Conjunctive Queries with Negations: Bridging Theory and Practice

by

Boyi Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Boyi Li 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Antijoin, given its significant expressive power, has numerous applications in relational data analytics. Notwithstanding its importance, there remains great research potential in antijoin processing. In practical database systems, existing techniques to process antijoins are still considered rudimentary, building upon heuristics and cost-based optimization strategies that offer no theoretical guarantees. Meanwhile, the database theory community has proposed algorithms for antijoins with strong theoretical guarantees, yet these algorithms build upon specialized, complicated data structures and have not made their way to practice. In light of such gap between theory and practice, we propose new algorithms for antijoin processing in this thesis. Not only do our new algorithms provide the same theoretical guarantees as the state-of-the-art algorithm, but they also use only basic relational operations. The latter property enables our new algorithms to be rewritten in basic SQL statements, allowing an easy, system-agnostic integration into any SQL-based database system. We then empirically evaluate one of our new algorithms, rewritten in SQL, over real-life graph datasets with a variety of SQL database systems. Experimental results show order-of-magnitude improvements of our new algorithm over vanilla SQL queries.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Xiao Hu, for her guidance, support, and encouragement throughout the course of my graduate studies. I am also grateful to my thesis committee, Prof. Grant Weddell and Prof. David Toman, for their valuable feedback and insights that have greatly improved the quality of this work.

I would also like to thank all the members in the Data System Group lab for creating a supportive environment. In particular, I am grateful to Haochen Sun, Yingke Wang, Zhiang Wu, and Shufan Zhang for their fridenship and companionship, which made my graduate experience more enjoyable and less monotonous.

Dedication

This is dedicated to the one I love.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables x

1 Introduction 1

2 Preliminaries 4

2.1 Problem Setting 4

2.1.1 Relational Algebra 4

2.1.2 Conjunctive Queries (CQ) 5

2.1.3 Conjunctive Queries with Negations ($\mathsf{CQ}^{\neg}$) 7

2.1.4 Hypergraph Representation 7

2.1.5 Counting, Evaluation and Enumeration 10

2.2 Computational Model and Complexity Measures 11

2.3 Important Properties of Hypergraph Acyclicity 12

2.4 Related Work 15

3 Warm-Up: An Old Counting Algorithm for α-Acyclic CQ 18

3.1 The Yannakakis Algorithm 18

3.2 Annotation and Aggregation 22

3.3 A Counting Variant of the Yannakakis Algorithm 25

4 A New Counting Algorithm for Signed-Acyclic $\mathsf{CQ}^\neg$ 28

4.1 A New Decision Algorithm for Signed-Acyclic $\mathsf{CQ}^\neg$ 29

4.2 The Inclusion-Exclusion Principle and Its Implications 35

4.2.1 Incremental Aggregation 39

4.2.2 Chunks 40

4.2.3 Chunk Construction: A Simple Scenario 42

4.2.4 Chunk Construction: A Difficult Scenario 44

4.3 The Complete Algorithm 50

5 Theoretical Guarantees 58

5.1 Correctness 58

5.2 Runtime Analysis 65

6 Experiments 67

6.1 Experimental Setups 67

6.2 Queries and Datasets 68

6.3 Results 71

6.4 Impact of Individual Variables 73

6.5 Memory Consumption 76

7 Extensions 78

7.1 An Evaluation Algorithm for Signed-Acyclic $\mathsf{CQ}^\neg$ 78

7.2 An Aggregation Algorithm for Free-Connex $\mathsf{CQ}^\neg$ 82

7.3 Difference of CQ 85

8 Conclusion and Future Directions **89**

References **91**

List of Figures

3.1 An illustration of the Yannakakis algorithm on Example 5 over a database instance. A blue label on a child-parent edge indicates the order it is visited in the algorithm. 21

3.2 An illustration of Algorithm 1 on Example 5 over the same database instance as Figure 3.1. A red arrow indicates a newly established child-parent edge. 27

4.1 β-step on linear chain $V \subsetneq U_1 \subseteq U_2$. 33

4.2 Aggregation updates that happen while chunk (V, U_1, U_2, W) is constructed. 44

4.3 Aggregation updates accompanying each chunk update in Example 11. . . 49

4.4 An illustration of Algorithm 6 on Example 13, using a signed-elimination sequence $\sigma = X_4X_3X_2X_1$. A red arrow indicates a newly established child-parent edge. 56

6.1 Processing Time of Test Queries with Different Database Systems 72

6.2 Impact of the Query Count on Processing Time 74

6.3 Impact of Maximum Intermediate Size on Processing Time 75

6.4 Impact of Number of Negative Relations 75

6.5 DuckDB Memory Consumption with Gnutella 77

List of Tables

2.1 Mathematical Notations . . . 5

2.2 Relational Notations . . . 6

2.3 Complexity Measure Notations . . . 11

6.1 Graph Datasets and Relevant Statistics . . . 70

Chapter 1

Introduction

Relational algebra [19] and the SQL language reside at the core of relational databases. The majority of database research evolves around how each relational or SQL operation can be carried out efficiently from both practical and theoretical lenses. Among these operations, join[1] has the power of combining results from multiple tables, making it an indispensable part of analytical queries that crucially support business intelligence and decision-making. Given such importance, join processing remains a highly active research area.

Gap between theory and practice in join processing. Practical relational database systems are predominantly based on the binary join architecture, repeatedly picking two tables to join and materializing the result as a new table until there is one remaining table, which then stores the final join result. A query optimizer is tasked to determine the join order via a combination of finely-tuned heuristics and cost-based optimization strategies.

Such architecture has virtually remained the same since the seminal work of Selinger [40] in 1979 on cost-based query optimization, despite two main drawbacks. First, the cost-based optimizer is notorious for choosing suboptimal plans due to inaccurate statistical information [32]. Second, binary joins are fundamentally limited as there are pathological cases [37] that reveal the suboptimality of any binary-join-based plan.

In the database theory community, however, many new multi-way join algorithms have been developed in the past decade. The most well-known ones are a suite of algorithms known as *worst-case optimal joins* [36, 42, 7] with asymptotic runtime matching the worst-case output size conditioned on input statistics (see [35] for a survey). Notwithstanding the theoretical guarantees, attempts to integrate these new algorithms into practical systems

[1] A *join* will be formally defined in Section 2 as a *conjunctive query*.

face many challenges. For instance, asymptotic runtime could hide an expensive constant factor [34, 43]; additionally, existing implementations of these algorithms rely on specialized data structures, which are not only expensive to pre-compute [24], but also require modifying the system's source code.

The aforementioned worst-case optimal join algorithms apply to *general* join queries. Meanwhile, there are recent interests in bridging theory and practice for a more restricted type of join queries, called the α-*acyclic* join queries. The reason is mainly twofold. First, most join queries in practice either are, or are close to being α-acyclic [25, 11]. Second, on the theory side, there exists a seminal algorithm for α-acyclic join queries, the Yannakakis algorithm [46], that not only provides worst-case guarantees but also uses purely binary relational operations. The latter property allows the Yannakakis algorithm to be expressed as SQL statements, thereby enabling a system-agnostic integration of the algorithm into any SQL-based database system. Such direction is shown to be very promising lately [25, 43]. Concurrently, a different line of research [45, 10, 11, 49] focuses on efficient implementations of the Yannakakis algorithm in individual database systems.

Antijoin and its state of research. Orthogonal to the research advances in join, many of its extensions, including aggregation [31, 6], union [17], comparison-based predicates [44], to name a few, have been proposed and extensively studied to enlarge the expressive power of join. Among the list, antijoin[2] stands out with two main reasons. First, antijoin exercises great expressive power as it captures difference [30], the only non-monotone operator in relational algebra. In SQL syntax, antijoins are expressed as Not Exists, Not In, Except, etc. Second, research interests on antijoin mainly stem from the theory community [12, 13, 15, 33, 16, 48], while practical exposure is limited.

The latest theory work on antijoin is by [48], which presents an algorithm to process *signed-acyclic* antijoin queries. Despite its state-of-the-art asymptotic runtime, the algorithm relies on complex data structures, exhibiting severe obstacles to practical implementation. Meanwhile, on the practice side, [30] studies how difference of join queries, a closely-related notion to antijoin, can be computed efficiently. Interestingly, their technique is based on SQL-rewriting.

Our contributions. An intriguing connection can be drawn between the state of research on join processing and antijoin processing. On the one hand, although existing algorithms for general join queries rely on specialized data structures, there exists an algorithm for α-acyclic join queries, the Yannakakis algorithm, that manifests the possibility of practical implementation via SQL-rewriting. On the other hand, existing algorithms for

[2] An *antijoin* will be formally defined in Section 2 as a *conjunctive query with negations*.

signed-acyclic antijoin queries build upon specialized data structures, yet SQL-rewriting has already been adopted in some special cases. Naturally, one may ask: can signed-acyclic antijoin queries benefit from SQL-rewriting? Our thesis provides an affirmative answer, with the main contributions as follows.

- We develop a Yannakakis-style *counting* algorithm for signed-acyclic antijoin queries, i.e. it counts the number of results produced by the query. We consider it Yannakakis-style, for it not only uses only binary relational operations but also shares a similar structure as a counting variant of the Yannakakis algorithm.
- We implement our counting algorithm via SQL-rewriting and empirically evaluate it on PostgreSQL [2], DuckDB [1], and SparkSQL [4], capturing both centralized and parallel environments. Experimental results then show the effectiveness of our counting algorithm on SNAP [3], a real-life graph dataset collection, with orders of magnitude improvement over vanilla SQL queries.
- To further demonstrate the potentials of SQL-rewriting techniques on signed-acyclic antijoin processing, we extend the scope beyond counting with two new algorithms that also use only binary relational operations: (i) one is an *evaluation* algorithm for signed-acyclic antijoin queries, i.e. it computes the results of the query; (ii) the other is an *aggregation* algorithm for free-connex antijoin queries, which covers both counting and evaluation of signed-acyclic antijoin queries as special cases. All our new algorithms match the runtime of the state-of-the-art algorithm [48].
- Lastly, given that antijoin queries and difference of join queries are closely related, we study and improve the SQL-rewrite technique of [30] by removing the previously needed union operation. As union poses a practical bottleneck due to de-duplication, such improvement further bridges the gap between theory and practice on the study of difference of join queries.

Thesis organization. In Section 2, we formalize the database-theoretic concepts and notations that appear in this thesis. In Section 3, we review the Yannakakis algorithm and detail a counting variant of it that motivates our new counting algorithm. We present our counting algorithm in Section 4, establish its theoretical guarantees in Section 5, and empirically evaluate it in Section 6. Lastly, in Section 7, we extend our scope beyond counting of signed-acyclic join queries by presenting new algorithms and techniques in (i) evaluation of signed-acyclic antijoin queries; (ii) aggregation of free-connex antijoin queries; and (iii) difference of join queries.

Chapter 2

Preliminaries

In this section, we introduce the mathematical and database-theoretical background for this thesis. Table 2.1 contains a list of frequently used set-theoretic notations.

2.1 Problem Setting

2.1.1 Relational Algebra

Relational algebra [19] is a theoretical framework for modelling structured data and queries. Structured data takes the form of tables, where each data entry spans a row and each column contains data of the same type.

Relational algebra models a table as a relation, its columns as attributes, and its rows as tuples. Let $\text{dom}(v)$ be the domain of an attribute v. A tuple t over an attribute set V, denoted $t \in \text{dom}(V)$ and also referred to as a V-tuple, is a function that maps each attribute $v \in V$ to a value in $\text{dom}(v)$. For any $U \subseteq V$, we use $\pi_U t$ to denote the projection of t to attributes U, and for any $W \cap V = \emptyset$ and $t' \in \text{dom}(W)$, we use $t \| t'$ to denote the extension of t with t' to attributes $V \uplus W$. A relation R over attributes V, which we compactly denote as R_V, is a finite set of tuples over attributes V. In this thesis, we reserve the letters U, V, W (optionally equipped with subscripts) to represent sets of attributes. Other letters, commonly R and N, are used to represent relations.

Table 2.2 lists the definitions of the notations above, as well as a series of operations that act on relations: projection π, (natural) join $\bowtie$, semijoin $\ltimes$, and antijoin $\triangleright$. Besides, given

Table 2.1: Mathematical Notations

Notation	Definition
$A \uplus B$	The union of two sets A and B, with $A \cap B = \emptyset$.
$A \sqcup B$	The disjoint union of two sets A and B; that is, if we have both $x \in A$ and $x \in B$, x's instances in the two sets are regarded as distinct, and x appears twice in the result.
2^A	The power set, i.e. the set of all subsets, of a set A.
$A \subseteq_C B$	For sets A, B, and C, we write $A \subseteq_C B$ to mean $A \subseteq B \cup C$.
$\text{var}(\cdot)$	The set of attributes appearing in a mathematical object.
$[n]$	For natural number n, define $[n] := \{1, 2, \dots, n\}$.

the set nature of relations, binary set operations, eg. union $\cup$, intersection $\cap$, difference $-$, may be defined naturally on relations over the *same* set of attributes.

Order of relational operations. π takes precedence over the rest, while an intermixing sequence of the aforementioned binary operators is evaluated from left to right. Inherited from the associativity and commutativity of logical conjunction, joins and antijoins associate and commute with each other. That is, given a sequence of relations $S_0, S_1, S_2, \dots, S_n$ and operators $o_1, o_2, \dots, o_n \in \{\bowtie, \triangleright\}$, for any permutation $\tau : [n] \to [n]$, it follows that

$$S_0 \ o_1 \ S_1 \ o_2 \ S_2 \cdots \ o_n \ S_n = S_0 \ o_{\tau(1)} \ S_{\tau(1)} \ o_{\tau(2)} \ S_{\tau(2)} \cdots \ o_{\tau(n)} \ S_{\tau(n)}$$

2.1.2 Conjunctive Queries (CQ)

Conjunctive queries (CQ) are a standard notion in database theory to capture joins followed by projections. In SQL, they can be described by Select-From statements with tables connected using Natural Join. Formally, a CQ takes the form

$$\begin{aligned} \mathcal{Q} &= \pi_F\big(R_{V_1} \bowtie R_{V_2} \bowtie \cdots \bowtie R_{V_n}\big) \\ &= \{\pi_F t : t \in \text{dom}(V), \forall i \in [n], \pi_{V_i} t \in R_{V_i}\} \end{aligned}$$

where $R_{V_1}, R_{V_2}, \dots, R_{V_n}$ are relations, $V := \bigcup_{i=1}^n V_i = \text{var}(\mathcal{Q})$ are all the attributes of $\mathcal{Q}$, and $F \subseteq V$ are the free/output attributes.

If $V_i = V_j$ for some distinct $i, j \in [n]$, i.e. R_{V_i}, R_{V_j} are defined on the same set of attributes, then we can update[1] $R_{V_i} \leftarrow R_{V_i} \cap R_{V_j}$ and delete R_{V_j} from the CQ, without

[1] Based on the computational model, which is detailed in Section 2.2, computing the intersection of a

Table 2.2: Relational Notations

Notation	Definition
$t \in \text{dom}(V)$	A tuple over attributes V, also referred to as a V-tuple. Mathematically, $t : V \to \bigsqcup_{v \in V} \text{dom}(v)$ is a function mapping each attribute $v \in V$ to its respective domain $\text{dom}(v)$.
$\pi_U t$	The projection of a V-tuple t on attributes $U \subseteq V$. $\pi_U t$ is a U-tuple such that $\pi_U t(v) = t(v)$ for all $v \in U$.
$t \| t'$	The extension of V-tuple t with W-tuple t', where $V \cap W = \emptyset$. $t \| t'$ is a $V \uplus W$-tuple, such that $t \| t'(v) = t(v)$ for all $v \in V$, and $t \| t'(v) = t'(v)$ for all $v \in W$. Given an attribute $w \notin V$ and a w-value $a \in \text{dom}(w)$, we denote $t \| a$ the extension of t with a singleton tuple $(w \mapsto a)$.
R_V	A relation over attributes V, which is a finite set of V-tuples.
$\pi_U R_V$	The projection of relation R_V to attributes $U \subseteq V$, which is a relation $\pi_U R_V = \{\pi_U t : t \in R_V\}$.
$R_V \bowtie R_U$	The join between relations R_V and R_U, which is a relation $R_V \bowtie R_U = \{t \in \text{dom}(V \cup U) : \pi_V t \in R_V \wedge \pi_U t \in R_U\}$.
$R_V \ltimes R_U$	The semijoin of relation R_V with R_U, which is a relation $R_V \ltimes R_U = \{t \in R_V : \pi_{V \cap U} t \in \pi_{V \cap U} R_U\}$.
$R_V \triangleright N_U$	The antijoin between relations R_V and N_U, which is a relation $R_V \triangleright N_U = \{t \in \text{dom}(V \cup U) : \pi_V t \in R_V \wedge \pi_U t \notin N_U\}$. See *safety condition* in Section 2.1.3 for more details.
$\mathcal{Q} = (F, \mathcal{E})$	A CQ with hyperedges $\mathcal{E}$ and output attributes F. Implicitly, we assume its relations are $\{R_V\}_{V \in \mathcal{E}^+}$.
$\mathcal{Q} = \mathcal{E}$	A full CQ. Equivalently, $\mathcal{Q} = (\text{var}(\mathcal{Q}), \mathcal{E})$.
$\mathcal{Q} = (F, \mathcal{E}^+, \mathcal{E}^-)$	A CQ$^\neg$ with positive hyperedges $\mathcal{E}^+$, negative hyperedges $\mathcal{E}^-$, and output attributes F. Implicitly, we assume its positive relations and negative relations of $\mathcal{Q}$ are $\{R_V\}_{V \in \mathcal{E}^+}$ and $\{N_V\}_{V \in \mathcal{E}^-}$, respectively.
$\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$	A full CQ$^\neg$. Equivalently, $\mathcal{Q} = (\text{var}(\mathcal{Q}), \mathcal{E}^+, \mathcal{E}^-)$.

affecting the query result. Hence, we assume relations in a CQ are always over distinct sets of attributes. When $F = V$, we call $\mathcal{Q}$ a *full* CQ. This thesis primarily focuses on full CQ.

2.1.3 Conjunctive Queries with Negations ($\mathsf{CQ}^{\neg}$)

The notion of conjunctive queries seems too restrictive compared to the rich syntax of relational algebra or SQL; as a result, numerous work in the theory community focuses on extensions of conjunctive queries to capture wider classes of queries. One such extension is conjunctive queries with negations ($\mathsf{CQ}^{\neg}$). Formally, a $\mathsf{CQ}^{\neg}$ has the form

$$\begin{aligned}\mathcal{Q} &= \pi_F\Big(R_{V_1} \bowtie R_{V_2} \bowtie \cdots \bowtie R_{V_n} \triangleright N_{W_1} \triangleright N_{W_2} \triangleright \cdots \triangleright N_{W_m}\Big)\\ &= \Big\{\pi_F t : t \in \mathrm{dom}(V), \Big(\forall i \in [n], \pi_{V_i} t \in R_{V_i}\Big) \text{ and } \Big(\forall j \in [m], \pi_{W_j} t \notin N_{W_j}\Big)\Big\}\end{aligned}$$

where $V := \big(\bigcup_{i=1}^n V_i\big) \cup \big(\bigcup_{j=1}^m W_j\big)$ are all the attributes, and $R_{V_1}, R_{V_2}, \ldots, R_{V_n}$, N_{W_1}, N_{W_2}, $\ldots, N_{W_m}$ are relations. We refer to each R_{V_i} as a positive **R**elation and each N_{W_j} as a **N**egative relation.

From before, we may assume that every V_i is distinct. Likewise, if $W_i = W_j$ for some distinct $i, j \in [m]$, then we can update[2] $N_{W_i} \leftarrow N_{W_i} \cup N_{W_j}$ and delete N_{W_j} from the query without affecting the query result. Hence, we also assume that every W_i is distinct, i.e. for all distinct $i, j \in [m]$, $W_i \neq W_j$.

Safety condition. If there exists some attribute $v \in V$ such that v does not exist in any positive relation, then the query would produce an infinite number of results if $\mathrm{dom}(v)$ is infinite. As we define relations to be finite sets of tuples, and it is also convenient to regard a query ('s result set) as a relation, we prevent such a scenario by assuming by default that $V = \bigcup_{i=1}^n V_i$. This condition is termed the *safety condition*, and we describe $\mathsf{CQ}^{\neg}$ satisfying it as *safe*.

2.1.4 Hypergraph Representation

In database theory, conjunctive queries are commonly represented by a graph-theoretic alternative — hypergraphs. Hypergraphs generalize the notion of graphs by allowing each

list of relations $R_1, R_2, \ldots, R_m$ over the same set of attributes takes $O\left(m \cdot \min\{|R_i|\}_{i=1}^m\right) = O\left(\sum_{i=1}^m |R_i|\right)$ time. Adding such a term will not affect the complexity of any problem studied in this thesis.

[2] Again, based on Section 2.2, computing the union of a list of relations $N_1, N_2, \ldots, N_m$ over the same set of attributes takes $O\left(\sum_{i=1}^m |N_i|\right)$ time, which does not affect the overall complexity of any problem studied in this thesis.

edge, now termed *hyperedge*, to contain a set of vertices. Formally, a hypergraph takes the form $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ are its vertices and $\mathcal{E} \subseteq 2^{\mathcal{V}}$ are its hyperedges. For convenience in definitions, we assume there is no isolated vertex, i.e. $\mathcal{V} = \bigcup_{V \in \mathcal{E}} V =: \text{var}(\mathcal{E})$.

Given a CQ $\mathcal{Q} = \pi_F(R_{V_1} \bowtie R_{V_2} \bowtie \cdots \bowtie R_{V_n})$, its *hypergraph* is defined as $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \text{var}(\mathcal{Q})$ and $\mathcal{E} = \{V_1, V_2, \ldots, V_n\}$. Since $\text{var}(\mathcal{Q}) = \bigcup_{i=1}^{n} V_i = \text{var}(\mathcal{E})$, the vertex set $\mathcal{V}$ is fully determined by the hyperedge set $\mathcal{E}$; thus, we omit the vertex set and represent $\mathcal{Q}$ as $\mathcal{Q} = (F, \mathcal{E})$. When $\mathcal{Q}$ is full, i.e. $F = \text{var}(\mathcal{Q})$, F is also fully determined by $\mathcal{E}$; hence, we drop F and write $\mathcal{Q} = \mathcal{E}$.

As previously discussed, we can always assume relations in a CQ are over distinct sets of attributes; as a result, there is a one-to-one correspondence between relations in a CQ and hyperedges in its hypergraph. Therefore, when we present a CQ only with hypergraph representation, we use R_V to denote the relation associated to V, for each hyperedge V.

The hypergraph representation of CQ sees important applications in database theory, as certain properties of a hypergraph would imply complexity results regarding its associated CQ. One well-known example is the AGM bound [8], which is a tight worst-case bound on the output size of a full CQ based on its input size and certain *width-measure* of the associated hypergraph. A subsequent breakthrough [36] gives an algorithm with a runtime matching the AGM bound (up to some poly-logarithmic factor).

Acyclicity. Hypergraph acyclicity, generalizing the notion of a graph having no cycle, is a class of properties with important implications on the theoretical complexity of its associated CQ [23]. In this thesis, we focus on two notions of acyclicity: α-acyclicity and β-acyclicity. A CQ is said to be α-acyclic (β-acyclic) if and only if its hypergraph is α-acyclic (β-acyclic).

There are many equivalent definitions of α- and β-acyclicity [14], and we adopt the ones commonly seen in database theory literature.

Definition 1. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph. $\mathcal{H}$ is α-acyclic, if either $\mathcal{E} = \emptyset$, or the following procedure, called the GYO-algorithm [27, 47], results in one final empty hyperedge:*

(1) Remove an attribute that only appears in one hyperedge.

(2) If there exists distinct $V_1, V_2 \in \mathcal{E}$ with $V_1 \subseteq V_2$, remove V_1 from $\mathcal{E}$.

(3) Repeat (1) and (2) until no further changes can be made.

$\mathcal{H}$ is β-acyclic, if all its sub-hypergraphs, i.e. $\mathcal{H}' = (\mathcal{V}, \mathcal{E}')$ for all $\mathcal{E}' \in 2^{\mathcal{E}}$, is α-acyclic.

Example 1. *Consider hypergraphs $\mathcal{H}_1$ with hyperedges $V_1 = \{A, B\}, V_2 = \{B, C\}, V_3 = \{A, C\}, V_4 = \{A, B, C\}$, and $\mathcal{H}_2$ with hyperedges $U_1 = \{A, B\}, U_2 = \{B, C\}, U_3 = \{C, D\}$.*

$\mathcal{H}_1$ is α-acyclic, because $V_1 \subseteq V_4, V_2 \subseteq V_4, V_3 \subseteq V_4$ enables us to remove V_1, V_2, V_3, respectively, in the GYO-algorithm. Since there is one remaining hyperedge V_4, we remove all attributes in it to obtain one final empty hyperedge.

However, $\mathcal{H}_1$ is not β-acyclic. Consider its sub-hypergraph with hyperedges V_1, V_2, V_3 only. The GYO-algorithm cannot eliminate any attribute or hyperedge from the very beginning.

On the other hand, $\mathcal{H}_2$ is β-acyclic. It is easy to verify that any hypergraph with two or fewer hyperedges is α-acyclic, leaving $\mathcal{H}_2$ itself as the only candidate to check for α-acyclicity. We can sequentially eliminate the attributes and hyperedges D, U_3, C, U_2, B, A in the GYO-algorithm.

Generalization to $\mathsf{CQ}^\neg$. Hypergraph representation and the notion of acyclicity have both been generalized to $\mathsf{CQ}^\neg$.

Definition 2 ([12, 48])**.** *A signed-hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^+)$ is a hypergraph with vertices $\mathcal{V}$ and hyperedges $\mathcal{E}^+ \sqcup \mathcal{E}^-$. We call $\mathcal{E}^+$ its positive hyperedges and $\mathcal{E}^-$ its negative hyperedges.*

A signed-hypergraph is a special type of hypergraph, in which each hyperedge is *colored* as either positive or negative. Given a $\mathsf{CQ}^\neg$ $\mathcal{Q} = \pi_F\left((\bowtie_{i=1}^n R_{V_i}) \triangleright N_{W_1} \triangleright \cdots \triangleright N_{W_m}\right)$, its signed-hypergraph is defined as $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ with $\mathcal{V} = \text{var}(\mathcal{Q})$, $\mathcal{E}^+ = \{V_1, V_2, \ldots, V_n\}$, and $\mathcal{E}^- = \{W_1, W_2, \ldots, W_m\}$. We represent $\mathcal{Q}$ as $\mathcal{Q} = (F, \mathcal{E}^+, \mathcal{E}^-)$, and if $\mathcal{Q}$ if full, we write $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$.

Similar to the case of CQ, negative relations in a $\mathsf{CQ}^\neg$ are assumed to be over distinct sets of attributes, implying a one-to-one correspondence between relations and hyperedges. Hence, when a $\mathsf{CQ}^\neg$ is given only in hypergraph representation, we use R_V and N_U to denote the relation associated with V, U, respectively, for each positive hyperedge V and negative hyperedge U.

Definition 3 ([12, 48])**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^+)$ be a signed-hypergraph. With a slight abuse of terminology, we call the hypergraph $\mathcal{H}' = (\mathcal{V}, \mathcal{E}^+ \cup \mathcal{E}')$, for all $\mathcal{E}' \in 2^{\mathcal{E}^-}$, the sub-hypergraphs of $\mathcal{H}$. $\mathcal{H}$ is signed-acyclic, if all its sub-hypergraphs are α-acyclic.*

Example 2. *Let $\mathcal{H}$ be a signed-hypergraph with positive hyperedges $V_1 = \{A, B\}, V_2 = \{B, C\}$ and negative hyperedge $U_1 = \{A, C\}$.*

To check signed-acyclicity, there are two candidate sub-hypergraphs: $\mathcal{H}_1$ with hyperedges V_1, V_2, and $\mathcal{H}_2$ with hyperedges V_1, V_2, U_1. While $\mathcal{H}_1$ is α-acyclic, $\mathcal{H}_2$ is not given the triangle $\{A, B, C\}$ formed. As a result, $\mathcal{H}$ is not signed-acyclic.

We say a $\mathsf{CQ}^{\neg}$ is signed-acyclic if its underlying signed-hypergraph is signed-acyclic.

2.1.5 Counting, Evaluation and Enumeration

Given a CQ $\mathcal{Q} = (F, \mathcal{E})$, the database theory community is particularly interested in three problems:

1. **Counting** is the problem of computing $|\mathcal{Q}|$, i.e. the number of F-tuples in $\mathcal{Q}$.
2. **Evaluation** is the problem of returning all F-tuples in $\mathcal{Q}$.
3. **Enumeration** is the problem of outputting the F-tuples in $\mathcal{Q}$ one-by-one after pre-processing. The performance measures of interest are typically (i) **pre-processing time**; and (ii) **delay**, which is the maximum time between outputting two consecutive tuples (including a special *start* and *end* tuple to indicate whether the enumeration has started and ended, respectively).

Each problem has its practical significance. Evaluation naturally aligns with one fundamental task of database systems — retrieving data based on a client-specified query. Counting sees its application in query optimization, for it enables a cost-based optimizer to know the size of an intermediate result without necessarily materializing it, during its planning stage. Lastly, enumeration is particularly useful for streaming-based applications, in which query results are streamed to the client one-by-one instead of being returned altogether.

Enumeration stems from a fine-grained understanding of evaluation algorithms. Conceptually, a CQ evaluation algorithm can be divided into two phases: (i) a pre-processing phase that reads the input relations and produces an intermediate data structure; and (ii) an enumeration phase that reads the intermediate data structure and generates the output tuple one-by-one [32]. Such division is closely related to many database research topics, for example, factorized databases [39] and dynamic query processing [38].

These problem definitions extend naturally to $\mathsf{CQ}^{\neg}$. This thesis focuses primarily on counting and evaluation, given their closer relevance to traditional database systems.

Table 2.3: Complexity Measure Notations

Notation	Definition
$\lvert R_V \rvert$	The number of tuples in a relation R_V.
IN	The input size of a given $\mathcal{Q} = (F, \mathcal{E}^+, \mathcal{E}^-)$. $\text{IN} := \sum_{V \in \mathcal{E}^+} \lvert R_V \rvert + \sum_{V \in \mathcal{E}^-} \lvert N_V \rvert$.
OUT	The output size of a given $\mathcal{Q} = (F, \mathcal{E}^+, \mathcal{E}^-)$. $\text{OUT} := \lvert \mathcal{Q} \rvert$.
$d(v)$	The number of relations containing the attribute v in a given $\mathcal{Q} = (F, \mathcal{E}^+, \mathcal{E}^-)$. $d(v) := \lvert \{V \in \mathcal{E}^+ \sqcup \mathcal{E}^- : v \in V\} \rvert$.
ϕ	The formula complexity of a given $\mathcal{Q}$. $\phi := \sum_{v \in \text{var}(\mathcal{Q})} d(v)$.

2.2 Computational Model and Complexity Measures

Following the convention in database theory, all computations are assumed to be under the RAM model. All tuples take $O(1)$ space, and memory read/write takes $O(1)$ time.

We make further assumptions on the underlying data structure of relations. Given a relation R_V and a V-tuple t, the following operations all take $O(1)$ time: (i) check whether $t \in R_V$; (ii) insert (delete) t into (from) R_V. Furthermore, the set $\{t \in \text{dom}(V) : t \in R_V\}$ can be enumerated with $O(1)$ delay. Last but not least, we assume the relations are conveniently indexed whenever needed, in the sense that given any W-tuple t, we can efficiently retrieve the tuples in R_V that joins with t; in other words, the set $R_V \ltimes \{t\} = \{t' \in R_V : \pi_{W \cap V} t' = \pi_{W \cap V} t\}$ can be enumerated with $O(1)$ delay.

In practice, such a data structure can be implemented using a combination of hash tables and linked lists, with a few limitations: (1) the $O(1)$ operations might take amortized $O(1)$ time instead; (2) in order to achieve a low hash collision rate, the data structure could take up space orders of magnitude larger than the number of tuples in the relation.

We derive the runtime of various relational operations based on these assumptions. Starting with an initially empty relation over the associated output attributes,

(i) $\pi_U R_V$ can be computed by inserting $\pi_U t$ for each $t \in R_V$, taking $O(|R_V|)$ time;

(ii) $R_V \bowtie R_U$ can be computed by inserting $t \| \pi_{U-V} t'$ for every $t \in R_V$ and every $t' \in R_U \ltimes \{t\}$ (or switching the role of R_V and R_U here), taking $O(\min\{|R_V|, |R_U|\} + |R_V \bowtie R_U|)$ time;

(iii) $R_V \ltimes R_U$ can be computed by inserting every $t \in R_V$ such that $R_U \ltimes \{t\}$ is non-empty, taking $O(|R_V|)$ time; and

(iv) $R_V \triangleright N_U$ *requires* that $U \subseteq V$[3] and can be computed by inserting every $t \in R_V$ such that $N_U \ltimes \{t\}$ is empty, taking $O(|R_V|)$ time.

Given a CQ $\mathcal{Q} = (F, \mathcal{E})$, we consider three complexity measures: (i) the *input size* $\text{IN} := \sum_{V \in \mathcal{E}} |R_V|$; (ii) the *output size* $\text{OUT} := |\mathcal{Q}|$, i.e. the number of F-tuples t such that $t \in \mathcal{Q}$; and (iii) the *formula complexity* $\phi := \sum_{V \in \mathcal{E}} |V|$, capturing how complex the query expression is. For each attribute $v \in \text{var}(\mathcal{Q})$, we denote $d(v) := \sum_{V \in \mathcal{E}} \mathbb{1}(v \in V)$ the number of relations containing it, where $\mathbb{1}(\cdot)$ is the indicator function; consequently, $\phi = \sum_{v \in \text{var}(\mathcal{Q})} d(v)$. In Table 2.3, we extend these measures to $\mathsf{CQ}^{\neg}$.

With these measures, we define two types of parameterized complexity:

1. **Data complexity** is expressed in terms of IN and OUT, with ϕ regarded as constant.
2. **Combined complexity** is expressed in terms of IN, OUT, and ϕ.

These terminologies date back to [41], where OUT was not considered in the original definitions; nevertheless, as the output size could vary significantly for a fixed query and fixed IN, additionally incorporating OUT allows a much coarser-grained analysis.

Many results in database theory assume data complexity mainly for two reasons. First, given the fast-growing nature of data, ϕ is relatively insignificant when compared to IN and OUT. Second, data complexity allows a natural optimality argument. For example, any evaluation algorithm would need $\Omega(\text{IN})$ time to read each input tuple and $\Omega(\text{OUT})$ time to write out the results. If the evaluation algorithm runs in time $O(\text{IN} + \text{OUT})$, typically referred to as *linear time*, optimality is automatically achieved.

2.3 Important Properties of Hypergraph Acyclicity

Recall that hypergraph α-acyclicity is defined with respect to the GYO-algorithm. It can be quickly checked that the GYO-algorithm runs in polynomial time (in terms of hypergraph size). On the other hand, both β-acyclicity and signed-acylicity are defined as a *hereditary closure of α-acyclicity*[4] [12]. These definitions offer no insight on whether β-acyclicity or signed-acyclicity can be decided in polynomial time.

[3] This is to comply with the safety condition on the $\mathsf{CQ}^{\neg}$ $R_V \triangleright N_U$; otherwise, the output size of the antijoin operator could be infinite.

[4] In the sense that all sub-hypergraphs of the hypergraph (signed-hypergraph resp.) are α-acyclic.

There exist alternative recursive characterizations of β-acyclicity and signed-acyclicity that eventually lead to polynomial-time decision algorithms. These recursive characterizations mimic the procedure of recursively removing leaves in a graph to decide whether it contains no cycles. α-acyclicity admits a similar characterization, hence we include it to make the treatment comprehensive.

Definition 4. *Let v be an attribute and $\mathcal{E}$ be a set of hyperedges. Define $\mathcal{E}_v := \{U \in \mathcal{E} : v \in U\}$ as the hyperedges in $\mathcal{E}$ incident to v, and $\mathcal{E}[\backslash v] := \{U - \{v\} : U \in \mathcal{E}\} - \{\emptyset\}$ as the set of hyperedges obtained by removing v from $\mathcal{E}$.*

For a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and $v \in \mathcal{V}$, define $\mathcal{H}[\backslash v] := (\mathcal{V} - \{v\}, \mathcal{E}[\backslash v])$ as the hypergraph obtained by removing v from $\mathcal{H}$.

For a signed-hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ and $v \in \mathcal{V}$, define $\mathcal{H}[\backslash v] := (\mathcal{V} - \{v\}, \mathcal{E}^+[\backslash v], \mathcal{E}^-[\backslash v])$ as the signed-hypergraph obtained by removing v from $\mathcal{H}$.

Definition 5 (α- and β-leaf)**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph and $v \in \mathcal{V}$ be a vertex in it.*

v is an α-leaf of $\mathcal{H}$, if there exists a maximum V in $\mathcal{E}_v$ with respect to $\subseteq$. Such V is called the pivot[5] *of v.*

v is a β-leaf of $\mathcal{H}$, if $\mathcal{E}_v$ can be linearly ordered by $\subseteq$.

Definition 6 (Signed-leaf)**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ be a signed-hypergraph and $v \in \mathcal{V}$ be a vertex in it.*

v is a signed-leaf of $\mathcal{H}$, if there exists a maximum V in $\mathcal{E}_v^+$ with respect to $\subseteq$, and $\{V\} \cup \{U \in \mathcal{E}_v^- : U \not\subseteq V\}$ can be linearly ordered by $\subseteq$. We similarly call V the pivot of v.

Lemma 1 ([27, 47, 22, 13, 14, 48])**.** *Let $\mathcal{H}$ be a (signed-)hypergraph, and $\tau \in \{\alpha, \beta, signed\}$.*

- *If $\mathcal{H}$ is empty, then $\mathcal{H}$ is τ-acyclic.*
- *If $\mathcal{H}$ is non-empty and τ-acyclic, then $\mathcal{H}$ has a τ-leaf.*
- *Let v be any τ-leaf of $\mathcal{H}$. Then $\mathcal{H}$ is τ-acyclic if and only if $\mathcal{H}[\backslash v]$ is τ-acyclic.*

With Lemma 1, for $\tau \in \{\alpha, \beta, \text{signed}\}$, τ-acyclicity can be decided by iteratively identifying and removing any τ-leaf, until either the empty (signed-)hypergraph is reached (then $\mathcal{H}$ is τ-acyclic) or no more τ-leaves can be found (then $\mathcal{H}$ is not τ-acyclic). It is easy to verify that this procedure runs in polynomial time.

[5] If there are multiple candidates for the pivot, we apply a consistent tie-breaking rule and select only one candidate as the pivot. For example, we may pre-index all hyperedges and always select the candidate with the largest index.

Definition 7. *Let* $(\kappa, \tau) \in \{(\epsilon, \alpha), (\epsilon, \beta), (\textit{signed}, \textit{signed})\}$ *and* $\mathcal{H}$ *be a* κ*-hypergraph.*

Let $\sigma = v_1 v_2 \cdots v_k$ *be a vertex ordering of all vertices in* $\mathcal{H}$*.*

Denote $\mathcal{H}^{(0)} := \mathcal{H}$ *and* $\mathcal{H}^{(i)} := \mathcal{H}^{(i-1)}[\backslash v_i]$ *for each* $i \in [k]$*.*

σ *is called a* τ*-elimination sequence of* $\mathcal{H}$*, if* v_i *is a* τ*-leaf of* $\mathcal{H}^{(i-1)}$ *for every* $i \in [k]$*.*

Example 3. *Consider the same hypergraph* $\mathcal{H}_1$ *in Example 1, which has hyperedges* $V_1 = \{A, B\}, V_2 = \{B, C\}, V_3 = \{A, C\}, V_4 = \{A, B, C\}$*. We have seen that* $\mathcal{H}_1$ *is* α*-acyclic but not* β*-acyclic. Furthermore, its* α*-acyclicity is witnessed by an* α*-elimination sequence* $\sigma = ABC$*:*

- $\mathcal{H}_1$*:* $V_1 = \{A, B\}, V_2 = \{B, C\}, V_3 = \{A, C\}, V_4 = \{A, B, C\}$*.* A *is an* α*-leaf with pivot* V_4*.*
- $\mathcal{H}_1[\backslash A]$*:* $V_1 = \{B\}, V_2 = \{B, C\}, V_3 = \{C\}, V_4 = \{B, C\}$*.* B *is an* α*-leaf with pivot* V_4*, where we employ the largest-index rule to break the tie between pivot candidates* V_2 *and* V_4*.*
- $\mathcal{H}_1[\backslash A][\backslash B]$*:* $V_2 = \{C\}, V_3 = \{C\}, V_4 = \{C\}$*.* C *is an* α*-leaf with pivot* V_4*.*

Notice that the hyperedges incident to A *are* V_1, V_3, V_4*, which cannot be linearly ordered by* $\subseteq$*, since* $V_1 \not\subseteq V_3$ *and* $V_3 \not\subseteq V_1$*. The cases of* B *and* C *are identical up to symmetry. Hence,* $\mathcal{H}_1$ *is not* β*-acyclic as it contains no* β*-leaf.*

Example 4. *Consider a signed-hypergraph* $\mathcal{H}$ *with positive hyperedges* $V_1 = \{A, B\}, V_2 = \{B, C\}, V_3 = \{C, D\}$ *and negative hyperedges* $U_1 = \{A, B, C\}, U_2 = \{A, B, C, D\}$*.*

$\mathcal{H}$ *is signed-acyclicity, as witnessed by a signed-elimination sequence* $\sigma = ADBC$*:*

- $\mathcal{H}$*: positive* $V_1 = \{A, B\}, V_2 = \{B, C\}, V_3 = \{C, D\}$*, negative* $U_1 = \{A, B, C\}, U_2 = \{A, B, C, D\}$*.*
  A *is a signed-leaf with pivot* V_1*, where* $V_1 \subseteq U_1 \subseteq U_2$*.*
- $\mathcal{H}[\backslash A]$*: positive* $V_1 = \{B\}, V_2 = \{B, C\}, V_3 = \{C, D\}$*, negative* $U_1 = \{B, C\}, U_2 = \{B, C, D\}$*.*
  D *is a signed-leaf with pivot* V_3*, where* $V_3 \subseteq U_2$*.*
- $\mathcal{H}[\backslash A][\backslash D]$*: positive* $V_1 = \{B\}, V_2 = \{B, C\}, V_3 = \{C\}$*, negative* $U_1 = \{B, C\}, U_2 = \{B, C\}$*.*
  B *is a signed-leaf with pivot* V_2*. There is no negative* U *such that* $U \not\subseteq V_2$*.*

- $\mathcal{H}[\setminus A][\setminus D][\setminus B]$: *positive* $V_2 = \{C\}, V_3 = \{C\}$, *negative* $U_1 = \{C\}, U_2 = \{C\}$. C *is a signed-leaf with pivot* V_3. *There is no negative* U *such that* $U \not\subseteq V_3$.

Last but not least, we remark a result that allows us to choose the signed-elimination sequence with a certain degree of freedom.

Lemma 2 ([13])**.** *Let* $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ *be a signed-acyclic signed-hypergraph and* $S \subseteq \mathcal{V}$ *be a set of attributes. If* $\mathcal{H}' = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^- \cup \{S\})$ *is signed-acyclic, then there exists a signed-elimination sequence of* $\mathcal{H}$ *where attributes in* S *appear last.*

2.4 Related Work

CQ and α-acyclicity. An important direction in database theory is to identify classes of CQ that admit linear time evaluation algorithms. It is well known that a full CQ can be evaluated in linear time if and only if it is α-*acyclic* (under a widely believed conjecture) [46, 9].

Following the pioneering Yannakakis algorithm [46] that attains optimality, i.e. linear time, on full α-acyclic CQ, CQ evaluation sees research interests from both theory and practice communities. On the theory side, the restriction to full α-acyclicity CQ is relaxed to incorporate cyclicity [8, 7] and projection [9, 21, 29], with numerous notions of *query-widths* developed to measure how far the CQ is from an ideal state that admits a linear time evaluation algorithm.

Meanwhile, there are numerous efforts to implement the Yannakakis algorithm in practice. These approaches can be classified as two types: (i) exploring how the Yannakakis algorithm can be efficiently implemented within specific database systems or frameworks [45, 11, 10, 49]; (ii) re-expressing the Yannakakis algorithm as SQL statements to empower an easy, system-agnostic integration [25, 43].

Despite the successes, these works also reveal a drawback of the Yannakakis algorithm, which partially explains why the 40-year-old algorithm has not yet been adopted in commercial systems — an expensive factor in ϕ is hidden in the data complexity. Combined complexity, therefore, serves as a better indicator of the practical performance of an algorithm. In this thesis, all complexity results will be presented in combined complexity unless stated otherwise.

Prior results on $\mathsf{CQ}^\neg$. The notion of $\mathsf{CQ}^\neg$ is first studied by Brault-Baron [12, 13]. [12] studies *negated conjunctive queries* (NCQ), a special case of $\mathsf{CQ}^\neg$ with only *singleton*

positive hyperedges, due to its close connection to CNF formula when restricting to Boolean domains, i.e. each attribute domain being $\{0, 1\}$. By converting an NCQ to a CNF formula and applying the Davis-Putnam resolution [20], [12] shows that a signed-acyclic NCQ can be decided in $O(\text{IN} \log \text{IN})$ time. The decision algorithm is then extended to cover signed-acyclic $\mathsf{CQ}^{\neg}$ [12] and to an enumeration algorithm [13].

Theorem 1 ([13])**.** *A full signed-acyclic* $\mathsf{CQ}^{\neg}$ *can be enumerated with either*

- $O(IN \log IN)$ *pre-processing time and* $O(\log IN)$ *delay; or*
- $O(IN \log^{\phi} IN)$ *pre-processing time and* $O(1)$ *delay.*

Therefore, a full signed-acyclic $\mathsf{CQ}^{\neg}$ *can be evaluated in either* $O((\text{IN} + \text{OUT}) \log \text{IN})$ *time or* $O(\text{IN} \log^{\phi} \text{IN} + \text{OUT})$ *time.*

The logarithmic factors are consequences of translating arbitrary attribute domains to Boolean domains. Later, several works related to $\mathsf{CQ}^{\neg}$ emerge in the theory community. [15] gives a polynomial-time counting algorithm for signed-acyclic NCQs, yet it is unknown whether the result can be extended to signed-acyclic $\mathsf{CQ}^{\neg}$. [33] studies characterization of $\mathsf{CQ}^{\neg}$ beyond signed-acyclicity, with the scope limited to decision problem on Boolean domains. [16] focuses on direct access, a task to retrieve the k-th answer from a set of ordered answers for arbitrary k, of $\mathsf{CQ}^{\neg}$, developing a novel circuit construction technique while generalizing the characterization studied by [33]. These works remain theory-centric, showing no implication of a practical algorithm that decides, counts, or evaluates a signed-acyclic $\mathsf{CQ}^{\neg}$.

The state-of-the-art result on signed-acyclic $\mathsf{CQ}^{\neg}$ comes from [48], which presents an enumeration algorithm that not only achieves the optimal data complexity $O(\text{IN} + \text{OUT})$ (when regarded as an evaluation algorithm) but also exercises great generality. Although their algorithm works on a more general query class than $\mathsf{CQ}^{\neg}$, because the focus of this thesis is on counting and evaluation of full $\mathsf{CQ}^{\neg}$, we distill their results to these cases.

Theorem 2 ([48])**.** *Let* $\mathcal{Q}$ *be a full signed-acyclic* $\mathsf{CQ}^{\neg}$*.*

$\mathcal{Q}$ *can be enumerated with* $O(\phi^3 + \phi \cdot \text{IN})$ *pre-processing time and* $O(\phi)$ *delay; consequently,* $\mathcal{Q}$ *can be evaluated in time* $O(\phi^3 + \phi \cdot \text{IN} + \phi \cdot \text{OUT})$*.*

$|\mathcal{Q}|$ *can be computed in time* $O(\phi^3 + \phi \cdot \text{IN})$*.*

[48]'s algorithm is not restricted to counting; in fact, it supports *aggregation over arbitrary commutative semirings*[6]. The generality, however, comes at a cost. The algorithm

[6] Aggregation extends the notion of counting and will be discussed in Section 3.2.

requires maintaining complex doubly-linked-list indices to skip intermediate tuples masked by the negative relations; moreover, to support aggregation, it builds upon a context-free grammar representation of the query and uses a RangeSum oracle [18] in a black-box way. These complications present great challenges to practical implementation, especially when our intended scope is the special case of counting and evaluation of full signed-acyclic $\mathsf{CQ}^{\neg}$.

Chapter 3

Warm-Up: An Old Counting Algorithm for α-Acyclic CQ

In this section, we review a counting variant of the Yannakakis Algorithm [46] for α-acyclic CQ. This variant shares the same fundamental idea as our new counting algorithm for signed-acyclic $\mathsf{CQ}^{\neg}$, which is detailed in Section 4 and serves as the core contribution of this thesis.

3.1 The Yannakakis Algorithm

Recall that we previously defined α-acyclicity in terms of the GYO-algorithm [27, 47], which iteratively reduces a hypergraph to one final empty hyperedge. Another common definition of α-acyclicity is with respect to a *join tree*, which has wide applications in database theory.

Definition 8. *Let $\mathcal{Q} = (F, \mathcal{E})$ be an α-acyclic CQ. A join tree $\mathcal{T}$ for $\mathcal{Q}$ is a rooted tree with nodes $N(\mathcal{T}) = \mathcal{E}$, such that for every attribute $v \in \mathrm{var}(\mathcal{Q})$, the set of nodes $\mathcal{E}_v = \{U \in \mathcal{E} : v \in U\}$ forms a connected component in $\mathcal{T}$. This is known as the connected property.*

$\mathcal{Q}$ is α-acyclic, if it admits a join tree.

Building a join tree. A join tree can be regarded as a concise representation of one execution of the GYO-algorithm [14]. Indeed, given an α-acyclic CQ, the GYO-algorithm can be slightly modified to build a join tree as follows:

(1) Remove an attribute that only appears in one hyperedge.

(2) If there exists distinct $V_1, V_2 \in \mathcal{E}$ with $V_1 \subseteq V_2$, make V_1 a child node of V_2 and remove V_1 from $\mathcal{E}$.

(3) Repeat (1) and (2) until there is one remaining hyperedge (which becomes the root).

Building a join tree via an α-elimination sequence. Recall that, an attribute v is an α-leaf if there exists a hyperedge V, $v \in V$, such that for any other hyperedge U with $v \in U$, we have $U \subseteq V$. Such V is called the pivot of v, and an α-elimination sequence (Definition 7) guarantees that the pivot can be found at the elimination of each attribute in the sequence. We can therefore modify the above join tree building procedure to be deterministic as follows:

(1) For each attribute v in an α-elimination sequence: (i) find the pivot V of v; (ii) for any other hyperedge U with $v \in U$, make U a child node of V and remove U; (iii) remove v from V.

(2) For all remaining hyperedges $V_1, V_2, \ldots, V_k$, make $V_2, \ldots, V_k$ as child nodes of V_1[1].

(3) Return V_1 as the root of the join tree.

Example 5. *Consider the following* CQ

$$\mathcal{Q} := R_1(X_1, X_2) \bowtie R_2(X_1, X_2, X_3) \bowtie R_3(X_2, X_3, X_4) \bowtie R_4(X_3, X_4, X_5) \bowtie R_5(X_4, X_5)$$

and denote its associated hypergraph $\mathcal{H}$ with hyperedges $V_1 = \{X_1, X_2\}, V_2 = \{X_1, X_2, X_3\}$, $V_3 = \{X_2, X_3, X_4\}, V_4 = \{X_3, X_4, X_5\}, V_5 = \{X_4, X_5\}$. $\mathcal{Q}$ is α-acyclic, as witnessed by an α-elimination sequence $\sigma = X_1X_5X_3X_2X_4$:

- *$\mathcal{H}$: X_1 is an α-leaf with pivot V_2.*
- *$\mathcal{H}[\backslash X_1]$: $V_1 = \{X_2\}, V_2 = \{X_2, X_3\}, V_3 = \{X_2, X_3, X_4\}, V_4 = \{X_3, X_4, X_5\}, V_5 = \{X_4, X_5\}$.*
  X_5 is an α-leaf with pivot V_4.

[1] At this stage, because there is no attribute left, these hyperedges can be arbitrarily made as children nodes of one another. To enforce determinism, we can apply pre-indexing and order these hyperedges by their indices.

- *$\mathcal{H}[\backslash X_1][\backslash X_5]$: $V_1 = \{X_2\}, V_2 = \{X_2, X_3\}, V_3 = \{X_2, X_3, X_4\}, V_4 = \{X_3, X_4\}, V_5 = \{X_4\}$.*
  X_3 is an α-leaf with pivot V_3.

- *$\mathcal{H}[\backslash X_1][\backslash X_5][\backslash X_3]$: $V_1 = \{X_2\}, V_2 = \{X_2\}, V_3 = \{X_2, X_4\}, V_4 = \{X_4\}, V_5 = \{X_4\}$.*
  X_2 is an α-leaf with pivot V_3.

- *$\mathcal{H}[\backslash X_1][\backslash X_5][\backslash X_3][\backslash X_2]$: $V_3 = \{X_4\}, V_4 = \{X_4\}, V_5 = \{X_4\}$. X_4 is an α-leaf with pivot V_5.*

Following σ also gives us a join tree of $\mathcal{Q}$: eliminating (i) X_1 creates child-parent edge (V_1, V_2), (ii) X_5 creates child-parent edge (V_5, V_4), and (iii) X_3 creates child-parent edges (V_2, V_3) and (V_4, V_3), resulting in a join tree with root V_3. This join tree is shown in Figure 3.1b.

The Yannakakis algorithm. One important application of join trees is the Yannakakis algorithm [46]. When given as input a join tree $\mathcal{T}$ of a full α-acyclic $\mathcal{Q}$, the algorithm evaluates $\mathcal{Q}$ by traversing $\mathcal{T}$ three times:

(1) (Bottom-up) for each child-parent edge (U, V), update $R_V \leftarrow R_V \ltimes R_U$.

(2) (Top-down) for each child-parent edge (U, V), update $R_U \leftarrow R_U \ltimes R_V$.

(3) (Bottom-up) for each child-parent edge (U, V), update $R_V \leftarrow R_V \bowtie R_U$.

Finally, the updated root relation is returned as the query result. The first two rounds of the Yannakakis algorithm are termed Yannakakis semijoin reductions, whose purpose is to ensure that every relation only contains tuples that contribute to the final query result; tuples that do not are called *dangling*. After the third round, the updated relation at each node V stores the result of joining all (original, non-updated) relations in the sub-tree rooted at V.

Example 6. *Figure 3.1 shows the Yannakakis algorithm running on $\mathcal{Q}$ from Example 5, using the join tree built on the α-elimination sequence $\sigma = X_1X_5X_3X_2X_4$. On the given database instance, we have $(a_1, b_2, c_1), (a_1, b_2, c_2)$ from R_2 are dangling, (b_2, c_1, d_2) from R_3 are dangling, and (c_1, d_2, e_1) from R_4 are dangling. During the bottom-up semijoin reduction, (c_1, d_2, e_1) and (b_2, c_1, d_2) are removed from R_4 and R_3, respectively. During the top-down semijoin reduction, the two dangling tuples in R_2 are removed. The last bottom-up join accumulates the final join result at the root, i.e. V_3, as desired.*

R_1

X_1	X_2
a_1	b_1
a_1	b_2
a_2	b_1

R_5

X_4	X_5
d_1	e_1
d_1	e_2

R_2

X_1	X_2	X_3
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_2
a_1	b_2	c_1
a_1	b_2	c_2

R_3

X_2	X_3	X_4
b_1	c_1	d_1
b_1	c_2	d_1
b_2	c_1	d_2

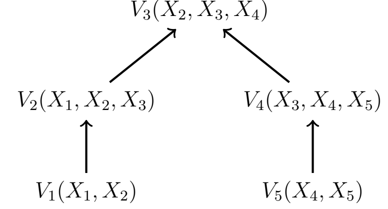
R_4

X_3	X_4	X_5
c_1	d_1	e_1
c_1	d_1	e_2
c_1	d_2	e_1
c_2	d_1	e_1

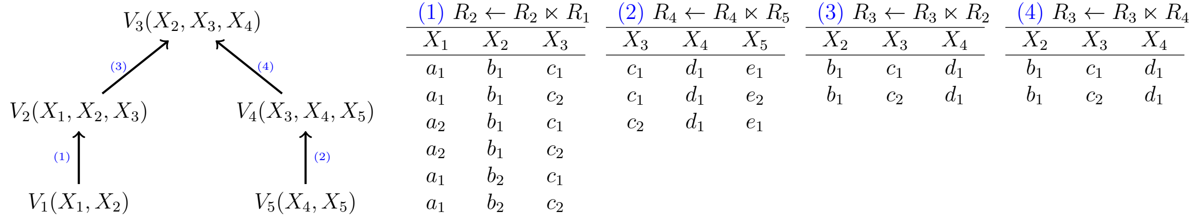
$\mathcal{Q} = \bowtie_{i=1}^{5} R_i$

X_1	X_2	X_3	X_4	X_5
a_1	b_1	c_1	d_1	e_1
a_1		"		e_2
a_2		"		e_1
a_2		"		e_2
a_1	b_1	c_2	d_1	e_1
a_2		"		e_1

(a) Database instance and query result.



(b) Join tree.



(1) $R_2 \leftarrow R_2 \ltimes R_1$

X_1	X_2	X_3
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_2
a_1	b_2	c_1
a_1	b_2	c_2

(2) $R_4 \leftarrow R_4 \ltimes R_5$

X_3	X_4	X_5
c_1	d_1	e_1
c_1	d_1	e_2
c_2	d_1	e_1

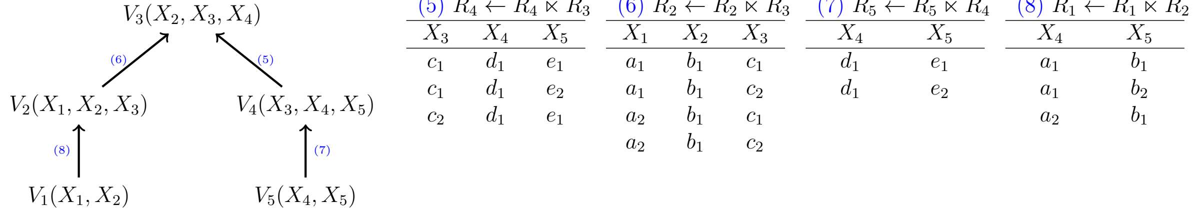
(3) $R_3 \leftarrow R_3 \ltimes R_2$

X_2	X_3	X_4
b_1	c_1	d_1
b_1	c_2	d_1

(4) $R_3 \leftarrow R_3 \ltimes R_4$

X_2	X_3	X_4
b_1	c_1	d_1
b_1	c_2	d_1

(c) Bottom-up semijoin reduction.



(5) $R_4 \leftarrow R_4 \ltimes R_3$

X_3	X_4	X_5
c_1	d_1	e_1
c_1	d_1	e_2
c_2	d_1	e_1

(6) $R_2 \leftarrow R_2 \ltimes R_3$

X_1	X_2	X_3
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_2

(7) $R_5 \leftarrow R_5 \ltimes R_4$

X_4	X_5
d_1	e_1
d_1	e_2

(8) $R_1 \leftarrow R_1 \ltimes R_2$

X_4	X_5
a_1	b_1
a_1	b_2
a_2	b_1

(d) Top-down semijoin reduction.

(1) $R_2 \leftarrow R_2 \bowtie R_1$

X_1	X_2	X_3
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_2
a_1	b_2	c_1
a_1	b_2	c_2

(2) $R_4 \leftarrow R_4 \bowtie R_5$

X_3	X_4	X_5
c_1	d_1	e_1
c_1	d_1	e_2
c_2	d_1	e_1

(3) $R_3 \leftarrow R_3 \bowtie R_2$

X_1	X_2	X_3	X_4
a_1	b_1	c_1	d_1
a_2		"	
a_1	b_1	c_2	d_1
a_2		"	

(4) $R_3 \leftarrow R_3 \bowtie R_4$

X_1	X_2	X_3	X_4	X_5
a_1	b_1	c_1	d_1	e_1
a_1		"		e_2
a_2		"		e_1
a_2		"		e_2
a_1	b_1	c_2	d_1	e_1
a_1		"		e_1

(e) Bottom-up join. Nodes are visited in the same order as the bottom-up semijoin reduction.

Figure 3.1: An illustration of the Yannakakis algorithm on Example 5 over a database instance. A blue label on a child-parent edge indicates the order it is visited in the algorithm.

The two semijoin reductions ensure that during the third round, any intermediate join result will be non-dangling; hence, when running on a full α-acyclic CQ, the size of any intermediate join result will not exceed OUT. Based on the runtime of join and semijoin derived in Section 2.2, we can derive that the Yannakakis algorithm attains the optimal $O(\text{IN} + \text{OUT})$ time in data complexity:

- **The first and second round.** For each node $V \in \mathcal{E}$, we update $R_V \leftarrow R_V \ltimes R_U$, which takes $O(|R_V|)$ time, once for each of its neighbor U in the join tree. In total, the two semijoin reductions take $O\left(\sum_{V\in\mathcal{E}} |R_V|\right) = O(\text{IN})$ time.
- **The third round.** Let $V \in \mathcal{E}$ be arbitrary and $U_1, \ldots, U_m$ be its children, in the order they are visited. Then, the algorithm iteratively updates $R_V \leftarrow R_V \bowtie R_{U_i}$, for each $i \in [m]$. Because the intermediate size never exceeds OUT, the runtime of this process process is bounded by $O\left(\sum_{i=0}^{m} \left|R_V \bowtie \left(\bowtie_{j=1}^{i} R_{U_j}\right)\right|\right) = O\left(|R_V| + \text{OUT}\right)$. Summing up the runtime at all nodes gives $O\left(\text{OUT} + \sum_{V\in\mathcal{E}} |R_V|\right) = O(\text{IN} + \text{OUT})$.

3.2 Annotation and Aggregation

Next, we show how the Yannakakis algorithm can be modified for counting. But before this, we need to introduce an important extension of CQ to incorporate *aggregation*. In SQL, this captures $\mathsf{Group\ By}$ equipped with an aggregate function. From a theoretical perspective, CQ with aggregation is a powerful framework that generalizes many classic problems in various fields, including database, constraint satisfaction, and matrix multiplication [6].

Annotation. *Annotated relations* [28, 31] is a technique that annotates each tuple in a relation with an additional weight. The weights are taken from a commutative semiring[2], which is an algebraic structure equipped with both addition and multiplication. Joins and projections are modified, such that a join *multiplies* the weights of two joining tuples, while a projection, now called an *aggregation*, *sums* the weights over all tuples which agree on the projected attributes.

For notation convenience, and given that our focus is on counting, we only consider integers equipped with the normal addition and multiplication as weights. That is, we define annotated relations only with respect to $(\mathbb{Z}, +, \times, 0, 1)$, a.k.a. the *counting ring*.

[2]A commutative semiring is a 5-tuple $(\mathbb{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, where $\oplus$, $\otimes$ are binary operators over $\mathbb{D}$ such that (1) $(\mathbb{D}, \oplus, \mathbf{0})$, $(\mathbb{D}, \otimes, \mathbf{1})$ are commutative monoids, where a monoid is an algebraic group without the invertibility condition; (2) $\otimes$ distributes over $\oplus$: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ for all $a, b, c \in \mathbb{D}$; (3) $\mathbf{0}$ is annihilating over $\otimes$: $a \otimes \mathbf{0} = \mathbf{0}$ for all $a \in \mathbb{D}$.

Definition 9. *An annotated relation R over a finite set of variables V, compactly denoted R_V, is a function $R : \mathrm{dom}(V) \to \mathbb{Z}$ that associates each V-tuple with an integer weight.*

For a V-tuple t, we write $t \in R_V$ if and only if $R_V(t) \neq 0$. We define $|R_V| := \{t \in \mathrm{dom}(V) : t \in R_V\}$ and restrict $|R_V|$ to be finite.

Let R_V, S_U, T_V be annotated relations, and $W \subseteq V$ be attributes. Define:

- *(Aggregation) The aggregation of R_V on W is an annotated relation*

$$\bigoplus_W R_V : \mathrm{dom}(V - W) \to \mathbb{D} : t \mapsto \sum_{t' \in \mathrm{dom}(W)} R_V(t \| t')$$

- *(Join) The join between R_V and S_U is an annotated relation*

$$R_V \otimes S_U : \mathrm{dom}(V \cup U) \to \mathbb{D} : t \mapsto R_V(\pi_V t) \times S_U(\pi_U t)$$

- *(Summation) The summation between R_V and T_V is an annotated relation*

$$R_V \oplus T_V : \mathrm{dom}(V) \to \mathbb{D} : t \mapsto R_V(t) + T_V(t)$$

- *(Subtraction)*[3] *The subtraction between R_V and T_V is an annotated relation*

$$R_V \ominus T_V : \mathrm{dom}(V) \to \mathbb{D} : t \mapsto R_V(t) - T_V(t)$$

Degree and aggregation. Given a U-tuple t and a relation R_V, the degree of t in R_V, denoted $\deg(R_V, t)$, is the number of tuples in R_V that agrees with t, i.e. $\deg(R_V, t) := \{t' \in R_V : \pi_{U \cap V} t' = \pi_{U \cap V} t\} = |R_V \ltimes \{t\}|$. Notice that $\deg(R_V, ()) = |R_V|$, where $()$ is the empty tuple.

Lemma 3. *Let R_V be a relation, and W be a set of attributes. Define the annotated relation $R'_V : \mathrm{dom}(V) \to \mathbb{Z} : t \mapsto \mathbb{1}(t \in R_V)$, where $\mathbb{1}(\cdot)$ is the indicator function. Then, for all $t \in \mathrm{dom}(W)$, $\deg(R_V, t) = \left(\bigoplus_{V-W} R'_V\right)(\pi_{V \cap W} t)$.*

Proof.

$$\begin{aligned}
\deg(R_V, t) &= |\{t' \in \pi_{V \cap W} R_V : t' = \pi_{V \cap W} t\}| = |\{t' \in \mathrm{dom}(V - W) : (\pi_{V \cap W} t) \| t' \in R_V\}| \\
&= \sum_{t' \in \mathrm{dom}(V-W)} \mathbb{1}\left((\pi_{V \cap W} t) \| t' \in R_V\right) = \sum_{t' \in \mathrm{dom}(V-W)} R'_V((\pi_{V \cap W} t) \| t') \\
&= \left(\bigoplus_{V-W} R'_V\right)(\pi_{V \cap W} t)
\end{aligned}$$

□

[3] For an arbitrary commutative semiring, we need the existence of additive inverse to define subtraction.

Lemma 3 reveals a natural connection between degree calculation and aggregation: given an unannotated relation R_V, we can annotate R_V in the most natural way, i.e. annotating any $t \in R_V$ by 1 and annotating any $t \notin R_V$ by 0; subsequently, for any attributes W, the aggregation $\bigoplus_{V-W} R_V$ gives us precisely the degree of any W-tuple in R_V. In particular, $\bigoplus_V R_V = |R_V|$. In what follows, with a slight abuse of notation, any unannotated relation R_V will be simultaneously equipped with such a natural annotation.

Next, we introduce a scenario in which we can combine two aggregation results into one. This result is inspired by [15], and it will become useful when we later present the core counting algorithm of this thesis.

Lemma 4. *Let $\mathcal{R}_1, \mathcal{R}_2$ be two disjoint sets of annotated relations, $V_1 := \mathrm{var}(\mathcal{R}_1), V_2 := \mathrm{var}(\mathcal{R}_2)$. Let $W \subseteq V_1 \cup V_2$ be arbitrary. If $V_1 \cap V_2 \subseteq W$, then*

$$\bigoplus_{V_1 \cup V_2 - W} \bigotimes_{R_V \in \mathcal{R}_1 \uplus \mathcal{R}_2} R_V = \left(\bigoplus_{V_1 - W} \bigotimes_{R_V \in \mathcal{R}_1} R_V \right) \otimes \left(\bigoplus_{V_2 - W} \bigotimes_{S_U \in \mathcal{R}_2} S_U \right)$$

Before proceeding to the proof, we explain its significance in the degree context. Suppose that we have a full CQ $\mathcal{Q}$ where its relations have been partitioned into $\mathcal{R}_1$ and $\mathcal{R}_2$. Let $\mathcal{Q}_1$ and Q_2 be full CQs associated to the joins among $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. Then, once the hypothesis of Corollary 4 is satisfied, the result entails that to aggregate the degrees of W-tuples in $\mathcal{Q}$, we can separately aggregate the degrees of $(V_1 - W)$-tuples in $\mathcal{Q}_1$ and $(V_2 - W)$-tuples in $\mathcal{Q}_2$, and combine their results via a join.

Proof. Since $V_1 \cap V_2 \subseteq W$, one can easily verify that $V_1 \cup V_2 - W = (V_1 - W) \uplus (V_2 - W)$. Hence, there exists a natural bijection between $\mathrm{dom}(V_1 \cup V_2 - W)$ and $\mathrm{dom}(V_1 - W) \times \mathrm{dom}(V_2 - W)$ that maps each t to $(\pi_{V_1 - W} t, \pi_{V_2 - W} t)$. Furthermore, for any $t \in \mathrm{dom}(W), t_1 \in \mathrm{dom}(V_1 - W), t_2 \in \mathrm{dom}(V_2 - W)$ and $R_V \in \mathcal{R}_1$, we have $\pi_V t \| t_1 \| t_2 = \pi_V t \| t_1$ (a symmetric argument applies to any $S_U \in \mathcal{R}_2$). Hence, using commutativity and distributivity in the semiring, for each $t \in \mathrm{dom}(W)$, we get

$$\begin{aligned}
&\left(\bigoplus_{V_1 \cup V_2 - W} \bigotimes_{R_V \in \mathcal{R}_1 \uplus \mathcal{R}_2} R_V \right)(t) \\
&= \bigoplus_{t' \in \mathrm{dom}(V_1 \cup V_2 - W)} \left(\bigotimes_{R_V \in \mathcal{R}_1 \uplus \mathcal{R}_2} R_V \right)(t \| t') \\
&= \bigoplus_{t'_1 \in \mathrm{dom}(V_1 - W)} \bigoplus_{t'_2 \in \mathrm{dom}(V_2 - W)} \left(\bigotimes_{R_V \in \mathcal{R}_1 \uplus \mathcal{R}_2} R_V \right)(t \| t'_1 \| t'_2)
\end{aligned}$$

$$= \bigoplus_{t'_1 \in \text{dom}(V_1 - W)} \bigoplus_{t'_2 \in \text{dom}(V_2 - W)} \bigotimes_{R_V \in \mathcal{R}_1 \uplus \mathcal{R}_2} R_V(\pi_V t \| t'_1 \| t'_2)$$

$$= \bigoplus_{t'_1 \in \text{dom}(V_1 - W)} \bigoplus_{t'_2 \in \text{dom}(V_2 - W)} \left(\bigotimes_{R_V \in \mathcal{R}_1} R_V(\pi_V t \| t'_1) \right) \otimes \left(\bigotimes_{S_U \in \mathcal{R}_2} S_U(\pi_U t \| t'_2) \right)$$

$$= \left(\bigoplus_{t'_1 \in \text{dom}(V_1 - W)} \bigotimes_{R_V \in \mathcal{R}_1} R_V(\pi_V t \| t'_1) \right) \otimes \left(\bigoplus_{t'_2 \in \text{dom}(V_2 - W)} \bigotimes_{S_U \in \mathcal{R}_2} S_U(\pi_U t \| t'_2) \right)$$

$$= \left(\left(\bigoplus_{V_1 - W} \bigotimes_{R_V \in \mathcal{R}_1} R_V \right) \otimes \left(\bigoplus_{V_2 - W} \bigotimes_{S_U \in \mathcal{R}_2} S_U \right) \right)(t)$$

□

3.3 A Counting Variant of the Yannakakis Algorithm

Let $\mathcal{Q}$ be a full α-acyclic CQ $\mathcal{Q}$. When the problem of interest is to decide $\mathcal{Q}$ (i.e. whether or not $\mathcal{Q}$ produces any result), or more generally to count $|\mathcal{Q}|$, only the first bottom-up Yannakakis semijoin reduction is necessary. The idea is to annotate each relation initially with 1 and replace each semijoin $R_V \leftarrow R_V \ltimes R_U$ with aggregation $R_V \leftarrow R_V \otimes \big(\bigoplus_{U-V} R_U\big)$. After the bottom-up traversal, the weight $R_V(t)$ for each $t \in R_V$ reflects the contribution of t, i.e. its degree, in the join result of all relations in the sub-tree rooted at V. Then, $|\mathcal{Q}|$ is obtained through a final aggregation at the root of $\mathcal{T}$.

Recall that a join tree is also built bottom-up in the modified GYO-algorithm. Hence, rather than building the join tree bottom-up and then traversing it bottom-up to perform counting, the two phases can be combined. Moreover, recall that a join tree can be built following an α-elimination sequence. Algorithm 1 puts together these ideas, where for each hyperedge V, it maintains an annotated relation S_V, which stores the degrees of all V-tuples in the join result of all relations in the sub-join tree rooted at V. In the algorithm, to make the notation clear, we explicitly keep track of the attributes eliminated and use the notation $A \subseteq_C B$ to mean $A - C \subseteq B$ (equivalently, $A \subseteq B \cup C$) for sets of attributes A, B, C. This is so that we can compactly denote $V - \{v\} \subseteq W$ as $V \subseteq_{\{v\}} W$ for some hyperedges V, W and attribute v. This notation is taken from [33].

Example 7. *Consider the same* CQ $\mathcal{Q} = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5$ *from Example 5, which admits an α-elimination sequence* $\sigma = X_1 X_5 X_3 X_2 X_4$.

Algorithm 1 BUILDANDCOUNT($\mathcal{Q} = \mathcal{E}$)

Input: full α-acyclic CQ $\mathcal{Q} = \mathcal{E}$

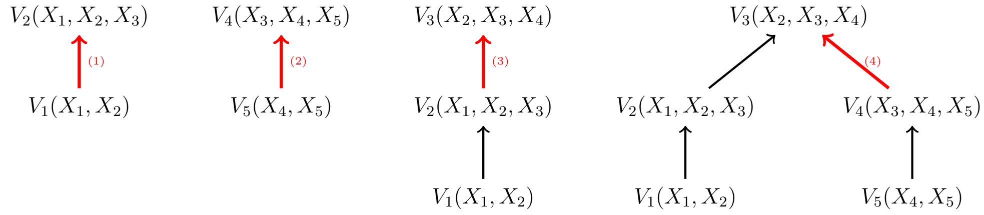
Output: a join tree for $\mathcal{Q}$, and annotated $\{S_V\}_{V\in\mathcal{E}}$ storing the degrees at each node

1: **procedure** MAKEPARENT(V, U) ▷ Helper function to update V as the parent of U.
2: $S_V \leftarrow S_V \otimes \left(\bigoplus_{U-V} S_U\right)$
3: $parent_U \leftarrow V$
4: **end procedure**
5: $\sigma \leftarrow$ find any α-elimination sequence of $\mathcal{Q}$
6: $V_{\text{elim}} \leftarrow \emptyset$ ▷ Explicitly track eliminated attributes.
7: **for all** $V \in \mathcal{E}$ **do**
8: $S_V \leftarrow \bigoplus_{\emptyset} R_V$ ▷ Annotate each $t \in R_V$ with weight 1.
9: $parent_V \leftarrow$ Null ▷ Parent-child relationship to be updated during the algorithm.
10: **end for**
11: **for all** $v \in \sigma$ **do**
12: $\mathcal{E}_v \leftarrow \{U \in \mathcal{E} : parent_U = \text{Null and } v \in U\}$
13: $V \leftarrow$ the hyperedge in $\mathcal{E}_v$, such that $U \subseteq_{V_{\text{elim}}} V$ for all $U \in \mathcal{E}_V$
14: **for all** $U \in \mathcal{E}_v - \{V\}$ **do** MAKEPARENT(V, U) **end for**
15: $V_{\text{elim}} \leftarrow V_{\text{elim}} \cup \{v\}$
16: **end for**
17: $\{V_1, V_2, \ldots, V_k\} \leftarrow$ all hyperedge in $\mathcal{E}$ with no parent node
18: **for all** $i = 2, 3, \ldots, k$ **do** MAKEPARENT(V_1, V_k) **end for**
19: **return** join tree built from $\{parent_V\}_{V\in\mathcal{E}}$, and annotated relations $\{S_V\}_{V\in\mathcal{E}}$

Figure 3.2 illustrates the execution of Algorithm 1 on $\mathcal{Q}$ and σ, using same database instance as Figure 3.1. S_i is the annotated relation maintained for R_i (which corresponds to hyperedge V_i), for each $i \in [5]$.

Notice that the creation order of child-parent edges precisely matches the order they are visited in the bottom-up Yannakakis semijoin reduction in Figure 3.1. Furthermore, the dangling tuples (c_1, d_2, e_1) in R_4 and (b_1, c_1, d_2) in R_3 have their annotation updated to 0 at the same time as their removal in the semijoin reduction. Lastly, the aggregation at the root V_3 implies $|\mathcal{Q}| = 4 + 2 + 0 = 6$, as desired.

Later, the idea of incrementally building a join tree while maintaining counts will appear again in our counting algorithm for signed-acyclic $\mathsf{CQ}^{\neg}$.



(a) Child-parent edges constructed during the elimination of α-leaves X_1, X_5, X_3.

(1) $S_2 \leftarrow R_2 \otimes R_3$

X_1	X_2	X_3	$\cdot$
a_1	b_1	c_1	1
a_1	b_1	c_2	1
a_2	b_1	c_1	1
a_2	b_1	c_2	1
a_1	b_2	c_1	1
a_1	b_2	c_2	1

(2) $S_4 \leftarrow R_4 \otimes R_5$

X_3	X_4	X_5	$\cdot$
c_1	d_1	e_1	1
c_1	d_1	e_2	1
c_1	d_2	e_1	0
c_2	d_1	e_1	1

(3) $S_3 \leftarrow R_3 \otimes (\bigoplus_{X_1} S_2)$

X_2	X_3	X_4	$\cdot$
b_1	c_1	d_1	2
b_1	c_2	d_1	2
b_2	c_1	d_2	1

(4) $S_3 \leftarrow S_3 \otimes (\bigoplus_{X_5} S_4)$

X_2	X_3	X_4	$\cdot$
b_1	c_1	d_1	4
b_1	c_2	d_1	2
b_2	c_1	d_2	0

(b) Annotated relations updated while child-parent edges are constructed.

Figure 3.2: An illustration of Algorithm 1 on Example 5 over the same database instance as Figure 3.1. A red arrow indicates a newly established child-parent edge.

Chapter 4

A New Counting Algorithm for Signed-Acyclic $\mathsf{CQ}^{\neg}$

One important aspect of theoretically optimal join algorithms is to ensure that any intermediate result produced during execution indeed contributes to the final result, i.e. it is *non-dangling*. For α-acyclic CQ, the celebrated Yannakakis algorithm [46] removes all dangling tuples from each relation via semijoins, followed by a series of binary joins to incrementally construct the query result. Yannakakis algorithm not only has theoretical guarantees but also uses only relational operators. The latter property inspires subsequent work [25, 43] to implement Yannakakis algorithm as a series of SQL statements, enabling an easy integration of the algorithm into any SQL-based database systems.

When it comes to $\mathsf{CQ}^{\neg}$, the negative relations add a layer of complexity. The state-of-the-art algorithm (Theorem 2) [48] enumerates all solutions to a signed-acyclic $\mathsf{CQ}^{\neg}$ by maintaining complicated doubly-linked lists to skip any dangling intermediate result. It is unknown whether this algorithm translates in practice, and in particular, whether it implies an algorithm using only relational operators.

We can regard these doubly-linked lists as capturing the *exact* set of query results that a tuple contributes to. When such a set is empty, we know the tuple is dangling. However, if all we want is to know whether a given tuple is dangling with respect to a $\mathsf{CQ}^{\neg}$, then instead of extraneously building these data structures, it suffices to simply *count* the contribution, i.e. degree, of the tuple in the final result.

This motivates us to study how degrees in a signed-acyclic $\mathsf{CQ}^{\neg}$ can be calculated, subsequently leading to our new counting algorithm.

Section outline. First, in Section 4.1, we demonstrate how a signed-acyclic $\mathsf{CQ}^\neg$ can be decided. Although the decision problem trivially reduces to the counting problem, our decision algorithm (i) develops a primitive that will be used repeatedly later; and (ii) introduces a general recipe to tackle signed-acyclic $\mathsf{CQ}^\neg$ through the lens of signed-elimination. Next, in Section 4.2, we explore the inclusion-exclusion principle and its implications. Eventually, these explorations lead to our counting algorithm in Section 4.3. The correctness proof and runtime analysis are presented in Section 5.

4.1 A New Decision Algorithm for Signed-Acyclic $\mathsf{CQ}^\neg$

Given an initial signed-acyclic $\mathsf{CQ}^\neg$ and a signed-elimination sequence, a general decision recipe, following [12], is to eliminate one attribute at a time, such that the $\mathsf{CQ}^\neg$ before the elimination contains a solution if and only if the $\mathsf{CQ}^\neg$ after the elimination contains a solution. [12]'s decision algorithm converts attribute domains into Boolean domains and uses the Davis-Putnam resolution [20]; in the meantime, [48]'s enumeration algorithm builds upon a similar elimination framework, in which additional data structures are maintained.

It is unknown how these approaches translate to a decision algorithm for signed-acyclic $\mathsf{CQ}^\neg$ in practice. To bridge such a gap, we introduce a new decision algorithm. Our decision algorithm shares a similar overall structure as [12] and [48], except it novelly incorporates degree calculations. These degree calculations can be easily realized in any practical database system.

Signed-leaf elimination. We first introduce the elimination procedure of one attribute, which is implemented in Algorithm 2. Consider a $\mathsf{CQ}^\neg$ $\mathcal{Q}_1 = (\mathcal{E}^+, \mathcal{E}^-)$, and suppose it has a signed-leaf v. The goal is to reduce $\mathcal{Q}_1$ to a new $\mathsf{CQ}^\neg$ $\mathcal{Q}_2$ (with the associated relations updated) that does not contain v, such that $\pi_{\text{var}(\mathcal{Q}_1)-\{v\}}\mathcal{Q}_1 = \mathcal{Q}_2$.

Recall that v is a signed-leaf, if there exists a $V \in \mathcal{E}_v^+$, called the pivot of v, such that (i) for all $U \in \mathcal{E}_v^+$, $U \subseteq V$; and (ii) $\{V\} \cup \{U \in \mathcal{E}_v^- : U \not\subseteq V\}$ can be linearly ordered by $\subseteq$. Following the naming convention of [48], the elimination of v is divided into two parts, α-step and β-step, to deal with hyperedges of types (i) and (ii), respectively.

α-step. Consider a query of the form $R_{V_1} \bowtie R_{V_2} \triangleright N_{V_3}$ with $V_2 \subseteq V_1$ and $V_3 \subseteq V_1$. Then, we can update $R_{V_1} \leftarrow R_{V_1} \ltimes R_{V_2}$ and $R_{V_1} \leftarrow R_{V_1} \triangleright N_{V_3}$ and remove R_{V_2}, N_{V_3}, respectively, without affecting the query result. α-step is to exploit this idea to remove any positive (negative resp.) hyperedge U such that $U \subseteq V$, by updating $R_V \leftarrow R_V \ltimes R_U$ ($R_V \leftarrow R_V \triangleright N_U$ resp.).

β-step. After the α-step, due to the pivot property of V, all remaining hyperedges that contain v are $\{V\} \cup \{U \in \mathcal{E}_v^- : U \not\subseteq V\}$ and can be linearly ordered by $\subseteq$. Let $V \subsetneq U_1 \subseteq U_2 \subseteq \cdots \subseteq U_m$ be a linear order on these hyperedges. The first goal of β-step is to remove the *redundant* tuples in the negative relation associated with each U_i, i.e. N_{U_i}. Now, when do we consider a tuple $t_i \in N_{U_i}$ redundant?

- Suppose there exists some $j < i$ (hence $U_j \subseteq U_i$) such that $\pi_{U_j} t_i =: t_j \in N_{U_j}$. Consider any invalid output tuple $t \notin \mathcal{Q}_1$ because it is masked by t_i; that is, $\pi_{U_i} t = t_i$. Then, $\pi_{U_j} t = \pi_{U_j} t_i = t_j$, meaning that t is also masked by $t_j \in N_{U_j}$; as a result, the filtering effect of t_i is redundant — we can remove t_i from N_{U_i} without affecting the query result.

- Suppose that $\pi_V t_i \notin R_V$. Again, consider any invalid output tuple $t \notin \mathcal{Q}_1$ masked by t_i, which implies that $\pi_V t = \pi_V t_i \notin R_V$. But $\pi_V t \notin R_V$ already entails that t cannot be a valid output tuple regardless of whether it is masked by t_i, which renders t_i redundant.

These observations suggest that we can remove all $t_i \in N_{U_i}$ such that (i) $\pi_{U_j} t_i \in N_{U_j}$ for any $j \in [i-1]$, or (ii) $\pi_V t_i \notin R_V$. To realize these removals, we update $N_{U_i} \leftarrow N_{U_i} \triangleright N_{U_j}$ for all $j \in [i-1]$ and $N_{U_i} \leftarrow N_{U_i} \ltimes R_V$.

Next, we remove v from each of $V, U_1, \ldots, U_m$, by constructing new relations $R_{V-\{v\}}$, $N_{U_1-\{v\}}, \ldots, N_{U_m-\{v\}}$ and using them to replace $R_V, N_{U_1}, \ldots, N_{U_m}$, respectively, in the reduced query $\mathcal{Q}_2$. Let us consider an invalid output tuple t of $\mathcal{Q}_2$, such that t agrees with all other relations that do not contain v, but t cannot be *extended* to a valid output of $\mathcal{Q}_1$. This is to say that, for all $a \in \text{dom}(v)$ such that $t \| a$ agrees with R_V, there exists some N_{U_i} that masks $t \| a$.

First, notice that there exists some $a \in \text{dom}(v)$ for which $t \| a$ agrees with R_V, if and only if $\pi_{V-\{v\}} t \in \pi_{V-\{v\}} R_v$; therefore, to exclude the case where t satisfies the above condition vacuously, we construct $R_{V-\{v\}} \leftarrow \pi_{V-\{v\}} R_V$ to be passed to $\mathcal{Q}_2$

Second, if for some i we have that $\pi_{U_i-\{v\}} t \notin \pi_{U_i-\{v\}} N_{U_i}$, then N_{U_i} will not mask any such extension of t, hence can be excluded from consideration. We therefore let $i \in [m]$ be the *largest* index for which $\pi_{U_i-\{v\}} t \in \pi_{U_i-\{v\}} N_{U_i}$. Then, in logic terms, t satisfies

$$\left(\pi_{U_i-\{v\}} t \in \pi_{U_i-\{v\}} N_{U_i}\right) \bigwedge \left(\forall a \in \text{dom}(v) \text{ s.t. } \pi_V t \| a \in R_V, \exists j \in [i] \text{ s.t. } \pi_{U_j} t \| a \in N_{U_j}\right)$$

Because $V \subseteq U_1 \subseteq \cdots \subseteq U_i$, any attribute in t outside of U_i has no impact on the latter clause. We can therefore truncate t to be over $\text{dom}(U_i - \{v\})$ in the latter clause to get

$$\left(t_i := \pi_{U_i-\{v\}} t \in \pi_{U_i-\{v\}} N_{U_i}\right) \bigwedge \left(\forall a \in \text{dom}(v) \text{ s.t. } \pi_V t_i \| a \in R_V, \exists j \in [i] \text{ s.t. } \pi_{U_j} t_i \| a \in N_{U_j}\right)$$

Third, consider any $b \in \text{dom}(v)$ and $j \in [i]$ such that $\pi_{U_j} t_i \| b \in N_{U_j}$. As we have previously removed all redundant tuples from these negative relations, we know that $\pi_V t_i \| b \in R_V$, and for all $k \in [i], k \neq j$, we must have $\pi_{U_k} t_i \| b \notin N_{U_k}$. Therefore, the condition simplifies to

$$\left(t_i := \pi_{U_i - \{v\}} t \in \pi_{U_i - \{v\}} N_{U_i}\right) \bigwedge \left(\left\{a : \pi_V t_i \| a \in R_V\right\} = \biguplus_{j=1}^{i} \left\{b : \pi_{U_j} t_i \| b \in N_{U_j}\right\}\right)$$

where the disjoint union nicely translates to *counting*

$$\left(t_i := \pi_{U_i - \{v\}} t \in \pi_{U_i - \{v\}} N_{U_i}\right) \bigwedge \left(\left|\{a : \pi_V t_i \| a \in R_V\}\right| = \sum_{j=1}^{i} \left|\{b : \pi_{U_j} t_i \| b \in N_{U_j}\}\right|\right)$$

and we can further simplify these cardinalities using degrees, since $\deg(R_V, t_i) = |\{a : \pi_V t_i \| a \in R_V\}|$ and $\deg(N_{U_j}, t_i) = |\{b : \pi_{U_j} t_i \| b \in N_{U_j}\}|$. The significance of redundancy removal now becomes obvious — it is to ensure that we do not overcount the effect of any negative tuple.

In summary, for such t to be an invalid output of $\mathcal{Q}_2$, it must be that $\pi_{U_i - \{v\}} t =: t_i \in \pi_{U_i - \{v\}} N_{U_i}$ and $\deg(R_v, t_i) = \sum_{j=1}^{i} \deg(N_{U_j}, t_i)$. Therefore, we insert all $t_i \in \pi_{U_i - \{v\}} N_{U_i}$ satisfying the degree equality into $N_{U_i - \{v\}}$ to be passed to the reduced query $\mathcal{Q}_2$. This completes the description of β-step.

Example 8. *Consider a* $\mathsf{CQ}^{\neg}$ $\mathcal{Q}_1 = (\{W, V\}, \{U_1, U_2\})$ *with positive hyperedges* $W = \{A, B, C\}, V = \{C, D\}$ *and negative hyperedges* $U_1 = \{B, C, D\}, U_2 = \{A, B, C, D\}$*. This will function as a running example in Sections 4.1 and 4.2.*

Figure 4.1 illustrates a database instance of $\mathcal{Q}_1$ *before and after the elimination of the signed-leaf* D*. The linear* $\subseteq$*-chain in the* β*-step is* $V \subsetneq U_1 \subseteq U_2$*.*

Since there is no other hyperedge U *with* $D \in U$ *and* $U \subseteq V$*,* α*-step does nothing.*

At the beginning of β*-step (i.e. after* α*-step),* R_V *becomes the only source of* D*-values in the database instance. Hence, because* $(b_2, c_3, d_3) \in N_{U_1}$ *and* $(a_2, b_2, c_4, d_4) \in N_{U_2}$ *do not agree with any tuple in* R_V*, these tuples' presence has no impact on the query result and are thus removed. Moreover,* $(b_1, c_1, d_1) \in N_{U_1}$ *already captures the masking effect of* $(a_1, b_1, c_1, d_1) \in N_{U_2}$*, so the latter is also removed.*

Now, we construct new relations $R_{V-\{D\}}, N_{U_1-\{D\}}, N_{U_2-\{D\}}$ *to be passed along to the reduced query* $\mathcal{Q}_2$*.* D *is aggregated away to provide efficient degree look-ups in* R_V, N_{U_1}, N_{U_2}*.*

- $R_{V-\{D\}}$*. Straightforward from projecting away* D *in* R_V*.*

Algorithm 2 REDUCE($\mathcal{Q}_1 = (\mathcal{E}^+, \mathcal{E}^-), v$)

Input: full CQ$^\neg$ $\mathcal{Q}_1 = (\mathcal{E}^+, \mathcal{E}^-)$, a signed-leaf v of $\mathcal{Q}_1$
Output: full CQ$^\neg$ $\mathcal{Q}_2 = (\mathcal{E}^+_{\text{red}}, \mathcal{E}^-_{\text{red}})$ such that $\mathcal{Q}_2 = \pi_{\text{var}(\mathcal{Q}_1)-\{v\}}\mathcal{Q}_1$
Global Variables: positive and negative relations $\{R_V\}_{V\in\mathcal{E}^+\cup\mathcal{E}^+_{\text{red}}}$, $\{N_U\}_{U\in\mathcal{E}^-\cup\mathcal{E}^-_{\text{red}}}$, respectively

1: $\mathcal{E}^+_{\text{red}} \leftarrow \mathcal{E}^+ - \mathcal{E}^+_v$, $\mathcal{E}^-_{\text{red}} \leftarrow \mathcal{E}^- - \mathcal{E}^-_v$
2: $V \leftarrow$ the pivot for signed-leaf v of $\mathcal{Q}_1$
3: **for all** $U \in \mathcal{E}^+_v \sqcup \mathcal{E}^-_v - \{V\}$ such that $U \subseteq V$ **do** ▷ α-step.
4: **if** $U \in \mathcal{E}^+$ **then**
5: $R_V \leftarrow R_V \ltimes R_U$
6: **else**
7: $R_V \leftarrow R_V \triangleright N_U$
8: **end if**
9: **end for**
10: $R_{V-\{v\}} \leftarrow \pi_{V-\{v\}}R_V$
11: $\mathcal{E}^+_{\text{red}} \leftarrow \mathcal{E}^+_{\text{red}} \cup \{V - \{v\}\}$
12: Find all $U_1, U_2, \ldots, U_m \in \mathcal{E}^-_v$ such that $V \subsetneq U_1 \subseteq U_2 \subseteq \cdots \subseteq U_m$
13: **for all** $i = 1, 2, \ldots, m$ **do** ▷ β-step.
14: **for all** $j = 1, 2, \ldots, i-1$ **do** $N_{U_i} \leftarrow N_{U_i} \triangleright N_{U_j}$ **end for**
15: $N_{U_i} \leftarrow N_{U_i} \ltimes R_V$
16: $N_{U_i-\{v\}} \leftarrow \left\{t \in \pi_{U_i-\{v\}}N_{U_i} : \deg(R_V, t) = \sum_{j=1}^{i} \deg(N_{U_j}, t)\right\}$
17: $\mathcal{E}^-_{\text{red}} \leftarrow \mathcal{E}^-_{\text{red}} \cup \{U_i - \{v\}\}$
18: **end for**
19: **return** $\mathcal{Q}_2 := (\mathcal{E}^+_{\text{red}}, \mathcal{E}^-_{\text{red}})$

- $N_{U_1-\{D\}}$. *Because* $\deg(N_{U_1}, (b_1, c_1)) = 2 = \deg(R_V, (b_1, c_1))$*, any candidate output tuple in* $\mathcal{Q}_2$ *that contains* (b_1, c_1) *would become invalid as all its extensions to* R_V *are masked by* N_{U_1}*. On the other hand,* $\deg(N_{U_1}, (b_1, c_2)) = 1 < 2 = \deg(R_V, (b_1, c_2))$*, so a candidate output tuple in* $\mathcal{Q}_2$ *containing* (b_1, c_2) *still has a chance of being valid, as not all its extensions to* R_V *are masked by* N_{U_1} *(in fact, there is precisely* $2 - 1 = 1$ *such extension).*

- $N_{U_2-\{D\}}$. *Because* $\deg(N_{U_2}, (a_1, b_1, c_2)) + \deg(N_{U_1}, (a_1, b_1, c_2)) = \deg(R_V, (a_1, b_1, c_2))$*,* (a_1, b_1, c_2) *needs to be masked in* $\mathcal{Q}_2$ *due to a collaborative effect of* N_{U_1} *and* N_{U_2}*. On the other hand,* $\deg(N_{U_2}, (a_1, b_1, c_3)) + \deg(N_{U_1}, (a_1, b_1, c_3)) = 1 + 0 < 2 = \deg(R_V, (a_1, b_1, c_3))$*.*

R_W		
A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_1	b_1	c_3

R_V	
C	D
c_1	d_1
c_1	d_2
c_2	d_1
c_2	d_2
c_3	d_1
c_3	d_2

N_{U_1}		
B	C	D
b_1	c_1	d_1
b_1	c_1	d_2
b_1	c_2	d_1
b_2	c_3	d_3

N_{U_2}			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_2	d_2
a_1	b_1	c_3	d_1
a_2	b_2	c_4	d_4

$\mathcal{Q}_1$			
A	B	C	D
a_1	b_1	c_3	d_2

(a) Database instance and query result of $\mathcal{Q}_1 = R_W \bowtie R_V \triangleright N_{U_1} \triangleright N_{U_2}$.

R_V	
C	D
c_1	d_1
c_1	d_2
c_2	d_1
c_2	d_2
c_3	d_1
c_3	d_2

N_{U_1}		
B	C	D
b_1	c_1	d_1
b_1	c_1	d_2
b_1	c_2	d_1

N_{U_2}			
A	B	C	D
a_1	b_1	c_2	d_2
a_1	b_1	c_3	d_1

(b) β-step on signed-leaf D, after removing redundant tuples in N_{U_1}, N_{U_2} via $N_{U_1} \leftarrow N_{U_1} \ltimes R_V$, $N_{U_2} \leftarrow N_{U_2} \ltimes R_V$, and $N_{U_2} \leftarrow N_{U_2} \triangleright N_{U_1}$.

$\bigoplus_D R_V$	
C	$\cdot$
c_1	2
c_2	2
c_3	2

$\bigoplus_D N_{U_1}$		
B	C	$\cdot$
b_1	c_1	2
b_1	c_2	1

$\bigoplus_D N_{U_2}$			
A	B	C	$\cdot$
a_1	b_1	c_2	1
a_1	b_1	c_3	1

(c) Aggregating away D in R_V, N_{U_1}, N_{U_2} to facilitate efficient degree look-up.

R_W		
A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_1	b_1	c_3

$R_{V-\{D\}}$
C
c_1
c_2
c_3

$N_{U_1-\{D\}}$	
B	C
b_1	c_1

$N_{U_2-\{D\}}$		
A	B	C
a_1	b_1	c_2

$\mathcal{Q}_2$		
A	B	C
a_1	b_1	c_3

(d) Relations and query result of the reduced query $\mathcal{Q}_2 = R_W \bowtie R_{V-\{D\}} \triangleright N_{U_1-\{D\}} \triangleright N_{U_2-\{d\}}$.

Figure 4.1: β-step on linear chain $V \subsetneq U_1 \subseteq U_2$.

The elimination of D is thus complete. One may see in Figure 4.1d that $\mathcal{Q}_2 = \pi_{ABC}\mathcal{Q}_1$.

Lastly, Lemma 5 establishes the correctness of Algorithm 2.

Lemma 5. *Algorithm 2 ensures that $\pi_{W-\{v\}}\mathcal{Q}_1 = \mathcal{Q}_2$, where $\mathcal{Q}_1, \mathcal{Q}_2$ are its input and output $\mathsf{CQ}^{\neg}$, respectively, v is its input signed-leaf of $\mathcal{Q}_1$, and $W := \mathrm{var}(\mathcal{Q}_1)$.*

Proof. It is easy to see that after α-step, the query result remains the same; hence, we directly compare the query results before and after β-step. In the remainder of the proof, we use the same notations as in Algorithm 2. Any relation other than R_V and $N_{U_1}, \dots, N_{U_m}$ does not contain attribute v and is unaffected by β-step; hence, we can assume without loss of generality that R_V and $N_{U_1}, \dots, N_{U_m}$ are the only relations in the query.

Let $t \in \pi_{W-\{v\}}\mathcal{Q}_1$. This means that there exists some $a \in \mathrm{dom}(v)$, such that $\pi_V t \| a \in R_V$ and $\pi_{U_1} t \| a \notin N_{U_1}, \dots, \pi_{U_m} t \| a \notin N_{U_m}$. By construction, we have $\pi_{V-\{v\}} t \in R_{V-\{v\}}$ and $\pi_{U_1-\{v\}} t \notin N_{U_1-\{v\}}, \dots, \pi_{U_m-\{v\}} t \notin N_{U_m-\{v\}}$. Therefore, $t \in \mathcal{Q}_2$.

Algorithm 3 DECIDEFULL($\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$)

Input: full signed-acyclic $\mathsf{CQ}^\neg$ $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$
Output: whether $\mathcal{Q}$ contains an output

1: $\sigma \leftarrow$ any signed-elimination sequence for $\mathcal{Q}$
2: **for all** $v \in \sigma$ **do** $\mathcal{Q} \leftarrow$ REDUCE($\mathcal{Q}, v$) **end for**
3: Evaluate $\mathcal{Q}$, which now contains only relations with no attributes
4: **if** $\mathcal{Q} \neq \emptyset$ **then return** True **else return** False **end if**

Conversely, let $t \in \mathcal{Q}_2$. Then $\pi_{V-\{v\}}t \in R_{V-\{v\}}$ and $\pi_{U_1-\{v\}}t \notin N_{U_1-\{v\}}, \ldots, \pi_{U_m-\{v\}}t \notin N_{U_m-\{v\}}$. Since $\pi_{V-\{v\}}t \in R_{V-\{v\}}$, there exists some $a \in \mathrm{dom}(v)$ such that $\pi_V t \| a \in R_V$. If $\pi_{U_1} t \| a \notin N_{U_1}, \ldots, \pi_{U_m} t \| a \notin N_{U_m}$, then $t \| a \in \mathcal{Q}_1$, giving $t \in \pi_{W-\{v\}}\mathcal{Q}_1$.

Otherwise, we can find the largest $i \in [m]$ for which $\pi_{U_i-\{v\}}t \in \pi_{U_i-\{v\}}N_{U_i}$. Because $\pi_{U_i-\{v\}}t \notin N_{U_i-\{v\}}$, by construction we know that there exists some $a \in \mathrm{dom}(v)$ such that $\pi_V t \| a \in R_V$ and $\pi_{U_1} t \| a \notin N_{U_1}, \ldots, \pi_{U_i} t \| a \notin N_{U_i}$. And because of the largest choice of such i, we have $\pi_{U_j-\{v\}}t \notin \pi_{U_j-\{v\}}N_{U_j}$ and thus $\pi_{U_j} t \| a \notin \pi_{U_j} N_{U_j}$, for each $j = i+1, \ldots, m$. Thus, $t \| a \in \mathcal{Q}_1$, which implies $t \in \pi_{W-\{v\}}\mathcal{Q}_1$. □

The decision algorithm. The decision algorithm is implemented in Algorithm 3, which iteratively invokes REDUCE to eliminate all attributes in the query. We omit its correctness proof, as it follows immediately from the correctness of REDUCE.

Lastly, we derive the runtime of our decision algorithm. Notice that [48]'s algorithm can be used to decide a signed-acyclic $\mathsf{CQ}^\neg$ by checking whether its count is non-zero, which has a runtime of $O(\phi^3 + \phi \cdot \mathrm{IN})$. Our decision algorithm matches this combined complexity.

Theorem 3. *Algorithm 3 decides a full signed-acyclic* $\mathsf{CQ}^\neg$ *in* $O\left(\phi^3 + \phi \cdot \mathrm{IN}\right)$ *time.*

Proof. First, a brute-force approach finds a signed-elimination sequence in $O\left(\phi^3\right)$ time [48].

Next, we consider the runtime of eliminating a signed-leaf v. Let $d(v)$ denote the number of relation which contains attribute v in the query. In α-step, both $R_V \ltimes R_U$ and $R_V \triangleright N_U$ can be computed by probing R_U or N_U with every $t \in R_V$. Thus, α-step takes a total of $O(d(v) \cdot |R_V|)$ time.

In β-step, for each N_{U_i}, the combined runtime to remove redundant tuples from it is similarly $O(d(v) \cdot |N_{U_i}|)$; meanwhile, for any $t \in \mathrm{dom}(U_i - \{v\})$, all of $\deg(R_V, t)$ and $\deg(N_{U_j}, t)$, $j \in [i]$, can be efficiently looked up in $O(1)$ time, if we pre-compute $\bigoplus_v R_V$

and $\bigoplus_v N_{U_1}, \ldots, \bigoplus_v N_{U_m}$ after redundant tuple removals. Such pre-computation takes $O(|R_V| + \sum_{i=1}^{m} |N_{U_i}|)$ time. Therefore, α-step and β-step combined take time

$$O\left((d(v)+1) \cdot |R_V| + \sum_{i=1}^{m} (d(v)+1) \cdot |N_{U_i}|\right) = O(d(v) \cdot \mathrm{IN})$$

Eliminating one signed-leaf v takes $O(d(v) \cdot \mathrm{IN})$ time. Subsequently, Algorithm 3 runs in time $O\left(\phi^3 + \sum_v d(v) \cdot \mathrm{IN}\right) = O\left(\phi^3 + \phi \cdot \mathrm{IN}\right)$ to decide $\mathcal{Q}$. □

4.2 The Inclusion-Exclusion Principle and Its Implications

Our new decision algorithm for signed-acyclic $\mathsf{CQ}^{\neg}$ already utilizes the idea of counting in the form of degree calculations. Now, we move on to the general counting problem of signed-acyclic $\mathsf{CQ}^{\neg}$.

The inclusion-exclusion principle. As observed in [13, 48], counting the solutions to a $\mathsf{CQ}^{\neg}$ can be done using the inclusion-exclusion principle. Concretely, let $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$ be a full $\mathsf{CQ}^{\neg}$ with positive hyperedges $\mathcal{E}^+$ and negative hyperedges $\mathcal{E}^-$. It follows that

$$|\mathcal{Q}| = \sum_{\mathcal{S} \subseteq \mathcal{E}^-} (-1)^{|\mathcal{S}|} \cdot |\mathcal{Q}_{\mathcal{S}}|$$

where $\mathcal{Q}_{\mathcal{S}}$ is the full CQ (without negations) with hyperedges $\mathcal{E}^+ \cup \mathcal{S}$. If $\mathcal{Q}$ is signed-acyclic, then each $\mathcal{Q}_{\mathcal{S}}$ is α-acyclic by definition; therefore, each $|\mathcal{Q}_{\mathcal{S}}|$ and thus $|\mathcal{Q}|$ all take $O(\mathrm{IN})$ time to compute in data complexity. Nevertheless, the data complexity hides an exponential dependency on $|\mathcal{E}^-|$, making this approach impractical.

To fully understand why the inclusion-exclusion principle leads to exponentially many sub-problems, we start with a simple observation.

Lemma 6. *Let $\mathcal{Q}$ be any full $\mathsf{CQ}^{\neg}$ and N be a relation such that* $\mathrm{var}(N) \subseteq \mathrm{var}(\mathcal{Q})$. *Then*

$$\mathcal{Q} = (\mathcal{Q} \bowtie N) \uplus (\mathcal{Q} \triangleright N)$$

Proof. Since $\mathrm{var}(N) \subseteq \mathrm{var}(\mathcal{Q})$, all of $\mathcal{Q}$, $\mathcal{Q} \bowtie N$, and $\mathcal{Q} \triangleright N$ are defined over the same attributes, hence the equation is well-defined. The partition holds by definitions of join and antijoin. □

On important consequence of Lemma 6 is that the disjoint union translates nicely to counting: $|\mathcal{Q}| = |\mathcal{Q} \bowtie N| + |\mathcal{Q} \triangleright N|$, or more importantly, $|\mathcal{Q} \triangleright N| = |\mathcal{Q}| - |\mathcal{Q} \bowtie N|$. That is, the count of a query with a negative relation N, i.e. $|\mathcal{Q} \triangleright N|$, is equal to the count of the query without it, i.e. $|\mathcal{Q}|$, subtracted by the count of the query when treating N as a positive relation, i.e. $|\mathcal{Q} \bowtie N|$.

This idea leads to a procedure to count $\mathsf{CQ}^\neg$ by iteratively unravelling its negative relations. To see this, suppose we are given a $\mathsf{CQ}^\neg$ $\mathcal{Q} = R_4 \bowtie R_0 \triangleright N_3 \triangleright N_2 \triangleright N_1$, i.e. R_4, R_0 are positive relations and N_3, N_2, N_1 are negative relations. The index we place on the relations corresponds to the order we unravel them, which will become useful soon.

By iteratively unraveling N_3, N_2, N_1 in the leftmost expression using Lemma 6, we get

$$\begin{aligned}
|\mathcal{Q}| &= |R_4 \bowtie R_0 \triangleright N_3 \triangleright N_2 \triangleright N_1| \\
&= |R_4 \bowtie R_0 \triangleright N_2 \triangleright N_1| - |R_4 \bowtie R_0 \bowtie N_3 \triangleright N_2 \triangleright N_1| \\
&= |R_4 \bowtie R_0 \triangleright N_1| - |R_4 \bowtie R_0 \bowtie N_2 \triangleright N_1| - |R_4 \bowtie R_0 \bowtie N_3 \triangleright N_2 \triangleright N_1| \\
&= |R_4 \bowtie R_0| - |R_4 \bowtie R_0 \bowtie N_1| - |R_4 \bowtie R_0 \bowtie N_2 \triangleright N_1| - |R_4 \bowtie R_0 \bowtie N_3 \triangleright N_2 \triangleright N_1|
\end{aligned}$$

In the end, there are still two expressions containing negative relations, namely $|R_4 \bowtie R_0 \bowtie N_2 \triangleright N_1|$ and $|R_4 \bowtie R_0 \bowtie N_3 \triangleright N_2 \triangleright N_1|$. We can recursively unravel them in a similar manner, until all remaining expressions consist only of positive relations. Indeed, this recovers precisely the inclusion-exclusion principle, which is unideal for generating exponentially many sub-problems.

Avoiding the exponential dependency. We offer a different angle to look at this process. Recall that joins and antijoins commute with one another; hence, we can re-order the relations in the previous equality based on a descending order of their indices:

$$|\mathcal{Q}| = |R_4 \bowtie R_0| - |R_4 \bowtie N_1 \bowtie R_0| - |R_4 \bowtie N_2 \triangleright N_1 \bowtie R_0| - |R_4 \bowtie N_3 \triangleright N_2 \triangleright N_1 \bowtie R_0|$$

Consider the index ordering on the relations $(R_0, N_1, N_2, N_3, R_4)$. For each index $i \in [4]$, define $\mathcal{Q}_i$ to be a $\mathsf{CQ}^\neg$ having the 0-th and i-th relations as positive, and any relation in between as negative. That is,

$$\begin{aligned}
\mathcal{Q}_1 &= N_1 \bowtie R_0 \\
\mathcal{Q}_2 &= N_2 \triangleright N_1 \bowtie R_0 \\
\mathcal{Q}_3 &= N_3 \triangleright N_2 \triangleright N_1 \bowtie R_0 \\
\mathcal{Q}_4 &= R_4 \triangleright N_3 \triangleright N_2 \triangleright N_1 \bowtie R_0 = \mathcal{Q}
\end{aligned}$$

Assume for now that every $\mathcal{Q}_i$ is a safe $\mathsf{CQ}^\neg$. There is a nice interpretation of these $\mathcal{Q}_i$: (i) $\mathcal{Q}_1$ captures tuples from R_0 that *do not survive* N_1 (i.e. they join with some negative tuple in N_1); (ii) $\mathcal{Q}_2$ captures tuples from R_0 that survive N_1 but do not survive N_2; (iii) $\mathcal{Q}_3$ captures tuples from R_0 that survive N_1, N_2 but do not survive N_3; finally, (iv) $\mathcal{Q}_4$ captures tuples from R_0 that survive N_1, N_2, N_3 and join with R_4[1], which are precisely what we want for $\mathcal{Q}$.

More importantly, the above equality involving $|\mathcal{Q}|$ translates to:

$$|\mathcal{Q}_4| = |\mathcal{Q}| = |R_4 \bowtie R_0| - |R_4 \bowtie \mathcal{Q}_1| - |R_4 \bowtie \mathcal{Q}_2| - |R_4 \bowtie \mathcal{Q}_3|$$

and we can similarly unravel each of $|\mathcal{Q}_3|, |\mathcal{Q}_2|, |\mathcal{Q}_1|$:

$$\begin{aligned}
|\mathcal{Q}_3| &= |N_3 \triangleright N_2 \triangleright N_1 \bowtie R_0| \\
&= |N_3 \triangleright N_1 \bowtie R_0| - |N_3 \bowtie N_2 \triangleright N_1 \bowtie R_0| \\
&= |N_3 \bowtie R_0| - |N_3 \bowtie N_1 \bowtie R_0| - |N_3 \bowtie N_2 \triangleright N_1 \bowtie R_0| \\
&= |N_3 \bowtie R_0| - |N_3 \bowtie \mathcal{Q}_1| - |N_3 \bowtie \mathcal{Q}_2| \\
|\mathcal{Q}_2| &= |N_2 \triangleright N_1 \bowtie R_0| \\
&= |N_2 \bowtie R_0| - |N_2 \bowtie N_1 \bowtie R_0| \\
&= |N_2 \bowtie R_0| - |N_2 \bowtie \mathcal{Q}_1| \\
|\mathcal{Q}_1| &= |N_1 \bowtie R_0| \\
&= |N_1 \bowtie R_0|
\end{aligned}$$

That is, each $|\mathcal{Q}_i|$ can be expressed in terms of the i-th relation in the index ordering, together with each of $|R_0|, |\mathcal{Q}_1|, \ldots, |\mathcal{Q}_{i-1}|$. Conceptually, this suggests an iterative algorithm to compute $|\mathcal{Q}_i|$ for each $i = 1, \ldots, 4$, using results obtained for $|\mathcal{Q}_1|, \ldots, |\mathcal{Q}_{i-1}|$. It seems that the counting problem of one $\mathsf{CQ}^\neg$, i.e. $|\mathcal{Q}|$ (note that $\mathcal{Q} = \mathcal{Q}_4$), has been decomposed into only quadratically many sub-problems, thereby overcoming the limitation of the inclusion-exclusion principle. Indeed, this can be generalized to allowing arbitrarily many negative relations.

Corollary 1. *Consider a sequence of hyperedges $V_0, V_1, \ldots, V_m, V_{m+1}$ ($m \geq 0$) with associated relations $S_0, S_1, \ldots, S_m, S_{m+1}$, respectively.*

For each $i \in [m+1]$, define $\mathsf{CQ}^\neg$ $\mathcal{Q}_i := (\{V_0, V_i\}, \{V_1, \ldots, V_{i-1}\})$.

[1] Or equivalently, they *do not survive* R_4. However, non-survival now becomes the desired behavior as R_4 is positive.

Assume every $\mathcal{Q}_i$ is safe. Then for each $i \in [m+1]$,

$$S_i \bowtie S_0 = \mathcal{Q}_i \uplus \biguplus_{j=1}^{i-1} (S_i \bowtie \mathcal{Q}_j)$$

As an immediate consequence of the disjoint union, for each $i \in [m+1]$,

$$|\mathcal{Q}_i| = |S_i \bowtie S_0| - \sum_{j=1}^{i-1} |S_i \bowtie \mathcal{Q}_j| = |S_i \bowtie S_0| - |S_i \bowtie \mathcal{Q}_1| - \cdots - |S_i \bowtie \mathcal{Q}_{i-1}| \qquad (*)$$

Proof. Iteratively applying Lemma 6 with $S_1, S_2, \ldots, S_{i-1}$ on the rightmost $\mathsf{CQ}^\neg$ gives:

$$\begin{aligned}
&S_i \bowtie S_0 \\
&= (S_i \bowtie S_1 \bowtie S_0) \uplus (S_i \triangleright S_1 \bowtie S_0) \\
&= (S_i \bowtie \mathcal{Q}_1) \uplus (S_i \bowtie S_2 \triangleright S_1 \bowtie S_0) \uplus (S_i \triangleright S_2 \triangleright S_1 \bowtie S_0) \\
&= (S_i \bowtie \mathcal{Q}_1) \uplus (S_i \bowtie \mathcal{Q}_2) \uplus \cdots \uplus (S_i \bowtie S_{i-1} \triangleright S_{i-2} \triangleright \cdots \triangleright S_1 \bowtie S_0) \uplus (S_i \triangleright S_{i-1} \triangleright \cdots \triangleright S_1 \bowtie S_0) \\
&= (S_i \bowtie \mathcal{Q}_1) \uplus (S_i \bowtie \mathcal{Q}_2) \uplus \cdots \uplus (S_i \bowtie \mathcal{Q}_{i-1}) \uplus \mathcal{Q}_i
\end{aligned}$$

□

Suppose that we are given a $\mathsf{CQ}^\neg$ with *exactly two* positive hyperedges and m negative hyperedges ($m \geq 0$), where we can find an ordering $(V_0, V_1, \ldots, V_m, V_{m+1})$ on its hyperedges such that (i) V_0, V_{m+1} are the positive hyperedges and $V_1, \ldots, V_m$ are the negative hyperedges; and (ii) all associated $\mathcal{Q}_i$ defined as in Corollary 1 are indeed safe $\mathsf{CQ}^\neg$. Then, ($*$) suggests an iterative procedure to compute $|\mathcal{Q}_i|$ as $|S_i \bowtie S_0| - \sum_{j=1}^{i-1} |S_i \bowtie \mathcal{Q}_j|$, for each $i = 1, 2, \ldots, m+1$. Eventually, $|\mathcal{Q}_{m+1}|$ precisely captures the count of the target $\mathsf{CQ}^\neg$.

Nevertheless, three concerns arise immediately with this approach.

(1) Given $|\mathcal{Q}_j|$ for $1 \leq j < i$, how can we even obtain $|S_i \bowtie \mathcal{Q}_j|$ in the first place?

(2) What if there are more than two[2] positive relations?

(3) Corollary 1 requires a hyperedge ordering, such that every resulted $\mathcal{Q}_i$ is a safe $\mathsf{CQ}^\neg$. How can we find such an ordering?

[2]The counting problem of a safe $\mathsf{CQ}^\neg$ with only one positive relation R_V is trivial: the safety condition implies $U \subseteq V$ for all negative U, hence applying $R_V \leftarrow R_V \triangleright N_U$ reduces the query into having precisely one relation R_V.

We address these concerns one by one. (1) and (2) imply a high-level counting procedure for signed-acyclic $\mathsf{CQ}^{\neg}$ that resembles the Yannakakis algorithm for α-acyclic CQ. Lastly, (3) resolves outstanding technical challenges and concretizes our new counting algorithm.

4.2.1 Incremental Aggregation

First, We demystify how $|S_i \bowtie \mathcal{Q}_j|$ can be obtained from $|\mathcal{Q}_j|$, for each $1 \leq j < i$. The idea is that rather than storing $|\mathcal{Q}_j|$, which is merely a number, we store an *aggregation* on $\mathcal{Q}_j$. Briefly recall the counting variant of the Yannakakis algorithm from Section 3.3, which incrementally aggregates degrees at each join tree node so that the final degrees at the join tree root imply the query count.

We revisit Corollary 1. Its disjoint union in fact implies a more general result involving aggregation on arbitrary attributes W:[3]

$$\bigoplus_{W}(S_i \otimes S_0) = \left(\bigoplus_{W} \mathcal{Q}_i\right) \oplus \left(\bigoplus_{j=1}^{i-1} \bigoplus_{W}(S_i \otimes \mathcal{Q}_j)\right)$$

where in the double-sum notation $\bigoplus\bigoplus$, the outer $\bigoplus$ indicates *summation* among annotated relations, and the inner $\bigoplus$ indicates *aggregation*. Because we use integers as annotations, which admit additive inverse, the equality can be rearranged as

$$\bigoplus_{W} \mathcal{Q}_i = \left(\bigoplus_{W}(S_i \otimes S_0)\right) \ominus \left(\bigoplus_{j=1}^{i-1} \bigoplus_{W}(S_i \otimes \mathcal{Q}_j)\right)$$

As a special case, ($*$) can be recovered by taking $W = \mathrm{var}(\mathcal{Q}_i)$.

Nevertheless, $\bigoplus_W(S_i \otimes S_0)$ and $\bigoplus_W(S_i \otimes \mathcal{Q}_j)$ in the above expressions still need to be handled: how can we efficiently compute these aggregations without actually materializing the join results of $S_i \bowtie S_0$ and $S_i \bowtie \mathcal{Q}_j$? Here, we bring Lemma 4 from Section 3.2 into the picture, which enables us to *combine* two aggregation results into one. In a nutshell, we will find aggregation attributes $W_0, W_1, \ldots, W_m, W_{m+1}$ on each $S_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_m, \mathcal{Q}_{m+1}$, respectively, such that for all $i \in [m+1]$ and $j \in [i-1]$, the aggregations can be decomposed

[3] We regard each relation (query) as naturally annotating its tuples (output tuples) as 1, as discussed in Section 3.2.

into:

$$\bigoplus_{W_i}(S_i \otimes S_0) = \left(\bigoplus_{W_i} S_i\right) \otimes \left(\bigoplus_{W_0} S_0\right)$$

$$\bigoplus_{W_i}(S_i \otimes \mathcal{Q}_j) = \left(\bigoplus_{W_i} S_i\right) \otimes \left(\bigoplus_{W_j} \mathcal{Q}_j\right)$$

We remark on one crucial condition that allows us to combine two aggregations into one. Suppose that we have two aggregations[4] $\bigoplus R$ and $\bigoplus S$, and we want to compute certain aggregation $\bigoplus(R \otimes S)$ via $(\bigoplus R) \otimes (\bigoplus S)$. The crucial condition is that any $v \in \text{var}(R) \cap \text{var}(S)$ *must not* be aggregated in either $\bigoplus R$ or $\bigoplus S$. Conceptually, the new aggregation $\bigoplus(R \otimes S)$ implies certain information on the join $R \bowtie S$; hence, if any join attribute between R and S has been aggregated/eliminated, then the information on how the two join with each other is permanently lost.

We also briefly mention our choice of $W_0, W_1, \ldots, W_{m+1}$ on $S_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_{m+1}$, respectively, which stems from a hyperedge ordering $(V_0, V_1, \ldots, V_{m+1})$. We will choose $W_0 := \text{var}(S_0) - V_0$ and $W_i := \text{var}(\mathcal{Q}_i) - V_i$ for each $i \in [m+1]$; that is, W_i is to aggregate any attributes *not in* V_i. As per Lemma 3, the aggregation $\bigoplus_{W_i} \mathcal{Q}_i$ then stores the degree of all V_i-tuples in $\mathcal{Q}_i$. Because V_i is associated with one input relation, one benefit of such W_i is that any $|\bigoplus_{W_i} \mathcal{Q}_i|$, i.e. the size of any aggregated result, will not exceed IN. Any correctness-related discussion regarding these $W_0, W_1, \ldots, W_{m+1}$ will be postponed to Section 4.3 and 5.

Substituting the claimed equalities then leads to the following for all $i \in [m+1]$:

$$\bigoplus_{W_i} \mathcal{Q}_i = \left[\left(\bigoplus_{W_i} S_i\right) \otimes \left(\bigoplus_{W_0} S_0\right)\right] \ominus \left[\bigoplus_{j=1}^{i-1} \left(\bigoplus_{W_i} S_i\right) \otimes \left(\bigoplus_{W_j} \mathcal{Q}_j\right)\right] \tag{$**$}$$

Similar to $(*)$, $(**)$ implies an iterative procedure to compute $\bigoplus_{W_i} \mathcal{Q}_i$ for each $i = 1, 2, \ldots, m+1$.

4.2.2 Chunks

We have seen how a $\mathsf{CQ}^{\neg}$ with two positive hyperedges can be processed. The idea is to order its hyperedges in a sequence with the first and last hyperedges as positive, while the

[4]Here, and in what follows, all irrelevant aggregation attributes will be omitted to simplify the notations.

other in between as negative. We call such a sequence a *chunk* from now on. When there are more than two positive hyperedges, the high-level idea is to decompose the query into multiple *chunks*, where we process individual chunks with ($**$) and combine their results.

We introduce two possible alignments, *sequential* and *hierarchical*, between two chunks and describe the corresponding strategy to combine their aggregation results.

Sequential Chunks. First, we consider a $\mathsf{CQ}^\neg$ with three positive hyperedges. Let us label its positive and negative hyperedges as $\{V_1, V_2, V_3\}$ and $\{U_{1,m}, \ldots, U_{1,m_1}, U_{2,1}, \ldots, U_{2,m_2}\}$, respectively. Suppose that we can order its hyperedges *sequentially* as

$$V_1, U_{1,1}, \ldots, U_{1,m_1}, V_2, U_{2,1}, \ldots, U_{2,m_2}, V_3$$

where any hypothesis of ($**$) holds on both $(V_1, U_{1,1}, \ldots, U_{1,m_1}, V_2)$ and $(V_2, U_{2,1}, \ldots, U_{2,m_2}, V_3)$. In other words, we are dividing the hyperedges into two *chunks* $C_1 := (V_1, U_{1,1}, \ldots, U_{1,m_1}, V_2)$ and $C_2 := (V_2, U_{2,1}, \ldots, U_{2,m_2}, V_3)$. Notice that the ending hyperedge of C_1, V_2, happens to be the starting hyperedge of C_2; hence, we say that C_1 and C_2 are ordered *sequentially*.

Let us look at the first chunk C_1 and apply ($**$) on it. The end result is an aggregation $\bigoplus \mathcal{Q}_{m_1+1}$, such that $\mathcal{Q}_{m_1+1} = (\{V_1, V_2\}, \{U_{1,1}, \ldots, U_{1,m_1}\})$, which concerns precisely all hyperedges in C_1.

We would then like to apply ($**$) on C_2. Notice that the endpoint V_2 of C_1 now becomes the start point. This corresponds to *substituting the previous* $\bigoplus \mathcal{Q}_{m+1}$ *of* C_1 *for the new* $\bigoplus S_0$ *of* C_2 — recall that we can always regard a query as a (positive) relation; here, we regard the previous $\mathcal{Q}_{m_1+1}$ of C_1 as the new S_0 of C_2. We then apply ($**$) on C_2. At the end, the obtained aggregation $\bigoplus \mathcal{Q}_{m_2+1}$ now concerns hyperedges in *both* C_1 and C_2, which then implies the count of the target $\mathsf{CQ}^\neg$.

We can interpret the procedure as follows. Applying ($**$) on C_1 gives us information on paths that start from V_1, survive all of $N_{1,1}, \ldots, N_{1,m_1}$, and reach V_2; then, the subsequent ($**$) on C_2 extends these paths at V_2 to further reach V_3 while surviving all of $N_{2,1}, \ldots, N_{2,m_2}$.

Hierarchical Chunks. Next, we look at another way of ordering hyperedges. Consider the same $\mathsf{CQ}^\neg$ with three positive hyperedges as above, but this time suppose we can order its hyperedges *hierarchically* as two chunks $C_1 := (V_1, U_{1,m}, \ldots, U_{1,m_1}, V_3)$ and $C_2 := (V_2, U_{2,1}, \ldots, U_{2,m_2}, V_3)$, where both chunks end at the same hyperedge V_3.

As before, suppose we first apply ($**$) on C_1, which gives an aggregation $\bigoplus \mathcal{Q}_{m_1+1}$ for $\mathcal{Q}_{m_1+1} = (\{V_1, V_3\}, \{U_{1,1}, \ldots, U_{1,m_1}\})$. Now, how can we apply ($**$) on C_2? We apply the same trick as before — regarding this $\mathcal{Q}_{m_1+1}$ as a (positive) relation in ($**$) for C_2. Notice

that $C_2 = (V_2, U_{2,1}, \ldots, U_{2,m_2}, V_3)$ shares the same end point V_3 as C_1, so now we *substitute the previous* $\bigoplus \mathcal{Q}_{m_1+1}$ *of* C_1 *for the new* $\bigoplus S_{m_2+1}$ *of* C_2 and apply $(**)$ on C_2. Likewise, the aggregation $\bigoplus \mathcal{Q}_{m_2+1}$ obtained at the end concerns both C_1 and C_2.

Generalization. We briefly mention how these aggregation strategies can be generalized to arbitrarily many chunks, or equivalently, arbitrary many positive hyperedges. We assume these positive hyperedges can be ordered in a rooted tree (the exact data structure will be given in Section 4.3), in which any child-parent edge (V, W) corresponds to a chunk $C_{V,W}$ that starts at V and ends at W.

The high-level aggregation procedure works as follows. We maintain a certain aggregation $\bigoplus S_V$ at each node V, and traverse the tree bottom-up. For each visited child-parent edge (V, W), we (i) apply $(**)$ on the chunk $C_{V,W}$ with $\bigoplus S_V$ *replacing* $\bigoplus S_0$ *and* $\bigoplus S_W$ *replacing* $\bigoplus S_{m+1}$; and (ii) *update* $\bigoplus S_W$ *by the result of* $\bigoplus \mathcal{Q}_{m+1}$ *from* $(**)$.

One may notice that this procedure resembles the counting variant of the Yannakakis algorithm from Section 3.3, which operates on a join tree with its nodes being positive hyperedges (recall that the Yannakakis algorithm is for CQ, i.e. there is no negative hyperedge). The algorithm maintains a degree aggregation $\bigoplus S_V$ at each node V and traverses the join tree bottom-up, where for each visited child-parent edge (V, W), it updates $\bigoplus S_W \leftarrow (\bigoplus S_W) \otimes (\bigoplus S_V)$. The only place our described procedure differs from the Yannakakis algorithm is that we apply a more complex update formula $(**)$ to take into consideration the effects of the negative hyperedges in between.

4.2.3 Chunk Construction: A Simple Scenario

We have drawn a close connection between our proposed procedure and the Yannakakis algorithm — both operating on a rooted tree with nodes being hyperedges. For the Yannakakis algorithm, such a tree is called a join tree, and we have seen in Section 3.1 that a join tree can be constructed following an α-elimination sequence. Hence, it should come as no surprise that our chunk construction follows a signed-elimination sequence. Like Algorithm 1, which combines join tree construction and degree aggregation in the Yannakakis algorithm, our goal is to combine chunk construction and $(**)$.

One of the conditions that a chunk must satisfy, for $(**)$ to hold, is that all its associated $\mathcal{Q}_i$ defined as in $(**)$ are indeed safe $\mathsf{CQ}^{\neg}$. We now establish a simple scenario and see how the safety condition can be satisfied using properties of a signed-leaf.

Recall that an attribute v is called a signed-leaf of a $\mathsf{CQ}^{\neg}$, if there exists a positive hyperedge V, called the pivot of v, such that (i) for all other positive hyperedge such that

$v \in U$, we have $U \subseteq V$; and (ii) all negative hyperedges U for which $U \not\subseteq V$ can be sorted as $U_1, U_2, \ldots, U_m$, such that $V \subsetneq U_1 \subseteq U_2 \subseteq \cdots \subseteq U_m$.

Now suppose v is the first signed-leaf that we eliminate from the $\mathsf{CQ}^{\neg}$. For any other positive/negative hyperedge U such that $U \subseteq V$, U can be removed from the query (after updating R_V with U's associated relation) without affecting the query result; therefore, we may assume that no such U exists. Then, we construct the first chunk *partially* as $C_1 = (V, U_1, U_2, \ldots, U_m)$. C_1 is considered incomplete, for it does not contain an ending positive hyperedge. Observe that the $\mathsf{CQ}^{\neg}$ $\mathcal{Q}_i := (\{U_i, V\}, \{U_1, \ldots, U_{i-1}\})$ at each chunk position i is indeed safe, because $U_1 \subseteq U_2 \subseteq \cdots \subseteq U_i$.

After eliminating v from the query, suppose now we have a signed-leaf w with pivot W, and that $U_m - \{v\} \subseteq W$. Recall that in Reduce (Algorithm 3), $U_m - \{v\}$ will be eliminated in the α-step of w using W. This corresponds to *adding* W to the end of C_1. Now $C_1 = (V, U_1, U_2, \ldots, U_m, W)$, which is a complete chunk as its ending hyperedge is positive. Furthermore, the new $\mathcal{Q}_{m+1} := (\{W, V\}, \{U_1, U_2, \ldots, U_m\})$ is a safe $\mathsf{CQ}^{\neg}$ because $V \subseteq U_1 \subseteq U_2 \subseteq \cdots \subseteq U_m \subseteq W \cup \{v\} = W \cup V$.

Example 9. *Consider the same $\mathsf{CQ}^{\neg}$ as Example 8 with positive hyperedges $W = \{A, B, C\}$, $V = \{C, D\}$ and negative hyperedges $U_1 = \{B, C, D\}, U_2 = \{A, B, C, D\}$, with the same database instance as Figure 4.1. We illustrate the aforementioned chunk construction strategy by eliminating D first. Figure 4.2 displays the aggregation updates that occur while the chunk is updated.*

The elimination of signed-leaf D sees the linear $\subseteq$-chain $V \subsetneq U_1 \subseteq U_2$ and constructs an incomplete chunk (V, U_1, U_2). The associated $\mathsf{CQ}^{\neg}$ are $\mathcal{Q}_1 := N_{U_1} \bowtie R_V$, $\mathcal{Q}_2 := N_{U_2} \triangleright N_{U_1} \bowtie R_V$.

Let S_i be the relation associated with each chunk position i. Then, the goal is to aggregate the degree of each $t \in S_i$ in $\mathcal{Q}_i$. For a shorthand in notation, we denote $S_i^{\oplus}$ the aggregated degrees. At the beginning, each $S_i^{\oplus}$ is initialized by annotating any $t \in S_i$ with 1.

For $\mathcal{Q}_1$, this is done by computing $N_{U_1} \otimes R_V^{\oplus}$. Notice that the tuple $(b_2, c_3, d_3) \in N_{U_1}$, previously considered redundant because it does not join with anything in R_V, is assigned a weight 0.

For $\mathcal{Q}_2$ and any $t \in N_{U_2}$, $(N_{U_2} \otimes R_V^{\oplus})(t)$ captures its extensions to R_V, and $(N_{U_2} \otimes N_{U_1}^{\otimes})(t)$ captures its extensions to R_V that do not survive N_{U_1}. The difference between the two reflects its valid extensions to R_V that survive N_{U_1}. Notice that the tuples (a_1, b_1, c_1, d_1) and (a_2, b_2, c_4, d_4), which were both identified as redundant by Reduce *previously, are assigned a weight 0.*

$N^{\oplus}_{U_1} \leftarrow N^{\oplus}_{U_1} \otimes R^{\oplus}_{V}$

B	C	D	$\cdot$
b_1	c_1	d_1	1
b_1	c_1	d_2	1
b_1	c_2	d_1	1
b_2	c_3	d_3	0

(a) U_1

$T_1 := N^{\oplus}_{U_2} \otimes R^{\oplus}_{V}$

A	B	C	D	$\cdot$
a_1	b_1	c_1	d_1	1
a_1	b_1	c_2	d_2	1
a_1	b_1	c_3	d_1	1
a_2	b_2	c_4	d_4	0

$T_2 := N^{\oplus}_{U_2} \otimes N^{\oplus}_{U_1}$

A	B	C	D	$\cdot$
a_1	b_1	c_1	d_1	1
a_1	b_1	c_2	d_2	0
a_1	b_1	c_3	d_1	0
a_2	b_2	c_4	d_4	0

$N^{\oplus}_{U_2} \leftarrow T_1 \ominus T_2$

A	B	C	D	$\cdot$
a_1	b_1	c_1	d_1	$1-1=0$
a_1	b_1	c_2	d_2	$1-0=1$
a_1	b_1	c_3	d_1	$1-0=1$
a_2	b_2	c_4	d_4	$0-0=0$

(b) U_2

$T_3 := R^{\oplus}_{W} \otimes (\bigoplus_D R^{\oplus}_{V})$

A	B	C	$\cdot$
a_1	b_1	c_1	2
a_1	b_1	c_2	2
a_1	b_1	c_3	2

$T_4 := R^{\oplus}_{W} \otimes (\bigoplus_D N^{\oplus}_{U_1})$

A	B	C	$\cdot$
a_1	b_1	c_1	2
a_1	b_1	c_2	1
a_1	b_1	c_3	0

$T_5 := R^{\oplus}_{W} \otimes (\bigoplus_D N^{\oplus}_{U_2})$

A	B	C	$\cdot$
a_1	b_1	c_1	0
a_1	b_1	c_2	1
a_1	b_1	c_3	1

$R^{\oplus}_{W} \leftarrow T_3 \ominus T_4 \ominus T_5$

A	B	C	$\cdot$
a_1	b_1	c_1	$2-2-0=0$
a_1	b_1	c_2	$2-1-1=0$
a_1	b_1	c_3	$2-0-1=1$

(c) U_3

Figure 4.2: Aggregation updates that happen while chunk (V, U_1, U_2, W) is constructed.

This completes the elimination of D. All remaining attributes A, B, C are valid signed-leaves with the same pivot W. Because $U_2 - \{D\} \subseteq W$, the chunk (V, U_1, U_2) is completed by adding W to the end. Now, we update the aggregation for $\mathcal{Q}_3 := R_W \triangleright N_{U_2} \triangleright N_{U_1} \bowtie R_V$. For any $t \in R_W$, the depicted T_3, T_4, T_5 captures its extensions to R_V, its extensions to R_V that do not survive N_{U_1}, and its extensions to R_V that survive N_{U_1} but do not survive N_{U_2}, respectively. Subtracting the other two from the first, we obtain its extensions to R_V surviving both N_{U_1} and N_{U_2}, as desired. One can see that the aggregation result $R^{\oplus}_{W}$ for $\mathcal{Q}_3$ reflects precisely the count of the target $\mathsf{CQ}^{\neg}$ $(\{W, V\}, \{U_1, U_2\})$, whose result set is shown in Figure 4.1a.

4.2.4 Chunk Construction: A Difficult Scenario

Example 9 seems to suggest that chunk construction can be done analogously as the α-step and β-step in Reduce: at each elimination step with a linear $\subseteq$-chain $V \subsetneq U_1 \subseteq \cdots \subseteq U_m$, (i) α-step *completes* any incomplete chunk ending at some other hyperedge U such that $U \subseteq V$, by adding V to that chunk; then, (ii) β-step *creates* a new incomplete chunk $(V, U_1, \ldots, U_m)$; finally, (iii) $(**)$ is used to ensure all aggregation results are maintained while chunks are updated. Let us consider another seemingly simple example, which in fact reveals many technical challenges with this procedure.

Example 10. *Consider a $\mathsf{CQ}^{\neg}$ $\mathcal{Q} = (\{V_1, V_2, V_3\}, \{U_1\})$ with positive hyperedges $V_1 = \{A, B\}, V_2 = \{B, C\}, V_3 = \{C, D\}$ and negative hyperedge $U_1 = \{A, B, C, D\}$. It can be easily verified that $\sigma = ABCD$ is a signed-elimination sequence of $\mathcal{Q}$.*

We consider the chunk construction that occurs when we follow σ, based on the α- and β-step described above:

- *A has pivot V_1. α-step does nothing (as there are no incomplete chunks ending at some other U for which $U \subseteq V_1$). β-step sees $V_1 \subsetneq U_1$ and creates an incomplete chunk $C_1 = (V_1, U_1)$.*
- *Now $V_1 = \{B\}, V_2 = \{B, C\}, V_3 = \{C, D\}, U_1 = \{B, C, D\}$.*
  B has pivot V_2. α-step does nothing (even thought $V_1 \subseteq V_2$, there is no incomplete chunk ending at V_1). β-step sees $V_2 \subsetneq U_2$ and creates an incomplete chunk $C_2 = (V_2, U_1)$.
- *Now $V_2 = \{C\}, V_3 = \{C, D\}, U_1 = \{C, D\}$.*
  C has pivot V_3. α-step sees $U_1 \subseteq V_3$ and adds V_3 to both C_1 and C_2. β-step does nothing.

At the end, we have two complete chunks $C_1 = (V_1, U_1, V_3)$ and $C_2 = (V_2, U_1, V_3)$.

Example 10 immediately raises an issue: so far, we have only seen chunks that do not share any negative hyperedge (i.e. sequential and hierarchical alignments); nevertheless, U_1 is shared by both C_1 and C_2. How can we apply ($**$)?

Let us revisit the signed-elimination step of B in Example 10. After the β-step, we have two incomplete chunks $C_1 = (V_1, U_1)$ and $C_2 = (V_2, U_1)$ that share the same ending U_1. According to our terminology in Section 4.2, C_1 and C_2 are ordered *hierarchically*, which we already have a way to compute an aggregation at U_1 that concerns both incomplete chunks — (i) apply ($**$) on C_1 to obtain an aggregation $\bigoplus \mathcal{Q}_1$ where $\mathcal{Q}_1 := (\{U_1, V_1\}, \emptyset)$; then (ii) substitute $\bigoplus \mathcal{Q}_1$ for $\bigoplus S_1$ in ($**$) for C_2 and obtain another aggregation $\bigoplus \mathcal{Q}_1$ (with a notation overload) where now $\mathcal{Q}_1 := (\{U_1, V_1, V_2\}, \emptyset)$.

The real challenge arises in the signed-elimination of C in Example 10. When V_3 is added to the end of both C_1 and C_2, how can we apply ($**$)? Conceptually, ($**$) is a formula that allows us to aggregate from one positive hyperedge to another positive hyperedge along a sequence of negative hyperedges. Since $C_1 = (V_1, U_1, V_3)$ and $C_2 = (V_2, U_1, V_3)$ both contain the negative hyperedge U_1, if we treat C_1 and C_2 as ordered hierarchically and apply ($**$) with our aforementioned strategy, then the effect of U_1 will be double-counted, ultimately rendering the aggregation result at V_3 incorrect.

Recursive aggregation. We propose a *recursive* solution to compute the aggregation. First, let us re-formulate Corollary 1 in a recursive manner. The proof directly follows from Lemma 6 and is thus omitted.

Corollary 2. *For any chunk $C_m := (V_0, V_1, \ldots, V_m)$ $(m \geq 0)$ with associated relations $S_0, S_1, \ldots, S_m$, respectively, define* $\mathsf{CQ}^\neg$ *(in hypergraph representation)*

$$\mathsf{ChunkCQ}^\neg(C_m) := \begin{cases} (\{V_0\}, \emptyset) & \text{if } m = 0 \\ (\{V_0, V_m\}, \{V_1, \ldots, V_{m-1}\}) & \text{if } m \geq 1 \end{cases}$$

Let $m \geq 1$ and define a new chunk $C'_{m-1} := (V_0, V_1, \ldots, V_{m-2}, V_m)$. It then follows that

$$\mathsf{ChunkCQ}^\neg(C'_{m-1}) = \mathsf{ChunkCQ}^\neg(C_m) \uplus (S_m \bowtie \mathsf{ChunkCQ}^\neg(C_{m-1}))$$

(where $C_{m-1} := (V_0, V_1, \ldots, V_{m-1})$) which then implies

$$|\mathsf{ChunkCQ}^\neg(C_m)| = |\mathsf{ChunkCQ}^\neg(C'_{m-1})| - |S_m \bowtie \mathsf{ChunkCQ}^\neg(C_{m-1})| \qquad (***)$$

Given a chunk $C_m = (V_0, V_1, \ldots, V_m)$, Corollary 2 defines $\mathsf{ChunkCQ}^\neg(C_m)$ to be the $\mathsf{CQ}^\neg$ concerning all hyperedges in C_m, which is no different from our prior settings. Our interest is in computing the count $|\mathsf{ChunkCQ}^\neg(C_m)|$. Then, $(***)$ suggests that the problem can be *recursively* decomposed into $|\mathsf{ChunkCQ}^\neg(C'_{m-1})|$ and $|S_m \bowtie \mathsf{ChunkCQ}^\neg(C_{m-1})|$, where both C'_{m-1} and C_{m-1} are chunks with one hyperedge less than C_m.

The chunk C_m is constructed by adding V_m to the end of $C_{m-1} = (V_0, V_1, \ldots, V_{m-1})$; hence, the sub-problem $|\mathsf{ChunkCQ}^\neg(C_{m-1})|$ should have been solved at this point. Section 4.2.1 then entails how we can *combine* its result with S_m to obtain $|S_m \bowtie \mathsf{ChunkCQ}^\neg(C_{m-1})|$.

Nevertheless, $C'_{m-1} = (V_0, V_1, \ldots, V_{m-2}, V_m)$ is a chunk that we have never seen before — how can we handle it? The base case is when $m = 1$, where the chunk $C'_{m-1} = (V_0, V_m)$ consists of precisely two positive hyperedges and no negative hyperedges, and $|\mathsf{ChunkCQ}^\neg(C'_{m-1})| = |S_0 \bowtie S_m|$. Section 4.2.1 again shows how this can be computed.

Now, we look at $m \geq 2$. Let us apply $(***)$ again on it to obtain

$$|\mathsf{ChunkCQ}^\neg(C'_{m-1})| = |\mathsf{ChunkCQ}^\neg(C''_{m-2})| - |S_m \bowtie \mathsf{ChunkCQ}^\neg(C_{m-2})|$$

where $C''_{m-2} = (V_0, V_1, \ldots, V_{m-3}, V_m)$ and $C_{m-2} = (V_0, V_1, \ldots, V_{m-2})$. That is, the sub-problems now are on chunks with two hyperedges less than C_m. As C_{m-2} is also a chunk encountered during the construction of C_m, we only need to solve the sub-problem on this new chunk C''_{m-2}. One may immediately observe that recursively unraveling these sub-problems indeed leads to $(*)$.

That is, in order to solve the aggregation problem on $C_m = (V_0, V_1, \ldots, V_{m-1}, V_m)$, we *take away the second last hyperedge* V_{m-1} from it and recursively solve the aggregation problem on the resulting chunk. Let us now revisit Example 10 with this recursive idea.

We have $C_1 = (V_1, U_1, V_3)$ and $C_2 = (V_2, U_1, V_3)$, where we already have an aggregation $\bigoplus(N_{U_1} \bowtie R_{V_1} \bowtie R_{V_2})$ at U_1. Our goal is to compute an aggregation at V_3 that concerns all hyperedges in C_1 and C_2. As our first step, we combine V_3 with the aggregation result at U_1 to obtain $\bigoplus(R_{V_3} \bowtie N_{U_1} \bowtie R_{V_1} \bowtie R_{V_2})$. Then, we *take away* U_1 *from* C_1 *and* C_2 to obtain $C_1' := (V_1, V_3), C_2' := (V_2, V_3)$, and solve the aggregation problem at V_3 on C_1' and C_2'. Notice that C_1' and C_2' are ordered *hierarchically*, which we know how to handle — now, we get an aggregation $\bigoplus(R_{V_3} \bowtie R_{V_1} \bowtie R_{V_2})$ from C_1' and C_2'. Finally,

$$\bigoplus(R_{V_3} \bowtie R_{V_1} \bowtie R_{V_2}) - \bigoplus(R_{V_3} \bowtie N_{U_1} \bowtie R_{V_1} \bowtie R_{V_2}) = \bigoplus(R_{V_3} \triangleright N_{U_1} \bowtie R_{V_1} \bowtie R_{V_2})$$

which gives the desired aggregation at V_3.

In a nutshell, the recursive aggregation at a hyperedge V works as follows, which occurs when we add V *on top of* some other U_m (i.e. appending V to every chunk that ends at U_m). The base case occurs when U_m is positive. Otherwise, for every current chunk of the form $C := (V_0, U_1, \ldots, U_{m-1}, U_m, V)$ (i.e. these were the chunks ending at U_m before we appended V), we *temporarily* create a chunk $C' := (V_0, U_1, \ldots, U_{m-1}, V)$ by removing U_m from C, and recursively aggregate at V on these temporary chunks. This is one of the core components of our new counting algorithm, and it is implemented in Algorithm 4.

Recursive chunks merging. Albeit clean, the above recursive aggregation does not work on the given chunk instances in Example 10. The correctness is violated when we try to aggregate on the hierarchically ordered chunks $C_1' := (V_1, V_3)$ and $C_2' := (V_2, V_3)$. Concretely, let W_3 be the aggregation attributes for V_3, where W_3 is to aggregate any attribute *not in* V_3 — this time, the aggregation attributes become relevant.

Suppose C_1' is aggregated first (the case where C_2' is aggregated first results in a similar issue). Then, this gives an aggregation $\bigoplus_{W_3}(R_{V_1} \bowtie R_{V_3})$, which is then substituted into (**) on C_2'. In a nutshell, (**) tries to compute an aggregation $\bigoplus_{W_3}(R_{V_1} \bowtie R_{V_2} \bowtie R_{V_3})$ via $\bigoplus_{W_3}(R_{V_1} \bowtie R_{V_3})$ and R_{V_2}. We briefly recall our discussion from Section 4.2.1: to combine two aggregation results into one, any join attribute must not be aggregated. Notice that $V_1 = \{A, B\}, V_2 = \{B, C\}, V_3 = \{C, D\}$, so $R_{V_1} \bowtie R_{V_3}$ and R_{V_2} join on attributes B, C. Since W_3 aggregates away any attribute not in V_3, and $B \notin V_3$ in particular, there is no way we can combine $\bigoplus_{W_3}(R_{V_1} \bowtie R_{V_3})$ and R_{V_2} to get $\bigoplus_{W_3}(R_{V_1} \bowtie R_{V_2} \bowtie R_{V_3})$.

The issue lies in the α-step while we eliminate B in Example 10. If we were to apply Reduce to eliminate B, because here $V_1 = \{B\}, V_2 = \{B, C\}$ and V_2 is the pivot of B, Reduce would eliminate V_1 using V_2 in its α-step. Nevertheless, our current chunk construction strategy does not capture such interaction between V_1 and V_2 — ideally, we want to *place* V_1 *underneath* V_2 *and aggregate from* V_1 *to* V_2 *to capture an aggregation* $\bigoplus(R_{V_2} \bowtie R_{V_1})$.

This inspires us to modify the α-, β-step design into a recursive procedure — the idea is that after we create the chunks $C_1 = (V_1, U_1), C_2 = (V_2, U_1)$ and do the aggregation at U_1, we need to continue merging the sub-chunks *beneath* U_1. We briefly describe how the elimination of B in Example 10 is modified. Initially, we see both an incomplete chunk $C_1 = (V_1, U_1)$ with end U_1, and the pivot V_2. Because $V_2 \subsetneq U_1$, we (i) create a new incomplete chunk $C_2 = (V_2, U_1)$ and aggregate at U_1 (on C_1, C_2). Next, we look at the sub-chunks $C_1' = (V_1), C_2' = (V_2)$ beneath U_1. Because V_2 is the pivot and $V_1 \subseteq V_2$, we (ii) create a new complete chunk $C_3 = (V_1, V_2)$ and aggregate at V_2; finally, we connect the end of C_3 to U_1 by (iii) creating a new incomplete chunk $C_4 = (V_2, U_1)$. We *do not* aggregate at U_1 again, as we already have an aggregation at U_1 that concerns all hyperedges underneath it after step (i). In the end, the remaining chunks are $C_3 = (V_1, V_2)$ and $C_4 = (V_2, U_1)$, and we have obtained the associated aggregation at both V_2 and U_1.

Now, we can unveil another core component of our new counting algorithm — a recursive chunk merging procedure. For technical reason, we consider every standalone positive hyperedge V as an incomplete chunk (V), and every standalone negative hyperedge U as an incomplete chunk $(\emptyset, U)$, where $\emptyset$ associates to a special positive relation $R_\emptyset = \{()\}$ containing the *empty tuple* — this is so that joining with $R_\emptyset$ has no impact to the join result.

At each elimination step of a signed-leaf v with a linear $\subseteq$-chain $V \subsetneq U_1 \subseteq \cdots \subseteq U_m$, the goal is to ($\beta$-step) create one incomplete chunk $(V, U_1, \ldots, U_m)$; and (α-step) use V to complete any other incomplete chunk that contains the attribute v. At each recursive execution on a given set of incomplete chunks, we (i) find those incomplete chunks $\{C_i\}_{i=1}^m$ ending at $\{V_i\}_{i=1}^m$, respectively, such that $v \in V_i$; (ii) identify the maximum $V \in \{V_i\}_{i=1}^m$ with respect to $\subseteq$; (iii) add V to the end of all other chunks $\{C_i : V \neq V_i\}$; and (iv) if V is negative, recursively merge all sub-chunks ending at V (otherwise V is the pivot of v, and recursive merging can stop). Conceptually, the recursion visits the chain $V \subsetneq U_1 \subseteq \cdots \subseteq U_m$ in reverse order. The implementation can be found in Algorithm 5.

Example 11. *Consider the same* $\mathsf{CQ}^{\neg}$ *and signed-elimination sequence* $\sigma = ABCD$ *in Example 10. We demonstrate such recursive chunk construction strategy on* σ*, where Figure 4.3 shows the accompanying aggregation with an example database instance:*

- *Hyperedges:* $V_1 = \{A, B\}, V_2 = \{B, C\}, V_3 = \{C, D\}, U_1 = \{A, B, C, D\}$.
  Incomplete Chunks: $C_1 := (V_1), C_2 := (V_2), C_3 := (V_3), C_4 := (\emptyset, U_1)$.
  We eliminate signed-leaf A. $V_1 \subsetneq U_1$*, so we add* U_1 *to* C_1 *and now* $C_1 = (V_1, U_1)$.
  Since U_1 *is now on top of some actual positive hyperedge, we remove* C_4*, which was here just for technical reasons. Now, there is only one chunk ending at* U_1*, so recursive merging stops at* U_1.

R_{V_1}

A	B
a_1	b_1
a_1	b_2
a_2	b_1
a_2	b_2

R_{V_2}

B	C
b_1	c_1
b_1	c_2
b_2	c_1
b_2	c_2

R_{V_3}

C	D
c_1	d_1
c_1	d_2
c_2	d_1
c_2	d_2

$\mathcal{Q}$

A	B	C	D
a_1	b_1	c_2	d_1
a_1	b_1	c_2	d_2
a_1	b_2	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_1	c_1	d_2
a_2	b_1	c_2	d_1
a_2	b_1	c_2	d_2
a_2	b_2	c_1	d_1
a_2	b_2	c_1	d_2
a_2	b_2	c_2	d_1
a_2	b_2	c_2	d_2

N_{U_1}

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_1	b_2	c_1	d_2
a_1	b_2	c_2	d_1
a_2	b_1	c_1	d_1
a_1	b_1	c_1	d_3
a_3	b_1	c_1	d_1

(a) Database instance and query result.

$N^{\oplus}_{U_1} \leftarrow N^{\oplus}_{U_1} \otimes R^{\oplus}_{V_1}$

A	B	C	D	$\cdot$
a_1	b_1	c_1	d_1	1
a_1	b_1	c_1	d_2	1
a_1	b_2	c_1	d_2	1
a_1	b_2	c_2	d_1	1
a_2	b_1	c_1	d_1	1
a_1	b_1	c_1	d_3	1
a_3	b_1	c_1	d_1	0

(b) U_1 on $C_1 = (V_1, U_1)$.

$N^{\oplus}_{U_1} \leftarrow N^{\oplus}_{U_1} \otimes R^{\oplus}_{V_2}$

A	B	C	D	$\cdot$
a_1	b_1	c_1	d_1	1
a_1	b_1	c_1	d_2	1
a_1	b_2	c_1	d_2	1
a_1	b_2	c_2	d_1	1
a_2	b_1	c_1	d_1	1
a_1	b_1	c_1	d_3	1
a_3	b_1	c_1	d_1	0

(c) U_1 on $C_2 = (V_2, U_1)$.

$R^{\oplus}_{V_2} \leftarrow R^{\oplus}_{V_2} \otimes (\bigoplus_A R^{\oplus}_{V_1})$

B	C	$\cdot$
b_1	c_1	2
b_1	c_2	2
b_2	c_1	2
b_2	c_2	2

(d) V_2 on $C_1 = (V_1, V_2)$.

$T_1 := R^{\oplus}_{V_3} \otimes (\bigoplus_B R^{\oplus}_{V_2})$

C	D	$\cdot$
c_1	d_1	4
c_1	d_2	4
c_2	d_1	4
c_2	d_2	4

$T_2 := R^{\oplus}_{V_3} \otimes (\bigoplus_{AB} N^{\oplus}_{U_1})$

C	D	$\cdot$
c_1	d_1	2
c_1	d_2	2
c_2	d_1	1
c_2	d_2	0

$R^{\oplus}_{V_3} \leftarrow T_1 \ominus T_2$

C	D	$\cdot$
c_1	d_1	$4 - 2 = 2$
c_1	d_2	$4 - 2 = 2$
c_2	d_1	$4 - 1 = 3$
c_2	d_2	$4 - 0 = 4$

(e) V_3 on $C_2 = (V_2, U_1, V_3)$.

Figure 4.3: Aggregation updates accompanying each chunk update in Example 11.

- *Hyperedges:* $V_1 = \{B\}, V_2 = \{B, C\}, V_3 = \{C, D\}, U_1 = \{B, C, D\}$.
  Incomplete Chunks: $C_1 = (V_1, U_1), C_2 = (V_2), C_3 = (V_3)$.
  We eliminate signed-leaf B. $V_2 \subsetneq U_1$, *so we add* U_1 *to* C_2 *and now* $C_2 = (V_2, U_1)$.
  Then, we merge the sub-chunks $C_1' := (V_1), C_2' := (V_2)$ *beneath* U_1. $V_1 \subseteq V_2$, *so we replace* U_1 *by* V_2 *in* C_1 *and now* $C_1 = (V_1, V_2)$, *which becomes complete and is excluded from consideration.*
  Since V_2 *is positive, recursive merging stops at* V_2.

- *Hyperedges:* $V_2 = \{C\}, V_3 = \{C, D\}, U_1 = \{C, D\}$.
  Incomplete Chunks: $C_2 = (V_2, U_1), C_3 = (V_3)$.
  We eliminate signed-leaf C. $U_1 \subseteq V_3$, *so we add* V_3 *to* C_2 *and now* $C_2 = (V_2, U_1, V_3)$, *which becomes complete.*
  Since V_3 *is positive, recursive merging stops proceed underneath* V_3.

- *Hyperedges:* $V_3 = \{D\}, U_1 = \{D\}$. *Incomplete Chunks:* $C_3 = (V_3)$.
  We eliminate signed-leaf D. *As there is only one remaining incomplete chunk, we do nothing.*

At the end, we have complete chunks $C_1 = (V_1, V_2), C_2 = (V_2, U_1, V_3)$ *and an incomplete*

chunk $C_3 = (V_3)$. The technical reason to have an incomplete C_3 at the end is that if there were additional hyperedges to be processed, we could keep building on top of C_3. One may verify in Figure 4.3 that the final aggregation at V_3, i.e. $R^{\oplus}_{V_3}$, implies $|\mathcal{Q}|$. More specifically, $\bigoplus_{AB} \mathcal{Q} = R^{\oplus}_{V_3}$.

4.3 The Complete Algorithm

In Section 4.2.2, we mentioned a rooted tree structure among positive hyperedges in which each child-parent edge (V, W) embeds a chunk $C_{V,W}$ that starts at V and ends at W. Now, we formalize such a data structure as *hyperedge dependency tree* (HDT). Intrinsically, an HDT is a rooted tree with its nodes being hyperedges, mimicking the notion of join trees in the context of α-acyclic CQ.

Definition 10. *Let $\mathsf{Q} = (\mathcal{E}^+, \mathcal{E}^-)$ be a $\mathsf{CQ}^\neg$.*

A hyperedge dependency tree (abbr. HDT) $\mathcal{T}$ is a rooted tree with nodes $N(\mathcal{T}) \subseteq \mathcal{E}^+ \sqcup \mathcal{E}^-$.

Let V be the root of $\mathcal{T}$ and W be any node in $\mathcal{T}$.

We call a path-to-W in $\mathcal{T}$ $P = (U_0, U_1, \ldots, U_m, W)$ a W-dependency path, if (i) U_0 is a descendant of W; (ii) $U_0 \in \mathcal{E}^+$; and (iii) $U_1, \ldots, U_m \in \mathcal{E}^-$.

We call $\mathcal{Q}_\mathcal{T} := (N(\mathcal{T}) \cap \mathcal{E}^+ \cup \{V\}, N(\mathcal{T}) \cap \mathcal{E}^- - \{V\})$ the $\mathsf{CQ}^\neg$ induced by $\mathcal{T}$.

HDT is a compact way to embed the chunks constructed throughout our algorithm. For any positive node W in an HDT, any W-dependency path corresponds to a *complete* chunk; on the other hand, if an HDT has a negative root U, then any U-dependency path corresponds to an *incomplete* chunk. Thus, HDT enables an efficient retrieval of all incomplete (sub)-chunks ending at any hyperedge, which is useful in both the recursive aggregation and the recursive chunks merging described in Section 4.2.4.

Meanwhile, the notion of induced $\mathsf{CQ}^\neg$ $\mathcal{Q}_\mathcal{T}$ is defined analogously to $\mathcal{Q}_i$ from Corollary 1 — the root V of $\mathcal{T}$ is regarded as positive in $\mathcal{Q}_\mathcal{T}$, mimicking that V_i is regarded as positive in $\mathcal{Q}_i$.

Example 12. *We consider the same $\mathsf{CQ}^\neg$ from Example 11. This time, we express all its associated chunk construction in the HDT terminology.*

- *Before: initially, incomplete chunks were $C_1 = (V_1), C_2 = (V_2), C_3 = (V_3), C_4 = (\emptyset, U_1)$.*
  Now: initialize four single-node HDT, one for each of V_1, V_2, V_3, U_1.

- *Before: when eliminating signed-leaf A, we added U_1 to C_1 to update $C_1 = (V_1, U_1)$. Now: set U_1 as the parent of V_1.*

- *Before: when eliminating signed-leaf B, we first added U_1 to C_2 to update $C_2 = (V_2, U_1)$, then replaced U_1 by V_2 in C_1 to update $C_1 = (V_1, V_2)$.
Now: first set U_1 as the parent of V_2, then update V_2 as the parent of V_1.*

- *Before: when eliminating signed-leaf C, we added V_3 to C_2 to update $C_2 = (V_2, U_1, V_3)$. Now: set V_3 as the parent of U_1.*

- *Before: in the end, we had complete chunks $C_1 = (V_1, V_2), C_2 = (V_2, U_1, V_3)$.
Now: we have an HDT $\mathcal{T}$ with child-parent edges $(V_1, V_2), (V_2, U_1), (U_1, V_3)$.*

The induced $\mathsf{CQ}^\neg$ by every sub-tree of $\mathcal{T}$ are $\mathcal{Q}_{\mathcal{T}_{V_3}} := (\{V_3, V_2, V_1\}, \{U_1\})$, $\mathcal{Q}_{\mathcal{T}_{U_1}} := (\{U_1, V_2, V_1\}, \emptyset)$, $\mathcal{Q}_{\mathcal{T}_{V_2}} := (\{V_2, V_1\}, \emptyset)$, and $\mathcal{Q}_{\mathcal{T}_{V_1}} := (\{V_1\}, \emptyset)$. These precisely match the associated $\mathcal{Q}_i$ at each hyperedge in the complete chunks $C_1 = (V_1, V_2), C_2 = (V_2, U_1, V_3)$.

The recursive aggregation and the recursive chunks merging from Section 4.2.4 are presented as Algorithm 4 (AggregateHDT) and Algorithm 5 (MergeHDT), respectively, using the HDT terminology. Similar to Algorithm 1 from Section 3.3, we explicitly keep track of the attributes eliminated (V_{elim}) and use $\subseteq_{V_{\text{elim}}}$ in place of $\subseteq$.

AggregateHDT computes the updated aggregation at hyperedge V when we add V *on top of* some hyperedge U, where the input HDT $\mathcal{T}$ compactly stores all chunks currently ending at U. When U is a negative hyperedge, we *temporarily* replace U by V in these chunks and aggregate at V on them. Since these chunks are ordered hierarchically, we (recursively) aggregate them one-by-one and use $\otimes$ to combine their results. Concretely, when U is a negative hyperedge, AggregateHDT visits all U-dependency paths.

MergeHDT aims at merging all incomplete chunks ending at some hyperedge that contains the signed-leaf v into one incomplete chunk. At each recursive step, we identify the maximum ending hyperedge V with respect to $\subseteq_{V_{\text{elim}}}$, and V is added to all other chunks. When V is positive, then we know that V is the pivot of v, hence the recursion stops; otherwise, sub-chunks beneath V are recursively merged.

The complete counting algorithm is given in Algorithm 6, which iteratively invokes MergeHDT on every attribute v in a signed-elimination sequence. The overall structure of Algorithm 6 resembles the α-acyclic CQ counting algorithm (Algorithm 1) from Section 3.3 — both algorithms incrementally construct some rooted trees having hyperedges as nodes (i.e. join tree and HDT), where certain annotation is maintained at each node to capture degree information.

Algorithm 4 AGGREGATEHDT(S_V, $\mathcal{T}$)

Input: aggregation S_V maintained at V, HDT $\mathcal{T}$ to be added as a child sub-tree of V
Output: new aggregation at V that includes all hyperedges in $\mathcal{T}$
Global Constants: $\mathsf{CQ}^{\neg}$ $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$, annotated relations $\{S_W\}_{W \in \mathcal{E}^+ \sqcup \mathcal{E}^-}$

1: $U \leftarrow$ root of $\mathcal{T}$
2: **if** $U \in \mathcal{E}^+$ **then return** $S_V \otimes (\bigoplus_{U-V} S_U)$ **end if**
3: $S'_V \leftarrow S_V$ ▷ Hierarchically aggregate over U's children.
4: **for all** child sub-tree $\mathcal{T}'$ of U in $\mathcal{T}$ **do** $S'_V \leftarrow$ AGGREGATEHDT(S'_V, $\mathcal{T}'$) **end for**
5: **return** $S'_V \ominus \left(S_V \otimes \left(\bigoplus_{U-V} S_U\right)\right)$

There are, however, some clear distinctions. In Algorithm 1, any established child-parent edge (U, V) persists throughout the algorithm, and the associated update at V takes the form $S_V \leftarrow S_V \otimes (\bigoplus_{U-V} S_U)$; in contrast, in Algorithm 6, an established child-parent edge (U, V) does not necessarily persist, and the associated update at V could also rely on aggregation underneath U, due to the recursive nature of MERGEHDT and AGGREGATEHDT, respectively.

Last but not least, we elaborate on the technical reason for having an input positive hyperedge W in Algorithm 6. Since $W \in \mathcal{E}^+$, it is easy to see that the hypothesis of Lemma 2 is satisfied; hence, there exists a signed-elimination sequence of $\mathcal{Q}$ that removes all attributes outside of W first. As a result, W remains a single-node HDT $\mathcal{T}$ until all attributes outside of W are eliminated. Then, at this point, we have $V \subseteq_{V_{\text{elim}}} W$ for any other outstanding $\mathcal{T}'$ rooted at some V; thus, we invoke an α-step to let W *complete* these $\mathcal{T}'$, resulting in one final HDT $\mathcal{T}$ that is both rooted at W and contains all hyperedges of $\mathcal{Q}$. Hence, its induced $\mathsf{CQ}^{\neg}$ $\mathcal{Q}_{\mathcal{T}}$ equals the input $\mathsf{CQ}^{\neg}$ $\mathcal{Q}$, and the aggregation at its root W implies $|\mathcal{Q}|$, as desired.

Without leaving a positive hyperedge intact throughout signed-elimination steps, we may end up with one final HDT rooted at some negative hyperedge. In such a case, its induced $\mathsf{CQ}^{\neg}$ would not be the same as Q (as the negative root would be regarded as a positive hyperedge in the induced $\mathsf{CQ}^{\neg}$), subsequently rendering the counting algorithm incorrect.

Example 13. *Consider a full signed-acyclic* $\mathsf{CQ}^{\neg}$ $\mathcal{Q} = (\{V_1, V_2, V_3\}, \{U_1, U_2, U_3, U_4\})$ *with positive hyperedges* $V_1 = \{X_1, X_2\}, V_2 = \{X_2, X_3\}, V_3 = \{X_3, X_4\}$ *and negative hyperedges* $U_1 = \{X_4\}, U_2 = \{X_3, X_4\}, U_3 = \{X_2, X_3, X_4\}, U_4 = \{X_1, X_2, X_3, X_4\}$*. It is easy to check that* $\sigma = X_4X_3X_2X_1$ *is a signed-elimination sequence of* $\mathcal{Q}$*. Figure 4.4 shows the execution of Algorithm 6 on* $\mathcal{Q}$ *with a database instance, using* σ*. This corresponds to taking* $W := \{X_1, X_2\} = V_1$ *in the algorithm.*

Elimination of X_4***.*** *Initially, every hyperedge is a single-node* HDT*.*

- MergeHDT *first selects* $\mathbf{T}_{\text{merge}} = \{V_3, U_1, U_2, U_3, U_4\}$ *(we only need the roots of* HDT *for this selection), and identifies* U_4 *as the maximum w.r.t.* $\subseteq$*. So all of* V_3, U_1, U_2, U_3 *are added as children of* U_4*.*
- MergeHDT *then selects* $\mathbf{T}_{\text{merge}} = \{V_3, U_1, U_2, U_3\}$ *from the children of* U_4*, and identifies* U_3 *as the maximum w.r.t.* $\subseteq$*. So all of* V_3, U_1, U_2 *are added as children of* U_3*.*
- MergeHDT *then selects* $\mathbf{T}_{\text{merge}} = \{V_3, U_1, U_2\}$ *from the children of* U_3*, and identifies* V_3 *as the maximum w.r.t.* $\subseteq$*. So both* U_1, U_2 *are added as children of* V_1*. Because* V_3 *is positive, the recursion stops.*

Elimination of X_3***.*** $V_1 = \{X_1, X_2\}, V_2 = \{X_2, X_3\}, V_3 = \{X_3\}, U_3 = \{X_2, X_3\}, U_4 = \{X_1, X_2, X_3\}$*. As* MergeHDT *would not go underneath a positive hyperedge, any hyperedge underneath a positive hyperedge can be omitted.*

- MergeHDT *first selects* $\mathbf{T}_{\text{merge}} = \{V_2, U_4\}$*, and identifies* U_4 *as the maximum w.r.t.* $\subseteq$*. So* V_2 *is added as a child of* U_3*.*
- MergeHDT *then selects* $\mathbf{T}_{\text{merge}} = \{V_2, U_3\}$ *from the children of* U_4*, and identifies* V_2 *as the maximum w.r.t.* $\subseteq$*. So* U_3 *is added as a child of* V_2*. Because* V_2 *is positive, the recursion stops.*

Now, all attributes outside of $W = \{X_1, X_2\} = V_1$ *are eliminated. We are left with two* HDT*s, one having* U_4 *as its root and one having* V_1 *as its only node. Then,* U_4 *is added as a child of* V_1*, and the algorithm terminates. The final aggregation at* V_1 *indeed matches the query count.*

Algorithm 5 MERGEHDT($\mathbf{T}$, v)

Input: set of HDT $\mathbf{T}$ to be merged on the elimination of signed-leaf v
Output: modified set of HDT
Global Constants: CQ$^{\neg}$ $\mathcal{Q} = (\mathcal{E}^{+}, \mathcal{E}^{-})$, set of eliminated attributes V_{elim}
Global Variables: annotated relations $\{S_W\}_{W \in \mathcal{E}^{+} \sqcup \mathcal{E}^{-}}$

```
1:  procedure COMPARE(𝒯1,𝒯2)  ▷ Helper function to determine which HDT has a larger
        root w.r.t. ⊆_{V_elim}.
2:      V1, V2 ← roots of 𝒯1, 𝒯2, respectively
3:      if V1 ⊊_{V_elim} V2 then return 𝒯2
4:      else if V1 ⊋_{V_elim} V2 then return 𝒯1
5:      else if V2 ∈ ℰ+ then return 𝒯2
6:      else return 𝒯1 end if
7:  end procedure
8:  T_merge ← {𝒯 ∈ T : the root of 𝒯 contains v}     ▷ HDTs to merge in the current step.
9:  T_rest ← T − T_merge                              ▷ HDTs to skip in the current step.
10: if 𝒯_merge = ∅ then return T_rest end if
11: 𝒯 ← maximum HDT among T_merge with respect to COMPARE
12: V ← root of 𝒯
13: for all 𝒯′ ∈ T_merge − {𝒯} do
14:     S_V ← AGGREGATEHDT(S_V, 𝒯′)    ▷ Update the aggregation at V to include 𝒯′.
15:     Update 𝒯 by setting 𝒯′ a child sub-tree of V
16: end for
17: if V ∈ ℰ− then
18:     T_child ← current children sub-trees of V in 𝒯
19:     T′_child ← MERGEHDT(T_child, v)
20:     Update 𝒯 by replacing children sub-trees of V by T′_child
21: end if
22: return {𝒯} ∪ T_rest
```

Algorithm 6 BUILDANDCOUNTHDT($\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$, W)

Input: full signed-acyclic $\mathsf{CQ}^{\neg}$ $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$, any positive hyperedge $W \in \mathcal{E}^+$

Output: an HDT for $\mathcal{Q}$ with root W, and aggregation $\{S_V\}_{V \in \mathcal{E}^+ \sqcup \mathcal{E}^-}$ at each node

1: $\sigma \leftarrow$ find any signed-elimination sequence of $\mathcal{Q}$ that leaves attributes W till the end

2: $V_{\text{elim}} \leftarrow \emptyset$ ▷ Explicitly track eliminated attributes.

3: $\mathbf{T} \leftarrow \emptyset$ ▷ HDTs constructed throughout the algorithm.

4: **for all** $V \in \mathcal{E}^+ \sqcup \mathcal{E}^-$ **do**

5: $\quad S_V \leftarrow \begin{cases} \bigoplus_{\emptyset} R_V, V \in \mathcal{E}^+ \\ \bigoplus_{\emptyset} N_V, V \in \mathcal{E}^- \end{cases}$ ▷ Annotate each tpule in the relation with weight 1.

6: $\quad \mathbf{T} \leftarrow \mathbf{T} \cup \{V$ as a single-node HDT$\}$

7: **end for**

8: **for all** $v \in \sigma$ **do**

9: $\quad$ **if** $v \in W$ **then** exit the loop **end if**

10: $\quad \mathbf{T} \leftarrow$ MERGEHDT($\mathbf{T}$, v)

11: $\quad V_{\text{elim}} \leftarrow V_{\text{elim}} \cup \{v\}$

12: **end for**

13: $\mathcal{T} \leftarrow$ single-node HDT with W as the root from $\mathbf{T}$

14: **for all** $\mathcal{T}' \in \mathbf{T} - \{\mathcal{T}\}$ **do**

15: $\quad S_W \leftarrow$ AGGREGATEHDT(S_W, $\mathcal{T}'$)

16: $\quad$ Update $\mathcal{T}$ by setting $\mathcal{T}'$ a child sub-tree of W

17: **end for**

18: **return** $\mathcal{T}$, and annotated relations $\{S_V\}_{V \in \mathcal{E}^+ \sqcup \mathcal{E}^-}$

R_{V_1}

X_1	X_2
a_1	b_1
a_1	b_2
a_2	b_1
a_2	b_2
a_3	b_3

R_{V_2}

X_2	X_3
b_1	c_1
b_1	c_2
b_2	c_1
b_2	c_2
b_3	c_3

R_{V_3}

X_3	X_4
c_1	d_1
c_1	d_2
c_2	d_1
c_2	d_2
c_3	d_3

N_{U_1}

X_4
d_3

N_{U_3}

X_2	X_3	X_4
b_2	c_1	d_1
b_2	c_2	d_2
b_3	c_3	d_3

$\mathcal{Q}$

X_1	X_2	X_3	X_4
a_1	b_1	c_1	d_1
a_1	b_1	c_2	d_2

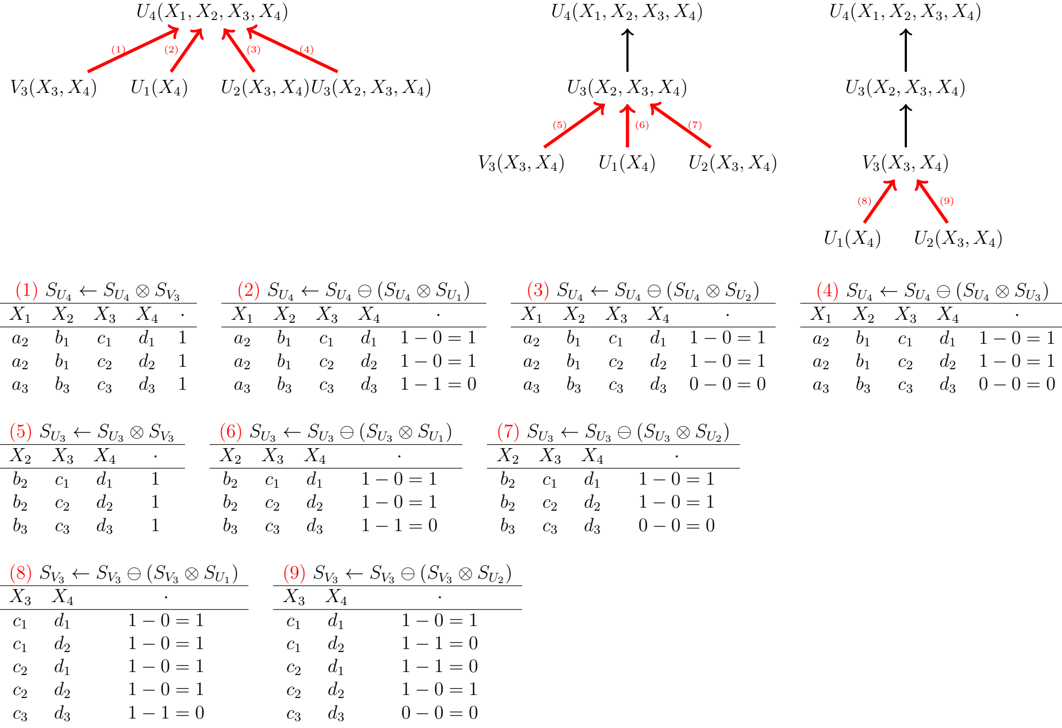
N_{U_2}

X_3	X_4
c_1	d_2
c_2	d_1
c_3	d_3

N_{U_4}

X_1	X_2	X_3	X_4
a_2	b_1	c_1	d_1
a_2	b_1	c_2	d_2
a_3	b_3	c_3	d_3

(a) Database instance and query result.



(1) $S_{U_4} \leftarrow S_{U_4} \otimes S_{V_3}$

X_1	X_2	X_3	X_4	$\cdot$
a_2	b_1	c_1	d_1	1
a_2	b_1	c_2	d_2	1
a_3	b_3	c_3	d_3	1

(2) $S_{U_4} \leftarrow S_{U_4} \ominus (S_{U_4} \otimes S_{U_1})$

X_1	X_2	X_3	X_4	$\cdot$
a_2	b_1	c_1	d_1	$1-0=1$
a_2	b_1	c_2	d_2	$1-0=1$
a_3	b_3	c_3	d_3	$1-1=0$

(3) $S_{U_4} \leftarrow S_{U_4} \ominus (S_{U_4} \otimes S_{U_2})$

X_1	X_2	X_3	X_4	$\cdot$
a_2	b_1	c_1	d_1	$1-0=1$
a_2	b_1	c_2	d_2	$1-0=1$
a_3	b_3	c_3	d_3	$0-0=0$

(4) $S_{U_4} \leftarrow S_{U_4} \ominus (S_{U_4} \otimes S_{U_3})$

X_1	X_2	X_3	X_4	$\cdot$
a_2	b_1	c_1	d_1	$1-0=1$
a_2	b_1	c_2	d_2	$1-0=1$
a_3	b_3	c_3	d_3	$0-0=0$

(5) $S_{U_3} \leftarrow S_{U_3} \otimes S_{V_3}$

X_2	X_3	X_4	$\cdot$
b_2	c_1	d_1	1
b_2	c_2	d_2	1
b_3	c_3	d_3	1

(6) $S_{U_3} \leftarrow S_{U_3} \ominus (S_{U_3} \otimes S_{U_1})$

X_2	X_3	X_4	$\cdot$
b_2	c_1	d_1	$1-0=1$
b_2	c_2	d_2	$1-0=1$
b_3	c_3	d_3	$1-1=0$

(7) $S_{U_3} \leftarrow S_{U_3} \ominus (S_{U_3} \otimes S_{U_2})$

X_2	X_3	X_4	$\cdot$
b_2	c_1	d_1	$1-0=1$
b_2	c_2	d_2	$1-0=1$
b_3	c_3	d_3	$0-0=0$

(8) $S_{V_3} \leftarrow S_{V_3} \ominus (S_{V_3} \otimes S_{U_1})$

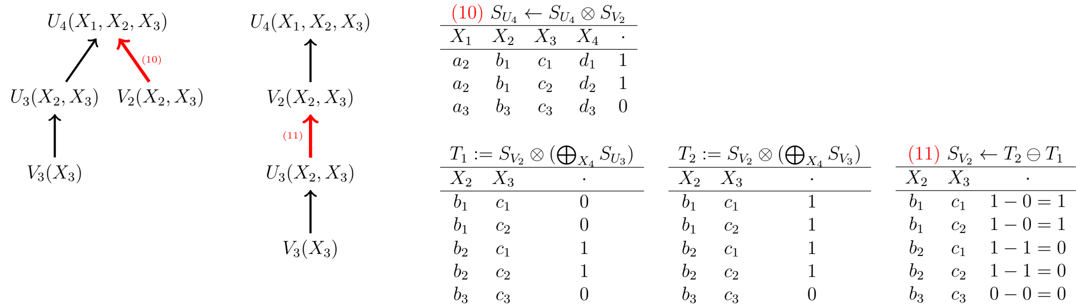
X_3	X_4	$\cdot$
c_1	d_1	$1-0=1$
c_1	d_2	$1-0=1$
c_2	d_1	$1-0=1$
c_2	d_2	$1-0=1$
c_3	d_3	$1-1=0$

(9) $S_{V_3} \leftarrow S_{V_3} \ominus (S_{V_3} \otimes S_{U_2})$

X_3	X_4	$\cdot$
c_1	d_1	$1-0=1$
c_1	d_2	$1-1=0$
c_2	d_1	$1-1=0$
c_2	d_2	$1-0=1$
c_3	d_3	$0-0=0$

(b) Elimination of X_4.

Figure 4.4: An illustration of Algorithm 6 on Example 13, using a signed-elimination sequence $\sigma = X_4X_3X_2X_1$. A red arrow indicates a newly established child-parent edge.



(10) $S_{U_4} \leftarrow S_{U_4} \otimes S_{V_2}$

X_1	X_2	X_3	X_4	$\cdot$
a_2	b_1	c_1	d_1	1
a_2	b_1	c_2	d_2	1
a_3	b_3	c_3	d_3	0

$T_1 := S_{V_2} \otimes (\bigoplus_{X_4} S_{U_3})$

X_2	X_3	$\cdot$
b_1	c_1	0
b_1	c_2	0
b_2	c_1	1
b_2	c_2	1
b_3	c_3	0

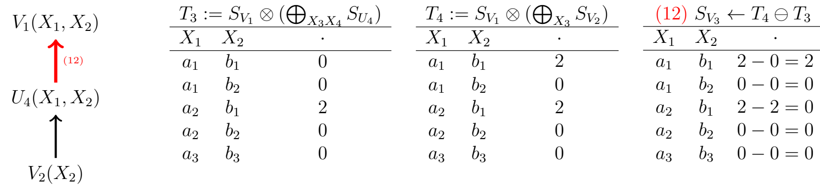
$T_2 := S_{V_2} \otimes (\bigoplus_{X_4} S_{V_3})$

X_2	X_3	$\cdot$
b_1	c_1	1
b_1	c_2	1
b_2	c_1	1
b_2	c_2	1
b_3	c_3	0

(11) $S_{V_2} \leftarrow T_2 \ominus T_1$

X_2	X_3	$\cdot$
b_1	c_1	$1-0=1$
b_1	c_2	$1-0=1$
b_2	c_1	$1-1=0$
b_2	c_2	$1-1=0$
b_3	c_3	$0-0=0$

(c) Elimination of X_3. Complete chunks and eliminated attributes are not shown.



$T_3 := S_{V_1} \otimes (\bigoplus_{X_3 X_4} S_{U_4})$

X_1	X_2	$\cdot$
a_1	b_1	0
a_1	b_2	0
a_2	b_1	2
a_2	b_2	0
a_3	b_3	0

$T_4 := S_{V_1} \otimes (\bigoplus_{X_3} S_{V_2})$

X_1	X_2	$\cdot$
a_1	b_1	2
a_1	b_2	0
a_2	b_1	2
a_2	b_2	0
a_3	b_3	0

(12) $S_{V_3} \leftarrow T_4 \ominus T_3$

X_1	X_2	$\cdot$
a_1	b_1	$2-0=2$
a_1	b_2	$0-0=0$
a_2	b_1	$2-2=0$
a_2	b_2	$0-0=0$
a_3	b_3	$0-0=0$

(d) Last step. Complete chunks and eliminated attributes are not shown.

Figure 4.4: An illustration of Algorithm 6 on Example 13, using a signed-elimination sequence $\sigma = X_4X_3X_2X_1$. A red arrow indicates a newly established child-parent edge (cont.).

Chapter 5

Theoretical Guarantees

We now establish the correctness of our counting algorithm and show that it has the same combined complexity as [48]'s algorithm.

5.1 Correctness

Proof outline. The correctness of our counting algorithm relies on correctly maintaining the aggregation at each hyperedge V. More specifically, given any HDT $\mathcal{T}$ rooted at V, we need to maintain the aggregation $S_V = \bigoplus_{\text{var}(\mathcal{T})-V} \mathcal{Q}_{\mathcal{T}}$, i.e. the degree of V-tuples in the $\mathsf{CQ}^{\neg}$ $\mathcal{Q}_{\mathcal{T}}$ induced by $\mathcal{T}$. At first, we need to ensure that any such $\mathsf{Q}_{\mathcal{T}}$ is indeed a safe $\mathsf{CQ}^{\neg}$.

Next, observe that aggregation at any V is updated whenever a new HDT $\mathcal{T}$ is added as a child sub-tree of V via Algorithm 4 (AggregateHDT); therefore, our next step is to establish the correctness of AggregateHDT.

With this goal in mind, our proof is laid out as follows. (1) First, we define a few properties over HDT; (2) next, we show that AggregateHDT, when given as inputs HDTs satisfying these properties, attains its correctness; (3) lastly, we prove that any HDT created by our algorithm indeed satisfies the desired properties. The reason we separate AggregateHDT from the correctness of the overall counting algorithm is that AggregateHDT would recursively proceed on some *temporary* instances of HDT — refer to Line 4 of AggregateHDT, which *temporarily* replaces U by V in $\mathcal{T}$ and aggregate (recursively) on it. These temporary HDTs are not materialized by our algorithm, hence we do not have (3) to grant them the desired properties.

Definition 11. *Let V_{elim} be the set of eliminated attributes.*

Let $\mathcal{T}_1, \mathcal{T}_2$ be HDT*s rooted at distinct hyperedges V_1, V_2, respectively. We say that:*

- *$\mathcal{T}_1$ is $\subseteq_{V_{\text{elim}}}$-ordered, if for any child-parent edge (U, V) in $\mathcal{T}$, we have $U \subseteq_{V_{\text{elim}}} V$. One immediate consequence is that $\text{var}(\mathcal{T}_1) = \bigcup_{V \in N(\mathcal{T}_1)} V \subseteq_{V_{elim}} V_1$.*
- *$\mathcal{T}_1$ is safe, if for any sub-tree $\mathcal{T}_1'$ in $\mathcal{T}_1$, its induced* $\mathsf{CQ}^\neg$ *$\mathcal{Q}_{\mathcal{T}_1'}$ is safe.*
- *$\mathcal{T}_1, \mathcal{T}_2$ are V_{elim}-separable, if for all attribute $v \in var(\mathcal{T}_1) \cap var(\mathcal{T}_2)$, $v \notin V_{\text{elim}}$; that is, $\mathcal{T}_1, \mathcal{T}_2$ do not share in common any eliminated attribute.*
- *V_1, V_2 are isolated, if (i) neither V_1 nor V_2 is a descendant of some positive hyperedge, and (ii) neither V_1 nor V_2 is a descendant of the other.*
- *$\mathcal{T}_1$ is recursively-V_{elim}-separable, if (i) V_1 is not a descendant of some positive hyperedge, and (ii) for all sub-tree pairs $(\mathcal{T}_1', \mathcal{T}_1'')$ in $\mathcal{T}_1$ with isolated roots, $\mathcal{T}_1', \mathcal{T}_1''$ are V_{elim}-separable.*

We briefly discuss the motivation behind these properties. $\subseteq_{V_{\text{elim}}}$-ordered-ness stems from Section 4.2.3, where we established a simple scenario in which chunk construction following the $\subseteq$-order seems to grant the safety condition on any of its induced $\mathsf{CQ}^\neg$. Meanwhile, V_{elim}-separability is inspired by Example 10, in which we saw that the eliminated attribute B causes troubles in subsequent aggregation — the high-level idea is that the recursive aggregation should not proceed to two parallel chunks that share an already eliminated attribute.

Ultimately, we want to show that any HDT throughout our counting algorithm is $\subseteq_{V_{\text{elim}}}$-ordered and safe, and the recursive aggregation will only be invoked on a recursively-V_{elim}-separable HDT.

We first show a helpful result that establishes a scenario where we can combine two aggregation results into one — this has been an outstanding mystery since the introduction of (∗∗). We are then ready to establish the conditional correctness of AggregateHDT.

Lemma 7. *For any two* HDT*s $\mathcal{T}_1, \mathcal{T}_2$ with roots V_1, V_2, respectively, if $\mathcal{T}_2$ is V_{elim}-ordered and $\mathcal{T}_1, \mathcal{T}_2$ are V_{elim}-separable, then (i) $\text{var}(\mathcal{T}_2) - V_2 \subseteq \text{var}(\mathcal{T}_2) - V_1$; and (ii) the difference between these two sets is precisely $V_2 - V_1$.*

Proof. Consider any $v \in \text{var}(\mathcal{T}_2) - V_2$. Since $\mathcal{T}_2$ is V_{elim}-ordered, we have $\text{var}(\mathcal{T}_2) \subseteq_{V_{\text{elim}}} V_2$; hence, $v \in V_{\text{elim}}$. Then, $\mathcal{T}_1, \mathcal{T}_2$ being V_{elim}-separable entails that $v \notin \text{var}(\mathcal{T}_1) \supseteq V_1$; thus, (i) holds.

We can decompose $\text{var}(\mathcal{T}_2)$ into three disjoint parts: $P_1 := V_2 \cap V_1$, $P_2 := V_2 - V_1$, and $P_3 := \text{var}(\mathcal{T}_2) - V_2$. Notice that $P_1 \subseteq V_1$, $P_2 \cap V_1 = \emptyset$, and the argument above shows that $P_3 \cap V_1 = \emptyset$. Therefore, $\text{var}(\mathcal{T}_2) - V_1 = P_2 \uplus P_3$, and the difference between $\text{var}(\mathcal{T}_2) - V_1$ and $\text{var}(\mathcal{T}_2) - V_2 = P_3$ is exactly $P_2 = V_2 - V_1$, as desired. □

Lemma 8 (Correctness of AGGREGATEHDT)**.** *Consider any two* HDT*s $\mathcal{T}_1, \mathcal{T}_2$ rooted at V_1, V_2, respectively, where V_1, V_2 are isolated*[1]*. Denote $\mathcal{T}$ the new* HDT *formed by making $\mathcal{T}_2$ a new child sub-tree of V_1. Assume the aggregation S_V is correctly maintained at any $V \in N(\mathcal{T}_1) \cup N(\mathcal{T}_2)$. Suppose that*

(i) $\mathcal{Q}_{\mathcal{T}}$ is a safe $\mathsf{CQ}^{\neg}$*;*

(ii) $\mathcal{T}_1, \mathcal{T}_2$ are both safe;

(iii) $\mathcal{T}_1, \mathcal{T}_2$ are both $\subseteq_{V_{\text{elim}}}$-ordered, with $V_2 \subseteq_{V_{\text{elim}}} V_1$;

(iv) $\mathcal{T}_1, \mathcal{T}_2$ are V_{elim}-separable; and

(v) $\mathcal{T}_2$ is recursively-V_{elim}-separable.

Then, AGGREGATE*$(S_V, \mathcal{T}_2)$ correctly computes $\bigoplus_{var(\mathcal{T})-V_1} \mathcal{Q}_{\mathcal{T}}$.*

Proof. We rely on a strong induction on the number of nodes in $\mathcal{T}$. We omit the base case where $\mathcal{T}$ contains two nodes, i.e. both $\mathcal{T}_1, \mathcal{T}_2$ are single-node HDT, for its proof is identical to what follows.

<u>Case $V_2 \in \mathcal{E}^+$ (induction not needed).</u> Notice that $\mathcal{Q}_{\mathcal{T}} = \mathcal{Q}_{\mathcal{T}_1} \bowtie \mathcal{Q}_{\mathcal{T}_2}$ in such case, which then implies

$$\bigoplus_{\text{var}(\mathcal{T})-V_1} \mathcal{Q}_{\mathcal{T}} = \bigoplus_{\text{var}(\mathcal{T}_1)\cup\text{var}(\mathcal{T}_2)-V_1} \mathcal{Q}_{\mathcal{T}_1} \otimes \mathcal{Q}_{\mathcal{T}_2}$$

From (iii), we get that $\text{var}(\mathcal{T}_1) \subseteq_{V_{\text{elim}}} V_1$; thus, $\text{var}(\mathcal{T}_1) \cap \text{var}(\mathcal{T}_2) \subseteq_{V_{\text{elim}}} V_1$. Subsequently, (iv) implies $\text{var}(\mathcal{T}_1) \cap \text{var}(\mathcal{T}_2) \subseteq V_1$. Then, by Lemma 4, we can break apart the aggregation as:

$$\bigoplus_{\text{var}(\mathcal{T}_1)\cup\text{var}(\mathcal{T}_2)-V_1} \mathcal{Q}_{\mathcal{T}_1} \otimes \mathcal{Q}_{\mathcal{T}_2} = \left(\bigoplus_{\text{var}(\mathcal{T}_1)-V_1} \mathcal{Q}_{\mathcal{T}_1}\right) \otimes \left(\bigoplus_{\text{var}(\mathcal{T}_2)-V_1} \mathcal{Q}_{\mathcal{T}_2}\right) = S_{V_1} \otimes \left(\bigoplus_{\text{var}(\mathcal{T}_2)-V_1} \mathcal{Q}_{\mathcal{T}_2}\right)$$

[1] Isolating roots guarantees that $\mathcal{T}_1, \mathcal{T}_2$ are non-overlapping.

Moreover, from (iii) and (iv), Lemma 7 entails $\bigoplus_{\mathrm{var}(\mathcal{T}_2)-V_1} \mathcal{Q}_{\mathcal{T}_2} = \bigoplus_{V_2-V_1} \left(\bigoplus_{\mathrm{var}(\mathcal{T}_2)-V_2} \mathcal{Q}_{\mathcal{T}_2}\right) = \bigoplus_{V_2-V_1} S_{V_2}$. That is, $\bigoplus_{\mathrm{var}(\mathcal{T})-V_1} \mathcal{Q}_{\mathcal{T}} = S_{V_1} \otimes (\bigoplus_{V_2-V_1} S_{V_2})$, which precisely matches Line 2 of AggregateHDT.

Case $V_2 \in \mathcal{E}^-$. Let $\mathcal{T}'_1, \mathcal{T}'_2, \ldots, \mathcal{T}'_m$ be all children sub-trees of V_2, rooted at hyperedges $W_1, W_2, \ldots, W_m$, respectively. Let us first derive a few properties on these $\mathcal{T}'_1, \mathcal{T}'_2, \ldots, \mathcal{T}'_m$. From $\mathcal{T}_2$ being safe, we have

$$\mathcal{T}'_1, \mathcal{T}'_2, \ldots, \mathcal{T}'_m \text{ are all safe} \tag{$\star$}$$

Next, $\mathcal{T}_2$ being $\subseteq_{V_{\mathrm{elim}}}$-ordered and $V_2 \subseteq_{V_{\mathrm{elim}}} V_1$ implies that

$$\mathcal{T}'_1, \mathcal{T}'_2, \ldots, \mathcal{T}'_m \text{ are all } \subseteq_{V_{\mathrm{elim}}} \text{-ordered, and } W_1, W_2, \ldots, W_m \subseteq_{V_{\mathrm{elim}}} V_1 \tag{$\dagger$}$$

Note that V_2 acts as a negative hyperedge in $\mathcal{Q}_{\mathcal{T}}$, and it acts as a positive hyperedge in $\mathcal{Q}_{\mathcal{T}_2}$. Since V_2 is not a descendant of any positive node (from V_1, V_2 being isolated), it is clear that $W_1, W_2, \ldots, W_m$ are mutually isolated. Then, $\mathcal{T}_1, \mathcal{T}_2$ being V_{elim}-separable and $\mathcal{T}_2$ being recursively-V_{elim}-separable together entail that

$$\mathcal{T}_1, \mathcal{T}'_1, \mathcal{T}'_2, \ldots, \mathcal{T}'_m \text{ are mutually } V_{\mathrm{elim}}\text{-separable} \tag{$\ddagger$}$$

$$\mathcal{T}'_1, \mathcal{T}'_2, \ldots, \mathcal{T}'_m \text{ are all recursively-}V_{\mathrm{elim}}\text{-separable} \tag{\S}$$

Let us define a new full $\mathsf{CQ}^\neg$ $\mathcal{Q}' = (N(\mathcal{T}) \cap \mathcal{E}^+ \cup \{V_1\}, N(\mathcal{T}) \cap \mathcal{E}^- - \{V_2\})$, i.e. this is the $\mathsf{CQ}^\neg$ obtained by removing the negative hyperedge V_2 from $\mathcal{Q}_{\mathcal{T}}$.

Because of (i), we have $V_2 \subseteq \mathrm{var}\left(N(\mathcal{T}) \cap \mathcal{E}^+ \cup \{V_1\}\right) \subseteq \mathrm{var}(\mathcal{Q}')$. Then by Lemma 6,

$$\mathcal{Q}' = \mathcal{Q}_{\mathcal{T}} \uplus (\mathcal{Q}_{\mathcal{T}_1} \bowtie \mathcal{Q}_{\mathcal{T}_2})$$

where the three queries from left to right correspond to removing V_2, regarding V_2 as a negative hyperedge, and regarding V_2 as a positive hyperedge, respectively. Since we assume aggregation over integers, which admit additive inverse, the disjoint union implies that

$$\bigoplus_{\mathrm{var}(\mathcal{T})-V_1} \mathcal{Q}_{\mathcal{T}} = \left(\bigoplus_{\mathrm{var}(\mathcal{T})-V_1} \mathcal{Q}'\right) \ominus \left(\bigoplus_{\mathrm{var}(\mathcal{T}_1)\cup\mathrm{var}(\mathcal{T}_2)-V_1} \mathcal{Q}_{\mathcal{T}_1} \otimes \mathcal{Q}_{\mathcal{T}_2}\right)$$

We have established in the previous case that $\bigoplus_{\mathrm{var}(\mathcal{T}_1)\cup\mathrm{var}(\mathcal{T}_2)-V_1} \mathcal{Q}_{\mathcal{T}_1} \otimes \mathcal{Q}_{\mathcal{T}_2} = S_{V_1} \otimes (\bigoplus_{V_2-V_1} S_{V_2})$; hence, it remains to show that Line 4 of AggregateHDT correctly computes $\bigoplus_{\mathrm{var}(\mathcal{T})-V_1} \mathcal{Q}'$.

We iteratively denote $\mathcal{T}_{1,0} := \mathcal{T}_1$ and $\mathcal{T}_{1,i}$ as the tree formed by making $\mathcal{T}'_i$ a new child sub-tree of V_1 in $\mathcal{T}_{1,i-1}$, for each $i = 1, 2, \ldots, m$. Then, observe that $\mathcal{Q}' = \mathcal{Q}_{\mathcal{T}_{1,m}}$, which then implies

$$\bigoplus_{\text{var}(\mathcal{T})-V_1} \mathcal{Q}' = \bigoplus_{\text{var}(\mathcal{T}_{1,m})-V_1} \mathcal{Q}_{\mathcal{T}_{1,m}}$$

Note that each $\mathcal{T}_{1,i}$ has strictly less nodes than $\mathcal{T}$. Therefore, the idea is to show, for each $i = 1, 2, \ldots, m$, that $\textsc{AggregateHDT}\Big(\bigoplus_{\text{var}(\mathcal{T}_{1,i-1})-V_1} \mathcal{Q}_{\mathcal{T}_{1,i-1}}\ ,\ \mathcal{T}'_i\Big)$ correctly computes $\bigoplus_{\text{var}(\mathcal{T}_{1,i})-V_1} \mathcal{Q}_{\mathcal{T}_{1,i}}$, by iteratively applying the inductive hypothesis. The only remaining part is to check that each HDT pair $\mathcal{T}_{1,i-1}, \mathcal{T}'_i$ indeed satisfies all (i)-(v). To distinguish between the notations, here we refer to them as (1)-(5), respectively.

(1) Suppose by way of contradiction that $\mathcal{Q}_{\mathcal{T}_{1,i}}$ is not safe. From (i), (ii), and ($\star$), $\mathcal{Q}_{\mathcal{T}}, \mathcal{Q}_{\mathcal{T}_1}, \mathcal{Q}_{\mathcal{T}'_1}, \ldots, \mathcal{Q}_{\mathcal{T}'_m}$ are all safe. Because $\mathcal{T}_{1,i}$ is formed by adding all of $\mathcal{T}'_1, \ldots, \mathcal{T}'_i$ to the root V_1 of $\mathcal{T}_1$, there must exists some attribute v from some $\mathcal{T}_j$, $j \in [i]$, such that v does not appear in any positive nodes in $\mathcal{T}_{1,i}$; in particular, $v \notin V_1$. From (†), we have $v \in \text{var}(\mathcal{T}'_j) \subseteq_{V_{\text{elim}}} W_j \subseteq_{V_{\text{elim}}} V_1$; hence, $v \in V_{\text{elim}}$. Then, immediately from (‡), we also get $v \notin \text{var}(\mathcal{T}_k)$ for all $k \in [m]$, $k \neq j$. But as $\mathcal{Q}_{\mathcal{T}}$ is a safe $\mathsf{CQ}^{\neg}$, where $\mathcal{T}$ consists of precisely $\mathcal{T}_1, \mathcal{T}'_1, \ldots, \mathcal{T}'_m, V_2$ and $V_2 \in \mathcal{E}^-$, v must be *guarded* from within $\mathcal{T}'_j$, i.e. there exists $W \in \mathcal{E}^+$ within $\mathcal{T}'_j$, such that $v \in W$. But such W also exists in $\mathcal{T}_{1,i}$, reaching a contradiction.

(2) We want to show that $\mathcal{T}_{1,i-1}, \mathcal{T}_i$ are both safe. From (ii) and ($\star$), $\mathcal{T}_1$ is safe and each of $\mathcal{T}'_1, \ldots, \mathcal{T}'_i$ is safe, so now the only remaining candidate to check for safety is $\mathcal{Q}_{\mathcal{T}_{1,i-1}}$. This shares the same proof as (1) above.

(3) Directly from (iii) and (†).

(4) From (iv) and (‡), we get

$$\begin{aligned}
\text{var}(\mathcal{T}_{1,i-1}) \cap \text{var}(\mathcal{T}'_i) \cap V_{\text{elim}} &= \left[\left(\text{var}(\mathcal{T}_1) \cup \bigcup_{j=1}^{i-1} \text{var}(\mathcal{T}'_j)\right) \cap \text{var}(\mathcal{T}'_i)\right] \cap V_{\text{elim}} \\
&= \left[(\text{var}(\mathcal{T}_1) \cap \text{var}(\mathcal{T}'_i)) \cup \bigcup_{j=1}^{i-1} \big(\text{var}(\mathcal{T}'_j) \cap \text{var}(\mathcal{T}'_i)\big)\right] \cap V_{\text{elim}} \\
&= [\text{var}(\mathcal{T}_1) \cap \text{var}(\mathcal{T}'_i) \cap V_{\text{elim}}] \cup \bigcup_{j=1}^{i-1} \big[\text{var}(\mathcal{T}'_j) \cap \text{var}(\mathcal{T}'_i) \cap V_{\text{elim}}\big]
\end{aligned}$$

$$\subseteq [\mathrm{var}(\mathcal{T}_1) \cap \mathrm{var}(\mathcal{T}_2) \cap V_{\mathrm{elim}}] \cup \bigcup_{j=1}^{i-1} \emptyset = \emptyset$$

i.e. $\mathcal{T}_{1,i-1}, \mathcal{T}'_i$ are V_{elim}-separable.

(5) Directly from (§).

□

Next, we establish the various properties each constructed HDT satisfies throughout Algorithm 6.

Lemma 9. *Throughout Algorithm 6, any* HDT $\mathcal{T}$ *is* V_{elim}*-ordered.*

Proof. Given a signed-leaf v and its associated $\subseteq_{\mathrm{elim}}$-chain $V \subseteq_{V_{\mathrm{elim}}} U_1 \subseteq_{V_{\mathrm{elim}}} \cdots \subseteq_{V_{\mathrm{elim}}} U_m$, in each recursive call of MergeHDT on v, the maximum HDT on Line 11 can be chosen if and only if any of its input HDT contains any hyperedge in $\{V, U_1, \ldots, U_m\}$ as its root. If no such HDT exists, then we know the recursive call of MergeHDT reaches underneath $V \in \mathcal{E}^+$, which is impossible by design.

Now, we show that throughout Algorithm 6, any HDT $\mathcal{T}$ is V_{elim}-ordered, using a strong induction on its size. The base case when $\mathcal{T}$ is single-node trivially holds.

There are two places[2] where $\mathcal{T}$ could be constructed: Line 15 and Line 20 in MergeHDT. In Line 15, $\mathcal{T}$ is constructed by combining two non-overlapping $\mathcal{T}_1, \mathcal{T}_2$ rooted at V_1, V_2, respectively, where a new child-parent edge (V_2, V_1) is constructed. Since $\mathcal{T}_1, \mathcal{T}_2$ are all smaller HDT compared to $\mathcal{T}$, the inductive hypothesis holds — $\mathcal{T}_1, \mathcal{T}_2$ are both V_{elim}-ordered. Then, the only candidate to check for V_{elim}-ordered-ness in $\mathcal{T}$ is the new edge (V_2, V_1). By design, we choose V_1 to be the maximum with respect to $\subseteq_{V_{\mathrm{elim}}}$; in particular, $V_2 \subseteq_{V_{\mathrm{elim}}} V_1$. Thus, $\mathcal{T}$ is indeed $\subseteq_{V_{\mathrm{elim}}}$-ordered.

Moving to Line 20, MergeHDT is applied again on children nodes of $\mathcal{T}$ (say $\mathcal{T}$ is rooted at V), which merges some children sub-trees of V into a new HDT $\mathcal{T}'_0$ (rooted at some V_0), while leaving other children sub-trees of V, denoted $\mathcal{T}'_1, \mathcal{T}'_2, \ldots, \mathcal{T}'_m$, intact. Since $\mathcal{T}$ was previously $\subseteq_{V_{\mathrm{elim}}}$-ordered, the only candidates to check for $\subseteq_{V_{\mathrm{elim}}}$-ordered in this updated $\mathcal{T}$ are $\mathcal{T}'_0$ and the child-parent edge (V_0, V). Since $\mathcal{T}'_0$ is smaller than $\mathcal{T}$, inductive hypothesis applies. In addition, because V_0 was also a node in the previous $\mathcal{T}$, we get $V_0 \subseteq_{V_{\mathrm{elim}}} V$. Therefore, the updated $\mathcal{T}$ remains $\subseteq_{V_{\mathrm{elim}}}$-ordered.

[2]There is also Line 16 of Algorithm 6. However, the arguments will be the same as Line 15 in MergeHDT.

Last but not least, because V_{elim} only includes more and more attributes throughout Algorithm 6, existing $\subseteq_{V_{\text{elim}}}$ relationships persist. As a result, $\mathcal{T}$ remains $\subseteq_{\text{elim}}$-ordered following its construction. □

We make an important observation that, whenever MERGEHDT merges two HDT, it is guaranteed that the roots of $\mathcal{T}_1, \mathcal{T}_2$ are isolated, i.e. neither root is an ancestor of the other or a descendant of some positive node (because MERGEHDT never reaches underneath a positive node). As a result, any ancestry relationship between two nodes (i.e. one staying as an ancestor of the other) remains intact once established. We repeatedly apply such observation in the remaining proofs.

Lemma 10. *Throughout Algorithm 6, for any two* HDT $\mathcal{T}_1, \mathcal{T}_2$ *rooted at isolated* V_1, V_2*, respectively,* $\mathcal{T}_1, \mathcal{T}_2$ *are* V_{elim}*-separable. Subsequently, any* HDT $\mathcal{T}$ *that is not a descendant of some positive hyperedge is recursively-*V_{elim}*-separable.*

Proof. Suppose by way of contradiction that there exist nodes U_1, U_2 in $\mathcal{T}_1, \mathcal{T}_2$, respectively, such that there exists $v \in U_1 \cap U_2$ with $v \in V_{\text{elim}}$, i.e. v has already been eliminated.

We trace back to the elimination step of v, denoting V its pivot and V'_{elim} the set of eliminated attributes at that point. We refer to the current time as t_1 and the time of eliminating v as t_2.

Consider t_2. If U_1, U_2 are part of the linear $\subseteq_{V'_{\text{elim}}}$ chain on top of pivot V, then either U_1 or U_2 becomes an ancestor of the other. As such ancestry relationship remains, and V_1 (V_2) is an ancestor of U_1 (U_2) at t_1, either V_1 or V_2 must be an ancestor of the other, contradicting V_1, V_2 being isolated.

Therefore, U_1, U_2 must be placed in the sub-tree rooted at V in t_2, remaining as descendants of V. As V_1, V_2 are isolated (so they cannot be descendants of the positive V), they must be ancestors of V in t_1, hence either of them is an ancestor of the other, reaching the same contradiction. □

Lemma 11. *Throughout Algorithm 6, any* HDT $\mathcal{T}$ *is safe.*

Proof. We prove by a strong induction on the size of $\mathcal{T}$, skipping the trivial base case.

In the inductive step, $\mathcal{T}$ is constructed by either Line 15 or Line 20 in MERGEHDT. In Line 15, $\mathcal{T}$ is constructed by combining two non-overlapping $\mathcal{T}_1, \mathcal{T}_2$ rooted at V_1, V_2, respectively, by adding a new child-parent edge (V_2, V_1). We also know that $V_2 \subseteq_{V_{\text{elim}}} V_1$. As $\mathcal{T}_1, \mathcal{T}_2$ are smaller than $\mathcal{T}$, inductive hypothesis suggests that both are safe. Hence, the only candidate in $\mathcal{T}$ to check for the safety condition is the $\mathsf{CQ}^{\neg}$ $\mathcal{Q}_{\mathcal{T}}$.

Suppose by way of contradiction that $\mathcal{Q}_{\mathcal{T}}$ is not safe. Comparing $\mathcal{Q}_{\mathcal{T}}$ with $\mathcal{Q}_{\mathcal{T}_1} \bowtie \mathcal{Q}_{\mathcal{T}_2}$, the only reason that these two $\mathsf{CQ}^{\neg}$ differ is if $V_2 \in \mathcal{E}^-$. Since $\mathcal{Q}_{\mathcal{T}_1}$ and $\mathcal{Q}_{\mathcal{T}_2}$ are both safe $\mathsf{CQ}^{\neg}$ due to $\mathcal{T}_1, \mathcal{T}_2$ being safe, respectively, it must be that $V_2 \in \mathcal{E}^-$.

Then, there must exist some attribute $v \in V_2$ such that, for all $W \in N(\mathcal{T}) \cap \mathcal{E}^+ \cup \{V_1\}$, $v \notin W$; in particular, $v \notin V_1$. Since $V_2 \subseteq_{V_{\text{elim}}} V_1$, we have $v \in V_{\text{elim}}$. We thus trace back to the elimination step of v, denoting its pivot V ($v \in V \in \mathcal{E}^+$) and the set of eliminated attributes at that time V'_{elim}.

We use a similar case analysis as the proof of Lemma 10: if V_2 is part of the linear $\subseteq_{V'_{\text{elim}}}$ chain on top of V, then V would have been embedded as a descendant of V_2; as a result, $V \in N(\mathcal{T}_2) \subseteq N(\mathcal{T})$, contradicting our choice of v; therefore, V must have been placed as an ancestor of V_2. Nevertheless, this means that V_2 remains a descendant of a positive node from this point on, which implies that V_1, V_2 are not isolated — MergeHDT would never combine $\mathcal{T}_1$ and $\mathcal{T}_2$ by design, reaching a contradiction.

Lastly, let $\mathcal{T}'$ be the HDT obtained after applying Line 20 to some $\mathcal{T}$. Similar to the proof of Lemma 10, the only induced $\mathsf{CQ}^{\neg}$ that could violate the safety condition (due to the strong induction) is $\mathcal{Q}_{\mathcal{T}'}$. But as $\mathcal{T}'$ is merely a reshuffling of non-root nodes of $\mathcal{T}$, we have $\mathcal{Q}_{\mathcal{T}'} = \mathcal{Q}_{\mathcal{T}}$, which is safe from the case above. Thus, $\mathcal{T}'$ is safe. □

By Lemma 9 and Lemma 11, for every HDT $\mathcal{T}$ constructed throughout Algorithm 6, $\mathcal{T}$ is $\subseteq_{V_{\text{elim}}}$-ordered and safe (in particular, $\mathcal{Q}_{\mathcal{T}}$ is safe). Moreover, if $\mathcal{T}$ is constructed via merging two HDT $\mathcal{T}_1, \mathcal{T}_2$, then we know that before the merging, $\mathcal{T}_1, \mathcal{T}_2$ have isolated roots; consequently, by Lemma 10, $\mathcal{T}_1, \mathcal{T}_2$ are V_{elim}-separable, and $\mathcal{T}_2$ is recursively-V_{elim}-separable. Thus, Lemma 8 implies that the annotated relation $S_V = \bigoplus_{\text{var}(\mathcal{T})-V} \mathcal{Q}_{\mathcal{T}}$ is correctly maintained upon the merging. Therefore, we conclude the correctness of Algorithm 6.

5.2 Runtime Analysis

We now show that our counting algorithm achieves the same combined complexity $O(\phi^3 + \phi \cdot \text{IN})$ as [48]'s algorithm (Theorem 2).

Theorem 4. *Given a full signed-acyclic* $\mathsf{CQ}^{\neg}$ $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$ *and a hyperedge* $W \in \mathcal{E}^+$*, Algorithm 6 can compute* $\bigoplus_{\text{var}(\mathcal{Q})-W} \mathcal{Q}$ *in* $O(\phi^3 + \phi \cdot \text{IN})$ *time.*

Subsequently, it computes $|\mathcal{Q}| = \bigoplus_W \bigoplus_{\text{var}(\mathcal{Q})-W} \mathcal{Q}$ *in* $O(\phi^3 + \phi \cdot \text{IN})$ *time.*

Proof. We can decompose the runtime into the overall costs of (i) finding the maximum HDT with respect to Compare in MergeHDT; and (ii) AggregateHDT.

We first analyze (i) in the elimination step of some attribute v. Notice that we do not need to compute the maximum from scratch in each recursive call of MergeHDT. Each recursive call of MergeHDT picks the maximum $\mathcal{T}$ from $\mathbf{T}_{\text{merge}}$, and, conceptually, all of $\mathbf{T}_{\text{merge}} - \{\mathcal{T}\}$ are passed to the next recursive call of MergeHDT, along with all children sub-trees of the root of $\mathcal{T}$. Hence, we can maintain a sorted list of $\mathbf{T}_{\text{merge}}$, where each recursive call of MergeHDT removes the maximum element and adds some new elements to the list. Since there are $d(v)$ hyperedges containing v, $\mathbf{T}_{\text{merge}}$ sees at most $d(v)$ elements at any time. The overall cost to maintain such a sorted $\mathbf{T}_{\text{merge}}$ is therefore $O(d(v)^2)$. Summing across all attributes, we arrive at $O(\sum_v d(v)^2) = O(\phi^2)$, which is dominated by the cost $O(\phi^3)$ to find an elimination sequence at the beginning of Algorithm 6.

Next, we look at (ii). First, we fix a hyperedge V. Notice that given any other hyperedge U, AggregateHDT computes $S_V \otimes (\bigoplus_{U-V} S_U)$ at most once (as the aggregation maintained at V will not double-count the effect of U). The cost of such computation can be decomposed into two parts: (a) the probing cost of each $t \in S_V$ to all $\bigoplus_{U-V} S_U$; and (b) the cost to compute all $\bigoplus_{U-V} S_U$.

For (a), each $t \in S_V$ will probe every other U at most once. Since there can be at most ϕ such U, the cost is $O(|S_V| \cdot \phi)$. Summing up such cost across all V, we get $O(\sum_{V \in \mathcal{E}^+ \sqcup \mathcal{E}^-} |S_V| \cdot \phi) = O(\phi \cdot \text{IN})$.

For (b), similarly, since there are at most ϕ such U, the overall cost to compute these $\bigoplus_{U-V} S_U$ is $O(\sum_{U \in \mathcal{E}^+ \sqcup \mathcal{E}^-} |S_U|) = O(\text{IN})$. We distribute such cost across all elimination steps: when we eliminate some v, only hyperedges containing v will take the role of V here, each requiring $O(\text{IN})$ time to compute all associated $\bigoplus_{U-V} S_U$. Summing up such cost across the elimination steps of all attributes, we get $O(\sum_v d(v) \cdot \text{IN}) = O(\phi \cdot \text{IN})$.

Therefore, the total cost of Algorithm 6 is $O(\phi^3 + \phi \cdot \text{IN})$, as claimed. □

Chapter 6

Experiments

6.1 Experimental Setups

Implementations. To demonstrate that our counting algorithm can be easily integrated into existing SQL-based systems, we implement a prototype program that takes in a signed-acyclic $\mathsf{CQ}^{\neg}$ and outputs a SQL query that produces its count. Although the original Yannakakis algorithm can already be converted to a series of SQL statements, it still takes extensive effort to optimize it for practical usage [25, 43]. Likewise, we note the enormous room for optimization beyond our prototype implementation.

We compare the SQL queries generated by our program with the vanilla SQL count queries. For example, for a query $R_1(A, B) \bowtie R_2(B, C) \triangleright N_1(A, B, C)$, its vanilla SQL count query is

Select Count($*$) From R_1, R_2
Where $R_1.B = R_2.B$ And Not Exists
(Select $*$ From N_1 Where $N_1.A = R_1.A$ AND $N_1.B = R_1.B$ AND $N_1.C = R_2.C$)

We also implement the inclusion-exclusion approach as an additional baseline. Recall that given a $\mathsf{CQ}^{\neg}$ $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$, the inclusion-exclusion principle entails that $|\mathcal{Q}| = \sum_{\mathcal{S} \subseteq \mathcal{E}^-} (-1)^{|\mathcal{S}|} \cdot |\mathcal{Q}_{\mathcal{S}}|$, where each $\mathcal{Q}_{\mathcal{S}} := \mathcal{E}^+ \cup \mathcal{S}$ is a CQ. When $\mathcal{Q}$ is signed-acyclic, each $\mathcal{Q}_{\mathcal{S}}$ is α-acyclic; hence, we can build one join tree for each $\mathcal{Q}_{\mathcal{S}}$ and utilize the Yannakakis bottom-up semijoin reduction on the join tree to obtain $|\mathcal{Q}_{\mathcal{S}}|$.

The inclusion-exclusion approach is similarly implemented as a program that generates SQL queries. Due to the exponential dependency on $\mathcal{E}^-$, the generated SQL query has to

undergo a computationally extensive procedure to aggregate the counts over exponentially many join trees. To provide additional advantage to the inclusion-exclusion approach, we observe that these join trees could share some sub-trees in common, and the aggregation results on common sub-trees can be cached to avoid some redundant computation. Implementing such a caching strategy results in a substantial speed-up for the inclusion-exclusion approach. Nevertheless, similar to the prototype implementation of our counting algorithm, we acknowledge the great potential of optimization strategies to further improve the inclusion-exclusion approach.

Systems. We compare the three methods, namely the vanilla SQL count query (Vanilla), the inclusion-exclusion approach (Base), and our counting algorithm (Ours), in PostgreSQL [2], DuckDB [1], and SparkSQL [4]. PostgreSQL is a row-based relational system with robust support for both transactional and analytical workloads, and it is widely adopted in both research and industrial settings; meanwhile, DuckDB is a columnar system highly optimized for analytical workloads. PostgreSQL and DuckDB are both centralized systems, while SparkSQL assumes a distributed environment and is optimized for parallel processing.

All experiments are run on a single machine equipped with an Apple M2 chip, having 8 cores and 16 GB of memory. We use PostgreSQL 15.13, DuckDB 1.3.0, and for SparkSQL, we use Java 17.0.15, Scala 2.13.16, and Spark 4.0.0. Experiments on these systems use all 8 cores.

Suggested by their respective algorithms, neither Base nor Ours would produce an intermediate result with size exceeding the input size; nevertheless, the cost-based query optimizers in these systems are unaware of such fact and could use NestedLoopJoin due to inaccurate cost estimates. NestedLoopJoin repeatedly scans one of its input relations, which severely impacts the performance of Base and Ours. Therefore, we disable the use of NestedLoopJoin by the query optimizer when running Base and Ours.

6.2 Queries and Datasets

Test Queries. Our counting algorithm is limited to signed-acyclic $\mathsf{CQ}^\neg$, and to the best of our knowledge, there are no existing datasets or queries specifically targeting signed-acyclic $\mathsf{CQ}^\neg$. We therefore compose our own suite of test queries, coming from three classes.

1. ***k*-paths.** Let $R_i(X_i, X_{i+1})$ be relation for every $i \in [k]$. Define $\mathcal{Q}_{k\text{-path}} := R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$ as the k-path CQ. A k-path $\mathsf{CQ}^\neg$ takes the form $\mathcal{Q}_{k\text{-path}} \triangleright N_1 \triangleright N_2 \triangleright \cdots \triangleright N_m$ for some integer $m \geq 0$, where for every $j \in [m]$, $\text{var}(N_j) = \{X_{\ell(j)}, X_{\ell(j)+1}, \ldots, X_{r(j)}\}$

for some $1 \leq \ell(j) \leq r(j) \leq k+1$. That is, each k-path $\mathsf{CQ}^{\neg}$ contains the k-path CQ as its positive part, and any of its negative relations covers a consecutive subsequence of $\{X_1, X_2, \ldots, X_k, X_{k+1}\}$. It can be verified easily that the latter is both a necessary and sufficient condition of signed-acyclicity for k-path $\mathsf{CQ}^{\neg}$.

From a graph perspective, a k-path $\mathsf{CQ}^{\neg}$ aims at identifying all length-k paths with additional constraints imposed on some of its subpaths, as specified by the negative relations $N_1, N_2, \ldots, N_m$.

2. **k-stars.** Let $S_i(X, Y_i)$ be relation for every $i \in [k]$. Define $\mathcal{Q}_{k\text{-star}} := S_1 \bowtie S_2 \bowtie \cdots \bowtie S_k$ as the k-star CQ. A k-star $\mathsf{CQ}^{\neg}$ takes the form $\mathcal{Q}_{k\text{-star}} \triangleright N_1 \triangleright N_2 \triangleright \cdots \triangleright N_m$ for some $m \geq 0$, where $Y \in \text{var}(N_j) \subseteq \{Y, X_1, X_2, \ldots, X_k\}$ for each $j \in [m]$. The necessary and sufficient condition of signed-acyclicity is that the CQ $\mathcal{Q}' := \{\text{var}(N_j) - \{Y\} : j \in [m]\}$ is β-acyclic.

3. **k-trees.** Let $T_{i,0}(X_i, X_{2i}), T_{i,1}(X_i, X_{2i+1})$ be relations for every $i \in [k]$. Define $\mathcal{Q}_{k\text{-tree}} := T_{1,0} \bowtie T_{1,1} \bowtie \cdots \bowtie T_{k,0} \bowtie T_{k,1}$ as the k-tree CQ. From a graph perspective, the k-tree CQ identifies all depth-k perfect binary trees.

   A k-tree $\mathsf{CQ}^{\neg}$ takes the form $\mathcal{Q}_{k\text{-star}} \triangleright N_1 \triangleright N_2 \triangleright \cdots \triangleright N_m$ for some $m \geq 0$, where $\text{var}(N_j) \subseteq \{X_1, X_2, \ldots, X_{2^k+1}\}$ for each $j \in [m]$. The necessary and sufficient condition of signed-acyclicity is that every $\text{var}(N_j)$ forms a connected component in the tree.

We use the following signed-acyclic k-path, k-star, and k-tree $\mathsf{CQ}^{\neg}$s as our test queries. For convenience in notation, $X_{[a:b]}$ denotes a consecutive subsequence of variables $\{X_a, X_{a+1}, \ldots, X_b\}$, for positive integers $a \leq b$.

- $\mathcal{Q}_1^{\neg} := \mathcal{Q}_{4\text{-path}} \triangleright N_1(X_{[1:3]}) \triangleright N_2(X_{[2:4]}) \triangleright N_3(X_{[3:5]})$.
- $\mathcal{Q}_2^{\neg} := \mathcal{Q}_{4\text{-path}} \triangleright N_1(X_{[1:3]}) \triangleright N_2(X_{[2:4]}) \triangleright N_3(X_{[3:5]}) \triangleright N_4(X_{[1:4]}) \triangleright N_5(X_{[2:5]})$.
- $\mathcal{Q}_3^{\neg} := \mathcal{Q}_{5\text{-path}} \triangleright N_1(X_{[1:3]}) \triangleright N_2(X_{[2:4]}) \triangleright N_3(X_{[3:5]}) \triangleright N_4(X_{[1:4]}) \triangleright N_5(X_{[2:5]})$.
- $\mathcal{Q}_4^{\neg} := \mathcal{Q}_{3\text{-star}} \triangleright N_1(X, Y_1, Y_2) \triangleright N_2(X, Y_2, Y_3)$.
- $\mathcal{Q}_5^{\neg} := \mathcal{Q}_{4\text{-star}} \triangleright N_1(X, Y_1, Y_2) \triangleright N_2(X, Y_2, Y_3) \triangleright N_3(X, Y_3, Y_4)$.
- $\mathcal{Q}_6^{\neg} := \mathcal{Q}_{2\text{-tree}} \triangleright N_1(X_1, X_2, X_3) \triangleright N_2(X_2, X_4, X_5) \triangleright N_3(X_3, X_6, X_7)$.

Our construction of these test queries is as follows. First, $\mathcal{Q}_1^{\neg}$ finds all length-4 paths where each of its length-2 subpaths satisfies some constraints (as imposed by N_1, N_2, N_3),

Table 6.1: Graph Datasets and Relevant Statistics

Dataset	#Edges	#Nodes	$\lvert\mathcal{Q}_{\text{4-path}}\rvert$	$\lvert\mathcal{Q}_{\text{5-path}}\rvert$	$\lvert\mathcal{Q}_{\text{3-star}}\rvert$	$\lvert\mathcal{Q}_{\text{4-star}}\rvert$	$\lvert\mathcal{Q}_{\text{2-tree}}\rvert$
Gnutella	88,328	36,682	4.95×10^{6}	1.92×10^{7}	1.04×10^{7}	1.57×10^{8}	1.87×10^{9}
WikiVote	103,689	7,115	9.15×10^{9}	4.13×10^{11}	4.80×10^{9}	2.56×10^{12}	1.22×10^{15}
Stanford	2,312,501	281,903	2.73×10^{10}	9.02×10^{11}	3.24×10^{9}	3.68×10^{11}	1.43×10^{14}

and $\mathcal{Q}_2^{\neg}$ extends $\mathcal{Q}_1^{\neg}$ by additionally constraining the length-3 subpaths. The pair $\mathcal{Q}_1^{\neg}, \mathcal{Q}_2^{\neg}$ demonstrates how the performance of the methods differs when the positive part (i.e. $\mathcal{Q}_{\text{4-path}}$) stays the same but more negative relations are considered. Next, $\mathcal{Q}_3^{\neg}$ can be equivalently expressed as $\mathcal{Q}_2^{\neg} \bowtie R_5(X_5, X_6)$, shedding light on how adding a positive relation could impact the performance. Then, we have two k-star $\mathsf{CQ}^{\neg}$s $\mathcal{Q}_4^{\neg}$ and $\mathcal{Q}_5^{\neg}$. Finally, $\mathcal{Q}_6^{\neg}$ finds all depth-2 perfect binary trees whose depth-1 perfect sub-trees satisfy some constraints.

Datasets. Given the graph nature of our test queries, we select three graph datasets from SNAP (Stanford Large Network Dataset Collection) [3] to measure the performance of our counting algorithm on real-life graph datasets. The selected datasets, along with some statistics, are reported in Table 6.1. Gnutella and WikiVote contain similar numbers of edges but vastly different numbers of nodes, which could provide insights into how well Vanilla, Base, and Ours perform on a sparse graph versus a dense graph. Moreover, testing on Stanford, a much larger graph, suggests the performance of these methods at scale.

All aforementioned relations $R_i, S_i, T_{i,0}, T_{i,1}$ in the test queries will be instantiated using the edge set of these graph datasets; meanwhile, any negative relation will be instantiated by sampling from its associated positive relations. For example, the positive part of $\mathcal{Q}_1^{\neg}$ is $\mathcal{Q}_{\text{4-path}} = R_1(X_1, X_2) \bowtie R_2(X_2, X_3) \bowtie R_3(X_3, X_4) \bowtie R_4(X_4, X_5)$. Then, $N_1(X_{[1:3]}) = N_1(X_1, X_2, X_3)$ will be instantiated by uniform samples from the results of $R_1(X_1, X_2) \bowtie R_2(X_2, X_3)$. To ensure that the I/O cost is not dominated by any specific relation, we constrain all positive and negative relations to have the same size. For instance, if we are using the Gnutella dataset, which contains $88,328$ edges, then all negative relations will contain $88,328$ uniform samples.

Understanding the Vanilla Query Plan. Before running the experiments, we gain some insights on how Vanilla will be executed by these systems, by looking at the execution plans of Vanilla on $\mathcal{Q}_1^{\neg}$ generated by PostgreSQL, DuckDB, and SparkSQL:

$$\text{PostgreSQL-Plan} = \bigoplus_{X_1,X_2,X_3,X_4,X_5} \Big(\big((R_1 \bowtie R_2 \triangleright N_1) \bowtie R_3 \triangleright N_2\big) \bowtie R_4 \triangleright N_3\Big)$$

$$\text{DuckDB-Plan} = \bigoplus_{X_1,X_2,X_3,X_4,X_5} (R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \triangleright N_1 \triangleright N_2 \triangleright N_3)$$

SparkSQL generates the same plan as PostgreSQL.

The execution plans are expressed in relational algebra, where the operations are left associative and aggregations take precedence over joins and antijoins. The first plan is as follows: (i) $R_1 \bowtie R_2$ is materialized, and the result is filtered with N_1; (ii) the intermediate result is joined with R_3 and filtered with N_2; (iii) the intermediate result is joined with R_4 and filtered with N_3; (iv) finally, all attributes are aggregated away at once to produce the answer. On the other hand, the second plan first materializes the join $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ before applying any antijoins. The first plan is superior to the second for pushing down the antijoins into the execution plan to reduce the intermediate result size; nevertheless, there is still room for improvement — for instance, we can additionally push down the aggregation operator and arrive at the following plan:

$$\bigoplus_{X_3,X_4,X_5}\Big(\bigoplus_{X_2}\big(\bigoplus_{X_1}(R_1 \bowtie R_2 \triangleright N_1) \bowtie R_3 \triangleright N_2\big) \bowtie R_4 \triangleright N_3\Big)$$

We give a brief explanation of this plan: after computing $R_1 \bowtie R_2 \triangleright N_1$, the attribute X_1 is no longer needed in subsequent processing (as the remaining relations R_3, N_2, R_4, N_3 do not contain X_1); hence, X_1 can be aggregated away early. Indeed, our proposed counting algorithm, i.e. Ours, can be regarded as a systematic approach to explore push-down opportunities of antijoins and aggregations.

As the query plans suggest, Vanilla is bottle-necked by the maximum intermediate result size. On the other hand, the runtimes of Base and Ours are dependent only on the input size and the formula complexity of the query. Therefore, we expect the trade-offs between these variables to reflect on the experimental results.

6.3 Results

Figure 6.1 reports the processing time of the SQL query generated by each method (Vanilla, Base, Ours) across different database systems. The processing time is measured as the time a database system takes to return an answer after receiving a SQL query, which includes the time for the query optimizer to parse the SQL query and determine a query plan, and the execution time of the plan. A bar reaching the top horizontal axis represents that the corresponding trial did not terminate within a time limit of $10,000$ seconds.

We first look at the results on centralized systems, i.e. PostgreSQL and DuckDB. The general trend is that Base outperforms Vanilla by several orders of magnitude, while Ours

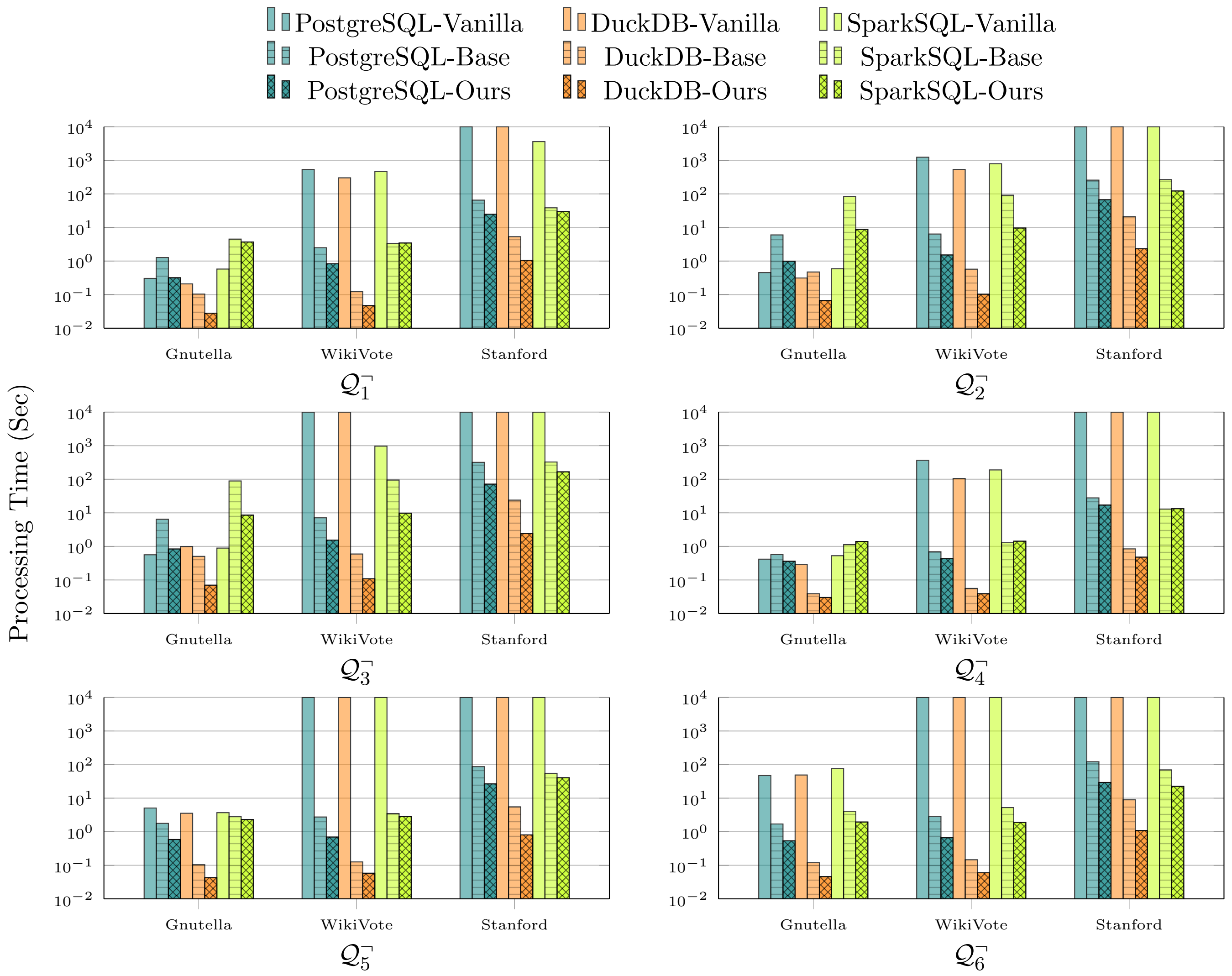


Figure 6.1: Processing Time of Test Queries with Different Database Systems

outperforms Vanilla by 1.3x to 10x. The exception occurs on PostgreSQL with the Gnutella dataset, where Vanilla outperforms Ours on $\mathcal{Q}_1^-$ by at most 2x on $\mathcal{Q}_1^-$, $\mathcal{Q}_2^-$ and $\mathcal{Q}_3^-$. From Table 6.1, $\mathcal{Q}_{\text{4-path}}$ and $\mathcal{Q}_{\text{5-path}}$ are relatively small on Gnutella; therefore, the intermediate result encountered by the query plan of Vanilla is not very expensive to materialize in these cases. Indeed, after moving to the WikiVote dataset, which contains a similar number of edges but is much denser, these intermediate sizes grow substantially, leading to a gap of at least 500x between Vanilla and Ours on these queries.

When fixing the query and the dataset, DuckDB achieves a much better performance than PostgreSQL with Base and Ours. The cause could be that DuckDB is optimized for analytical workloads, and both Base and Ours involve a large number of aggregations. With Vanilla, as PostgreSQL recognizes some push-down opportunities of antijoins whereas DuckDB does not, PostgreSQL sometimes achieve a better performance than DuckDB, for example, on $\mathcal{Q}_3^-$ with Gnutella.

Moving to SparkSQL, we observe some intriguing cases. First, we notice a similar phenomenon as with PostgreSQL on Gnutella, where Vanilla takes the least amount of time on $\mathcal{Q}_1^-$, $\mathcal{Q}_2^-$, $\mathcal{Q}_3^-$, and $\mathcal{Q}_4^-$, due to the sparse nature of Gnutella. Another finding is that Base and Ours noticeably consume more time compared to the centralized setting. Since SparkSQL assumes a distributed environment, relations are shuffled to compute joins and aggregations. This poses challenges to Base and Ours as they frequently compute inter-relation aggregations, resulting in high shuffling cost among the parallel executors. Nevertheless, on most queries and datasets, we still observe the general trend in which Base outperforms Vanilla by several orders of magnitude, while Ours attains improvements from 1.3x to 10x over Base.

6.4 Impact of Individual Variables

When we change the test query and the dataset, the performance of each of Vanilla, Base, and Ours is affected differently. Taking $\mathcal{Q}_1^-$ as an example, as we change the dataset from Gnutella to WikiVote, Vanilla suddenly incurs an over 1,000x increase in its processing time, while Base and Ours are not impacted as much. This motivates us to empirically evaluate how different variables impact these three methods.

Impact of Query Count. We first study whether the count of a query would affect the processing time of any of the three methods on $\mathcal{Q}_1^-$. To minimize the effect of other variables as much as possible, we fix IN and try to control the maximum intermediate size produced by Vanilla.

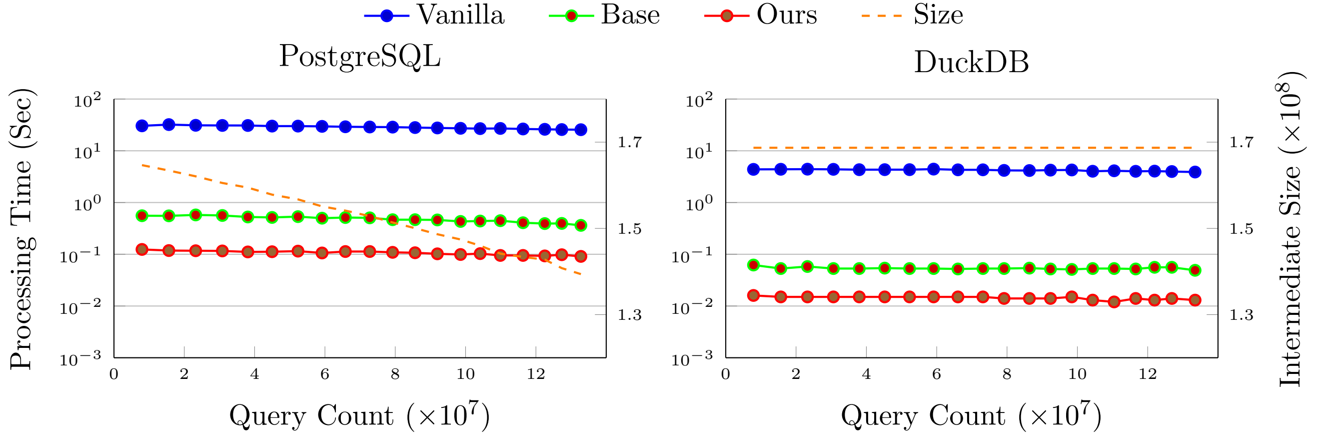


Figure 6.2: Impact of the Query Count on Processing Time

First, observe that $\mathcal{Q}_1^{\neg}$ can be equivalently expressed as

$$\mathcal{Q}_1^{\neg} = (R_1 \bowtie R_2 - N_1) \bowtie (R_2 \bowtie R_3 - N_2) \bowtie (R_3 \bowtie R_4 - N_3)$$

Suppose all the positive relations R_1, R_2, R_3, R_4 are fixed, and N_i is instantiated by uniform samples from $R_i \bowtie R_{i+1}$ for each $i \in [3]$. By varying $|N_1|, |N_2|, |N_3|$ while fixing $|N_1| + |N_2| + |N_3|$, we can vary $|\mathcal{Q}_1^{\neg}|$ with IN fixed. For example, if $|R_2 \bowtie R_3| > |R_3 \bowtie R_4|$, then choosing a smaller $|N_2|$ and a larger $|N_3|$ results in a smaller $|\mathcal{Q}_1^{\neg}|$ value in expectation.

Based on such idea, we instantiate each of R_1, R_2, R_3, R_4 by picking edges from the WikiVote dataset, such that $|R_2 \bowtie R_3|$ is much larger than $|R_3 \bowtie R_4|$. Then, N_i is constructed with uniform samples from $R_i \bowtie R_{i+1}$ for each $i \in [3]$. We let $|R_1| = |R_2| = |R_3| = |R_4| = |N_1| = 20,000$ and obtain $|R_1 \bowtie R_2| = 1,237,052$, $|R_2 \bowtie R_3| = 572,691$, $|R_3 \bowtie R_4| = 96,051$, and $|\mathcal{Q}_{\text{4-path}}| = 168,700,221$. Then, to vary $|\mathcal{Q}_1^{\neg}|$, we take $|N_2| = \tau$ and $|N_3| = 96,000 - \tau$ for different choices of $\tau \in \left\{\left\lceil \frac{i}{21} \times 96,000\right\rceil : i \in [20]\right\}$.

The result is reported in Figure 6.2, where we plot the processing time of each method as $|\mathcal{Q}_1^{\neg}|$ changes. We also plot the maximum intermediate size generated in the Vanilla plan. As DuckDB materializes $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ on Vanilla, such size remains the same; meanwhile, the largest intermediate result materialized by PostgreSQL is $\big((R_1 \bowtie R_2 \triangleright N_1) \bowtie R_3 \triangleright N_2\big) \bowtie R_4$. Under our setting, such result's size differs by at most 18% among the trials, which is at a much smaller scale compared to the changes in $|\mathcal{Q}_1^{\neg}|$ and would induce little impact on Vanilla. All other variables of consideration, i.e. the query expression and the input size, are fixed. The plots suggest that a varying query count value has no visible impact on the processing time of any of the three methods.

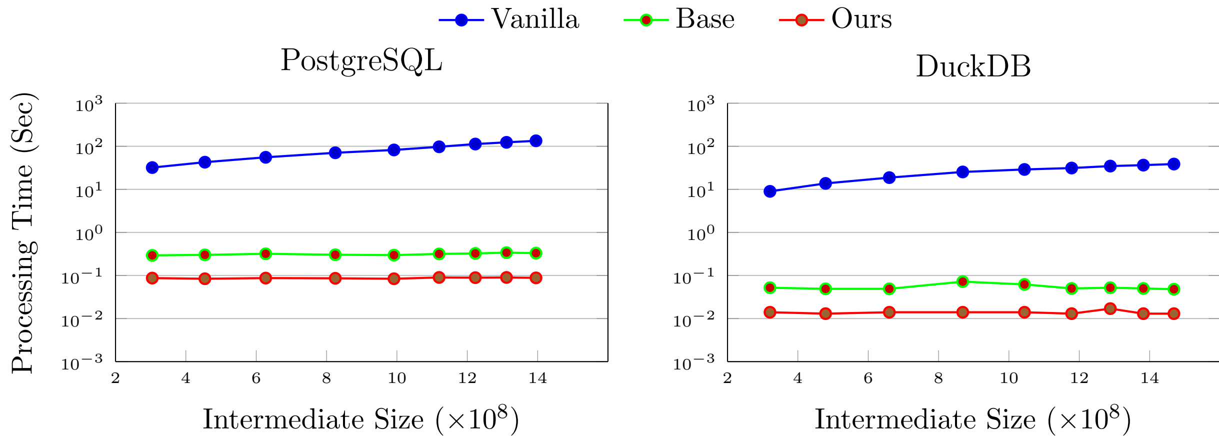


Figure 6.3: Impact of Maximum Intermediate Size on Processing Time

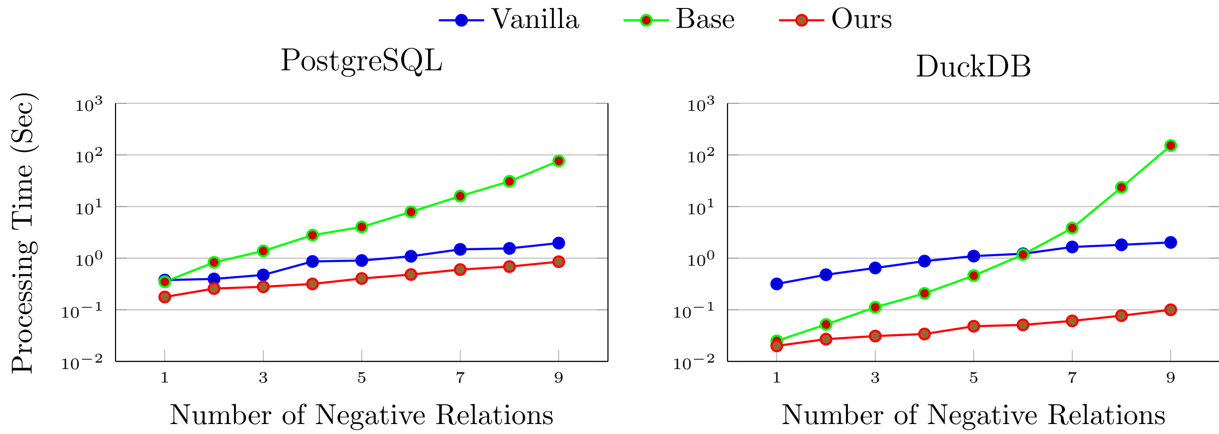


Figure 6.4: Impact of Number of Negative Relations

Impact of Maximum Intermediate Size. Next, we investigate how the processing time of Vanilla scales with the maximum intermediate size produced in its execution plan. Likewise, we test on $\mathcal{Q}_1^{\neg}$. We fix R_1, R_2, R_3, and use different edge-picking strategies on R_4 to vary $|R_3 \bowtie R_4|$, subsequently varying the maximum intermediate size. Although this change would induce a change in $|\mathcal{Q}_1^{\neg}|$, the previous result concludes that $|\mathcal{Q}_1^{\neg}|$ does not significantly impact the processing time of any method. We use the WikiVote dataset, and we have $|R_i| = |N_j| = 20,000$ for every $i \in [4], j \in [3]$.

Figure 6.3 reports how each method is affected, along with changes of the maximum possible intermediate size that would be produced by each system on $\mathcal{Q}_1^{\neg}$. As Vanilla materializes such an intermediate result, its processing time scales approximately linearly with such size. On the other hand, Base and Ours remain unaffected. This aligns with theoretical results that both Base and Ours have a data complexity of $O(\text{IN})$, independent of how large such an intermediate result might grow.

Impact of Number of Negative Relations. The data complexity $O(\text{IN})$ of Base hides its exponential dependency on the number of negative relations, motivating us to study the impact of such a variable. We consider the following array of negative relations:

$$N_1(X_{[1:3]}), N_2(X_{[2:4]}), N_3(X_{[3:5]}), N_4(X_{[4:6]}), N_5(X_{[1:4]}),$$
$$N_6(X_{[2:5]}), N_7(X_{[3:6]}), N_8(X_{[1:5]}), N_9(X_{[2:6]})$$

and use the test queries $\mathcal{Q}_{(i)}^{\neg} := \mathcal{Q}_{\text{5-path}} \triangleright N_1 \triangleright \cdots \triangleright N_i$, $i \in [9]$. We use Gnutella to instantiate every R_i in $\mathcal{Q}_{\text{5-path}}$, giving $|R_i| = 88,328$. To fix IN across these $\mathcal{Q}_{(i)}^{\neg}$, we construct each N_j to be roughly the same size while fixing $\sum_{j=1}^{i} |N_j| = 200,000$.

The result, shown in Figure 6.4, demonstrates an exponential increase in the processing time of Base as the number of negative relations grows, which is as expected. Meanwhile, the processing times of Vanilla and Ours scale at much lower rates.

6.5 Memory Consumption

At last, we measure the amount of memory used by DuckDB to execute our test queries on the Gnutella dataset. The result is reported in Figure 6.5. We see that Vanilla consumes the least amount of memory on $\mathcal{Q}_1^{\neg}$, $\mathcal{Q}_2^{\neg}$, $\mathcal{Q}_3^{\neg}$, and $\mathcal{Q}_4^{\neg}$. As previous experiments and discussion suggest, Vanilla is bottle-necked by the computation of intermediate results whose size could go well beyond the input size; meanwhile, Ours runs in a time dependent on the input size and the query's formula complexity. Given the sparse nature of Gnutella, the

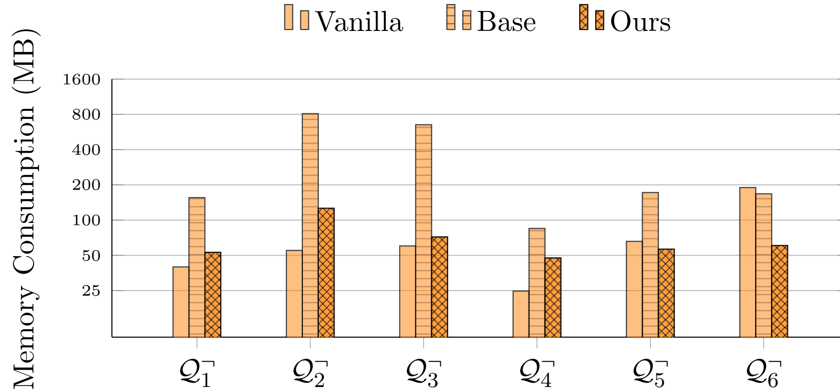


Figure 6.5: DuckDB Memory Consumption with Gnutella

intermediate results are inexpensive to compute, resulting in a smaller memory footprint for these queries. When moving to $\mathcal{Q}_5^{\neg}$ and $\mathcal{Q}_6^{\neg}$ that produce relatively large intermediate results, Vanilla's memory consumption increases substantially; meanwhile, Ours's memory consumption remains relatively stable as all test queries have similar input sizes and formula complexities.

On the other side, Base consumes more memory than Ours across all test queries, with the difference being enormous on $\mathcal{Q}_2^{\neg}$ and $\mathsf{Q}_3^{\neg}$. As $\mathcal{Q}_2^{\neg}$ and $\mathcal{Q}_3^{\neg}$ contain the largest number of negative relations among the test queries, Base's memory consumption is exacerbated significantly due to its exponential dependency on the number of negative relations.

Chapter 7

Extensions

7.1 An Evaluation Algorithm for Signed-Acyclic $\mathsf{CQ}^{\neg}$

We describe an evaluation algorithm for signed-acyclic $\mathsf{CQ}^{\neg}$ using the previous Reduce developed for our decision algorithm. Reduce is built on degree calculations, which can be easily realized in SQL with Group By; as a result, our evaluation algorithm can also be expressed as SQL statements and executed by any SQL-based database system.

Recall that, given a $\mathsf{CQ}^{\neg}$ $\mathcal{Q}_1$ with a signed-leaf v, Reduce outputs a new $\mathsf{CQ}^{\neg}$ $\mathcal{Q}_2$ with attribute v removed, such that $\pi_{\mathrm{var}(\mathcal{Q}_1)-\{v\}}\mathcal{Q}_1 = \mathcal{Q}_2$. This implies that given any output $t \in \mathcal{Q}_2$, t can be extended to a valid output of $\mathcal{Q}_1$; in other words, there exists some $a \in \mathrm{dom}(v)$ for which $t \| a \in \mathcal{Q}_1$. The goal of the evaluation algorithm is then to *recover* the lost information regarding these $a \in \mathrm{dom}(v)$ due to Reduce. We observe that in order to reconstruct $\mathcal{Q}_1$, all we need is to join (and antijoin) $\mathcal{Q}_2$ with the relations from $\mathcal{Q}_1$ that contain v.

Lemma 12. *Let $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$ be a $\mathsf{CQ}^{\neg}$ and v be its signed-leaf. Let V be the pivot of v, and $V \subsetneq U_1 \subseteq U_2 \subseteq \cdots \subseteq U_m$ be a linear $\subseteq$-order on $\{V\} \cup \{U \in \mathcal{E}_v^- : U \not\subseteq V\}$. Then,*

$$\mathcal{Q} = \left(\pi_{\mathrm{var}(\mathcal{Q})-\{v\}}\mathcal{Q}\right) \bowtie R'_V \triangleright N_{U_1} \triangleright \cdots \triangleright N_{U_m}$$

where R'_V is the updated relation associated to V after α-step in Reduce.

Proof. Recall that α-step removes all hyperedges $U \in \mathcal{E}^+ \sqcup \mathcal{E}^-$ such that $U \subseteq V$ and updates R_V without affecting the query result. After α-step, the only remaining hyperedges in the

query that contain v are precisely $V, U_1, U_2, \ldots, U_m$. For convenience, we denote the query on the right-hand side as $\mathcal{Q}_\star$.

$\subseteq$-direction. Let $t \in \mathcal{Q}$. This implies that $\pi_{\text{var}(\mathcal{Q})-\{v\}} t \in \pi_{\text{var}(\mathcal{Q})-\{v\}} \mathcal{Q}$, and (i) for all $W \in \mathcal{E}^+$, $\pi_W t \in R_W$; (ii) for all $U \in \mathcal{E}^-$, $\pi_U t \notin N_U$. (i) and (ii) cover V and each of $U_1, \ldots, U_m$, respectively; hence, we get $t \in \mathcal{Q}_\star$.

$\supseteq$-direction. Let $t \in \mathcal{Q}_\star$, and suppose by way of contradiction that $t \notin \mathcal{Q}$. Then, there must exist some hyperedge W in $\mathcal{Q}$ for which t violates. Without loss of generality, assume $W \in \mathcal{E}^+$, i.e. $\pi_W t \notin R_W$. Since $V, U_1, \ldots, U_m$ are the only hyperedges that contain attribute v, and they all appear in $\mathcal{Q}_\star$, we must have that $v \notin W$. But then $\pi_{\text{var}(\mathcal{Q})-\{v\}} t \in \pi_{\text{var}(\mathcal{Q})-\{v\}} \mathcal{Q}$ implies that there exists some $a \in \text{dom}(v)$ for which $t \| a \in \mathcal{Q}$, which then implies $\pi_W t \| a \in R_W$. Since $v \notin W$, $\pi_W t = \pi_W t \| a \in R_W$, reaching a contradiction. $\square$

Let us standardize the notations used in what follows: $\mathcal{Q}_1$ is a $\mathsf{CQ}^\neg$ with signed-leaf v, $\mathcal{Q}_2$ is the $\mathsf{CQ}^\neg$ obtained by eliminating v from $\mathcal{Q}_1$ via Reduce, $V \subsetneq U_1 \subseteq \cdots \subseteq U_m$ is the linear $\subseteq$-order Reduce uses in its β-step, with $R'_V, N_{U_1}, \ldots, N_{U_m}$ being the respective relations in the database instance after Reduce applies its α-step. Lemma 12 implies that $\mathcal{Q}_1 = \mathcal{Q}_2 \bowtie R'_V \triangleright N_{U_1} \triangleright \cdots \triangleright N_{U_m}$, suggesting a recursive procedure to evaluate $\mathcal{Q}_1$: (i) use Reduce to obtain the database instance of $\mathcal{Q}_2$; (ii) recursively evaluate $\mathcal{Q}_2$; (iii) evaluate $\mathcal{Q}_1$ as $\mathcal{Q}_2 \bowtie R'_V \triangleright N_{U_1} \triangleright \cdots \triangleright N_{U_m}$. Then, the remaining problem is: how can we compute the join/antijoin in (iii) in a way that is *efficient*, e.g. achieving $O(\text{IN} + \text{OUT})$ data complexity? We remark a series of simple observations inspired by [30].

Lemma 13. *Let R_V, R_W, N_U be relations, such that $V \cup W = U$. Then,*

(i) $R_V \bowtie R_W \triangleright N_U = R_V \bowtie R_W - N_U$;

(ii) $|R_V \bowtie R_W| \leq |R_V \bowtie R_W - N_U| + |N_U|$; *and*

(iii) if R_V is non-dangling with respect to the join $R_V \bowtie R_W$, then $|R_V| \leq |R_V \bowtie R_W|$. Subsequently, $|R_V| \leq |R_V \bowtie R_W - N_U| + |N_U|$.

Proof. (i) When $V \cup W = U$, the definitions of antijoin and difference are equivalent.

(ii) In the difference $R_W \bowtie R_W - N_U$, at most $|N_U|$ tuples from $R_W \bowtie R_W$ will be filtered. That is, $|R_V \bowtie R_W| - |R_V \bowtie R_W - N_U| \leq |N_U|$.

(iii) R_V being non-dangling with respect to the join $R_V \bowtie R_W$ means that $R_V \subseteq \pi_V(R_V \bowtie R_W)$. Hence, $|R_V| \leq |\pi_V(R_V \bowtie R_W)| \leq |R_V \bowtie R_W|$. (ii) then implies the latter inequality. $\square$

Lemma 13 reveals that, if we can evaluate an antijoin in the form of *difference*, then we could obtain certain size bounds on intermediate results produced during the process. To see how antijoins can be converted to differences in our computation, let us first focus on the case $m = 1$, i.e. there is only one negative relation N_{U_1} to consider. Then, notice that

$$\begin{aligned}\mathcal{Q}_2 \bowtie R'_V \triangleright N_{U_1} &= \mathcal{Q}_2 \bowtie \left(\pi_{U_1-\{v\}}\mathcal{Q}_2\right) \bowtie R'_V \triangleright N_{U_1} \\ &= \mathcal{Q}_2 \bowtie \left(\left(\pi_{U_1-\{v\}}\mathcal{Q}_2\right) \bowtie R'_V - N_{U_1}\right)\end{aligned}$$

where we have *patched* R'_V with a portion of $\mathcal{Q}_2$, in a way that precisely covers attributes U_1; then, Lemma 13 (i) suggests that we can change the antijoin to a difference. Because $\mathcal{Q}_2$ already appears in the query, additionally joining the query with a projection on it does not affect the query result.

Moving to an arbitrary m, we can apply this idea iteratively to change the antijoins of $N_{U_1}, N_{U_2}, \ldots, N_{U_m}$ to differences. Let us denote $U_0 := V$ and iteratively denote $R'_{U_i} := \left(\pi_{U_i-\{v\}}\mathcal{Q}_2\right) \bowtie R'_{U_{i-1}} - N_{U_i}$ for each $i = 1, 2, \ldots, m$. Because $U_0 \subsetneq U_1 \subseteq \cdots \subseteq U_m$, every R'_{U_i} is indeed well-defined. Then,

$$\begin{aligned}\mathcal{Q}_1 &= \mathcal{Q}_2 \bowtie R'_V \triangleright N_{U_1} \triangleright \cdots \triangleright N_{U_m} \\ &= \mathcal{Q}_2 \bowtie \left(\left(\pi_{U_1-\{v\}}\mathcal{Q}_1\right) \bowtie R'_V - N_{U_1}\right) \triangleright N_{U_2} \triangleright \cdots \triangleright N_{U_m} \\ &= \mathcal{Q}_2 \bowtie R'_{U_1} \triangleright N_{U_2} \triangleright \cdots \triangleright N_{U_m} \\ &= \mathcal{Q}_2 \bowtie \left(\left(\pi_{U_2-\{v\}}\mathcal{Q}_2\right) \bowtie R'_{U_1} - N_{U_2}\right) \triangleright N_{U_3} \triangleright \cdots \triangleright N_{U_m} \\ &= \mathcal{Q}_2 \bowtie R'_{U_2} \triangleright N_{U_3} \triangleright \cdots \triangleright N_{U_m} \\ &= \cdots \\ &= \mathcal{Q}_2 \bowtie R'_{U_m}\end{aligned}$$

Algorithm 7 implements these ideas to recursively evaluate a full signed-acyclic $\mathsf{CQ}^{\neg}$. We omit the correctness proof, for it follows directly from our exploration above.

Next, we establish the runtime of Algorithm 7, which also turns out to be the same as [48]'s algorithm. The crucial part of the runtime analysis lies in understanding the computation cost of each intermediate result R'_{U_i}, for which we make an important observation.

Lemma 14. *In Algorithm 7, for each* $i \in [m]$, $|R'_{U_i}| \le \mathrm{IN} + \mathrm{OUT}$.

Proof. We examine each of $R'_{U_m}, R'_{U_{m-1}}, \ldots, R'_{U_1}$, i.e. in reverse order of their computation.

The edge case of $m = 0$ is trivial: $|R'_{U_m}| = |R'_V| \le \mathrm{IN}$. Next, we consider $m \ge 1$.

Algorithm 7 $\textsc{EvaluateFull}(\mathcal{Q}_1 = (\mathcal{E}^+, \mathcal{E}^-),\ \sigma = v \cdot \sigma')$

Input: full signed-acyclic $\mathsf{CQ}^\neg$ $\mathcal{Q}_1 = (\mathcal{E}^+, \mathcal{E}^-)$, a signed-elimination sequence σ of $\mathcal{Q}_1$ with v being the first vertex and σ' being the rest

Output: result of evaluating $\mathcal{Q}_1$

1: **if** $\sigma' = \emptyset$ **then return** evaluate $\mathcal{Q}_1$ naïvely **end if** ▷ v is the only attribute left in $\mathcal{Q}_1$.
2: $\mathcal{Q}_2 \leftarrow \textsc{Reduce}(\mathcal{Q}_1, v)$
3: Let $V \subsetneq U_1 \subseteq \cdots \subseteq U_m$ be the linear $\subseteq$-order in β-step in $\textsc{Reduce}$
4: $R'_V, N_{U_1}, \ldots, N_{U_m} \leftarrow$ relations associated to $V, U_1, \ldots, U_m$, respectively
5: $\mathcal{Q}_2 \leftarrow \textsc{EvaluateFull}(\mathcal{Q}_2, \sigma')$ ▷ Evaluate $\mathcal{Q}_2$ recursively.
6: $U_0 \leftarrow V$
7: **for all** $i = 1, 2, \ldots, m$ **do** $R'_{U_i} \leftarrow \left(\pi_{U_i - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_i - \{v\}} - N_{U_i}$ **end for**
8: **return** $\mathcal{Q}_2 \bowtie R'_{U_m}$

First, we consider R'_{U_m} in the join $\mathcal{Q}_1 = \mathcal{Q}_2 \bowtie R'_{U_m}$. We claim that R'_{U_m} is non-dangling with respect to this join, i.e. for every $t \in R'_{U_m}$, $\pi_{U_m \cap \mathrm{var}(\mathcal{Q}_2)} t \in \pi_{U_m \cap \mathrm{var}(\mathcal{Q}_2)} \mathcal{Q}_2$. To see this, consider any $t \in R'_{U_m}$. Recall that $R'_{U_m} = \left(\pi_{U_m - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_{m-1}} - N_{U_m}$; therefore, $\pi_{U_m - \{v\}} t \in \pi_{U_m - \{v\}} \mathcal{Q}_2$. Notice that $U_m \cap \mathrm{var}(\mathcal{Q}_2) = U_m - \{v\}$, thereby proving the claim.

By Lemma 13 (iii), we then have $|R'_{U_m}| \leq |\mathcal{Q}_2 \bowtie R'_{U_m}| = |\mathcal{Q}_1| = \mathrm{OUT}$, as desired.

Next, consider any $i \in [m-1]$. We claim that R'_{U_i} is non-dangling with respect to the join $\left(\pi_{U_{i+1} - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_i}$. This can be proven similarly by noting that $R'_{U_i} = \left(\pi_{U_i - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_{i-1}} - N_{U_i}$, and that $U_i - \{v\}$ are precisely the attributes shared between $\pi_{U_{i+1} - \{v\}} \mathcal{Q}_2$ and R'_{U_i} because $U_i \subseteq U_{i+1}$. Then by Lemma 13 (iii),

$$|R'_{U_i}| \leq |\left(\pi_{U_{i+1} - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_i} - N_{U_{i+1}}| + |N_{U_{i+1}}| = |R'_{U_{i+1}}| + |N_{U_{i+1}}|$$

This inequality holds for all $i \in [m-1]$. So for each $i \in [m-1]$, we get

$$\begin{aligned} |R'_{U_i}| &\leq |R'_{U_{i+1}}| + |N_{U_{i+1}}| \\ &\leq |R'_{U_{i+2}}| + |N_{U_{i+2}}| + |N_{U_{i+1}}| \\ &\leq \cdots \\ &\leq |R'_{U_m}| + \sum_{j=i+1}^{m} |N_{U_j}| \leq \mathrm{OUT} + \mathrm{IN} \end{aligned}$$

□

Theorem 5. *Including the time to find a signed-elimination sequence, Algorithm 7 evaluates a full signed-acyclic* CQ *in* $O(\phi^3 + \phi \cdot \mathrm{IN} + \phi \cdot \mathrm{OUT})$ *time.*

Proof. We consider the non-recursive cost incurred by Algorithm 7 on a signed-leaf v, which consists of three parts: (i) running REDUCE to eliminate v from the input $\mathsf{CQ}^\neg$ $\mathcal{Q}_1$; (ii) iteratively computing intermediate results $R'_{U_1}, R'_{U_2}, \ldots, R'_{U_m}$; and (iii) computing $\mathcal{Q}_1$ as $\mathcal{Q}_2 \bowtie R'_{U_m}$. We analyze these parts separately.

(i) This takes $O(d(v) \cdot \text{IN})$ time (directly from the proof of Theorem 3).

(ii) For each $i \in [m]$, R'_{U_i} is computed as

$$R'_{U_i} = \left(\pi_{U_i - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_{i-1}} - N_{U_i}$$

Recall from Section 2.2 that (a) given any two relations R_1, R_2, the cost to compute $R_1 \bowtie R_2$ is $O(\min\{|R_1|, |R_2|\} + |R_1 \bowtie R_2|)$; and (b) given any two relations R, N with $\text{var}(N) \subseteq \text{var}(R)$, the cost to compute $R \triangleright N$ is $O(|R|)$. Because difference is a special form of antijoin, the cost of computing R'_{U_i} is dominated by the join $\left(\pi_{U_i - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_{i-1}}$. By Lemma 13 (ii) and Lemma 14, we can bound the join size as

$$|\left(\pi_{U_i - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_{i-1}}| \leq |R'_{U_i}| + |N_{U_i}| \leq 2 \cdot \text{IN} + \text{OUT}$$

which then implies that the join takes time

$$O\left(|R'_{U_{i-1}}| + |\left(\pi_{U_i - \{v\}} \mathcal{Q}_2\right) \bowtie R'_{U_{i-1}}|\right) = O(3 \cdot \text{IN} + 2 \cdot \text{OUT}) = O(\text{IN} + \text{OUT})$$

Last but not least, we have $m \leq d(v)$. Hence, the total cost to compute these intermediate results $R'_{U_1}, \ldots, R'_{U_m}$ is $O(d(v) \cdot (\text{IN} + \text{OUT}))$.

(iii) Computing $\mathcal{Q}_1 = \mathcal{Q}_2 \bowtie R'_{U_m}$ similarly takes time $O(|\mathcal{Q}_1| + |\mathcal{Q}_2|) = O(\text{OUT})$.

Hence, the non-recursive cost to handle a signed-leaf v is $O(d(v) \cdot (\text{IN} + \text{OUT}))$. Summing up such cost across all attributes gives $O\left(\sum_v d(v) \cdot (\text{IN} + \text{OUT})\right) = O(\phi \cdot \text{IN} + \phi \cdot \text{OUT})$.

Lastly, a signed-elimination sequence can be found in $O(\phi^3)$ time with a brute-force approach [48]. We then arrive at the claimed combined complexity of Algorithm 7. □

7.2 An Aggregation Algorithm for Free-Connex $\mathsf{CQ}^\neg$

So far, we have only focused on full signed-acyclic $\mathsf{CQ}^\neg$. Specifically, given a full signed-acyclic $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$, we have that (i) Algorithm 7 computes the result of $\mathcal{Q}$; and (ii)

Algorithm 6 computes the aggregation $\bigoplus_{\text{var}(\mathcal{Q})-W} \mathcal{Q}$ for any $W \in \mathcal{E}^+$. Our next question is, to what extent can we incorporate projection? Concretely, when given some attributes $F \subseteq \text{var}(\mathcal{Q})$, can we *efficiently*[1] compute the result of $\pi_F \mathcal{Q}$, or more generally, the aggregation $\bigoplus_{\text{var}(\mathcal{Q})-F} \mathcal{Q}$?

In the case of CQ, *free-connexity* defines a class of CQ that admits efficient evaluation algorithms [9]. This notion has also been extended to $\mathsf{CQ}^\neg$ [48].

Definition 12. *Let $\mathcal{Q} = (F, \mathcal{E}^+, \mathcal{E}^-)$ be a signed-acyclic $\mathsf{CQ}^\neg$. We say $\mathcal{Q}$ is free-connex, if the signed-hypergraph $\mathcal{H}' = (\text{var}(\mathcal{Q}^\neg), \mathcal{E}^+, \mathcal{E}^- \cup \{F\})$ is signed-acyclic.*

In fact, [48]'s algorithm works for free-connex $\mathsf{CQ}^\neg$ with aggregation. Under our setting of counting ring, their algorithm enumerates all output tuples of a free-connex $\mathsf{CQ}^\neg$ along with their degrees in the associated full $\mathsf{CQ}^\neg$.

Theorem 6 ([48])**.** *Let $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$ be a full signed-acyclic $\mathsf{CQ}^\neg$, and $F \subseteq \text{var}(\mathcal{Q})$ be a set of attributes, such that $\mathcal{Q}_F := (F, \mathcal{E}^+, \mathcal{E}^-)$ is free-connex.*

Then, $\bigoplus_{\text{var}(\mathcal{Q})-F} \mathcal{Q} = \{(t \mapsto \deg(\mathcal{Q}, t)) : t \in \mathcal{Q}_F\}$ can be enumerated with $O(\phi^3 + \phi \cdot \text{IN})$ pre-processing and $O(\phi)$ delay; subsequently, $\bigoplus_{\text{var}(\mathcal{Q})-F} \mathcal{Q}$ can be computed in time $O(\phi^3 + \phi \cdot \text{IN} + \phi \cdot \text{OUT})$, where $\text{OUT} := |\mathcal{Q}_F|$.

We propose a new algorithm, Algorithm 8, to compute $\bigoplus_{\text{var}(\mathcal{Q})-F} \mathcal{Q}$. First, notice that the definition of free-connexity is closely connected to Lemma 2: $\mathcal{Q}_F = (F, \mathcal{E}^+, \mathcal{E}^-)$ being free-connex implies that there exists a signed-elimination sequence σ of $\mathcal{Q}$ that leaves attributes in F till the last.

Our new algorithm builds upon such σ, using our previously developed algorithms as primitives: (i) use Reduce (Algorithm 2) to remove any non-output attribute v, i.e. $v \notin F$, from $\mathcal{Q}$ to obtain $\mathcal{Q}_F$; (ii) use EvaluateFull (Algorithm 7) to materialize $\mathcal{Q}_F$; (iii) add a new positive hyperedge F to $\mathcal{Q}$ with associated relation $R_F := \mathcal{Q}_F$, then use BuildAndCountHDT (Algorithm 6) with $W := F$ to obtain the aggregation $\bigoplus_{\text{var}(\mathcal{Q})-F} \mathcal{Q}$. We remark that (i) and (iii) can be combined, as both of them can go through the same elimination steps to remove any attribute $v \notin F$. Doing so leads to a modified version of BuildAndCountHDT; however, because the overall combined complexity of the algorithm remains unaffected, for simplicity, we only present the implementation that follows exactly (i)-(iii) as described above.

[1] We want efficiency in terms of linear time, i.e. $O(\text{IN} + \text{OUT})$ data complexity. Here OUT here is the size of the query result, i.e. $\text{OUT} = |\pi_F \mathcal{Q}|$ instead of $|\mathcal{Q}|$. In theory, we could have a much smaller $|\pi_F \mathcal{Q}|$ than $|\mathcal{Q}|$.

Algorithm 8 AGGREGATEFREECONNEX($\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$, F)

Input: full signed-acyclic $\mathsf{CQ}^\neg$ $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$, attributes $F \subseteq \text{var}(\mathcal{Q})$ such that $\mathcal{Q}_F := (F, \mathcal{E}^+, \mathcal{E}^-)$ is free-connex

Output: aggregation $\bigoplus_{\text{var}(\mathcal{Q})-F} \mathcal{Q}$

1: $\sigma \leftarrow$ find any signed-elimination sequence of $\mathcal{Q}$ that leaves attributes in F till the end
2: $\mathcal{Q}_F \leftarrow \mathcal{Q}$ ▷ $\mathsf{CQ}^\neg$ to be iteratively reduced until it only contains output-attributes.
3: **for all** $v \in \sigma, v \notin F$ **do** $\mathcal{Q}_F \leftarrow$ REDUCE($\mathcal{Q}_F$, v) **end for**
4: $\sigma_F \leftarrow$ suffix of σ with only attributes in F
5: $R_F \leftarrow$ EVALUATEFULL($\mathcal{Q}_F$, σ_F) ▷ Materialize $\mathcal{Q}_F$ as a new relation.
6: $\mathcal{T}, \{S_W\}_{W \in N(\mathcal{T})} \leftarrow$ BUILDANDCOUNTHDT($\mathcal{Q}' := (\mathcal{E}^+ \cup \{F\}, \mathcal{E}^-)$, F)
7: **return** S_F ▷ The root of $\mathcal{T}$ is F, which stores the final aggregation result.

The correctness of Algorithm 8 builds upon the correctness of its primitives and the fact that $\mathcal{Q} = \mathcal{Q} \bowtie (\pi_F \mathcal{Q})$, i.e. we can join $\mathcal{Q}$ with a projection of itself without affecting the join result.

Theorem 7. *Given a full signed-acyclic* $\mathsf{CQ}^\neg$ $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-)$ *and attributes* $F \subseteq \text{var}(\mathcal{Q})$ *such that the* $\mathsf{CQ}^\neg$ $\mathcal{Q}_F := (F, \mathcal{E}^+, \mathcal{E}^-)$ *is free-connex, Algorithm 8 computes* $\bigoplus_{\text{var}(\mathcal{Q})-F} \mathcal{Q}$ *in* $O(\phi^3 + \phi \cdot \text{IN} + \phi \cdot \text{OUT})$ *time, where* $\text{OUT} := |\mathcal{Q}_F|$.

Proof. As per Theorem 3 and Theorem 5, everything up to Line 6 takes $O(\phi^3 + \phi \cdot \text{IN} + \phi \cdot \text{OUT})$ time.

In Line 6, we pass on to our counting algorithm a new $\mathsf{CQ}^\neg$ $\mathcal{Q}'$ that contains one additional positive hyperedge F with $|R_F| = \text{OUT}$, when compared to the input $\mathsf{CQ}^\neg$ $\mathcal{Q}$. The formula complexity and the input size of $\mathcal{Q}'$ are therefore at most $2 \cdot \phi$ and $\text{IN} + |R_F| = \text{IN} + \text{OUT}$, respectively, where ϕ and IN are complexity measures defined on $\mathcal{Q}$. By Theorem 4, Line 6 takes $O((2\phi)^3 + (2\phi) \cdot (\text{IN} + \text{OUT})) = O(\phi^3 + \phi \cdot \text{IN} + \phi \cdot \text{OUT})$ time. □

To conclude, Algorithm 8 attains the same combined complexity as [48]. Moreover, as it builds upon primitives that can be expressed as SQL statements, Algorithm 8 can also be expressed as SQL statements.

7.3 Difference of CQ

Lastly, we look at a different class of conjunctive queries that is closely related to $\mathsf{CQ}^{\neg}$, termed *difference of multiple conjunctive queries* (DMCQ). This section is not directly related to the algorithms developed in the earlier sections and can be read on its own; however, it shares a similar philosophy: (i) we identify a gap between theory and practice in a previous technique; (ii) we develop a technique based on a similar idea of signed query partitioning idea as before; and (iii) building on our result on signed-acyclic $\mathsf{CQ}^{\neg}$, the developed technique can also be converted into SQL statements with theoretical guarantees. Our results augment a prior work [30] on DMCQ.

A DMCQ takes the form $\mathcal{Q}^{-} = \mathcal{Q}_0 - \mathcal{Q}_1 - \cdots - \mathcal{Q}_m$, where $m \geq 0$ is an integer and $\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_m$ are full CQs such that $\mathrm{var}(\mathcal{Q}_0) = \mathrm{var}(\mathcal{Q}_1) = \cdots = \mathrm{var}(\mathcal{Q}_m)$. We are interested in the problem of *efficiently* evaluating $\mathcal{Q}^{-}$. An immediate solution, which is indeed the approach of existing commercial database systems, is to materialize each $\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_m$ separately and apply sequentially the difference operators. When any of these $\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_m$ is large while $\mathcal{Q}^{-}$, the final result, is at a much smaller scale, the materialization of these $\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_m$ seems unnecessarily wasteful.

[30] is the first to study the problem of DMCQ evaluation. Their high-level strategy is to decompose the DMCQ into a union of subqueries and study how and when each subquery can be evaluated efficiently, with a primary focus on the case of $m = 1$.

Lemma 15 ([30])**.** *Let $\mathcal{Q}_i = \mathcal{E}_i$ be full CQ for each $i \in \{0\} \cup [m]$, with $\mathrm{var}(\mathcal{Q}_0) = \mathrm{var}(\mathcal{Q}_1) = \cdots = \mathrm{var}(\mathcal{Q}_m)$. Let R_{i,V_i} be the relation associated to hyperedge $V_i \in \mathcal{E}_i$ in query $\mathcal{Q}_i$. Then*

$$\mathcal{Q}_0 - \mathcal{Q}_1 - \cdots - \mathcal{Q}_m = \bigcup_{(V_1,\ldots,V_m) \in \mathcal{E}_1 \times \cdots \times \mathcal{E}_m} \left(\mathcal{Q}_0 \triangleright R_{1,V_1} \triangleright \cdots \triangleright R_{m,V_m} \right)$$

At this point, we know that $\mathcal{Q}_0 \triangleright R_{1,V_1} \triangleright \cdots \triangleright R_{m,V_m}$ is nothing special but a full $\mathsf{CQ}^{\neg}$; in addition, the union implies that $|\mathcal{Q}_0 \triangleright R_{1,V_1} \triangleright \cdots \triangleright R_{m,V_m}| \leq |\mathcal{Q}_0 - \mathcal{Q}_1 - \cdots - \mathcal{Q}_m| =: \mathrm{OUT}$. Therefore, we can express this result in terms of signed-acyclicity.

Theorem 8 ([30, 48])**.** *Consider any DMCQ $\mathcal{Q}^{-} = \mathcal{Q}_0 - \mathcal{Q}_1 - \cdots - \mathcal{Q}_m$ with $\mathcal{Q}_0 = \mathcal{E}, \mathcal{Q}_1 = \mathcal{E}_1, \ldots, \mathcal{Q}_m = \mathcal{E}_m$. If for all $(V_1, \ldots, V_m) \in \mathcal{E}_1 \times \cdots \times \mathcal{E}_m$, the $\mathsf{CQ}^{\neg}$ $\mathcal{Q}^{\neg} := (\mathcal{E}, \{V_i\}_{i=1}^{m})$ is signed-acyclic, then $\mathcal{Q}^{-}$ can be evaluated in $O(\mathrm{IN} + \mathrm{OUT})$ data complexity, under the assumption that taking the union does not affect the complexity.*

Based on the assumed computational model from Section 2.2, taking the union $R_V \cup S_V$ of two relations R_V, S_V can be done in time $O(|R_V| + |S_V|)$, which would not affect any

linear time algorithm. Nevertheless, from a practical perspective, the Union operator could pose a bottleneck: when these subqueries produce a large number of results, de-duplication is typically implemented via merge sort (instead of hash-based methods), causing a non-negligible logarithmic factor in the runtime.

We propose a conceptually simple fix — instead of decomposing the DMCQ into a union of subqueries, we decompose it into a *disjoint* union of subqueries; as a result, the potentially expensive de-duplication is avoided.

Lemma 16. *Let $\mathcal{Q}^- = \mathcal{Q}_0 - \mathcal{Q}_1 - \cdots - \mathcal{Q}_m$ be a* DMCQ*, and $\{R_{i,1}, R_{i,2}, \ldots, R_{i,n_i}\}$ be an arbitrary order of relations in $\mathcal{Q}_i$, for all $i \in [m]$. Then*

$$\mathcal{Q}^- = \biguplus_{(s_1,\ldots,s_m)\in[n_1]\times\cdots\times[n_m]} \left(\mathcal{Q}_0 \bowtie \left(\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}\right) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}\right)$$

Proof. Since $\text{var}(R_{i,j}) \subseteq \text{var}(\mathcal{Q}_i) = \text{var}(\mathcal{Q}_0)$ for every i, j, all queries are over the same attributes, hence the equation is well-defined. First, notice that

$$\mathcal{Q}_0 \bowtie \left(\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}\right) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m} \subseteq \mathcal{Q}_0 \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}$$

Therefore, by Lemma 15,

$$\begin{aligned}
&\bigcup_{(s_1,\ldots,s_m)\in[n_1]\times\cdots\times[n_m]} \left(\mathcal{Q}_0 \bowtie \left(\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}\right) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}\right)\\
&\subseteq \bigcup_{(s_1,\ldots,s_m)\in[n_1]\times\cdots\times[n_m]} \left(\mathcal{Q}_0 \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}\right)\\
&= \mathcal{Q}^-
\end{aligned}$$

Next, consider any $t \in \mathcal{Q}^-$. Clearly, $t \in \mathcal{Q}_0$, and $t \notin \mathcal{Q}_i$ for each $i \in [m]$. For every $i \in [m]$, the latter implies that there must exists some relation $R_{i,j}$ in $\mathcal{Q}_i$, such that $t \bowtie R_{i,j} = \emptyset$. Let $s_i \in [n_i]$ be the smallest choice of such j; hence, $t \bowtie R_{i,j} \neq \emptyset$ for all $j \in [s_i - 1]$ and $t \bowtie R_{i,s_i} = \emptyset$. Consequently, we get $t \in \mathcal{Q}_0 \bowtie (\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}$, which implies

$$\mathcal{Q}^- \subseteq \bigcup_{(s_1,\ldots,s_m)\in[n_1]\times\cdots\times[n_m]} \left(\mathcal{Q}_0 \bowtie \left(\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}\right) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}\right)$$

Finally, consider two tuples $t_1, t_2 \in \mathcal{Q}^-$ such that their respective choices of indices $(s_1, \ldots, s_m)$, $(s'_1, \ldots, s'_m)$, as per the procedure above, are distinct. Without loss of generality, let $j \in [m]$ be such that $s_j < s'_j$. Then, $t_1 \bowtie R_{j,s_j} = \emptyset$ while $t_2 \bowtie R_{j,s_j} \neq \emptyset$, which implies $t_1 \neq t_2$. Therefore, the subqueries $\mathcal{Q}_0 \bowtie (\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}$ defined over distinct array of indices $(s_1, \ldots, s_m)$ are disjoint, as desired. □

The signed-acyclicity of $\mathcal{Q}_0 \bowtie \left(\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}\right) \triangleright (\triangleright_{i=1}^{m} R_{i,s_i})$ immediately implies an efficient evaluation algorithm. Furthermore, the disjoint union grants that

$$|\mathcal{Q}^-| = \sum_{(s_1,\ldots,s_m) \in [n_1] \times \cdots \times [n_m]} \left| \mathcal{Q}_0 \bowtie \left(\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j} \right) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m} \right|$$

Hence, the signed-acyclicity also implies an efficient counting algorithm.

Alternatively, instead of evaluating each subquery $\mathcal{Q}_0 \bowtie \left(\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}\right) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}$ as a whole, we can first evaluate $\mathcal{Q}_0 \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}$ and filter the result with the positive relations $\bigcup_{i=1}^{m} \bigcup_{j=1}^{s_j-1} \{R_{i,j}\}$. Due to Lemma 15 and Lemma 16, neither approach produces an intermediate result with a size exceeding $\text{OUT} = |\mathcal{Q}^-|$. Therefore, the signed-acyclicity of either $\mathcal{Q}_0 \bowtie \left(\bowtie_{i=1}^{m} \bowtie_{j=1}^{s_i-1} R_{i,j}\right) \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}$ or $\mathcal{Q}_0 \triangleright R_{1,s_1} \triangleright \cdots \triangleright R_{m,s_m}$ implies a linear evaluation time under data complexity.

There remain two caveats. First, as the subqueries of Lemma 16 are dependent on the ordering of hyperedges, can some ordering yield only signed-acyclic subqueries while another ordering yields some signed-cyclic subqueries? Second, how do the hypotheses of Lemma 15 and Lemma 16 compare with each other?

Example 14. *Consider the* DMCQ $\mathcal{Q}^- = \mathcal{Q}_0 - \mathcal{Q}_1$*, where* $\mathcal{Q}_0 = R_1(A, C) \bowtie R_2(B, C) \bowtie R_3(B, D)$ *and* $\mathcal{Q}_1 = R_4(A, B, D) \bowtie R_5(A, B, C)$*. According to Lemma 16,* $\mathcal{Q}^-$ *can be decomposed into either* $\mathcal{Q}^- = (\mathcal{Q}_0 \triangleright R_4) \uplus (\mathcal{Q}_0 \bowtie R_4 \triangleright R_5)$*, or* $\mathcal{Q}^- = (\mathcal{Q}_0 \triangleright R_5) \uplus (\mathcal{Q}_0 \bowtie R_5 \triangleright R_4)$*.*

Both subqueries in the second decomposition are signed-acyclic. Nevertheless, neither $\mathcal{Q}_0 \triangleright R_4$ *nor* $\mathcal{Q}_0 \bowtie R_4 \triangleright R_5$ *is signed-acyclic.*

On the other, Lemma 15 decomposes $\mathcal{Q}^-$ *into* $\mathcal{Q}^- = (\mathcal{Q}_0 \triangleright R_4) \bowtie (\mathcal{Q}_0 \triangleright R_5)$*, and* $\mathcal{Q}_0 \triangleright R_4$ *is not signed-acyclic.*

The above example shows that different orderings could lead to subqueries of different difficulties (i.e. signed-acyclic vs. signed-cyclic). Moreover, it illustrates an example in which the hypothesis of Lemma 16 holds, whereas the hypothesis of Lemma 15 does not; hence, one might wonder whether the former is a weaker condition than the latter. The next example gives a counterexample.

Example 15. *Consider the* DMCQ $\mathcal{Q}^- = \mathcal{Q}_0 - \mathcal{Q}_1$*, where* $\mathcal{Q}_0 = R_1(A,B) \bowtie R_2(C)$ *and* $\mathcal{Q}_1 = R_3(A,C) \bowtie R_4(B,C)$*. Then, Lemma 15 decomposes* $\mathcal{Q}^-$ *into* $\mathcal{Q}^- = (\mathcal{Q}_0 \triangleright R_3) \cup (\mathcal{Q}_0 \triangleright R_4)$*, where both subqueries are signed-acyclic.*

On the other hand, Lemma 16 decomposes $\mathcal{Q}^-$ *into either* $\mathcal{Q}^- = (\mathcal{Q}_0 \triangleright R_3) \uplus (\mathcal{Q}_0 \bowtie R_3 \triangleright R_4)$ *or* $\mathcal{Q}^- = (\mathcal{Q}_0 \triangleright R_4) \uplus (\mathcal{Q}_0 \bowtie R_4 \triangleright R_3)$*, where the second subquery in each case is not signed-acyclic.*

Now, we see that the two hypotheses are incomparable. Putting everything together gives the following results, which improve Theorem 8 by strictly relaxing its hypothesis and removing the need for de-duplication.

Theorem 9. *Consider any* DMCQ $\mathcal{Q}^- = \mathcal{Q}_0 - \mathcal{Q}_1 - \cdots - \mathcal{Q}_m$ *with* $\mathcal{Q}_0 = \mathcal{E}, \mathcal{Q}_1 = \mathcal{E}_1, \ldots, \mathcal{Q}_m = \mathcal{E}_m$*. We define two conditions:*

(i) For all $(V_1, \ldots, V_m) \in \mathcal{E}_1 \times \cdots \times \mathcal{E}_m$*,* $\mathcal{Q} := (\mathcal{E}, \{V_i\}_{i=1}^m)$ *is signed-acyclic.*

(ii) There exists an ordering $\mathcal{E}_i = \{V_{i,1}, V_{i,2}, \ldots, V_{i,n_i}\}$ *for all* $i \in [m]$*, such that for all* $(s_1, \ldots, s_m) \in [n_1] \times \cdots \times [n_m]$*,* $\mathcal{Q} := \left(\mathcal{E} \cup \bigcup_{i=1}^m \{V_{i,j}\}_{j=1}^{s_i - 1}, \{V_{i,s_i}\}_{i=1}^m\right)$ *is signed-acyclic.*

Then,

- *If either (i) or (ii) holds,* $\mathcal{Q}^-$ *can be evaluated in time* $O(\mathrm{IN} + \mathrm{OUT})$ *under data complexity.*
- *If (ii) holds,* $|\mathcal{Q}^-|$ *can be counted in time* $O(\mathrm{IN})$ *under data complexity.*

Chapter 8

Conclusion and Future Directions

Efficient join processing has been actively researched by both theory and practice communities; meanwhile, it remains unknown whether there exist algorithms for antijoin processing that not only enjoy theoretically-bounded runtime but also admit practical implementation. Our thesis makes the first step towards such goal by proposing a new suite of algorithms with both theoretical and practical significance — on the one hand, the runtime of all our new algorithms match the state-of-the-art theory result on antijoin processing; on the other hand, our new algorithms use standard relational algebra, making them possible to be adopted into any practical database systems via SQL-rewriting. To further show that this direction is promising, we implement a prototype SQL-rewriter for our new counting algorithm. Subsequent empirical evaluation on real-life graph datasets shows substantial performance improvement of our approach over vanilla SQL queries.

We conclude the thesis with some potential future directions:

1. Our new counting algorithm can be viewed as analogous to the seminal Yannakakis algorithm for join processing, both operating on a rooted tree structure (i.e. HDT vs. join tree). Join trees see their applications in various aspects of join processing, including counting, evaluation, and enumeration of results of a join query; nevertheless, our newly proposed HDT is only used for counting the results of an antijoin query. We wonder whether it is possible to similarly develop evaluation and enumeration algorithms based on HDT.

2. We propose new counting, evaluation, and aggregation algorithms for antijoin processing; however, our experiment only covers the counting algorithm on some graph

datasets. To enhance the scope of empirical evaluation, we plan to similarly implement SQL-rewriting prototypes for our evaluation and aggregation algorithms, as well as extend the experiment to cover additional analytical workloads, for instance, the TPC-H benchmark [5]. Moreover, our current SQL-rewriter for our counting algorithm is still prototypical, which leaves a large room for optimizations.

3. All our new algorithms target signed-acyclic (free-connex) antijoin queries. Because there are already theoretical results that extend the scope of antijoin processing beyond signed-acyclicity [33], an intriguing future direction is to incorporate these results and improve our algorithms to handle signed-cyclic antijoin queries, via techniques such as *hypertree decomposition* from join processing [26].

References

[1] https://duckdb.org/, DuckDB.

[2] https://www.postgre.org/, PostgreSQL.

[3] https://snap.stanford.edu/snap/, SNAP.

[4] https://spark.apache.org/sql/, SparkSQL.

[5] https://www.tpc.org/tpch/, TPC-H.

[6] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28, 2016.

[7] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 429–444, 2017.

[8] A Atserias, M Grohe, and D Marx. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, 2008.

[9] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

[10] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *Proc. VLDB Endow.*, 18(8):2413–2426, 2025.

[11] Altan Birler, Alfons Kemper, and Thomas Neumann. Robust join processing with diamond hardened joins. *Proceedings of the VLDB Endowment*, 17(11):3215–3228, 2024.

[12] Johann Brault-Baron. A negative conjunctive query is easy if and only if it is beta-acyclic. In *Computer Science Logic (CSL'12)-26th International Workshop/21st Annual Conference of the EACSL (2012)*, pages 137–151. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012.

[13] Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.

[14] Johann Brault-Baron. Hypergraph acyclicity revisited. *ACM Computing Surveys (CSUR)*, 49(3):1–26, 2016.

[15] Johann Brault-Baron, Florent Capelli, and Stefan Mengel. Understanding model counting for β-acyclic cnf-formulas. *arXiv preprint arXiv:1405.6043*, 2014.

[16] Florent Capelli, Nofar Carmeli, Oliver Irwin, and Sylvain Salvati. Direct access for conjunctive queries with negations. *arXiv preprint arXiv:2310.15800*, 2023.

[17] Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. *ACM Transactions on Database Systems (TODS)*, 46(2):1–41, 2021.

[18] Bernard Chazelle and Burton Rosenberg. The complexity of computing partial sums off-line. *International Journal of Computational Geometry & Applications*, 1(01):33–45, 1991.

[19] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[20] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[21] Shaleen Deep, Hangdong Zhao, Austen Z Fan, and Paraschos Koutris. Output-sensitive conjunctive query evaluation. *Proceedings of the ACM on Management of Data*, 2(5):1–24, 2024.

[22] David Duris. Some characterizations of γ and β-acyclicity of hypergraphs. *Information Processing Letters*, 112(16):617–620, 2012.

[23] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM (JACM)*, 30(3):514–550, 1983.

[24] 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.

[25] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. Structure-guided query evaluation: Towards bridging the gap from theory to practice. *arXiv preprint arXiv:2303.02723*, 2023.

[26] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32, 1999.

[27] Marc H. Graham. On the universal relation. Technical report, University of Toronto, 1979.

[28] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, 2007.

[29] Xiao Hu. Output-optimal algorithms for join-aggregate queries. *Proceedings of the ACM on Management of Data*, 3(2):1–27, 2025.

[30] Xiao Hu and Qichen Wang. Computing the difference of conjunctive queries efficiently. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.

[31] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 91–106, 2016.

[32] Paraschos Koutris, Shaleen Deep, Austen Fan, and Hangdong Zhao. The quest for faster join algorithms (invited talk). In *28th International Conference on Database Theory (ICDT 2025)*, pages 1–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.

[33] Matthias Lanzinger. Tractability beyond β-acyclicity for conjunctive queries with negation and sat. *Theoretical Computer Science*, 942:276–296, 2023.

[34] Thomas Neumann. Closing the gap between theory and practice in query optimization. In *Companion of the 43rd Symposium on Principles of Database Systems*, pages 4–4, 2024.

[35] Hung Q Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 111–124, 2018.

[36] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48, 2012.

[37] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *Acm Sigmod Record*, 42(4):5–16, 2014.

[38] Dan Olteanu. Recent increments in incremental view maintenance. In *Companion of the 43rd Symposium on Principles of Database Systems*, pages 8–17, 2024.

[39] Dan Olteanu and Maximilian Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

[40] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[41] Moshe Y Vardi. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146, 1982.

[42] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.

[43] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees. *arXiv preprint arXiv:2504.03279*, 2025.

[44] Qichen Wang and Ke Yi. Conjunctive queries with comparisons. In *Proceedings of the 2022 International Conference on Management of Data*, pages 108–121, 2022.

[45] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. Predicate transfer: Efficient pre-filtering on multi-join queries. In *14th Conference on Innovative*

Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024. www.cidrdb.org, 2024.

[46] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

[47] Clement Tak Yu and Meral Z Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.*, pages 306–312. IEEE, 1979.

[48] Hangdong Zhao, Austen Z Fan, Xiating Ouyang, and Paraschos Koutris. Conjunctive queries with negation and aggregation: A linear time characterization. *Proceedings of the ACM on Management of Data*, 2(2):1–19, 2024.

[49] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.