(70, 70));
    quadTree.insert(Point(80, 80));

    // Print points stored in each node
    quadTree.printPoints();

    return 0;
}
```

Output:



```
Node Boundary: (0, 0) - (100, 100)
Points stored in this node:
(10, 10)
(20, 20)
(30, 30)
(40, 40)

  Node Boundary: (0, 0) - (50, 50)
Points stored in this node:
(5, 5)
(15, 15)
(35, 35)
(45, 45)

  Node Boundary: (50, 0) - (100, 50)
Points stored in this node:

  Node Boundary: (0, 50) - (50, 100)
Points stored in this node:

  Node Boundary: (50, 50) - (100, 100)
Points stored in this node:
(90, 90)
(60, 60)
(70, 70)
(80, 80)
```

Experiment 6

Title: Implementation of datalog queries for deductive databases.

Theory:

Datalog is a relational query language inspired by Prolog, the well known logic programming language; indeed, the notation follows Prolog.

Rule1:
Components(Part, Subpart) :- Assembly(Part, Subpart, Qty).

Rule 2:
Components(Part, Subpart) :- Assembly(Part, Part2, Qty),
Components(Part2, Subpart).

The first rule should be read as follows:
For all values of Part, Subpart, and Qty,
if there is a tuple Part, Subpart, Qty in Assembly,
then there must be a tuple Part, Subparti in Components.

The second rule should be read as follows:
For all values of Part, Part2, Subpart, and Qty,
if there is a tuple Part, Part2, Qty in Assembly and
a tuple Part 2, Subparti in Components,
then there must be a tuple hPart, Subparti in Components.

The part to the right of the :- symbol is called the **body** of the rule, and the part to the left is called the **head** of the rule. The symbol :- denotes **logical implication**; if the tuples mentioned in the body exist in the database, it is implied that the tuple mentioned in the head of the rule must also be in the database. (Note that the body could be empty; in this case, the tuple mentioned in the head of the rule must be included in the database.) Therefore, if we are given a set of Assembly and Components tuples, each rule can be used to **infer, or deduce,** some new tuples that belong in Components. This is why database systems that support Datalog rules are often called **deductive database systems.**

Example:

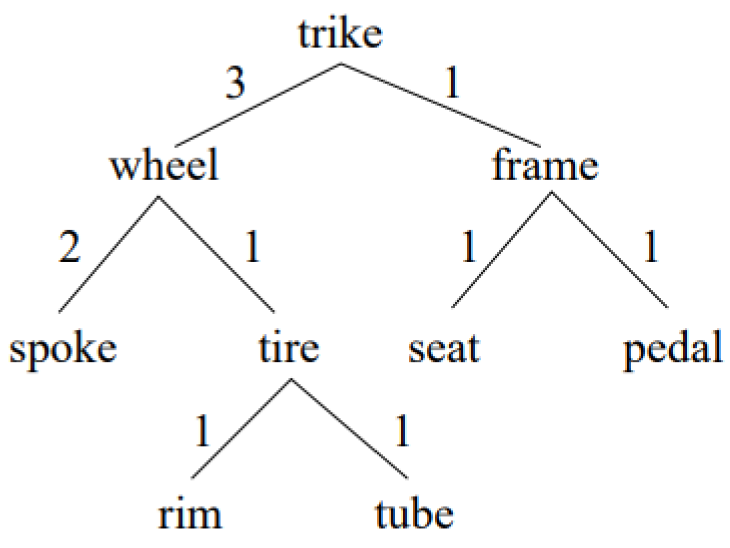


Figure 27.2 Assembly Instance Seen as a Tree

part	*subpart*
trike	spoke
trike	tire
trike	seat
trike	pedal
wheel	rim
wheel	tube

Figure 27.3 Components Tuples Obtained by Applying the Second Rule Once

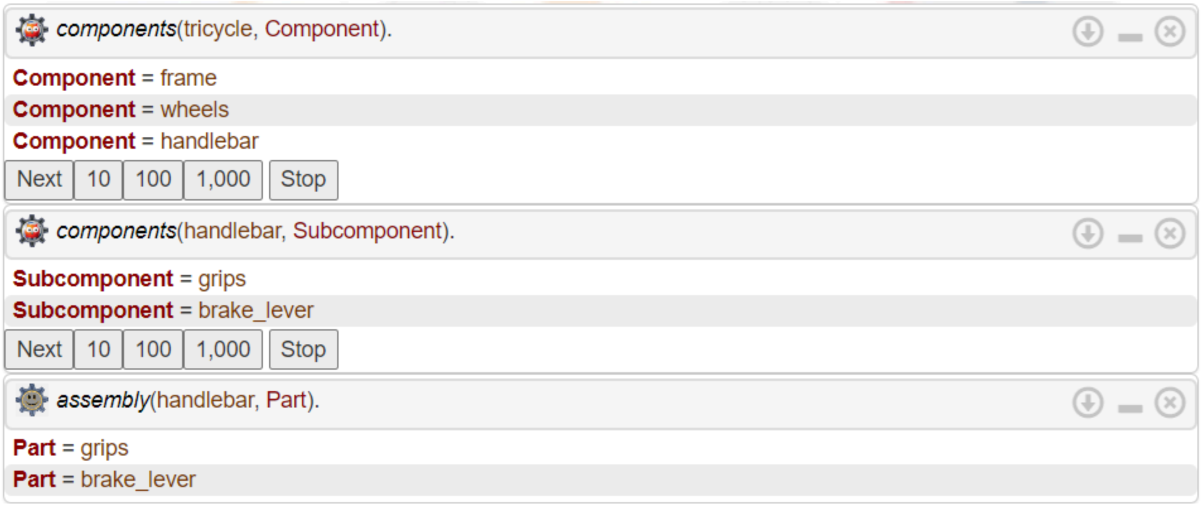
part	*subpart*
trike	spoke
trike	tire
trike	seat
trike	pedal
wheel	rim
wheel	tube
trike	rim
trike	tube

Figure 27.4 Components Tuples Obtained by Applying the Second Rule Twice

Implementation:

```
% Facts: Assembly relationships
assembly(tricycle, frame).
assembly(tricycle, wheels).
assembly(tricycle, handlebar).
assembly(frame, pedal).
assembly(frame, seat).
assembly(frame, mudguard).
assembly(wheels, front_wheel).
assembly(wheels, rear_wheel).
assembly(front_wheel, spokes).
assembly(rear_wheel, spokes).
assembly(handlebar, grips).
assembly(handlebar, brake_lever).
assembly(brake_lever, hydraulic_hose).

% Rules: Components
components(Part, Subpart) :- assembly(Part, Subpart).
components(Part, Subpart) :- assembly(Part, Intermediate), components(Intermediate,
Subpart).
```

Output:



```
components(tricycle, Component).
Component = frame
Component = wheels
Component = handlebar
components(handlebar, Subcomponent).
Subcomponent = grips
Subcomponent = brake_lever
assembly(handlebar, Part).
Part = grips
Part = brake_lever
```

Expt 2 option 2PC

```
import socket

class CoordinatorServer:
    def _init_(self, num_participants):
        self.server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.participants = []
        self.num_participants = num_participants
        self.subordinate_responses = {}

    def run(self):
        try:
            self.server_socket.bind(('localhost', 12345))
            self.server_socket.listen(5)

            print("Waiting for the Participants...\n")
            while len(self.participants) < self.num_participants:
                participant_socket, participant_addr = self.server_socket.accept()
                self.participants.append(participant_socket)
                print(f"Participant {len(self.participants)} connected from:
{participant_addr}")

            self.start_two_phase_commit()

            self.receive_responses_from_participants()
            self.make_decision()
            self.receive_acknowledgments()
            self.write_terminate_message()
            self.close_connections()
        except Exception as e:
            print(e)

    def start_two_phase_commit(self):
        print("\nStarting Two-Phase Commit protocol...")
        self.send_prepare_message()

    def send_prepare_message(self):
        print("\nSending PREPARE message to all participants...")
        for participant in self.participants:
            participant.send("PREPARE".encode())

    def receive_responses_from_participants(self):
        print()
        for participant in self.participants:
            response = participant.recv(1024).decode()
            self.subordinate_responses[participant] = response
            print(f"Received {response} response from participant:
{participant.getpeername()}")

    def make_decision(self):
        if all(response == "YES" for response in
self.subordinate_responses.values()):
            print("\nTransaction COMMITTING...")
            self.send_message_to_all("COMMIT")
        else:
            print("\nTransaction aborting...")
            self.send_message_to_all("ABORT")

    def send_message_to_all(self, message):
        for participant in self.participants:
            participant.send(message.encode())

    def receive_acknowledgments(self):
        print("\nWaiting for acknowledgments from all participants...")
        ack_count = 0
        while ack_count < len(self.participants):
            for participant in self.participants:
                acknowledgment = participant.recv(1024).decode()
                if acknowledgment == "ACK":
                    ack_count += 1
                    print(f"Received ACK from participant:
{participant.getpeername()}")

    def write_terminate_message(self):
        print("\nTransaction terminated successfully.")

    def close_connections(self):
        try:
            self.server_socket.close()
            for participant in self.participants:
                participant.close()
        except Exception as e:
            print(e)

if _name_ == "_main_":
    num_participants = int(input("Enter the number of participants: "))
    coordinator_server = CoordinatorServer(num_participants)
    coordinator_server.run()
```