Media Engineering and Technology Faculty
German University in Cairo

Algorithms for Conjunctive Queries

Bachelor Thesis

Author: Youssef Atef Mourad Gad
Supervisors: Prof. Haythem Osman Ismail

Submission Date: 1 June, 2023

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor degree

(ii) due acknowlegement has been made in the text to all other material used

---

Youssef Atef Mourad Gad
1 June, 2023

Acknowledgments

Firstly, I thank God for His grace, care, and blessings He has given me throughout this work. I'd like to thank Dr. Haythem Ismail for his continuous guidance and assistance. Last but not least, I thank my family for their genuine support and encouragement for me.

Abstract

Conjunctive queries are a crucial class of queries in the database field, allowing efficient joining of tables in a relational database to retrieve specific data tuples satisfying given conditions. Numerous algorithms have been proposed in the literature for processing conjunctive queries, each employing different data structures and optimization techniques. This work aims to provide a comprehensive study of these algorithms by implementing them and analyzing their performance. Specifically, we will examine three algorithms, including one that targets a specific instance of conjunctive queries and two that are more general. It will also present a detailed analysis of the data structures and optimizations employed by each algorithm.

Contents

Chapter 1

Introduction

1.1 Motivation

Traditional join methods, such as binary joins where we take every row in a table and compare it with every row in another table in terms of a specific property, can become expensive for large datasets due to their time complexity. As the number of tables being joined increases, the computational cost of the join operation can quickly become prohibitively expensive as it will become a series of binary joins together, even for relatively small datasets.

Suppose we have three tables representing different aspects of an e-commerce system, *Customer*, *Order*, and *Product*, each representing different aspects of the system. The *Customer* table contains information such as customer names and addresses, the *Order* table stores important details about each order (e.g., order ID, customer ID, date), and the *Product* table provides descriptions of the products available in the inventory.

Consider having a query that aims to retrieve the names of customers who have placed orders and also obtain the descriptions of the products within those orders. To accomplish this, we need to perform matching operations between the tables. In particular, we start by joining the *Customer* table with the *Order* table, matching customer IDs in both tables to identify customers who have placed orders. This join operation involves comparing each record in the *Customer* table with each record in the *Order* table. Since both tables contain 100 records, this initial matching operation would require 100 * 100 = 10,000 comparison operations. Once we have obtained the resulting table from the previous join, we proceed to perform another matching operation between this intermediate table and the *Product* table. The goal is to retrieve the descriptions of the products associated with each order. Similar to the previous join, this operation entails comparing each record in the intermediate table with each record in the *Product* table. The second matching operation would incur a total of 10,000 * 100 = 1,000,000 comparison operations.

The example demonstrates the substantial increase in computational comparisons when applying a basic matching algorithm to relatively small-sized tables.

Also another factor that can affect the size of computational comparison is the order of which table you choose to match first. We basically have two options for table join orders, either we join *Customer* with *Order* first and then join in *Product*, or we join *Product* with *Order* first and then join in *Customer*. Generally, starting by joining the two smaller tables together will reduce the total number of rows getting passed around for the remainder of the matching process.

In summary, traditional join methods like binary joins can be computationally expensive for large datasets, even for relatively small-sized tables. Joining multiple tables sequentially involves performing a series of binary joins, which can quickly accumulate a significant number of comparison operations. Furthermore, the order in which the tables are joined can impact the efficiency of the matching process. By strategically choosing the join order, such as starting with the smaller tables, we can potentially reduce the number of rows involved in subsequent joins, leading to fewer comparison operations and improved query performance.

1.2 Objective

In this study, three distinct conjunctive query processing algorithms are implemented, analyzed, and compared in terms of their runtime with traditional binary join, focusing on their performances in handling complex and large queries.

One of the implemented algorithms tackles a specific class of queries called Loomis-Whitney instances. The other two handle general queries with no specifications or constraints.

Due to time complications, one of these two algorithms was not fully completed. However, significant progress has been made on this algorithm, and we will provide an overview of the work that has been accomplished thus far.

1.3 Organization

The following chapter gives a general background, overviewing general notations and data structures used in this work. After that, the algorithms chapters demonstrate each algorithm discussion and implementation. Next, The results chapter mentions the tests used, the results, and their interpretations.

Chapter 2

Background

In preparation for the implementation and analysis of efficient join algorithms for conjunctive queries, it is essential to establish a strong foundation in the underlying concepts and data structures used in database literature. In this chapter, we will delve into the fundamental concepts of relational databases exploring the various notations associated with them, the statement of conjunctive query problem, and important data structures used in the join algorithm.

2.1 Formal Description for Relational Database

A relational database refers to a structured collection of data items that are organized in formally-described tables from which data can be accessed or retrieved, where it has the ability to establish links, or relationships, between information by joining tables. Thus, the utilization of the word "relation" to refer to a table [3].

A relation is a table structure where the data is stored in. The columns are units of named data that have a particular data type (e.g., number, text, or date) referred to as "attribute", while the rows are records of data having values for each column. Each row is referred to as "tuple". Many tuples can have the same values for a particular attribute.

A join is the process of combining rows from two or more tables, based on a shared attribute between those tables, where these rows must share the same value for that attribute.

A semi-join operation is a type of join operation performed on two tables to retrieve only the rows from the first table that have matching values in the second table. Unlike other types of joins that return a combination of rows from both tables, a semi-join only focuses on the rows from the first table and filters them based on the presence of matching values in the second table.

Example 2.1

Suppose we have two tables, table *Student* on a set of attributes that describe student info ID and $Name$ and the course that the student is enrolled in $CourseID$, and table $Course$ on the set of attributes $CourseID$ and $CourseName$.

Table 2.1: Student Table.

Student ID	Name	Course ID
1	John Smith	101
2	Jane Doe	102
3	Michael Johnson	101
4	Dianna Prince	104

Table 2.2: Course Table.

Course ID	Course Name
101	Math
102	Physics
103	English

When joining these two tables, there should be at least a common attribute between them in this case it is the attribute $CourseID$. This join operation translates to a query that associates each student with the course name in which he/she is enrolled, creating a third table in the process.

Table 2.3: Joined Table (Student-Course).

Student ID	Name	Course Name
1	John Smith	Math
2	Jane Doe	Physics
3	Michael Johnson	Math

2.2 Notation

To discuss and develop efficient algorithms for database management, it is necessary to define some notations that are commonly used in this field. A relational database is made up of tables or relations, and we use capitalized letters such as $R, S, T, ...$ to denote a relation. Each relation contains attributes, or columns, denoted by small letters such as $a, b, c, ...$ The set of attributes of a certain relation R is denoted as $\overline{A}(R)$.

We define a certain relation as $R(a, b)$ where a and b are the attributes on the relation R. Also, the projection of a relation R on an attribute a is denoted by $\pi_a(R)$ which returns only the tuples of a on R.

The joining process is denoted by the symbol "⋈" between two or more relations. indicating the join between two relations, such as R and S, the notation $R \bowtie S$ is commonly used.

semi-join operation is a type of join operation performed on two tables to retrieve only the rows from the first table that have matching values in the second table, denoted by the symbol "⋉" between two or more relations.

2.3 Conjunctive Query Problem Statement

A Conjunctive Query (CQ) is a type of query used to describe a joining problem in relational databases and to retrieve data that satisfies multiple conditions simultaneously. It is composed of several sub-queries, each of which specifies a condition that must be met for a tuple to be included in the returned result set. The sub-queries are combined using the logical operator 'AND', hence the name "conjunctive". The result of a CQ is a set of tuples that satisfy all of the specified conditions [6].

We denote a CQ q as

$$q(x_1, ..., x_k) : R_1(y_1), ..., R_n(y_n). \tag{2.1}$$

The expression $q(x_1, ..., x_k)$ is called the *head* of the query, q denotes an instance of a CQ, and $R_1(y_1), ..., R_n(y_n)$ is called the *body* of the query. Each expression $R_i(y_i)$ is called an *atom*, where $y_i \subseteq \overline{A}(R_i)$. Notice that the atom is different from a relation since many atoms can correspond to the same relation.

Example 2.2

$$q_1(a, c) : R(a, b), S(b, c).$$
$$q_2() : R(a, b), S(b, c), T(a).$$

In the CQ q_1, The query is executed by finding the common values of attribute b in both relations, R and S. It is basically the joining of these relations on attribute b, and the tuples that satisfy the condition are returned. We observe that the head of the query specifies the attributes a and c as the primary focus of interest. This indicates that the resulting set of tuples obtained from executing q_1 will be projected or selected based on these specific attributes. Therefore, the query outcome will include only those tuples that possess values for attributes a and c, while excluding other attributes that are not explicitly mentioned in the head of q_1.

In the CQ q_2, the head of the query is represented by an empty set, indicating the absence of specific attributes. This implies that the purpose of q_2 is solely to inquire whether there exist any tuples in the database that satisfy the specified query conditions or not. This type of query is called a *boolean CQ*.

2.4 Data Structures

In this section, we will investigate some data structures used in the joining algorithms.

2.4.1 Hyper Graphs

A mathematical construction that expands on the idea of a graph is called a hypergraph. Instead of just two vertices as in a conventional graph, edges in a hypergraph can link any number of vertices (also known as nodes or points). A set of vertices V and a set of hyperedges E make up a formal hypergraph $H = (V, E)$. Each hyperedge connects any number of vertices and is a subset of vertices [4].

A hypergraph for a CQ q, *H(q) = (V, E)*, can be used to describe the query where the set of variables V consists of all attributes in the body of the query, while the set E of hyperedges contains, for each atom in the query, the set of attributes that appear in this atom.

Example 2.3

$$q() : R(a, b), S(b, c), T(c, d).$$

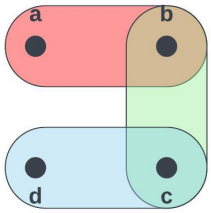


Figure 2.1: Hypergraph for a query.

The figure above is a hypergraph $H = (V, E)$ representation of the conjunctive query q in the example, where the attributes present in the body are represented as vertices $V = \{a, b, c, d\}$, and each set of attributes that appear in each atom is an edge $E = \{\{a, b\}, \{b, c\}, \{c, d\}\}$.

The hypergraph notation is used to describe a conjunctive query to introduce the *fractional edge cover* that plays a central role in the implementation of join algorithms.

2.4.2 Tries

A trie is a tree-like data structure used to describe a relation on its set of attributes. Suppose relation $R(x, y, z)$, a trie for such relation is built by a root node having children for x values and each child in set of x has children for y values and so on until reaching the last attribute for the relation and its values are the leaf nodes of the trie. Therefore, It can be said that each attribute in the relation occupies a level in the trie structure referred to as *depth*.

To navigate through the nodes of the trie a pointer is used called *a trie iterator*. This iterator gets the value of the node it points to and, also it is implemented having several methods such as:

1. open(): Proceed to the first key at the next depth.
2. up(): Return to the parent key at the previous depth.
3. next(): goes to the next child of the same parent at a certain depth.
4. atEnd(): if it is at the last child of the same parent at a certain depth.

Example 2.4

Table 2.4: Relation $R(x, y, z)$.

x	y	z
1	3	4
1	3	5
1	4	6
1	4	8
1	4	9
1	5	2
3	5	2

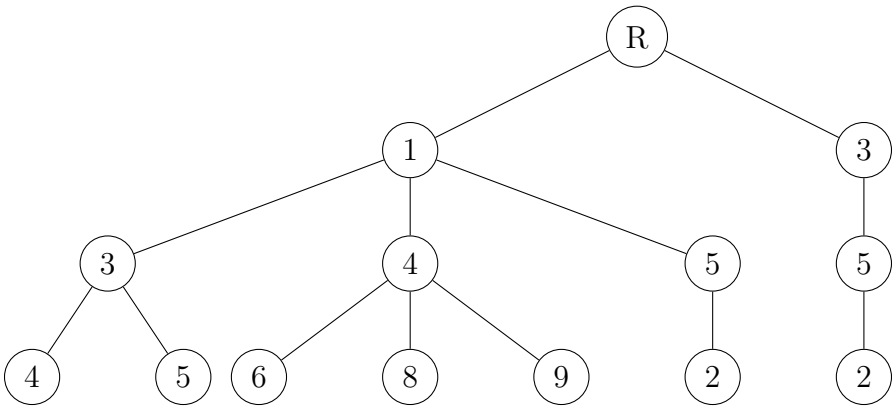


Figure 2.2: Trie representation of relation R.

In the above example, invoking open() thrice on an iterator positioned at R would move to the leaf node (1, 3, 4); next() would then move the iterator to leaf node (1, 3, 5); another next() would result in the iterator being atEnd(). The sequence up(), next(), open() would then advance the iterator to the leaf node (1, 4, 6).

2.5 Fractional Edge Cover

A fractional cover is a collection of non-negative weights y assigned to its hyperedges E such that for each vertex $v \in V$ in the hypergraph $H = (V, E)$, the sum of weights of hyperedges containing that vertex is at least 1 [5]. The term *fractional edge cover* is used to differentiate it from the *integral edge cover*, where each hyperedge is assigned a weight of either 0 or 1.

To find the *minimal fractional cover*, the fractional cover problem involves satisfying two conditions simultaneously.

$$\sum_{e:v \in e} y_e \geq 1, \quad \text{for all } v \in V. \tag{2.2}$$

$$y_e \geq 0, \quad \text{for any } e \in E. \tag{2.3}$$

A fractional cover is said to be minimal if there is no other fractional cover solution with a smaller total weight. This is considered a linear programming problem where the objective is to minimize the summation of weights as much as possible.

Example 2.5

Suppose having a conjunctive query $q(a,b,c) = R(a,b), S(b,c), T(a,c)$, so the hypergraph for q will be $H(q) = (V, E)$, where $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}, \{a, c\}\}$. The below figure describes $H(q)$.

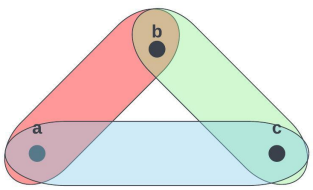


Figure 2.3: Hypergraph for a triangle query.

A possible fractional edge cover is $y_R = y_S = 1$, and $y_T = 0$. In this case, the sum of the weights is 2. Another possible solution is $y_R = y_S = y_T = 1/2$ making the sum of weights $3/2$ which is smaller than the previous solution.

Chapter 3

Algorithm for Loomis-Whitney Instances

In this chapter, we will go into great depth in the domain of Loomis-Whitney (LW) instances, a specialized class of hypergraphs, and we will discuss an implemented algorithm that solves the joining problem in an efficient way for these instances.

3.1 Definition of a LW Instance

A LW instance is a hypergraph H for a joining problem, where $H = (V, E)$ such that E is the collection of all subsets of V of size $|V| - 1$ [6], letting $n \geq 2$ then $V = [n]$. So we can use a LW instance to describe a certain class of CQs, each relation in the query has a number of attributes that is one less than the total number of relations in the query. Furthermore, it is ensured that no relation shares the same set of attributes as any other relation within the query.

Example 3.1

Consider three relations *R, S, and T* and the total attributes between them are the set of *a, b, and c*. For a CQ q to be a LW instance it has to satisfy the above conditions, $q() = R(a,b),S(b,c),T(a,c)$. This is also can be called a *triangle query*, as the hypergraph representing q will form a triangular shape.

3.2 Intuition Behind the Algorithm

This algorithm tries to efficiently find the result of the query by not including tuples of a size that will surpass a certain bound $P = \prod_{e \in E} N_e^{\frac{1}{n-1}}$ (the size bound from LW

inequality) [6], where R_e is the set of input relations to the query and $|Re| = Ne$, that would blow up the size of an intermediate relation.

The algorithm uses two sets. One to accommodate joining results in, and the other to push tuples, that will exceed bound P, in and then deal with it later to avoid any size/time expansion.

3.3 Data Structures and Notations used for the Algorithm

We construct a binary tree T whose set of leaves is exactly V and each internal node has exactly two children. Any binary tree over this leaf set can be used. For any internal node x, a left child is referred to as $lc(x)$ and a right child as $rc(x)$, and there is a labeling function where $label(x) \subseteq V$ is defined inductively as follows: $label(x) = V \setminus x$ for a leaf node $x \in V$, and for every internal node x: $label(x) = label(lc(x)) \cap label(rc(x))$, It is immediate that for any internal node x we have $label(lc(x)) \cup label(rc(x)) = V$ and that $label(x) = \emptyset$ if and only if x is the root of the tree. For any three relations R, S, and T, the notation $R \bowtie_S T = (R \bowtie T) \ltimes S$.

Example 3.2

For *q() = R(a,b), S(b,c), T(a,c)*, Any binary tree T necessary for the algorithm [6] can be constructed for q as follows:

(a) An instance for T.

(b) Another instance of T.

Figure 3.1: Different binary trees for the same CQ.

It is evident that both trees fulfill the conditions outlined in the preceding section.

3.4 Algorithm for LW instances

Algorithm 1 begins by initializing the bound P and invoking the LW method on the root node of the binary tree T. The method returns two sets, denoted as C and D. This process is repeated recursively for each node until a leaf node is reached. Upon reaching a leaf node, the method returns an empty set for C and assigns the relation whose attributes correspond to the label of the leaf node to set D.

For nodes other than the leaf, a series of computations are performed. First, the algorithm calculates the intersection F between the projection of the node label for the set of D obtained from the left child of the node and the projection of the node label for the set of D obtained from the right child of the node. Additionally, it determines set G, which consists of tuples in F where the length of the tuple present in D from the left child plus one is smaller than or equal to the rounded-up value of P divided by the length of the set D from the right child.

Furthermore, the algorithm assigns set C as the union of the sets D obtained from the two children, combined with the sets C of the children. Finally, it checks whether the current node is the root or an internal node. If it is the root, set D is set to be an empty set. Otherwise, set D is updated to be F with the removal of set G from it.

Finally, we get set $J \subseteq C(u)$ then we filter out the tuples that do not satisfy the conditions specified by the attribute sets in E, which is the pruning step.

Algorithm 1 Algorithm for LW Instances

Require: An LW instance: Re for $e \in \binom{V}{|V|-1}$ and $Ne = |Re|$

1: $P \leftarrow \prod_{e \in E} N_e^{\frac{1}{n-1}}$ (the size bound from LW inequality)
2: $u \leftarrow \text{root}(T)$
3: $(C(u), D(u)) \leftarrow \text{LW}(u)$

1: LW(x) : $x \in T$ returns (C, D)
2: **if** x is a leaf **then**
3: **return** $(\emptyset, R_{label(x)})$
4: **else**
5: $(C_L, D_L) \leftarrow \text{LW}(\text{lc}(x))$ and $(C_R, D_R) \leftarrow \text{LW}(\text{rc}(x))$
6: $F \leftarrow \pi_{\text{label}(x)}(D_L) \cap \pi_{\text{label}(x)}(D_R)$
7: $G \leftarrow \{t \in F : |D_L[t]| + 1 \leq \frac{P}{|D_R|}\}$ // $F = G = \emptyset$ if $|D_R| = 0$
8: **if** x is the root of T **then**
9: $C \leftarrow (D_L \bowtie D_R) \cup C_L \cup C_R$
10: $D \leftarrow \emptyset$
11: **else**
12: $C \leftarrow (D_L \bowtie_G D_R) \cup C_L \cup C_R$
13: $D \leftarrow F \setminus G$
14: **return** (C, D)

15: "Prune" $C(u)$ and return

While the mentioned algorithm addresses certain scenarios of LW instances, it exhibits limitations in certain cases. These limitations arise from factors such as which instance of the binary tree was constructed, as any binary tree on the set of leaves should output the same result for the same query [6], and the presence of duplicate values within tuples for a specific attribute in a relation. To illustrate this limitation, we will provide a detailed worked example that highlights the specific case where the algorithm fails to produce the desired outcome.

Example 3.3

Recall our example input CQ *q() = R(a,b), S(b,c), T(a,c)*, where

a	b
7	4

Table 3.1: Relation R.

b	c
4	1
4	3
4	5
4	9

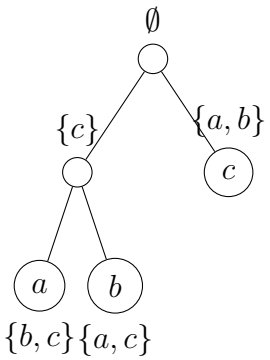
Table 3.2: Relation S.

a	c
7	0
7	2
7	5
7	7

Table 3.3: Relation T.

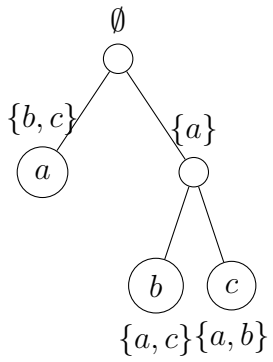
When applying the traditional binary join method to solve the query q, the correct result $J = [(7, 4, 5)]$ will be obtained. However, when using the LW algorithm, the outcome may vary depending on the specific instance of the binary tree constructed for q. In this example, the output may not be consistent due to the order in which the joins are performed in Algorithm 1. Furthermore, the presence of duplicate values for a specific attribute in the relations S and T contributes to this inconsistency.

Case 1: Consider this instance of the binary tree T was constructed.



Tracing algorithm 1 on this binary tree, we recursively call *LW()* method on each node till we reach to the leaf nodes. Upon reaching node a, the method returns an empty set C and the tuples of relation S represented as $D = [\{4, 1\}, \{4, 3\}, \{4, 5\}, \{4, 9\}]$. Similarly, for node b, the method returns an empty set C and the tuple of relation T as $D = [\{7, 0\}, \{7, 2\}, \{7, 5\}, \{7, 7\}]$. In the method call on the node above, the C set is the joining between the tuples in sets D of both children nodes a and b unioned with the C sets of the children which are empty so, $C = [\{7, 4, 5\}]$, and since $F = [\{5\}]$ and $G = [\{5\}]$, then set D will be an empty set. In the method call on the node above (root node), we have $D_L = \emptyset$, $D_R = [\{7, 4\}]$, $C_L = [\{7, 4, 5\}]$ and $C_R = \emptyset$, so the $C = [\{7, 4, 5\}]$ which will be the final output of the algorithm. Notice that this is the correct answer for q as it's equal to the output J from the binary join method.

Case 2: Now, we will consider the other instance for T



Tracing algorithm 1 on this binary tree, we recursively call *LW()* method on each node till we reach the leaf nodes. Upon reaching node b, the method returns an empty set C and the tuples of relation T represented as $D = [\{7, 0\}, \{7, 2\}, \{7, 5\}, \{7, 7\}]$. Similarly, for node c, the method returns an empty set C and the tuple of relation R as $D = [\{7, 4\}]$. In the method call on the node above, the C set is the joining between the tuples in sets D of both children nodes b and c unioned with the C sets of the children which are empty so, $C = [\{7, 4, 0\}, \{7, 4, 2\}, \{7, 4, 5\}, \{7, 4, 7\}]$, and since $F = [\{7\}]$ and $G = [\{7\}]$, then set D will be an empty set. In the method call on the node above (root node), we have $D_R = \emptyset$, $D_L = [\{4, 1\}, \{4, 3\}, \{4, 5\}, \{4, 9\}]$, $C_R = [\{7, 4, 0\}, \{7, 4, 2\}, \{7, 4, 5\}, \{7, 4, 7\}]$ and $C_L = \emptyset$, so the $C = [\{7, 4, 0\}, \{7, 4, 2\}, \{7, 4, 5\}, \{7, 4, 7\}]$ which will be the final output of the algorithm. Notice that this is not the correct answer for q as it's not equal to J and also it's a different output for the same CQ.

For this problem, a revised algorithm for LW instances was implemented. It has the same intuition as algorithm 1 with just a few changes that ensure that the correct output is returned no matter which instance of a binary tree is created and whether or not any relation has duplicate values for an attribute.

Algorithm 2 Revised Algorithm for LW Instances

Require: An LW instance: Re for $e \in \binom{V}{|V|-1}$ and $Ne = |Re|$

$P \leftarrow \prod_{e \in E} N_e^{\frac{1}{n-1}}$ (the size bound from LW inequality)
$u \leftarrow \text{root}(T)$
$(C(u), D(u)) \leftarrow \text{LW}(u)$

LW(x) : $x \in T$ returns (C, D)
    if x is a leaf **then**
        return $(\emptyset, R_{label(x)})$
    else
        $(C_L, D_L) \leftarrow \text{LW}(\text{lc}(x))$ and $(C_R, D_R) \leftarrow \text{LW}(\text{rc}(x))$
        $F \leftarrow \pi_{\text{label}(x)}(D_L) \cap \pi_{\text{label}(x)}(D_R)$
        $G \leftarrow \{t \in F : |D_L[t]| + 1 \leq \frac{P}{|D_R|}\}$ // $F = G = \emptyset$ if $|D_R| = 0$
        if $lc(x)$ is a leaf **then**
            $K_L \leftarrow D_L$
        else
            $K_L \leftarrow C_L$
        if $rc(x)$ is a leaf **then**
            $K_R \leftarrow D_R$
        else
            $K_R \leftarrow C_R$
        if x is the root of T **then**
            $C \leftarrow K_L \bowtie K_R$
            $D \leftarrow \emptyset$
        else
            $C \leftarrow K_L \bowtie_G K_R$
            $D \leftarrow F \setminus G$
        return (C, D)

"Prune" $C(u)$ and return

The key distinction between Algorithm 1 and the proposed algorithm lies in their approach to selecting the tuples sets for joining.

Tracing the previous input example on algorithm 2 using the binary tree constructed in **case 2**. we recursively call *LW()* method on each node till we reach the leaf nodes. Upon reaching node b, the method returns an empty set C and the tuples of relation T represented as $D = [\{7, 0\}, \{7, 2\}, \{7, 5\}, \{7, 7\}]$. Similarly, for node c, the method returns an empty set C and the tuple of relation R as $D = [\{7, 4\}]$, until this point algorithm 1 and and algorithm 2 are identical. In the method call on the above node, we choose which set of tuples to be joined. That ensures the correct tuples sets are assigned for the join. If the left child of the node x is a leaf, then D_L will be selected for the join, else then C_L is selected. So $F = [\{7\}]$, $G = [\{7\}]$ and the set $C = [\{7, 4, 0\}, \{7, 4, 2\}, \{7, 4, 5\}, \{7, 4, 7\}]$, and finally set $D = \emptyset$. Going up to the method call on the root node, since its left child is a leaf node, $D_L = [\{4, 1\}, \{4, 3\}, \{4, 5\}, \{4, 9\}]$ is assigned for the join, while its right

child is not, then $C_R = [\{4, 1\}, \{4, 3\}, \{4, 5\}, \{4, 9\}]$ is assigned for the join. Returning set C after computing the join which will be $C = [\{7, 4, 5\}]$, a correct returned out as the binary join method.

The proposed algorithm surpasses algorithm 1 as it ensures consistency in outputting correct results regardless of which binary tree is constructed.

Chapter 4

A Join Algorithm for All Conjunctive Queries

In this chapter, we will discuss a more general algorithm that handles all instances of CQ. We will view the data structures used in the algorithm, the reason behind considering this algorithm to be better than the binary join method, and the intuition behind it.

4.1 Worst Case Optimal Join Algorithms

Worst-Case Optimal Join (WCOJ) algorithms are algorithms that, in theory, offer asymptomatically a better runtime than traditional binary join methods. This optimization is achieved by several approaches, such as doing the join method on an optimized order and avoiding enumerating large intermediate results by processing multiple input relations in a single multi-way join [1].

The algorithm discussed in this chapter and the algorithm in the following chapter are both considered WCOJ algorithms.

4.2 Intuition Behind the Algorithm

The efficiency of this algorithm heavily relies on the order of the attributes by which the join is executed on. This order is obtained from the *query plan tree*, which is a data structure that divides the overall query into smaller subproblems. Also, this algorithm optimizes the running time as in each multi-join operation on a set of attributes, it checks which of the relations, being joined on, is the smallest in size and accordingly searches in the other relations for every tuple appearing in the smallest relation. Another reason for the algorithm's effectiveness is for some of its subproblems are solved on a "size check" basis [6].

4.3 Necessary Data Structures for the Algorithm

To fully understand the usages of the data structure for this algorithm and how it is implemented. We can, at first, propose an input example for the algorithm and explain around it the implementation for such structures for our proposed algorithm.

Example 4.1

Consider the CQ *q() = R(a,b,d,e), S(a,c,d,f), T(a,b,c), G(b,d,f), Z(c,e,f).* The hypergraph representing it *H(q) = (V,E)*, having 5 attributes $V = \{a, b, c, d, e, f\}$ and 3 relations *R, S, T, G* and *Z* represent in hyperedges $E = \{\{a, b, d, e\}, \{a, c, d, f\}, \{a, b, c\}, \{b, d, f\}, \{c, e, f\}\}$. Each input relation in the query q can be referred to as R_e, where $e \in E$. The sets V and E will be used in the implementation of the following tree structure.

4.3.1 Query Plan Tree

A query plan tree T is a binary tree having its nodes labeled by the set of the hyperedge E (set of relations), and each node in this tree is associated with a *universe* U where $U \subseteq V$.

In the beginning, we get an arbitrary order of the hyperedges E. The root node has universe V. We visit each edge in the order. The query plan tree is built recursively as follows: each node x has two children *lc(x)* and *rc(x)*, each labeled by the next edge in the arbitrary order obtained previously. the universe for *lc(x)* will be the set U except for the set of attributes contained in the edge, while the universe for *rc(x)* will be the set of attribute present in U and contained in the edge. We stop the intersection between the attributes for the remaining node edges and U is an empty set.

Here is an algorithmic implementation for the query plan tree:

Algorithm 3 Constructing the query plan tree T

1: Fix an arbitrary order $e_1, e_2, \ldots, e_m$ of all the hyperedges in E.
2: $T \leftarrow$ build-tree(V, m)

1: build-tree(U, k) :
2: **if** $e_i \cap U = \emptyset$, $\forall i \in [k]$
3: **return** nil
4: Create a node u with label$(u) \leftarrow E_k$ and univ$(u) \leftarrow U$
5: **if** $k > 1$ and $\exists i \in [k]$ such that $U \nsubseteq e_i$:
6: lc$(u) \leftarrow$ build-tree$(U \setminus e_k, k-1)$
7: rc$(u) \leftarrow$ build-tree$(U \cap e_k, k-1)$
8: **return** u

For our input example, we can assume an arbitrary order of hyperedges is $[Z, G, T, S, R]$. The constructed query plan tree, considering that order, will be as follows:

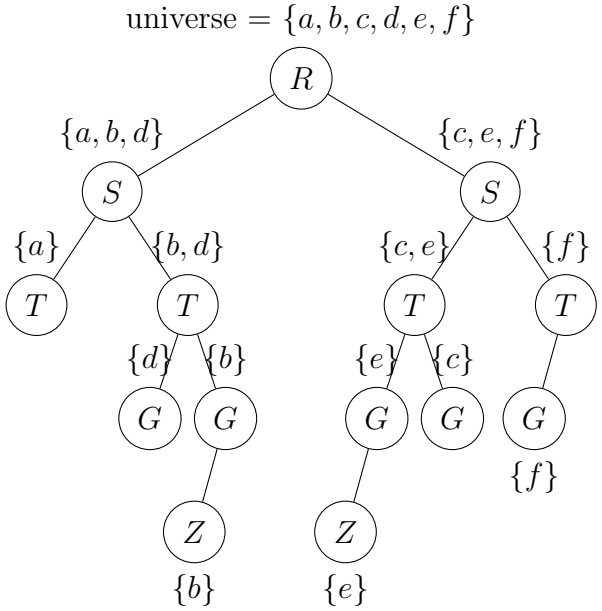


Figure 4.1: Query Plan Tree for query q

The query plan tree serves two purposes:

1. Gets the recursive structure for the algorithm, where each node in the tree can be considered as a subproblem.

2. It presents the total order of attributes.

4.3.2 Total Order of Attributes

The total order of attributes obtained from the query plan is the universe of each leaf node starting from the leftmost node. For the input example, the total order would be $\{a, d, b, e, c, f\}$. The following algorithm is used to recursively get the total order from any query plan tree.

Algorithm 4 Computing a total order of attributes in V

```
1: Let T be the query plan tree with root node u, where univ(u) = V
2: print-attribs(u):
    1: if u is a leaf node of T then
    2:     print all attributes in univ(u) in an arbitrary order
    3: else if lc(u) = nil then
    4:     print-attribs(rc(u))
    5: else if rc(u) = nil then
    6:     print-attribs(lc(u))
    7:     print all attributes in univ(u) \ univ(lc(u)) in an arbitrary order
    8: else
    9:     print-attribs(lc(u))
   10:     print-attribs(rc(u))
```

The next step in preparations for the algorithm is building search trees

4.3.3 Search Tree

The search tree is built for each relation and serves as an indexing structure for the relation over its set of attributes. Each level of the tree is indexed over an attribute of the relations, we choose the attribute on which the levels are indexes over respecting the total order.

Considering our input example. For relation R_R, the top level of the tree is indexed over attribute a, the next two levels are b and d, and the last level is indexed over e. For R_S, the first one is indexed over a then the second over c, and the last two on d and f. For R_T, the order is a, b,then c, and so on.

The building of a search tree for each relation will improve the running time of the WCOJ algorithm.

4.4 Implementation of the Algorithm

Before discussing the implementation of this WCOJ algorithm, we define some necessary terminologies. For a given tuple, denoted as t, on a specific attribute set A, we represent it as t_A to indicate that the tuple is associated with the attributes in set A. In other words, $t_A = (t_a)_{a \in A}$, where t_a represents the value of tuple t for attribute a. Consider any relation R with attribute set S and $A \subset S$ and t_A is a particular tuple, then the notation $R[t_A]$ denotes the "t_A-section" of R. This section is a relation on the attributes $S - A$, consisting of all tuples t_{S-A} such that $(t_A, t_{S-A}) \in R$. In particular, $R[t_\emptyset] = R$.

Also, it is necessary to have the minimal fractional cover solution of the hypergraph $H(q)$ as one of the input parameters for the WCOJ. Alongside this fractional cover

solution, the algorithm also takes the query plan tree and the total order of attributes as inputs.

Despite making a significant effort in implementing this WCOJ algorithm, this work falls short of fully implementing and evaluating the algorithm. Due to time constraints and limited resources, the implementation of the algorithm was mostly completed. However, due to these constraints, the debugging phase required to ensure the correct output of the algorithm was not sufficiently carried out. The complexity of the algorithm posed a major obstacle to its completion. The pseudocode for this algorithm is written by Ngo [6].

Chapter 5

Leapfrog Triejoin Algorithm

Leapfrog triejoin is another WCOJ algorithm that is currently being used in the commercial database system LogicBlox[2]. In this chapter, we will see how the leapfrog triejoin uses the pre-implemented leapfrog join as described in the paper by Todd Veldhuizen[7].

5.1 Leapfrog Join

The basic building block of leapfrog triejoin is a unary join which we call leapfrog join. The unary leapfrog join is a variant of sort-merge join which simultaneously joins unary relations $A_1(x), ..., A_k(x)$ on the same attribute.

5.1.1 Leapfrog Join Implementation

It is a condition for the relations in the join to be presented in sorted order so upon initialization, we sort each relation. Each relation in the leapfrog join is associated with *a linear iterator* which is an index used to navigate through values in the relation. These are the method that must be implemented in the linear iterator interface:

int key()	Returns the key at the current iterator position
next()	Proceeds to the next key
seek(int seekKey)	Position the iterator at a least upper bound for seekKey, i.e. the least key $\geq$ seekKey, or move to end if no such key exists. The sought key must be $\geq$ the key at the current position.
bool atEnd()	True when iterator is at the end.

The algorithm initializes an array of instances to those iterators. In operation, the join tracks the smallest and largest keys at which iterators are positioned and repeatedly move an iterator at the smallest key to the least upper bound for the largest key, 'leapfrogging' the iterators until they are all positioned at the same key. When the leapfrog join iterator is constructed, the leapfrog-init method (Algorithm 5) is used to initialize the state and find the first result. The leapfrog-init method is provided an array of iterators; it ensures the iterators are sorted according to the key at which they are positioned, an invariant that is maintained throughout.

The main engine is leapfrog-search (Algorithm 6), which finds the next key in the intersection $A_1 \cap ... \cap A_k$ Immediately following leapfrog-init(), the leapfrog join iterator is positioned at the first result, if any; following results are obtained by calling leapfrog-next() (Algorithm 7). To complete the linear iterator interface, we construct a leapfrog-seek() function which finds the first element on the intersection of relations which is $\geq$ seekKey (Algorithm 8).

Algorithm 5 leapfrog-init()

```
1: if atEnd() is true for any iterator
2:     atEnd = true
3: else
4:     atEnd = false
5:     sort the array Iter[0,...,k-1] by keys at which the iterator are positioned
6:     p = 0
7:     call leapfrog-search()
```

Algorithm 6 leapfrog-search()

```
1: x'= Iter[(p-1) mod k].key() //Max key of any iterator
2: While true
3:     x = Iter[p]key()
4:     if x = x' then
5:         key = x
6:         return
7:     else
8:         Iter[p].seek(x')
9:         if Iter[p].atEnd() then
10:            atEnd = true
11:            return
12:        else
13:            x' = Iter[p].key()
14:            p = p + 1 mod k
```

Algorithm 7 leapfrog-next()

```
1: Iter[p].next()
2: if Iter[p].atEnd() is true then
3:     atEnd = true
4: else
5:     p = p + 1 mod k
6:     call leapfrog-search()
```

Algorithm 8 leapfrog-seek(int seekKey)

```
1: Iter[p].seek(seekKey)
2: if Iter[p].atEnd() is true then
3:     atEnd = true
4: else
5:     p = p + 1 mod k
6:     call leapfrog-search()
```

Applying the above algorithm (Leapfrog join) on input of any number of unary relations (having the same singular attribute). It returns a list of all common values across these relations.

Example 5.1

Here is an example that illustrates the works of Leapfrog join:

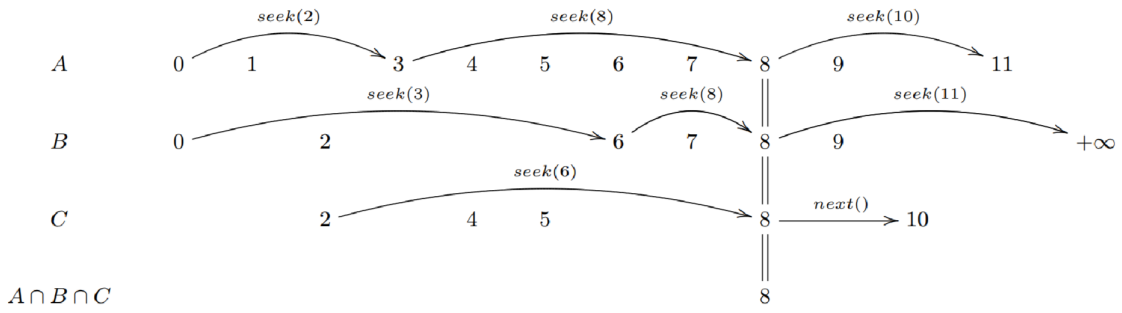


Figure 5.1: Visualization of leapfrog join.

Consider a leapfrog join involving three relations A, B, and C. Let's assume that $A = \{0, 1, 3, 4, 5, 6, 7, 8, 9, 11\}$, and the contents of relations B and C are shown in the second and third rows, respectively. Initially, the iterators for A, B, and C are positioned at 0, 0, and 2, respectively. The iterator for A performs a $seek(2)$ operation, which positions it at element 3. Subsequently, the iterator for B performs a $seek(3)$ operation, landing it at element 6. Finally, the iterator for C executes a $seek(6)$ operation, positioning it at element 8, and so on until reaching the end of one of the relations.

5.2 Leapfrog TrieJoin

Now to handle relations of arity greater than 1, A trie data structure is used to represent each one of these relations. Relations such as $A(x, y, z)$ are presented as tries with each tuple $(x, y, z) \in A$. The triejoin algorithm requires the optimizer to choose a variable ordering, i.e., some permutation of the variables appearing in the join. For example, in the join $R(a, b), S(b, c), T(a, c)$ we might choose the variable ordering $[a, b, c]$.

5.2.1 Intuition Behind the Algorithm

The triejoin algorithm begins by identifying the first common value among tries that share the initial attribute in the specified order. If a trie does not contain this common value in its table, the corresponding key is disregarded from the join operation. The algorithm then progresses to the next level in the trie, applying the same approach to find a common value among the tries sharing the subsequent attribute. This process continues iteratively until reaching the last level. The leapfrog join is performed on the children of the parent node, extracting the common keys between them until reaching the last child. At this point, bindings for each attribute in the preceding tries and the last attribute are established. To progress further, the algorithm backtracks to the previous attribute level, seeking bindings for its next value within that level. This backtracking process continues for each preceding attribute until reaching the last child associated with the first attribute in the order. Triejoin can be conceptualized as a backtracking search through the binding tries, exploring various combinations of bindings at each level until all possibilities have been examined. Overall, the triejoin algorithm efficiently identifies and combines common values among tries based on their shared attributes, facilitating an optimized join operation.

5.2.2 Implementation of the Leapfrog Triejoin Algorithm

In the triejoin algorithm, the next step involves initializing an array of trie iterator instances, with each instance corresponding to an attribute in the join operation. Specifically, if the join involves attributes *a, b, and c*, then the array would contain three

instances representing *a, b, and c*, respectively. A trie iterator instance of a relation Z can be referred to as Itr_Z

For instance, consider the join operation $R(a,b), S(b,c), T(a,c)$. In this case, the leapfrog join instance for variable b would be given pointers to the trie-iterators for relations R and S. However, it is important to note that the trie-iterator for relation R is shared by both the leapfrog joins for variables a and b.

Hence, the array of trie iterator instances would have three elements, corresponding to the three attributes in the join operation. Each element would consist of pointers to the tries in which the attribute appears. In this example, the array would look like $[[Itr_R, Itr_T], [Itr_R, Itr_S], [Itr_S, Itr_T]]$, indicating that the attribute a appears in tries S and T, while the attribute b appears in tries R and S, and the attribute c appears in tries S and T.

This array of trie iterator instances allows for efficient traversal and comparison of the tries at each attribute level, enabling the identification of common values among the related tries.

Two more methods should be implemented The *open()* method is responsible for preparing the trie iterators at the current depth. It increments the depth counter and opens the trie iterators associated with the attribute at that depth. This step ensures that the iterators are ready for comparison.

Algorithm 9 triejoin-open()

```
1: depth = depth + 1
2: for each iter in the leapfrog join at current depth
3:     iter.open()
4: call leapfrog-init() for leapfrog join at current depth
```

Conversely, the *up()* method facilitates moving up one level in the trie structure. It closes the trie iterators at the current depth and decrements the depth counter. This operation is essential for backtracking and exploring alternative paths during the join process.

Algorithm 10 triejoin-up()

```
1: for each iter in the leapfrog join at current depth
2:     iter.up()
3: depth = depth − 1 //Backtrack to previous var
```

The triejoin algorithm incorporates a variable called depth, which serves the purpose of keeping track of the current variable being considered for binding. Initially, the depth is set to -1, indicating that the triejoin is positioned at the root of the binding trie, before the first variable in the variable ordering. As the algorithm progresses, the depth value increases, representing the subsequent variables in the ordering. Each depth value corresponds to a specific variable, such as the first variable at depth 0, the second variable at depth 1, and so on.

Algorithmic Breakdown of Leapfrog Triejoin Operation

The algorithm is explained in plain English for simplification, and to avoid any complex or misleading notations.

1. Firstly, assign t as an empty list, then enter an always $True$ loop
2. Check if the current depth is equal to the length of the triejoin's array minus one.
3. If true, perform a leapfrog join operation:
   (a) Collect values of nodes from each trie iterator at the current depth, which are leaf nodes, and store them in a nested list called leap.
   (b) Create a new leapfrog join instance with leap as input.
   (c) Recursively call the leapfrog join algorithm to obtain possible bindings.
   (d) Append each obtained binding to the list t and add the result to the final output list.
4. Check if the trie iterator at the current depth is at the end and has been visited.
5. If true, remove the last value from the list t (if not empty) and move to the previous depth by calling the *up* method.
6. Check if the trie iterator at the current depth is not at the end and has been visited.
7. If true, advance the trie iterator to the next value and update the variable *main* with the current key value.
8. Search among the other trie iterator at the current depth for the *main* values using search search method, as all the node for a same depth are sorted.
9. Loop through the other trie iterators and check if any reach the end with a different key value than *main*.
10. If true, move to the next value for the first trie iterator.
11. Check if the first trie iterator has reached the end.
12. If true, remove the last value from t (if not empty), move to the previous depth, and set the flag b to break from the loop.
13. Check if the trie iterator at the current depth has not been visited.
14. If true, mark it as visited, update the *main* value, and binary search for *main* in other trie iterators.
15. Open the trie iterators at the next depth by calling the *open* method.

16. If the current depth is not the last depth, add the current value to the list t.

17. Check if the first value in t is *None*.

18. If true, terminate the loop.

19. Check if the first trie iterator at the current depth has reached the end and has been visited, and the current depth is 0.

20. If true, terminate the loop.

Example 5.2

We will illustrate the workings of the leapfrog triejoin on a simple example, Consider a conjunctive query $q = R(a, b), S(b, c), T(a, c)$, where

a	b
0	0
0	1
0	2
1	0
1	0

Table 5.1: Relation R.

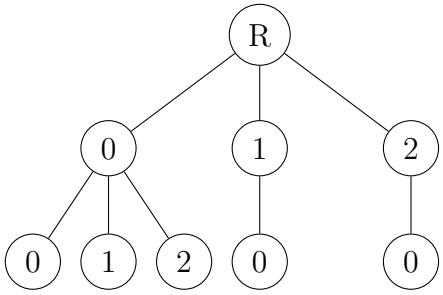
b	c
0	0
0	1
0	2
1	0
1	0

Table 5.2: Relation S.

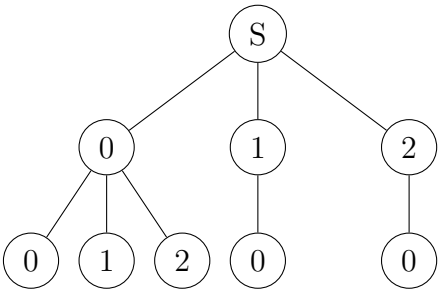
a	c
0	0
0	1
0	2
1	0
1	0

Table 5.3: Relation T.

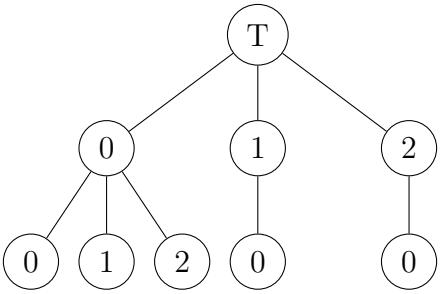
then the trie representations for *R,S, and* T:



(a) Trie representation of R.



(b) Trie representation of S.



(c) Trie representation of T.

Figure 5.2: Trie representations for each relation.

Initially, each iterator is positioned at the root of the trie, and we have an initialized array of trie iterator instances: $[[R, T], [R, S], [S, T]]$. We begin by invoking the *open()* function on $[R, T]$ to position their iterators at node 0 for attribute a in both iterators. Similarly, we do the same for $[R, S]$ and position its iterator at node 0 for attribute b.

As we progress through the attributes, when we reach the last attribute c, we obtain the values of the level in $[S, T]$, which are [[0, 1, 2], [0, 1, 2]]. We perform a leapfrog join between these values, resulting in the tuples (0, 0, 0), (0, 0, 1), and (0, 0, 2), which are appended to the output.

Next, we backtrack to attribute b and position the iterators in $[R, S]$ at node 1. Upon reaching attribute c again, we obtain the values of the level in $[S, T]$, which are [[0], [0, 1, 2]]. Applying the leapfrog join on these values, we find the common value 0, resulting in the tuple (0, 1, 0), which is appended to the output.

This process continues until we reach the last node in each trie. The final output is $[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (0, 2, 0), (1, 0, 0), (2, 0, 0)]$.

Chapter 6

Results

In this chapter, we conduct comprehensive testing of the algorithms and analyze their performance in terms of runtime. This evaluation includes different and various inputs and instances for processing CQ to imitate real-world scenarios. Also, It includes data representations for visual understanding. The aim of this evaluation is to gain insights into the efficiency of the implemented algorithms. By comparing the performance of these algorithms, we aim to identify their strengths and determine which algorithm is most fit for different scenarios and datasets.

The testing environment utilized a machine with *16GB* of RAM, an Intel core *i7 @ 1.8GHz 1.99GHz*, and a 64-bit operating system.

6.1 Benchmark for Comparing Performance

Throughout this work, the traditional binary join method was used as a reference comparison to each algorithm output correctness. We will also use it as a performance benchmark in evaluating the implemented algorithms.

We revise how a binary join method works so that we keep this chapter self-contained. It operates by comparing each tuple from one relation with each tuple from the other relations, identifying matching tuples on a specific attribute, and creating a new relation that incorporates the combined information. This method is not limited to joining only two relations; it can handle any number of relations involved in the join operation.

6.2 Datasets Used in the Evaluation

In this work, Random generator algorithms were implemented to test a wide range of relation and attribute combinations, where one can specify the range of the number of relations, attributes, and size of each relation, and also the range of values of the data generated as numbers were used as data for easier and more convenient evaluation.

6.3 Data Representation Used

A scatter plot is used to represent the relationship between the input number of relations in a CQ and the runtime (in milliseconds) it takes to execute and get the output, where it gives a clear view of the runtime each run takes.

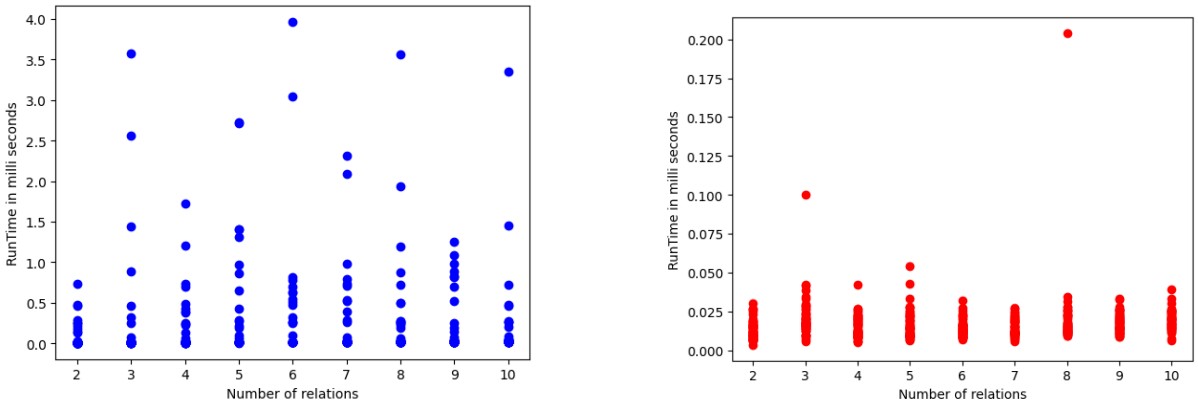
6.4 Comparative Analysis: LW and Binary Join

In chapter 3, we discussed that the intuition of the LW algorithm is that it avoids doing the join operation on relations that would lead to the explosion and exceeding a certain bound P.

In this test run, we will generate input for two scenarios. The first generates a dataset that would not lead to an explosion in the join, and the other will ensure that the dataset will cause the exceeding to occur.

6.4.1 Normal Run Case

We tested the two algorithms on 500 runs for each, we generate data for each run, where the number of relations and attributes vary between 2 and 10. It resulted in the following:



(a) Results of LW

(b) Results of Binary join

Figure 6.1: Loomis-Whitney VS Binary join

We notice that the binary join results are significantly better than the algorithm for LW, as the binary join averages an approximate of 0.0167 milliseconds per run, while the LW averages an approximate of 0.1762 milliseconds per run, making the binary join almost 90% times faster than LW.

6.4.2 Disscusion of the Result

the previous result between LW algorithm and binary join method can be justified for the following reasons:

1. The LW algorithm has the basic join operation in its core, so it will not be better than the actual join method.
2. The number of recursive calls on each node in the binary tree constructed.

6.4.3 Explosive Run Case

In this run case, a significantly larger input was used of in terms the relations sizes to ensure an explosion in the join process.

Fewer runs were conducted compared to the previous run case, due to limitations of resources in terms of the machine's capability. However, these runs resulted in the following:

The LW algorithm actually performed way better than in its previous run case but also falls short in comparison with the binary join method. In this run case the binary join method performed almost 58% better than the LW algorithm.

6.5 Comparative Analysis: Leapfrog Triejoin and Binary Join

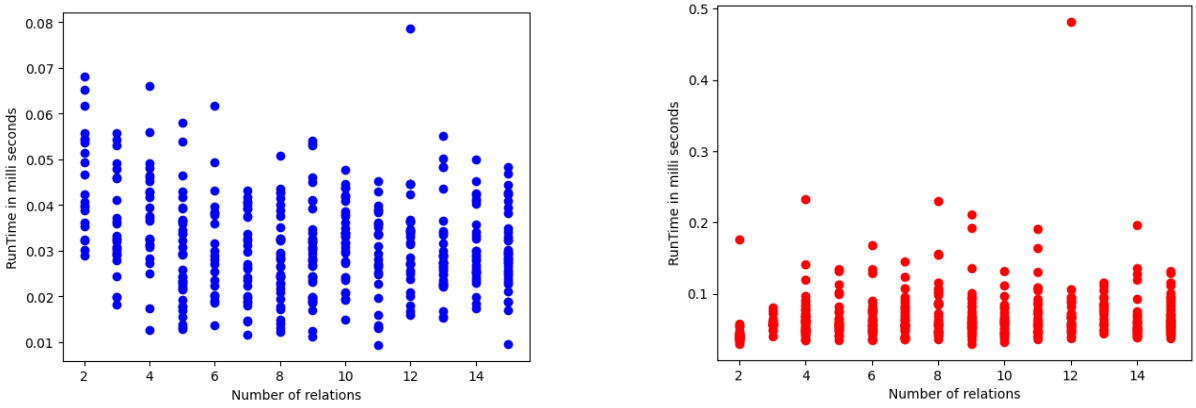
6.5.1 Run Case

We tested the two algorithms on 500 runs for each, we generate data for each run, where the number of relations varies between 2 and 15, while the number of tuples and the number of attributes in each relation varies between 1 and 6. It resulted in the following:

We notice that the leapfrog triejoin results are overall better than the results of binary join, as the leapfrog triejoin averages an approximate of 0.032 milliseconds for each run, while the binary join averages an approximate of 0.111 milliseconds per run, making the leapfrog triejoin almost 70% faster than the binary join.

6.5.2 Disscusion of the Result

the previous result between the leapfrog triejoin and binary join method can be justified for the following reasons:



(a) Results of Leapfrog Triejoin

(b) Results of Binary join

Figure 6.2: Leapfrog Triejoin VS Binary join

1. The leapfrog triejoin does not do any computational comparisons, as it only searches for specific values using efficient searching methods (binary-search) and accommodates values, in contrast with the binary join which is based on comparing tuples.
2. Skipping relations that do not have values for specific attributes.
3. the absence of multiple inner loops in the leapfrog triejoin, unlike the binary join.

Chapter 7

Conclusion

The target of this work was to discuss and analyze three distinct algorithms that were designed to improve CQ processing. Two of them were implemented fully and output results to analyze, and only one was not fully completed. Also, proposed implementations, for algorithms mentioned in previously published work [6] and [7], were given in this work.

The LW algorithm was slower than the traditional binary join method, while the leapfrog triejoin, which is a WCOJ algorithm, gives better performance compared to the binary join.

7.1 Future Work

As a recommendation for future work, I suggest the following:

1. The problem is still open for the design of new exact and approximation algorithms for CQ in future works.
2. Coming up with a more efficient algorithm for LW instances.
3. Complete the implementation of the first WCOJ algorithm. It is recommended that future researchers aim for advancing the investigation of the algorithm's approach from this point onward.
4. Expand the range of testing by running these algorithms on a powerful machine to process a larger number of input variables than used in the results.

Appendix

List of Abbreviations

CQ	Conjunctive Query
LW	Loomis-Whitney
WCOJ	Worst-Case Optimal Join

List of Figures

List of Tables

Bibliography

[1] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(12):1891–1904, 2020.

[2] Shan Shan Huang. Leapfrog triejoin: A worst-case optimal join algorithm. https://developer.logicblox.com/2012/10/leapfrog-triejoin-a-worst-case-optimal-join-algorithm/, 2012.

[3] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, 1(6):1–5, 2012.

[4] Paris Koutris Lecturer. Acyclic conjunctive queries.

[5] Paris Koutris Lecturer. Size bounds for joins.

[6] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[7] Todd L Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.