

Media Engineering and Technology Faculty
German University in Cairo



Comparison of Algorithms for Conjunctive Query Evaluation

Bachelor Thesis

Author: Omar Hesham Abdelnaby Youssef
Supervisors: Dr. Haythem O. Ismail

Submission Date: 19 May, 2024

Media Engineering and Technology Faculty
German University in Cairo



Comparison of Algorithms for Conjunctive Query Evaluation

Bachelor Thesis

Author: Omar Hesham Abdelnaby Youssef
Supervisors: Dr. Haythem O. Ismail

Submission Date: 19 May, 2024

This is to certify that:

- (i) the thesis comprises only my original work toward the Bachelor Degree
- (ii) due acknowledgement has been made in the text to all other material used

Omar Hesham Abdelnaby Youssef
19 May, 2024

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Dr. Haythem, without the constant support and guidance of whom this thesis would have never been possible. He always went above and beyond in offering as much help and accommodation as possible, and often times more.

I would also like to thank all my friends and colleagues over the last 4 years, without the support, help, and company of whom I would not have reached this point.

Abstract

In this paper, we conduct a review of two algorithms for evaluating join operations in relational databases: one specifically designed for Loomis-Whitney instances and another applicable to all join queries. The Loomis-Whitney algorithm leverages insights from the loomis-whitney geometric inequality to optimize joins for specific structured data instances, while the generalized algorithm employs a broader approach, applicable to any join query.

Contents

1	Introduction	1
1.1	Motivation	1
2	Background	2
2.1	Definitions and Problem Statement	2
3	Modeling and Implementation Details in Java	4
3.0.1	Tuples	4
3.0.2	Relations	6
4	Algorithm for Loomis-Whitney Instances	8
4.0.1	Algorithm Description	9
4.0.2	Running Example	11
5	Algorithm for All Join Queries	13
5.1	Constructing the Query Plan Tree	13
5.2	Constructing a search tree for every relation	16
5.3	The Recursive-Join Procedure	19
5.4	Algorithm for Computing the Final Join	21
5.4.1	The Fractional Cover Problem	21
6	Conclusion	22
	Appendix	23
A	Lists	24
	List of Abbreviations	24
	List of Figures	25

Chapter 1

Introduction

1.1 Motivation

Conjunctive queries are a type of query in first-order logic that make use of the logical conjunction operator \wedge . Conjunctive queries have many applications in various fields, one of them being in relational databases.

Not only can many queries in a relational database management system (RDBMS) be written as conjunctive queries, but some of the most fundamental operations in RDBMSs are also conjunctive queries at heart, such as the natural join between two or more relations (written as \bowtie).

Naturally, this makes developing efficient methods for the evaluation of conjunctive queries of great interest. A large, modern database system may be running hundreds of thousands of queries every day, and it would be safe to assume that a large portion of those queries are either pure conjunctive queries, or they make use of the join operation between one or more relations.

Our work in this thesis will focus on exploring different algorithms for evaluating joins. We will first begin exploring an algorithm for what are called Loomis-Whitney instances, which are specific instances of the Optimal Join problem, then we will move on to explore an algorithm for general join processing (i.e for solving the general case of the OJ problem.)

Chapter 2

Background

2.1 Definitions and Problem Statement

Before we begin our work in examining algorithms for efficient join processing between relations in a relational database system, we first need to make some definitions which we will later rely on.

All the algorithms we will be looking at are algorithms for solving what is called the Optimal Join problem, which is defined as follows:

Optimal Worst-case Join Evaluation Problem (Optimal Join Problem). Given a fixed database schema $\bar{R} = \{R_i(\bar{A}_i)\}_{i=1}^m$ and an m-tuple of integers $\bar{N} = (N_1, \dots, N_m)$. Let q be the natural join query joining the relations in \bar{R} and let $I(\bar{N})$ be the set of all instances such that $|R_i^I| = N_i$ for $i = 1, \dots, m$. Define $U = \sup_{I \in I(\bar{N})} |q(I)|$. Then, the optimal worst-case join evaluation problem is to evaluate q in time $O(U + \sum_{i=1}^m N_i)$.

We assume the existence of a set of attribute names $\mathcal{A} = A_1, \dots, A_n$ with associated domains $\mathbf{D}_1, \dots, \mathbf{D}_n$ and infinite set of relational symbols R_1, R_2, \dots . A relational schema for the symbol R_i of arity k is a tuple $\bar{A}_i = (A_{i_1}, \dots, A_{i_k})$ of distinct attributes that defines the attributes of the relation. A relational database schema is a set of relational symbols and associated schemas denoted by $R_1(\bar{A}_1), \dots, R_m(\bar{A}_m)$. A relational instance for $R(A_{i_1}, \dots, A_{i_k})$ is a subset of $\mathbf{D}_{i_1} \times \dots \times \mathbf{D}_{i_k}$. A relational database I is an instance for each relational symbol in schema, denoted by R_i^I . A natural join query (or simply query) q is specified by a finite subset of relational symbols $q \subseteq \mathbb{N}$, denoted by $\bowtie_{i \in q} R_i$. Let $\bar{A}(q)$ denote the set of all attributes that appear in some relation in q , that is $\bar{A}(q) = \{A \mid A \in \bar{A}_i \text{ for some } i \in q\}$. Given a tuple \mathbf{t} we will write $\mathbf{t}_{\bar{A}}$ to emphasize that its support is the attribute set \bar{A} . Further, for any $\bar{S} \subset \bar{A}$ we let $\mathbf{t}_{\bar{S}}$ denote \mathbf{t} restricted to \bar{S} . Given a database instance I , the output of the query q on I is denoted $q(I)$ and is defined as 4

$$q(I) \stackrel{\text{def}}{=} \left\{ \mathbf{t} \in \mathbf{D}^{\bar{A}(q)} \mid \mathbf{t}_{\bar{A}_i} \in R_i^I \text{ for each } i \in q \right\}$$

where $\mathbf{D}^{\bar{A}(q)}$ is a shorthand for $\times_{i:A_i \in \bar{A}(q)} \mathbf{D}_i$. We also use the notion of a semijoin: Given two relations $R(\bar{A})$ and $S(\bar{B})$ their semijoin $R \ltimes S$ is defined by

$$R \ltimes S \stackrel{\text{def}}{=} \{\mathbf{t} \in R : \exists \mathbf{u} \in S \text{ s.t. } \mathbf{t}_{\bar{A} \cap \bar{B}} = \mathbf{u}_{\bar{A} \cap \bar{B}}\}.$$

Additionally, we define the join between any two relations $R(\bar{A})$ and $S(\bar{B})$ $R \bowtie S$ to be:

$$R \bowtie S = \{t : \exists t_R \in R, \exists t_S \in S, \forall \alpha \in \bar{A} \cap \bar{B}, t(\alpha) = t_R(\alpha) = t_S(\alpha)\}$$

and the join $R \bowtie_G S$ to be

$$R \bowtie_G S \stackrel{\text{def}}{=} (R \bowtie S) \ltimes G.$$

For any relation $R(\bar{A})$, and any subset $\bar{S} \subseteq \bar{A}$ of its attributes, let $\pi_{\bar{S}}(R)$ denote the projection of R onto \bar{S} , i.e.

$$\pi_{\bar{S}}(R) = \{\mathbf{t}_{\bar{S}} \mid \exists \mathbf{t}_{\bar{A} \setminus \bar{S}}, (\mathbf{t}_{\bar{S}}, \mathbf{t}_{\bar{A} \setminus \bar{S}}) \in R\}.$$

For any tuple $\mathbf{t}_{\bar{S}}$, define the $\mathbf{t}_{\bar{S}}$ -section of R as

$$R[\mathbf{t}_{\bar{S}}] = \pi_{\bar{A} \setminus \bar{S}}(R \ltimes \{\mathbf{t}_{\bar{S}}\}).$$

From Join Queries to Hypergraphs A query q on attributes $\bar{A}(q)$ can be viewed as a hypergraph $H = (V, E)$ where $V = \bar{A}(q)$ and there is an edge $e_i = \bar{A}_i$ for each $i \in q$. Let $N_e = |R_e|$ be the number of tuples in R_e . From now on we will use the hypergraph and the original notation for the query interchangeably.

Chapter 3

Modeling and Implementation Details in Java

The algorithms examined in this thesis were implemented using Java, namely Java 22. Additionally, the database instances on which they were run were also modeled in Java. This section will highlight the main parts of the modeling of the database, and the modeling and implementation of structures required for each algorithm will be covered in the chapter covering that algorithm.

3.0.1 Tuples

Recall that a tuple is an ordered set of n elements $(a_1, a_2, a_3, \dots, a_n)$ where a_i represents the i th element in the tuple. The order of elements in the tuple is important, and the values of a tuple are always assumed to be in the same order of the attributes of the relation of which the tuple is a part. For example, for a relation R with attributes $R(A, B, C)$ and a tuple $t \in R = (1, 3, 5)$, it is implied that this tuple has the value 1 for the attribute A, 3 for the attribute B, and 5 for the attribute C.

The Tuple class in Java

We model a tuple in our database as a class with the following header:

```
public class Tuple
```

A tuple has the following instance variables:

```
private final String[] values;  
private final String[] attributes;  
private final HashMap<String, String> valueMap;
```

The array `values` represents the values of the tuple. Similarly, the array `attributes` represents the attributes of the relation two which this tuple belongs. Note that the inclusion of this array `attributes` is not really necessary, as the attributes could always be inferred from the relation to which the tuple belongs. It has only been included to simplify the implementation.

The `HashMap valueMap` simply maps every attribute to its value. This is also not strictly required, as a value for any given attribute a_k can be found by obtaining the index of the attribute from the relation (say that, in this example, it happens to be at index k), then its corresponding value in the tuple would be the element at `values[k]`. However, in order to facilitate finding the value of a given attribute in a tuple in constant time without having to first scan the relation's attributes to find the index of the given attribute, we resort to making this map. This map is automatically created with the construction of each tuple.

We also make use of the following constructors:

```
public Tuple(String[] attributes, String[] values)
public Tuple()
```

The first constructor is the default option for creating a new tuple. The second one is used to create an empty tuple that may later be populated with attributes and values.

The tuple class contains the following methods:

```
public String[] getValue()
public String[] getAttributes()
public String[] setValues(String[] values)
public String[] setAttributes(String[] attributes)
public String toString()

@Override
public int equals(Object o)
```

The first four methods are simple getters and setters, the `toString()` override simply prints the attributes and values of a given tuple in a neat display, and the `equals(Object o)` compares the equality of the tuple on which it is invoked to the Object o . Note that o is first cast into a tuple (if this casting fails, an exception is thrown), before being compared. Two tuples are defined to be equal if they have the same attribute and value arrays, in the same order.

3.0.2 Relations

A relation R is defined by an ordered set of attributes $A = (a_1, a_2, a_3, \dots, a_n)$ and a set of tuples $T = \{t_1, t_2, t_3, \dots, t_n\}$. Note that for all $t \in T$, `t.attributes` = $R.A$. In other words, a relation cannot contain a tuple with attributes different from its own attributes.

The Relation Class

The `Relation` class has the following instance variables:

```
private Set<Tuple> tuples;
private String[] attributes;
```

And the following constructors:

```
public Relation()
public Relation(String[] attributes)
public Relation(Set<Tuple> tuples, String[] attributes)
```

The constructors are similar to the ones outlined in the `Tuple` class. The first one is used to create an empty relation into which attributes and tuples can be later inserted, the second one is much the same, only choosing to define the attributes of the relation initially (which cannot be later changed, and hence dictate what tuples are allowed in this relation), and the third one creates a relation with the given attributes and tuples.

The `Relation` class includes the following basic methods:

```
@Override
public String toString()

public String[] getAttributes()
public Set<Tuple> getTuples()
public void addTuple(String[] values)
public void addTuple(Tuple newTuple)
public void removeTuples(Set<Tuple> tuples)
```

The first three methods are easy enough to understand on their own. `addTuple` adds a tuple either by values (in which case it constructs a new tuple the attributes of which will be the attributes of the relation and the values of which will be the given `values[]` array, or simply being given a tuple and adding it to the relation (making sure to first check that the attributes match the attributes of the relation it is being added to.) `removeTuples` removes the given tuples from the relation if they exist.

Most importantly, the `Relation` has the following methods, which are crucial to the implementation of the algorithms that will follow:

`public static Relation join(Relation R1, Relation R2)` computes the join ($R1 \bowtie R2$).

`public static Relation semijoin(Relation R1, Relation R2)` computes the semi-join $R1 \ltimes R2$

`public static Relation project(Relation R, String[] attributes)` computes the projection $\pi_{\text{attributes}} R$

Chapter 4

Algorithm for Loomis-Whitney Instances

A Loomis-Whitney instance is a special case of the optimal join problem where the relations in the query have a specific relationship to each other. More specifically, a Loomis-Whitney instance is defined for a hypergraph $H = (V, E)$, such that E is the collection of all subsets of V of size $|V| - 1$. Let n be the number of relations (i.e. $n = |E|$), N_e be the size of the relation with attributes e for any $e \in E$. The join $\bowtie_{e \in E} R_e$ can be computed in time $O(n^2 \cdot (\prod_{e \in E} N_e)^{1/(n-1)} + n^2 \sum_{e \in E} N_e)$.

Algorithm 1 Algorithm for Loomis-Whitney Instances

```

1: Input: An LW instance  $H = (V, E)$ .
2: Output:  $(C, D)$ .
3:  $P = \prod_{e \in E} N_e^{1/(n-1)}$  (the size bound from LW-inequality)
4:  $u = \text{root}(T)$ ;  $C(u) = D(u) = \emptyset$  (initially empty sets)
5:  $((C(u), D(u)) \leftarrow \text{LW}(u)$ 
6: Prune  $C(u)$  and return
7: LW(Node  $x \in T$ ):
8: if  $x$  is a leaf then
9:   return  $(\emptyset, R_{\text{label}(x)})$ 
10: end if
11:  $(C_L, D_L) \leftarrow \text{LW}(\text{LC}(x))$ 
12:  $(C_R, D_R) \leftarrow \text{LW}(\text{RC}(x))$ 
13:  $F \leftarrow \pi_{\text{label}(x)}(D_L) \cap \pi_{\text{label}(x)}(D_R)$ 
14:  $G \leftarrow t \in F : |D_L[t]| + 1 \leq \lceil P/|D_R| \rceil$  //  $F = G = \emptyset$  if  $|D_R| = 0$ 
15: if  $x$  is the root of  $T$  then
16:    $C \leftarrow D_L \bowtie D_R \cup C_L \cup C_R$ 
17:    $D \leftarrow \emptyset$ 
18: else
19:    $C \leftarrow (D_L \bowtie_G D_R) \cup C_L \cup C_R$ 
20:    $D \leftarrow F/G$ 
21: end if
22: return  $(C, D)$ 

```

4.0.1 Algorithm Description

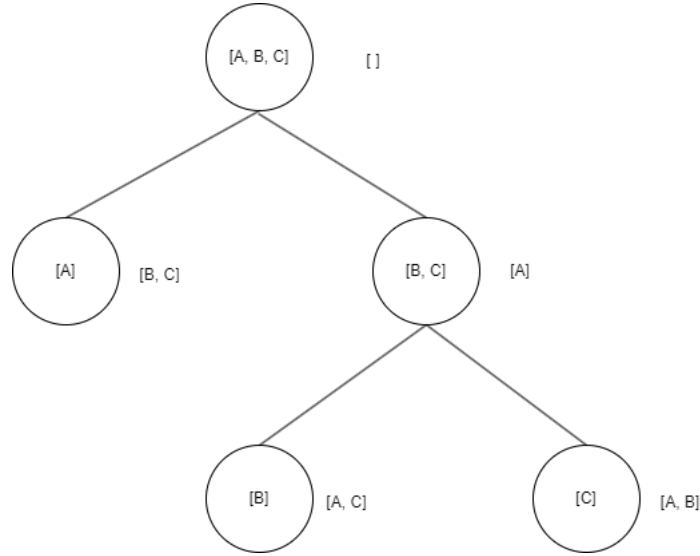
The algorithm begins by constructing a binary tree T over the vertex set V of the hypergraph (Figure 3.1). Any tree over this set can be used. Every vertex $v \in V$ appears in this tree as a leaf. Let $LT(x)$ denote the left child of x and $RT(x)$ denote the right child of x . Every node $x \in T$ has a label $\text{Label}(x) \subset V$ that is defined inductively as follows:

The BinaryTreeNode class in Java, which represents a node in the tree

$$\text{Label}(x) = \begin{cases} V/\{x\} & \text{if } x \text{ is a leaf node} \\ \text{Label}(\text{LC}(x)) \cap \text{Label}(\text{RC}(x)) & \text{if } x \text{ is an internal node} \end{cases}$$

From the above definition, it is easy to see that $\text{Label}(u) = \emptyset$ and for any internal node x , $\text{Label}(\text{RC}(x)) \cup \text{Label}(\text{LC}(x)) = V$.

For each node in the tree, we compute a set $C(x)$ that at each stage contains candidate tuples and an auxiliary set $D(x)$. The set $D(x)$ will intuitively allow us to deal with

Figure 4.1: A binary tree over the vertex set V

```

public static class BinaryTreeNode {
    private Set<String> label;
    private BinaryTreeNode left;
    private BinaryTreeNode right;

    public BinaryTreeNode(Set<String> label) {
        this.label = label;
    }
}

```

Figure 4.2: The BinaryTreeNode class in Java, used to represent a node of the tree

those tuples that would blow up the size of an intermediate relation. The key novelty in Algorithm 1 is the construction of the set G that contains all those tuples that are in some sense light, i.e., joining over them would not exceed the size/time bound P by much. In particular, it contains some tuples from the set F such that for each tuple $t \in F$, the number of tuples t_1 in the set D_L such that $t_{A \cap B} = t_{1A \cap B}$, where A is the set of attributes of F and B is the set of attributes of D_L , satisfies the inequality in line 14. The elements that are not light are postponed to be processed later by pushing them to the set $D(x)$.

4.0.2 Running Example

To illustrate the workings of the algorithm, we will consider a practical example with three relations $R(A, B)$, $S(B, C)$, and $T(A, C)$ containing the tuples:

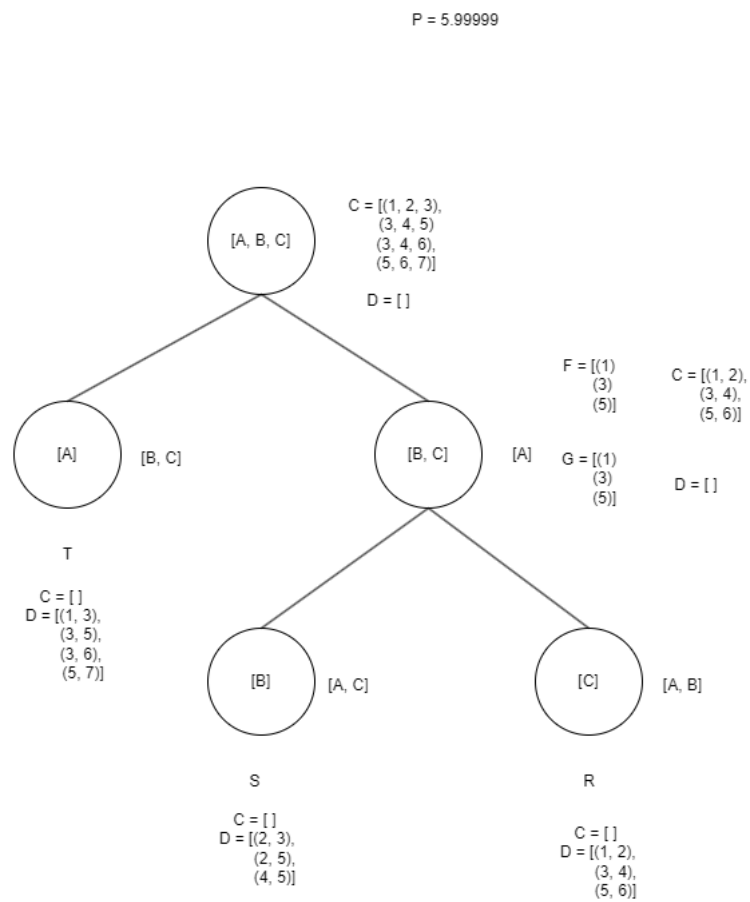
$$R = \langle (1, 2), (3, 4), (5, 6) \rangle$$

$$S = \langle (2, 3), (2, 5), (4, 5) \rangle$$

$$T = \langle (1, 3), (3, 5), (3, 6), (5, 7) \rangle$$

This database instance can be represented with the hypergraph $H = (V, E) = (\{A, B, C\}, \{\{A, B\}, \{B, C\}, \{A, C\}\})$. Figure 4.1 shows the binary tree constructed on the vertex set V .

Figure 3.2 shows the progression of the algorithm as it runs, starting from the leaf nodes and terminating at the root where the set C of the root node represents the result of the join query. The set next to a node $V/\{x\}$ is the label of that node (the set x being shown drawn inside the node.) The sets C and D for each node are shown beside it, as well as the sets F and G when applicable.

Figure 4.3: Running example of the algorithm V

Chapter 5

Algorithm for All Join Queries

In the previous chapter, we examined an algorithm for computing the join for loomis-whitney instances, which are a special case of the Optimal Join problem subject to some constraints of the structure of the hypergraph. In this chapter, we will examine an algorithm for solving the general instance of the OJ-Problem.

5.1 Constructing the Query Plan Tree

Much like the algorithm for Loomis-Whitney instances, this algorithm also begins with the construction of a (different) tree on the set of edges E .

We construct what we call a query plan tree. Every node in the query plan tree is labeled by a hyperedge, except for the leaf nodes, which are each labeled with a subset of hyperedges. Each node also has an associated universe $U \subset V$, a collection of vertices that is a subset of the vertex set, as well as an integer *label*.

```
public class QueryPlanTree {  
  
    private static ArrayList<ArrayList<String>> E;  
  
    public QueryPlanTree() {  
        E = new ArrayList<>();  
    }  
  
    public static class QueryPlanTreeNode {  
        private final ArrayList<String> universe;  
        private QueryPlanTreeNode left;  
        private QueryPlanTreeNode right;  
        private int label;  
  
        public QueryPlanTreeNode(ArrayList<String> universe) {  
            this.universe = universe;  
            this.left = null;  
            this.right = null;  
        }  
    }  
}
```

Figure 5.1: The QueryPlanTree class, contained within it is the QueryPlanTreeNode class to represent a node in the tree

Algorithm 2 Constructing the query plan tree \mathcal{T}

```

1: ConstructQueryPlanTree( $V, E$ )
2: Fix an arbitrary order  $e_1, e_2, \dots, e_m$  of all the hyperedges in  $E$ .
3:  $\mathcal{T} \leftarrow \text{BUILD-TREE}(V, m)$ 
4: return  $\mathcal{T}$ 
   BUILD-TREE( $U, k$ )
5: if  $e_i \cap U = \emptyset$  for all  $i \in [k]$  then
6:   return NIL
7: end if
8: Create a node  $u$  with  $\text{Label}(u) \leftarrow k$  and  $\text{UNIV}(u) = U$ 
9: if  $k > 1$  and  $\exists i \in [k]$  such that  $U \not\subseteq e_i$  then
10:   $\text{LC}(u) \leftarrow \text{BUILD-TREE}(U \setminus e_k, k - 1)$ 
11:   $\text{RC}(u) \leftarrow \text{BUILD-TREE}(U \cap e_k, k - 1)$ 
12: end if
13: return  $u$ 

```

We construct the query plan tree recursively as follows. We first arbitrarily order the hyperedges. The root node has universe V . We visit the edges one by one in the order we fixed.

If every remaining hyperedge (i.e every hyperedge that we have not yet visited) contains the universe V , then label the node with all remaining hyperedges and stop. In this case, the node is a leaf node. Otherwise, we consider the next hyperedge in the order. Label the root with e , and create two children of the root e . The left child's universe will be $V \setminus e$, and the right child's will be e . We recursively build the left tree starting from the next hyperedge (i.e., d) in the ordering, but only restricting to the smaller universe $V \setminus e$. Similarly, we build the right tree starting from the next hyperedge (d) in the ordering, but only restricting to the smaller universe e .

To better understand the tree building process, let us consider a worked example. Consider the relations $R_a(1, 4, 5, 6)$, $R_b(1, 4, 5)$, $R_c(2, 3)$, $R_d(3, 6)$. This would correspond to the hypergraph $\{1, 2, 3, 4, 5, 6\}, \{\{1, 4, 5, 6\}, \{1, 4, 5\}, \{2, 3\}, \{3, 6\}\}$. For simplicity, suppose we fix the order of the relations to be $R_d(3, 6)$, $R_c(2, 3)$, $R_b(1, 4, 5)$, $R_a(1, 4, 5, 6)$. This would correspond to the hypergraph $\{1, 2, 3, 4, 5, 6\}, \{\{1, 4, 5, 6\}, \{1, 4, 5\}, \{2, 3\}, \{3, 6\}\}$. Refer to Figure 5.2 to see the constructed tree. Each node on the figure are labelled by the subscript of the relation we are currently at in the order (i.e node a corresponds to relation R_a). Every node has its universe written above.

We begin with relation R_d , which corresponds to the root of the tree on the figure. We examine all the succeeding relations in the order to see if any of their hyperedges contain the universe of the current node. Since the universe of the root node is V , this is trivially false. We therefore construct a left child with universe $V \setminus e = \{1, 2, 3, 4, 5, 6\} \setminus \{3, 6\} = \{1, 2, 4, 5\}$ and a right child with universe $V \cap e = \{1, 2, 3, 4, 5, 6\} \cap \{3, 6\} = \{3, 6\}$. Now, let us consider the root's left child (i.e the node with the universe $\{1, 2, 4, 5\}$). We are

now at relation R_c in the order. Therefore, we check whether all the hyperedges of the succeeding relations R_b and R_a contain the universe of the current node. Since they do not, the same recursive construction process begins again. Notice that for this node's next child (the one labeled b with a universe $\{1, 4, 5\}$, the remaining relations in the total order (in this case, only R_a do(es) indeed contain this node's universe, so the recursive construction process does not apply to this node. For right child with the label b and the universe $\{2\}$, although R_a does not contain its universe of $\{2\}$, since its universe is of size 1, the construction stops here too. As a sidenote, a query construction tree need not be balanced. It just happens to be that the one we use as an example here is balanced.

After we construct the query plan tree, we compute what we call a total order of the attributes, by traversing the tree in a specific order. We then use this specific order of attributes to construct a search tree on every relation in the next step. Algorithm 3 shows the computation of the total order for this tree, which for this example is 1, 4, 5, 6, 2, 3.

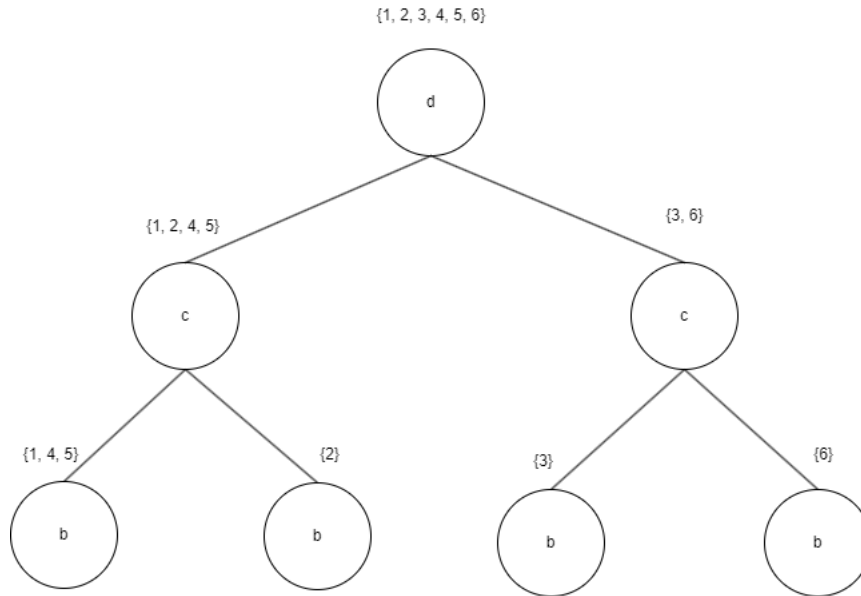


Figure 5.2: The query plan tree

5.2 Constructing a search tree for every relation

The next step in the algorithm is to construct a search tree for every relation. For each relation $R_e, e \in E$, we order all attributes in R_e such that the internal order of attributes in R_e is consistent with the total order of all attributes computed by Algorithm 4. More concretely, suppose R_e has k attributes ordered a_1, \dots, a_k , then a_i must come before a_{i+1} in the total order, for all $1 \leq i \leq k-1$. Then, we build a search tree (or any indexing data structure) for every relation R_e using the internal order of R_e 's attributes: a_1 indexes

Algorithm 3 Computing a total order of attributes in V

```

Let  $\mathcal{T}$  be the query plan tree with root node  $u$ , where  $\text{UNIV}(u) = V$ 
PRINT-ATTRIBS( $u$ )
PRINT-ATTRIBS( $u$ )
if  $u$  is a leaf node of  $\mathcal{T}$  then
    print all attributes in  $\text{UNIV}(u)$  in an arbitrary order
else if  $\text{LC}(u) = \text{NIL}$  then
    PRINT-ATTRIBS( $\text{RC}(u)$ )
else if  $\text{RC}(u) = \text{NIL}$  then
    PRINT-ATTRIBS( $\text{LC}(u)$ )
    print all attributes in  $\text{UNIV}(u) \setminus \text{UNV}(\text{LC}(u))$  in an arbitrary order
else
    PRINT-ATTRIBS( $\text{LC}(u)$ )
    PRINT-ATTRIBS( $\text{RC}(u)$ )
end if

```

level 1 of the tree, a_2 indexes the next level, \dots, a_k indexes the last level of the tree. The search tree for relation R_e is constructed to satisfy the following three properties. Let i and j be arbitrary integers such that $1 \leq i \leq j \leq k$. Let $\mathbf{t}_{\{a_1, \dots, a_i\}} = (t_{a_1}, \dots, t_{a_i})$ be an arbitrary tuple on the attributes $\{a_1, \dots, a_i\}$.

1. We can decide whether $\mathbf{t}_{\{a_1, \dots, a_i\}} \in \pi_{\{a_1, \dots, a_i\}}(R_e)$ in $O(i)$ -time (by "stepping down" the tree along the t_{a_1}, \dots, t_{a_i} path).
2. We can query the size $|\pi_{\{a_{i+1}, \dots, a_j\}}(R_e[\mathbf{t}_{\{a_1, \dots, a_i\}}])|$ in $O(i)$ time.
3. We can list all tuples in the set $\pi_{\{a_{i+1}, \dots, a_j\}}(R_e[\mathbf{t}_{\{a_1, \dots, a_i\}}])$ in time linear in the output size if the output is not empty.

The total running time for building all the search trees is $O(n^2 \sum_e N_e)$.

Constructing an example tree

Suppose we want to construct the search tree for the relation $R_a(1, 4, 5, 6)$ from the previous example. Suppose that the relation has the following tuples: $\langle (3, 2, 4, 5), (1, 3, 2, 3), (5, 2, 3, 1), (2, 5, 4, 3), (5, 1, 4, 8), (5, 3, 2, 3) \rangle$. Since this relation has four attributes, its tree will have four levels. Since the attributes appear in the total order in the order 1, 4, 5, 6, the first level of the tree will be indexed on the attribute 1, the second level on the attribute 4, the third on the attribute 5, and the fourth on the attribute 6.

Algorithm 4 shows the procedure to construct the search tree. Figure 5.3 shows the construction of the tree on R_a . The tree starts out with an empty root node, the children of which are the indexes on the first level. Let us consider the first level of the tree, which is indexed on the attribute 1. The level consists of the values 3, 1, 5, 2. Notice that values that exist in multiple tuples only appear once. Now, let us consider the next level indexed on the attribute 4. In particular, let us look at the first-level node with the value 5's children. Notice the three tuples (namely (5, 2, 3, 1), (5, 1, 4, 8), and (5, 3, 2, 3)) that contain the value 5 for their attribute of 1. The node labeled 5 has all the values for the next attribute in the index (in this case 4) as children. This way, any of those tuples can be reproduced by tracing a path from the root. Notice already that the first two values of these tuples can be reproduced by tracing all the paths through the node 5. This is also the way by which we can decide whether $\mathbf{t}_{\{a_1, \dots, a_i\}} \in \pi_{\{a_1, \dots, a_i\}}(R_e)$ in $O(i)$ -time.

Algorithm 4 Construct Search Tree

Require: *relation*: A relation, *order*: an ordering of the relation's attributes

```

1: tree  $\leftarrow$  new SearchTree()
2: for each tuple in relation do
3:   currentNode  $\leftarrow$  tree.root
4:   for each attribute in order do
5:     attributeValue  $\leftarrow$  tuple.valueMap[attribute]
6:     if attributeValue not in currentNode.children then
7:       newChild  $\leftarrow$  new Node(attributeValue)
8:       currentNode.children[attributeValue]  $\leftarrow$  newChild
9:     end if
10:    currentNode  $\leftarrow$  currentNode.children[attributeValue]
11:  end for
12: end for
13: return tree

```

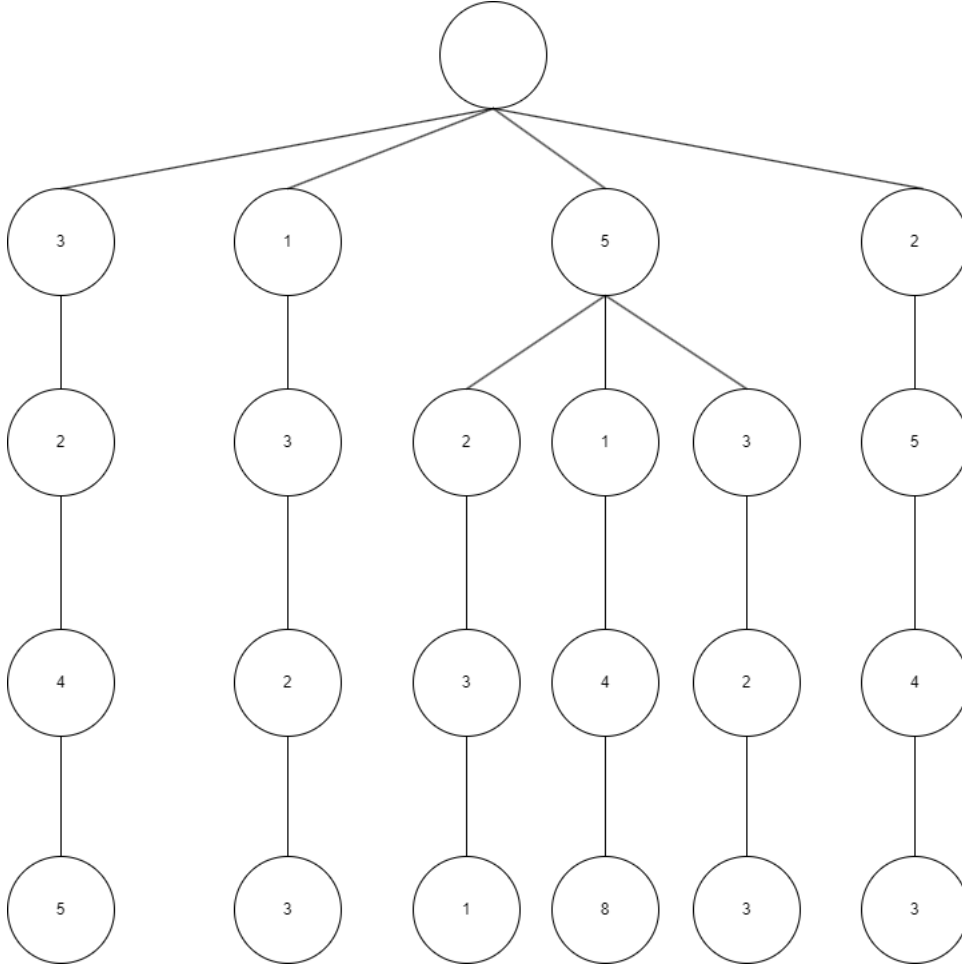
Algorithm 5 Check Membership in Projection

Require: *u*: root node of the search tree, *t*: tuple, *order*: list of attribute names

```

1: currentNode  $\leftarrow$  u
2: for each attribute in order[1 : i] do
3:   attributeValue  $\leftarrow$  t.valueMap[attribute]
4:   if attributeValue not in currentNode.children then
5:     return false
6:   end if
7:   currentNode  $\leftarrow$  currentNode.children[attributeValue]
8: end for
9: return true

```

Figure 5.3: The search tree constructed for the relation R_a

5.3 The Recursive-Join Procedure

Recursive-Join is a procedure that takes three arguments:

1. a node u from the query plan tree \mathcal{T} whose label is k for some $k \in [m]$.
2. a fractional cover solution $\mathbf{y}_{E_k} = (y_{e_1}, \dots, y_{e_k})$ of the hypergraph $(\text{UNIV}(u), E_k)$. Here, we only take the restrictions of hyperedges of E_k onto the universe $\text{UNIV}(u)$. Specifically,

$$\sum_{e \in E_k: i \in e} y_e \geq 1, \text{ for any } i \in \text{UNIV}(u)$$

$$y_e \geq 0, \text{ for any } e \in E_k$$

3. a tuple $\mathbf{t}_S = (t_i)_{i \in S}$ where S is the set of all attributes in V which precede $\text{UNIV}(u)$ in the total order. If there is no attribute preceding $\text{univ}(u)$ then this argument is NIL.

Algorithm 6 List Tuples in Projection**Require:** u : root node of the search tree, t : tuple, $order$: list of attribute names

```

1:  $currentNode \leftarrow u$ 
2: for each  $attribute$  in  $order[1 : i]$  do
3:    $attributeValue \leftarrow t.valueMap[attribute]$ 
4:   if  $attributeValue$  not in  $currentNode.children$  then
5:     return empty list
6:   end if
7:    $currentNode \leftarrow currentNode.children[attributeValue]$ 
8: end for
9: return listTuples( $currentNode$ )

```

Procedure 5 RECURSIVE-JOIN($u, \mathbf{y}, \mathbf{t}_S$)

```

1: Let  $U = \text{UNIV}(u)$ ,  $k = \text{LABEL}(u)$ 
2:  $Ret \leftarrow \emptyset$  //  $Ret$  is the returned tuple set
3: if  $u$  is a leaf node of  $\mathcal{T}$  then // note that  $U \subseteq e_i, \forall i \leq k$ 
4:    $j \leftarrow \text{argmin}_{i \in [k]} \{|\pi_U(R_{e_i}[\mathbf{t}_{S \cap e_i}])|\}$ 
5:   // By convention,  $R_e[\text{NIL}] = R_e$  and  $R_e[\mathbf{t}_\emptyset] = R_e$ 
6:   for each tuple  $\mathbf{t}_U \in \pi_U(R_{e_j}[\mathbf{t}_{S \cap e_j}])$  do
7:     if  $\mathbf{t}_U \in \pi_U(R_{e_i}[\mathbf{t}_{S \cap e_i}])$ , for all  $i \in [k] \setminus \{j\}$  then
8:        $Ret \leftarrow Ret \cup \{(\mathbf{t}_S, \mathbf{t}_U)\}$ 
9:   return  $Ret$ 
10: if  $\text{LC}(u) = \text{NIL}$  then //  $u$  is not a leaf node of  $\mathcal{T}$ 
11:    $L \leftarrow \{\mathbf{t}_S\}$ 
12:   // note that  $L \neq \emptyset$  and  $\mathbf{t}_S$  could be NIL (when  $S = \emptyset$ )
13: else
14:    $L \leftarrow \text{RECURSIVE-JOIN}(\text{LC}(u), (y_1, \dots, y_{k-1}), \mathbf{t}_S)$ 
15:    $W \leftarrow U \setminus e_k$ ,  $W^- \leftarrow e_k \cap U$ 
16:   if  $W^- = \emptyset$  then
17:     return  $L$ 
18:   for each tuple  $\mathbf{t}_{S \cup W} = (\mathbf{t}_S, \mathbf{t}_W) \in L$  do
19:     if  $y_{e_k} \geq 1$  then
20:       go to line 27
21:     if  $\left( \prod_{i=1}^{k-1} |\pi_{e_i \cap W^-}(R_{e_i}[\mathbf{t}_{(S \cup W) \cap e_i}])|^{\frac{y_{e_i}}{1-y_{e_k}}} < |\pi_{W^-}(R_{e_k}[\mathbf{t}_{S \cap e_k}])| \right)$  then
22:        $Z \leftarrow \text{RECURSIVE-JOIN}\left(\text{RC}(u), \left(\frac{y_{e_i}}{1-y_{e_k}}\right)_{i=1}^{k-1}, \mathbf{t}_{S \cup W}\right)$ 
23:       for each tuple  $(\mathbf{t}_S, \mathbf{t}_W, \mathbf{t}_{W^-}) \in Z$  do
24:         if  $\mathbf{t}_{W^-} \in \pi_{W^-}(R_{e_k}[\mathbf{t}_{S \cap e_k}])$  then
25:            $Ret \leftarrow Ret \cup \{(\mathbf{t}_S, \mathbf{t}_W, \mathbf{t}_{W^-})\}$ 
26:     else
27:       for each tuple  $\mathbf{t}_{W^-} \in \pi_{W^-}(R_{e_k}[\mathbf{t}_{S \cap e_k}])$  do
28:         if  $\mathbf{t}_{e_i \cap W^-} \in \pi_{e_i \cap W^-}(R_{e_i}[\mathbf{t}_{(S \cup W) \cap e_i}])$  for all  $e_i$  such that  $i < k$  and  $e_i \cap W^- \neq \emptyset$  then
29:            $Ret \leftarrow Ret \cup \{(\mathbf{t}_S, \mathbf{t}_W, \mathbf{t}_{W^-})\}$ 
30:   return  $Ret$ 

```

5.4 Algorithm for Computing the Final Join

Putting all of the previous steps together, algorithm 7 computes the final join.

5.4.1 The Fractional Cover Problem

The fractional cover problem is the following linear programming problem: Given a hypergraph $(\text{Univ}(u), E_k)$

$$\begin{array}{ll}
 \text{maximize} & \sum_{e \in E} \log N_e \cdot x_e \\
 \text{subject to:} & \\
 & x_e \geq 0 \quad \text{for every } e \in E \\
 & \sum_{e \in E_v} x_e \geq 1 \quad \text{for every } v \in V
 \end{array}$$

where E_v is the set of all hyperedges that contain v .

The solution to the fractional cover problem is a set of variables $x_1, x_2, x_3, \dots, x_e$ one associated with every hyperedge $e \in E$. We first put the problem in standard linear programming form and use the java SSC library in order to solve it.

Algorithm 7 Computing the join $\bowtie_{e \in E} R_e$

- 1: **Input:** Hypergraph $H = (V, E)$, $|V| = n$, $|E| = m$
 - 2: **Input:** Fractional cover solution $\mathbf{x} = (x_e)_{e \in E}$
 - 3: **Input:** Relations R_e , $e \in E$
 - 4: Compute the query plan tree \mathcal{T} , let u be \mathcal{T} 's root node
 - 5: Compute a total order of attributes
 - 6: Compute a collection of hash indices for all relations
 - 7: **return** RECURSIVE-JOIN($u, \mathbf{x}, \text{NIL}$)
-

Chapter 6

Conclusion

In conclusion, we have examined two algorithms for computing joins: one for loomis-whitney instances, and another for general join queries, and delved into some of the details of their implementations.

Appendix

Appendix A

Lists

List of Figures

4.1	A binary tree over the vertex set V	10
4.2	The BinaryTreeNode class in Java, used to represent a node of the tree .	10
4.3	Running example of the algorithm V	12
5.1	The QueryPlanTree class, contained within it is the QueryPlanTreeNode class to represent a node in the tree	14
5.2	The query plan tree	16
5.3	The search tree constructed for the relation R_a	19

References

Bibliography

- [1] Martin Grohe and Daniel Marx. Constraint solving via fractional edge covers. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 289–298. ACM Press, 2006.
- [2] Hung Q. Ngo, Ely Porat, Christopher Re, and Atri Rudra. Worst-case optimal join algorithms. *arXiv preprint arXiv:1203.1952*, page 31, 2012.
- [3] SSC Lab. SSC Lab. <https://www.ssclab.org/en/index.html>, Accessed 2024.