On the Complexity of Queries with Intersection Joins

Antonia Kormpa

Exeter College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2022

Στον Γιώργο

Acknowledgements

I am grateful to Prof. Michael Benedikt, for providing me with advice on this thesis, and for the invaluable encouragement he gave me during the last years of my studies. In addition, I am grateful to Prof. Dan Olteanu and Mahmoud Abo Khamis for the time they invested in meetings and for the fruitful discussions. I truly value the advice and support I received from Stefano Cavazzi, and the funding provided by Ordnance Survey. I would also like to express my sincere gratitute to Prof. Jared Tanner from Exeter College for all the great support I received. Last but not least, I would like to thank my thesis committee, and the University of Oxford.

Saying thank you to my family, including my parents, my siblings and my uncle, is not enough to express my gratitude to them for loving me and supporting me during my whole life. Vivi and Aristea, apart from the above, I will always remember the cheerful, endless video calls that we had during my time at Oxford. They were making me much happier! It goes without saying that I am a fortunate person to have Yiorgos, always by my side during this challenging journey. I would like to thank him for all the understanding, and support. Finally, I cannot omit to say thank you to all my friends for making my days with chats, walks, drinks, and more.

Abstract

This thesis studies the complexity of join processing on interval data. It defines a class of queries, called Conjunctive Queries with Intersections Joins (IJQs). An IJQ is a query in which the variables range both over scalars and intervals with real-valued endpoints. The joins are expressed through intersection predicates; an intersection predicate over a multi-set that consists of both scalars and intervals is a true assertion, if the elements in the multi-set intersect; otherwise, it is false. The class of IJQs includes queries that are often asked in practice.

This thesis introduces techniques for obtaining reductions from the problem of evaluating IJQs to the problem of evaluating Conjunctive Queries with Equality Joins (CQs). The key idea is the rewriting of an intersection predicate over a set of intervals into an equivalent predicate with equality conditions. This rewriting is achieved by building a segment tree where the nodes hierarchically encode intervals using bit-strings. Given a multi-set of intervals, their intersection is captured by certain equality conditions on the encoding of the nodes. Following that, it turns out that the problem of evaluating an IJQ on an input database containing intervals can be reduced to the problem of evaluating a union of CQs on a database containing scalars and vice versa. Such reductions lead to upper and lower bounds for the data complexity of Boolean IJQs, given upper and lower bounds for the data complexity Boolean CQs. The upper bounds are obtained using a reduction called forward reduction, which reduces any Boolean IJQ to a disjunction of Boolean CQs. The lower bounds are obtained by a reduction called backward reduction, in which any Boolean CQ from the aforementioned disjunction is reduced to the input Boolean IJQ. Overall, the two findings suggest that a Boolean IJQ is as difficult as the forward disjunctions' most difficult Boolean CQ. Last but not least, this thesis identifies an interesting subclass of Boolean IJQs that admit quasi-linear time computation in data complexity. They are referred to as ι-acyclic IJQs.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The amount of spatio-temporal data produced by various sources, such as phones, cars, and user-based applications, has increased significantly in recent years [14, 26]. Large spatio-temporal databases have also emerged as a result of advancements in mapping and navigation technologies [14]. Complex data from such sources are typically represented and approximated by intervals [6, 57].

This thesis studies the complexity of join query processing on interval data. In the literature, the join operation on intervals is referred to by a variety of names, including spatial join, intersection join, temporal join, and interval join [34, 40, 27]. In this thesis, the term intersection join is used to refer to any of the terms mentioned above. Furthermore, a database is considered to be an extension of a relational database that stores both scalars and intervals with real-valued endpoints. A Conjunctive Query with Intersection Joins (IJQ) applied on a database returns the tuples from the database which satisfy the intersection join predicates in the query. An intersection join predicate over a multi-set consisting of intervals is a true assertion if the elements in the multi-set intersect; otherwise, it is false. IJQs are often asked in spatio-temporal databases [6, 57], because such databases store objects that can be represented or approximated by intervals, such as road networks, hydrography, woodland areas, urban areas, timestamps, or time intervals. An IJQ applied on a spatio-temporal database asks for tuples that describe objects that intersect in terms of space, or they co-occur in terms of time.

This thesis investigates the complexity of IJQs by introducing techniques for obtaining reductions from the problem of evaluating IJQs to the problem of evaluating Conjunctive Queries with Equality Joins (CQs). The main idea is the rewriting of an intersection predicate over a multi-set of intervals into an equivalent predicate with equality conditions. This rewriting is achieved by building a segment tree whose nodes hierarchically encode intervals using bit-strings. Given a set of intervals, their intersection is captured by certain equality conditions on the encoding of the nodes. Following that, it turns out that the problem of evaluating an IJQ on a database which contains intervals can be reduced to the problem of evaluating a union of CQs on a database which contains scalars. Using the equivalence of the two predicates as building block, one can derive upper and lower bounds on the complexity of IJQs by reducing the intersection join problem to an equality join problem and vice versa.

1.1 Motivation

The motivation for this thesis stems from the lack of theoretical understanding on the complexity of Boolean IJQs, as well as the substantial progress in understanding the complexity of CQs.

Lack of General and Efficient Approaches

The class of IJQs includes queries that are often asked in databases and have practical applications. There exists a variety of intersection join algorithms developed in the context of temporal [27], and spatial databases [40, 34]. Most of them compute two-way intersection joins, whereas others compute multi-way intersection joins by combining two-way intersection joins [41]. Two-way intersection join algorithms can be classified into the following categories: plane-sweep algorithms [7, 50]; index-based algorithms [42, 38, 17, 24], and partitioning-based algorithms [47, 39, 37]. Combining two-way joins to solve a multi-way intersection join, has the drawback of producing large intermediate results in certain database instances, leading to sub-optimal running times. This is analogous to the suboptimal complexity of combining two-way equality joins to solve queries with multi-way equality joins [45]. A multi-way intersection join algorithm that avoids producing large intermediate results have been developed recently by Hu et al. [33], who investigated a special case of the IJQ problem, taking into account CQs with a single multi-way intersection join that involves every relation in the database. They proposed an approach based on worst-case optimal join algorithms [45]. However, this approach is not general enough to be used for solving IJQs with arbitrary intersection joins, as they are defined later in this thesis.

Another approach that provides an efficient solution for IJQs is to express intersection joins as inequalities, and then use efficient algorithms that solve CQs with inequalities. In particular, an IJQ can be rewritten as a disjunction of CQs with inequalities, where the non-empty intersection of two intervals can be determined by comparing their starting and ending points. In particular, given two intervals $[x.\mathsf{start}, x.\mathsf{end}]$ and $[y.\mathsf{start}, y.\mathsf{end}]$ one can check if they intersect by using the expression $(x.\mathsf{start} \leq y.\mathsf{start} \leq x.\mathsf{end}) \vee (y.\mathsf{start} < x.\mathsf{start} \leq y.\mathsf{end})$. This observation is formally expressed and expanded for more than two intervals in Section 4.2, Remark 4.2.11. The drawback of this approach is that there are certain IJQs for which even efficient algorithms for solving queries with inequality joins, such as FAQ-AI [2], do not guarantee optimality. To provide evidence, this thesis introduces an algorithm that solves the Boolean Triangle IJQ in sub-quadratic data complexity in the worst case, whereas FAQ-AI, using the above encoding with inequalities, solves the same query in quadratic data complexity times a polylogarithmic factor [36].

Progress on Understanding the CQ Complexity

Despite the significant progress in understanding the complexity of CQs in recent decades, the data complexity of IJQs is still not fully understood. A few findings that improved our understanding of the data complexity of CQs are reviewed in the following. The α-acyclic Boolean CQs are decidable in linear time, and the α-acyclic full CQs can be computed in linear time preprocessing, followed by a constant delay enumeration of the tuples in the output [58]. Boolean CQs Q can be computed in $\tilde{\mathcal{O}}(N^{\mathsf{subw}(Q)})$ time, where $\mathsf{subw}(Q)$ denotes the sub-modular width of Q [5]. Full CQs can be computed in $\tilde{\mathcal{O}}(N^{\mathsf{fhtw}(Q)})$ time preprocessing, where $\mathsf{fhtw}(Q)$ denotes the fractional hypertree width of Q, followed by a constant delay enumeration of the tuples in the output [31, 9, 12]. The developments mentioned above for CQs raise questions about the complexity of IJQs: what is the analogous of sub-modular width and fractional hypertree width for IJQs; moreover, what is the analogous of α-acyclicity for IJQs.

1.2 Research Questions

The research questions answered by this thesis are the following.

Research Question 1. The segment tree data structure is used to index sets of intervals [15, 22]. Can one use the properties of segment trees to divide the intersection join computation problem into sub-problems involving queries with equality joins? Could such an approach lead to non-trivial data complexity bounds for the intersection join computation?

Research Question 2. What is the data complexity of Boolean IJQs? Is it possible to find upper and lower bounds for the computation? Can the upper bounds be extended for non-Boolean IJQs?

Research Question 3. Is it possible to define a dichotomy between Boolean IJQs that admit quasi-linear time computation in data complexity, and those that do not, based on their syntax? Such a characterisation would provide a simple way to distinguish the IJQs that are easy to compute.

1.3 Summary of Contributions

This thesis establishes data complexity bounds for the computation of IJQs, based on known bounds for the data complexity of CQs. Those bounds are based on the idea of reducing the problem of evaluating an IJQ to the problem of evaluating a union of CQs, and vice versa. The segment tree is a building block in all algorithms introduced by this thesis.

1.3.1 Using Segment Trees for Solving IJQs

Chapter 4 describes two fundamental algorithms: (1) an enumeration algorithm that computes the intersection join of k sets of intervals of size N using $\mathcal{O}(k^2 \cdot N \cdot \log(k \cdot N))$ preprocessing time and constant delay enumeration of the tuples in the output; and (2) a simple reduction from the problem of evaluating any IJQ, to the problem of evaluating a union of CQs, called Intersection Join Decomposition (IJDEC). Interestingly, this reduction reveals that the Boolean Triangle IJQ can be computed in time $\tilde{\mathcal{O}}(N^{3/2})$ in data complexity, and the set of the all tuples that satisfy the Triangle IJQ (full Triangle IJQ) can be computed in $\tilde{\mathcal{O}}(N^{3/2})$ preprocessing time followed by constant delay enumeration of the tuples. The IJDEC reduction is improved in Chapter 5, to get more precise upper bounds in data complexity. Nevertheless, the IJDEC reduction has an advantage over its refinement; it generates fewer CQs, which is useful in practice.

$$\begin{tikzcd}\boxed{Q(D)} \arrow[r, "\equiv"] & \boxed{\tilde{Q}^{(1)}(\tilde{D}) \vee \cdots \vee \tilde{Q}^{(\ell)}(\tilde{D})}\end{tikzcd}$$

(a) Forward Reduction

$$\begin{tikzcd}\boxed{Q(B)} & \boxed{\tilde{Q}^{(i)}(\tilde{B})} \arrow[l, "\equiv"']\end{tikzcd}$$

(b) Backward Reduction

Figure 1.1: Forward and backward reductions. The query Q is a Boolean IJQ, and the queries $\tilde{Q}^{(1)}, \dots, \tilde{Q}^{(\ell)}$ are Boolean CQs. The databases $D, B, \tilde{D}$ and $\tilde{B}$ match the structures of the corresponding queries.

1.3.2 The Data Complexity of Boolean IJQs

Chapter 5 investigates the data complexity of Boolean IJQs. It establishes a reduction of any Boolean IJQ to a disjunction of Boolean CQs, called forward reduction, and a reduction from any Boolean CQ from the disjunction mentioned above, back to the input Boolean IJQ, called backward reduction, cf. Figure 1.1. The forward and backward reductions allow for solving Boolean IJQs using existing algorithms for solving Boolean CQs, as well as expressing the complexity of Boolean IJQs using known complexity upper and lower bounds for Boolean CQs.

Forward Reduction — Upper Bounds

The forward reduction takes as an input a Boolean IJQ Q, and a database D, which matches the structure of Q. It reduces the input query Q to a disjunction of Boolean CQs, denoted by $\tilde{Q}$, and the input database D to a new database, denoted by $\tilde{D}$, which matches the structure of $\tilde{Q}$. The problem of evaluating Q on D is equivalent to the problem of evaluating $\tilde{Q}$ on $\tilde{D}$ (Proposition 5.2.12). Both the number and the size of the generated CQs only depend on the size of Q, therefore, they are considered to be constants (Proposition 5.2.11). The size of the new database $\tilde{D}$ is $\tilde{\mathcal{O}}(|D|)$. The forward reduction enables using efficient algorithms for the computation of Boolean CQs, to solve IJQs (Theorem 5.2.14).

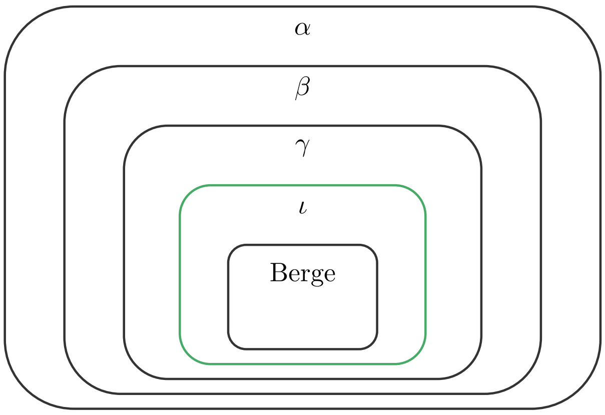


Figure 1.2: The hierarchy of acyclicity notions for hypergraphs. ι-acyclicity is a novel notion of acyclicity introduced by this thesis.

Backward Reduction — Lower Bounds

The backward reduction takes as an input a Boolean CQ $\tilde{Q}^{(i)}$, whose structure matches the structure of one of the Boolean CQs in $\tilde{Q}$, obtained by the forward reduction of Q, and an arbitrary database $\tilde{B}$ with scalars, chosen independently from $\tilde{D}$ and D mentioned above. The query $\tilde{Q}^{(i)}$ is reduced to a Boolean IJQ Q, whose structure matches that of the input Boolean IJQ, and the database $\tilde{B}$ is reduced to some database B, whose structure matches the structure of Q, and its size is $\mathcal{O}(|B|)$. The problem of evaluating $\tilde{Q}^{(i)}$ on $\tilde{B}$, is equivalent to the problem of evaluating Q on D (Proposition 5.3.2). The backward reduction shows that the lower bound of any Boolean CQ constructed by the forward reduction can be used as a lower bound for the input Boolean IJQ (Theorem 5.3.4).

Conclusion

The findings mentioned above suggest that any Boolean IJQ is as difficult as the most difficult Boolean CQ generated by the forward reduction. As a result, using the forward reduction, one can obtain optimal algorithms for Boolean IJQs, given optimal algorithms for Boolean CQs.

1.3.3 Iota Acyclicity

Chapter 6 establishes a new notion of acyclicity, called ι-acylicity (Figure 1.2). An IJQ is ι-acyclic if and only if its hypergraph includes no Berge-cycle of length greater than or equal to three (Theorem 6.1.2). It is proven that Boolean IJQs that are ι-acyclic are decidable in quasi-linear time. Furthermore, the Boolean IJQs that are not ι-acyclic are not decidable in linear time, assuming that the 3-SUM hypothesis is true [48] (Theorem 6.2.1). Last but not least, it is proven that ι-acyclicity is a tractable property (Theorem 6.3).

1.4 Thesis Outline

The remainder of the thesis is outlined as follows. Chapter 2 formally defines the IJQ evaluation problem. Chapter 3 summarizes the background and the related work. Chapters 4-6 describe the main results. Chapter 7 concludes and suggests future research directions.

Chapter 2

Problem Statement

This chapter describes formally the IJQ evaluation problem. Section 2.1 introduces the main notations and terminology. Additional notations and terminology are given at the beginning of each of the following chapters whenever is needed. Section 2.2 gives the definition of Conjunctive Queries with Intersection Joins (IJQ). Finally, two example IJQs are discussed in Section 2.3, which are used to illustrate concepts thorough the thesis.

2.1 Notation and Terminology

Consider a finite set $\mathcal{U}$ of variables. Variables are denoted by lowercase possibly subscripted letters, for example u, u_i. Variables range over, or in other words represent values from a domain.

Consider a finite set $\mathcal{R}$ of relations, where $\mathcal{R}$ is disjoint from $\mathcal{U}$. Relations are denoted with upper case possibly subscripted letters, for example R, R_i. A relation R_i is associated with a non-empty finite set of variables $e_i = \{u_1, \ldots, u_m\} \subseteq \mathcal{U}$. The expression $R_i(e_i)$ is called the schema of the relation R_i. Sometimes, in order to be clear about the variables included, the relation schema is written explicitly as $R_i(\{u_1, \ldots, u_m\})$ or by an abuse of notation as $R_i(u_1, \ldots, u_m)$ to avoid the clutter that the set notation may introduce. The names of the variables in a relation schema are considered to be part of the corresponding relation to indicate the names of different columns of the relation, hence, they can be used by a query language. However, the order of the variables does not play any role. In standard databases this approach is called named perspective [1].

In the context of a relation $R_i(e_i)$, each variable $u \in e_i$ is assigned to a domain $\mathsf{dom}_{R_i}(u)$, which is either a set of intervals $[x, y]$ with scalar endpoints, or a set of scalars. In the case where the domain consists of intervals, u is an interval variable, or in other words it has interval type in R_i. In the case where the domain consists of scalars, u is a scalar variable, or in other words it has scalar type in R_i. It is possible that a variable u is in the schemas of two different relations R_i and R_j, where in R_i it has interval type, whereas in R_j it has scalar type. In order to indicate its type, the variable is written as $[u]$ in the schema of R_i, and as is in the schema of R_j. For example, given two relation schemas $R_1([u], w)$, and $R_2(u, w)$, u and $[u]$ refer to the same variable, but the key difference is that u has interval type in R_1, while it has scalar type in R_2.

Consider a finite set of variables $e \subseteq \mathcal{U}$. A tuple t of a relation R with schema $R(e)$ is a function that maps each variable $u \in e$ to a value from $\mathsf{dom}_R(u)$. As mentioned before, the variable u in R has interval or scalar type, that is $\mathsf{dom}_R(u)$ can be either a domain with intervals or a domain with scalars respectively. In practice this means that a tuple in a relation has both interval and scalar values. Since t is defined as a mapping, $t(u)$ is used to denote the value of u in t. For a subset $a \subseteq e$, $t[a]$ denotes the tuple s over a where $s(u) = t(u)$ for each $u \in a$. A tuple t in R may be written in a linear syntax as a sequence, for example $t = (t_1, \ldots, t_m)$, where it is assumed that the order of the values corresponds to the order in which the variables appear in the explicit schema $R(\{u_1, \ldots, u_k\})$ or $R(u_1, \ldots, u_m)$ of R. An instance of the relation R with schema $R(e)$ is a finite set which consists of tuples over e. Therefore, a relation stores intervals in the columns that correspond to interval variables, and respectively scalars in the columns that correspond to scalar variables. Considering a relation R with schema $R(e)$, and a subset $a \subseteq e$, the projection of R on a, denoted by $\pi_a(R)$, is the set $\{t[a] \mid t \in R\}$.

A database schema is a set of relation schemas. A database D over a schema $\{R_1(e_1), \ldots, R_n(e_n)\}$, where $e_1, \ldots, e_n$ are finite sets of variables, is a finite non-empty set of relation instances $\{R_1, \ldots, R_n\}$. Each relation instance in a database corresponds to a relation schema in the corresponding database schema. From the above definitions it follows that the databases considered by this thesis are extensions of the standard relational databases, that store both scalars and intervals with scalar endpoints, rather that just scalars.

Considering an integer k, an intersection predicate on a multi-set $\{x_1, \ldots, x_k\}$ of intervals is the assertion

$$\left(\bigcap_{1 \leq i \leq k} x_i\right) \neq \emptyset.$$

In this thesis intersection predicates are applied on multi-sets that include both scalars and intervals with scalar endpoints. In this context, scalars are viewed as intervals with coinciding endpoints. In general, an interval and a scalar intersect if and only if the scalar is contained in the interval, and two scalars intersect if and only if they are equal.

2.2 Conjunctive Queries with Intersection Joins

The formal definition of an IJQ follows.

2.2.1 Syntax

An IJQ is an expression of the form

$$Q = \langle e, R_1(e_1) \wedge \cdots \wedge R_n(e_n) \rangle, \tag{2.1}$$

where $R_i(e_i)$ for each $1 \leq i \leq n$ is a relation schema, and $e \subseteq \bigcup_{1 \leq i \leq n} e_i$, is the set of free variables of the query. The main body of an IJQ consists of the set of free variables and a conjunction of relation schemas. As mentioned earlier the type of a variable in a relation is indicated in the corresponding relation schema by enclosing the variable in brackets or not. Since relation schemas are used in the query syntax, this notation is inherited in the query syntax as well. However, this does not mean that the query determines the types of variables; those are determined in the database on which this query is executed. The bracket notation in the query syntax can be thought as a reminder of the types of the variables in the relations in the database.

The following definitions will be used to assist the analysis in the subsequent sections. Define $\mathsf{free}(Q) = e$ be the set of free variables in the query, $\mathsf{vars}(Q) = \bigcup_{1 \leq i \leq n} e_i$ be the set of all variables in the query, and $\mathsf{schema}(Q) = \{R_i(e_i) \mid 1 \leq i \leq n\}$ be the set of all relation schemas in the query. The latter set is called the schema of the query. Lastly, define $\mathsf{ind}(Q, u) = \{i \mid u \in e_i, 1 \leq i \leq n\}$.

A hypergraph $\mathcal{H}$ is a tuple $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite set of vertices, and $\mathcal{E}$ is a finite multi-set of non-empty subsets of $\mathcal{V}$, called hyperedges. Hypergraphs provide an abstract way to describe the structure of a query; the relation schemas in the query correspond to the hyperedges of the hypergraph and the variables of the query correspond to vertices of the hypergraph. In particular, the hypergraph of the query Q is defined by $\mathcal{H}(Q) = (\mathcal{V}(Q), \mathcal{E}(Q))$, where $\mathcal{V}(Q) = \mathsf{vars}(Q)$, and $\mathcal{E}(Q) = \{e_1, \ldots, e_n\}$. Because any two relation schemas in the query can be defined over the same set of variables we have that $\mathcal{E}(Q)$ is a multi-set. Furthermore, the set of vertices $\mathcal{V}(Q)$

equals to the set of variables in the query therefore vertex is another term for variable and a the term hyperedge is used to refer to the set of variables in the schema of some relation in the query. Whenever it is clear form the context the hypergraph for an IJQ Q is just denoted by $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ for simplicity.

2.2.2 Semantics

The notion of semantics of a query refers to the meaning or the interpretation of the query when applied on a database. In other words, it refers to results the supposed to produce. Consider a database D. The result of the query Q applied on D, denoted by $Q(D)$, is a set of tuples

$$\left\{ (t_1[e], \dots, t_n[e]) \middle| t_i \in R_i(e_i) \text{ for each } 1 \leq i \leq n, \text{ and} \right.$$
$$\left. \text{for each } u \in \mathsf{vars}(Q) : \left(\bigcap_{i \in \mathsf{ind}(Q,u)} t_i(u) \right) \neq \emptyset \right\}. \quad (2.2)$$

One way to obtain an intuition of the above semantics, is to think that the multiple occurrences of the same variable, no matter what its type is, in different relation schemas in the query indicates an intersection predicate. The above evaluation problem is Boolean, or in other words a decision problem, if the set of free variables in the query is empty, hence, $e = \mathsf{free}(Q) = \emptyset$. In this case, $Q(D)$ is true if and only if there exists at least one tuple $(t_1, \dots, t_n) \in R_1 \times \cdots \times R_n$ that satisfies tha query, in other words, the set from Equation (2.2) is non-empty. In the following, in order to simplify the syntax of Boolean IJQs, the empty set of free variables together with the "$\langle$", and "$\rangle$" are omitted, therefore, the IJQ is written just as a conjunction of relation schemas, that is $Q = R_1(e_1) \wedge \cdots \wedge R_n(e_n)$.

A consequence of the above definitions, is that the IJQ problem subsumes the CQ problem, where the latter problem is the result of applying a CQ on a database. In particular, the CQ problem is a special case of an IJQ problem where all the variables in all of their occurrences in relations are of scalar type. In this case, the intersection join predicate

$$\left(\bigcap_{i \in \mathsf{ind}(Q,u)} t_i(u)\right) \neq \emptyset \tag{2.3}$$

in the expression of Equation (2.2) is equivalent to the statement "$t_i(u) = \xi$ for each $i \in \mathsf{ind}(Q, u)$, where ξ is a scalar". As a result, CQs are special cases of IJQs. Note that in the case of a CQ evaluation problem, the proposed representation of the output (Equation (2.2)) is non-standard [1]; the standard way is to represent the output $Q(D)$ as a relation over the free variables of the query $\mathsf{free}(Q)$ or e. Although the representations are different, in practice, they represent the same result; there is a one to one correspondence between the tuples in the standard representation and the tuples in the representation from Equation (2.2). In particular, each tuple t' from the relational representation of the output would correspond to a tuple t from the set of Equation (2.2), and t' can be obtained by assigning to $t'(u)$ the value ξ, for each $u \in \mathsf{free}(Q)$. In general, it is more practical to use the relational representation of the output when a CQ evaluation problem in encountered [1]. In the following, whenever the relational representation of the output is employed, it is stated explicitly.

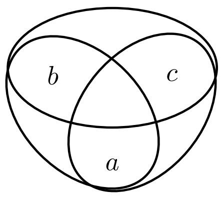
Naive Evaluation

The naive approach to evaluating an IJQ Q on a database examines all possible combinations of tuples that satisfy the query. This approach does not take advantage of any indexing or optimization techniques, and can be highly inefficient for large databases. Assuming that N is the size of each relation in the database, the naive approach can be implemented using nested loops ([51]), and requires $\mathcal{O}(N^{|\mathsf{schema}(Q)|})$ steps. In combined complexity, that is an exponential number of steps, and it is natural to wonder whether there exists a subclass that admits polynomial time computation. In data complexity, that is a polynomial in the size of the data number of steps. In this case, a reasonable goal is to make the exponent lower. Because the query is often much smaller than the database, data complexity is widely seen more relevant to database research [46].

2.3 Examples

Two running example queries are introduced next. They are also used to explain the newly introduced theory in the following chapters.

Example 2.3.1 (The Triangle IJQ). *Consider the full Triangle IJQ*



$$Q_\triangle = \langle \{a, b, c\}, R_1([a], [b]) \wedge R_2([b], [c]) \wedge R_3([a], [c]) \rangle$$

The query $Q_\triangle$ has three intersection joins on $a, b,$ and c. Furthermore, it has a hypergraph $\mathcal{H}(Q) = (\mathcal{V}(Q), \mathcal{E}(Q))$ with $\mathcal{V}(Q) = \{a, b, c\}$, and $\mathcal{E}(Q) = \{\{a, b\}, \{b, c\}, \{c, a\}\}$, which is illustrated above. Consider a database $D = \{R_1, R_2, R_3\}$, where R_1 stores intervals in both columns a and b, R_2 stores intervals both columns b and c, and R_3 stores intervals in both columns a and c. An example database, together with the result of the query are illustrated in Figure 2.1. By the semantics of IJQs (Section 2.2), the result $Q_\triangle(D)$ is a set of tuples such that $(r, s, t) \in Q_\triangle(D)$ if and only if $(r, s, t) \in R_1 \times R_2 \times R_3$ and $(r(a) \cap t(a)) \neq \emptyset$; and $(r(b) \cap s(b)) \neq \emptyset$; and $(s(c) \cap t(c)) \neq \emptyset$.

Example 2.3.2 (A Generic IJQ). *Consider the Boolean IJQ*

R_1

	a	b
r_1:	$[1,5]$	$[1,8]$
r_2:	$[5,7]$	$[2,8]$
r_3:	$[5,8]$	$[1,8]$

R_2

	b	c
s_1:	$[3,4]$	$[5,6]$
s_2:	$[4,4]$	$[8,10]$
s_3:	$[4,8]$	$[10,11]$
s_4:	$[10,12]$	$[2,6]$

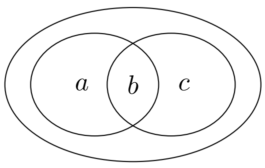
R_3

	a	c
t_1:	$[2,6]$	$[1,6]$
t_2:	$[6,10]$	$[11,15]$

Result

r_1	s_1	t_1
r_2	s_1	t_1
r_2	s_3	t_2
r_3	s_1	t_1
r_3	s_3	t_2

Figure 2.1: A database instance for the Triangle IJQ.



$$Q = R_1(a, [b], [c]) \wedge R_2(a, [b]) \wedge R_3(b, [c])$$

The query Q is Boolean, and has three intersection joins on a, b, and c. Furthermore, it has a hypergraph $\mathcal{H}(Q) = (\mathcal{V}(Q), \mathcal{E}(Q))$ with $\mathcal{V} = \{a, b, c\}$, and $\mathcal{E}(Q) = \{\{a, b, c\}, \{a, b\}, \{b, c\}\}$, which is illustrated above. Consider a database $D = \{R_1, R_2, R_3\}$, where R_1 stores scalars in column a, and intervals in both columns b and c, R_2 stores scalars in column a and intervals in column b, and R_3 stores scalars in column b and intervals in column c. By the semantics of IJQs (Section 2.2), the result $Q(D)$ is true if and only if there exists a tuple $(r, s, t) \in R_1 \times R_2 \times R_3$ such that, $r(a) = s(a)$, since a ranges over scalars in both R_1 and R_2; $t(b) \in (r(b) \cap s(b))$, since b ranges over scalars in R_3 and over intervals in R_2 and R_2; and $(r(c) \cap t(c)) \neq \emptyset$, since c ranges over intervals in both R_1 and R_3.

Chapter 3

Background & Related Work

This chapter provides a summary of the background materials used in the thesis, and the related work.

Notation. The symbol $\tilde{\mathcal{O}}$ means $\mathcal{O}$ with hidden logarithmic factors. Given an interval i, its left endpoint is denoted by $i.\mathsf{start}$ and its right endpoint is denoted by $i.\mathsf{end}$. Given any tree $\mathcal{T}$, $V(\mathcal{T})$ is used to denote the set of nodes in the tree, and $\mathsf{root}(\mathcal{T})$ is used to denote the root of the tree. Given any node u of a binary tree, $\mathsf{parent}(u)$ denotes the parent of u; $\mathsf{anc}(u)$ denotes the set of ancestors of u, including u, $\mathsf{lc}(u)$ (respectively $\mathsf{rc}(u)$) denotes the left child (respectively right child) of u.

Terminology. When one studies the complexity of database queries, it is critical to specify whether the combined or data complexity is being considered. In combined complexity, both the query and the database sizes are considered as parameters in the complexity. On the other hand, in data complexity, the size of the query is assumed to be constant, therefore only the size of the database is considered as a parameter in the complexity [46]. An algorithm runs in quasi-linear time if $T(N) = \mathcal{O}(N \cdot \log^k N)$ for some positive constant k. An enumeration algorithm for query evaluation consists of two phases, preprocessing, and enumeration. The algorithm performs some preprocessing on the query and the database during the preprocessing phase. The algorithm enumerates all tuples in the result during the enumeration phase [12]. The 3SUM conjecture states that it cannot be determined if a given set of N real numbers contains three elements that sum to zero, in time $\mathcal{O}(N^{2-\epsilon})$, for some $\epsilon \geq 0$ [48].

Organisation. Section 3.1 reviews existing acyclicity notions for CQs or hypergraphs, such as, alpha (α); gamma (γ); and Berge acyclicity, and width measures, such as the fractional hypertree width (fhtw); and the sub-modular width (subw). Section 3.2 reviews algorithms for the evaluation of CQs. The definition of the segment tree and its properties are given in Section 3.3. Section 3.4 provides a literature review on algorithms for the evaluation of join on interval data.

3.1 Hypergraphs

Hypergraphs are widely considered in studies on query evaluation [25, 10, 58, 8, 31, 29], because they provide a simple means of describing the structure of a query. The formal definition of a hypergraph follows.

Definition 3.1.1 (Hypergraph). *A (multi-)hypergraph $\mathcal{H}$ is a tuple $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite set of vertices and $\mathcal{E}$ is a multi-set of non-empty subsets of $\mathcal{V}$ called hyperedges, in other words, $\mathcal{E} \subseteq 2^{\mathcal{V}} \setminus \{\emptyset\}$, where $2^{\mathcal{V}}$ is the power set of $\mathcal{V}$.*

A query or a database schema is associated with a hypergraph in the following way: each variable is associated to a vertex of the hypergraph, and each relation schema is associated to a hyperedge of the hypergraph. As a result, certain properties of a query or a database schema can be studied by taking into account the properties of its hypergraph. For example, acyclicity and width are two important properties that can be studied using hypergraphs.

3.1.1 Acyclicity

An acyclic hypergraph corresponds to a schema without cyclic dependencies between relations. The absence of cycles can simplify query processing and optimization, making it easier to find efficient query evaluation strategies. A query or database schema is acyclic if its associated hypergraph is acyclic. Several types of acyclicity relevant to this thesis are defined in the following, including alpha (α), gamma (γ), and Berge. Figure 1.2 illustrates their relationship.

Alpha

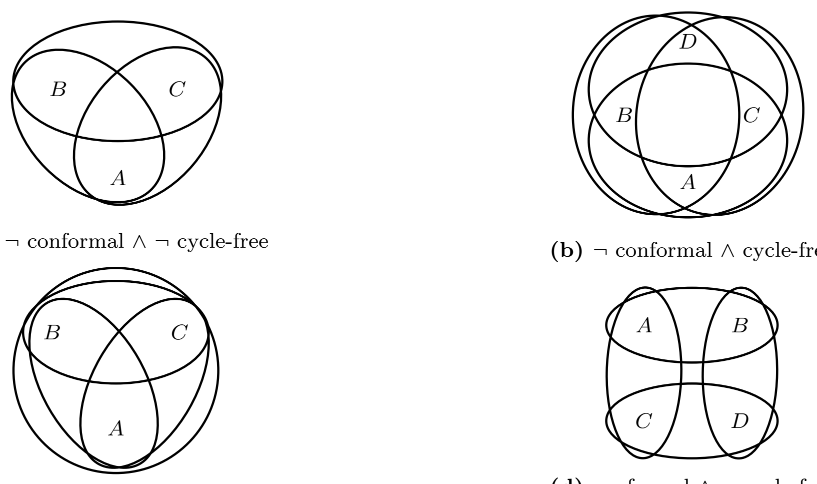
The class of α-acyclic CQs has been found to admit efficient evaluation algorithms, both in combined and in data complexity [1]. Previous work achieved several characterizations of α-acyclicity [1, 16, 25]. Definition 3.1.8 includes those that are relevant to this thesis. However, before moving on to the definition, it is necessary to define some preliminary concepts and describe how they lead to those characterizations.

Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. The induced sub-hypergraph of $\mathcal{H}$ on $S \subseteq \bigcup \mathcal{E}$ is obtained by removing from $\mathcal{H}$ the vertices in $\mathcal{V} \setminus S$.

Definition 3.1.2 (Induced Sub-hypergraph [16])**.** *Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. The induced hypergraph of $\mathcal{H}$ on $S \subseteq \bigcup \mathcal{E}$ is a hypergraph, denoted by $\mathcal{H}[S]$, whose multi-set of hyperedges, denoted by $\mathcal{E}[S]$, is defined by $\{e \cap S \mid e \in \mathcal{E}\} \setminus \{\emptyset\}$.*

Furthermore, the minimization of the hypergraph $\mathcal{H}$ is obtained by removing from $\mathcal{H}$ the hyperedges that are included in another hyperedge.

Definition 3.1.3 (Minimisation of a Hypergraph [16])**.** *Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. The minimization of $\mathcal{H}$ is a hypergraph, denoted by $\mathcal{M}(\mathcal{H})$, whose set of hyperedges, denoted by $\mathcal{M}(\mathcal{E})$, is defined by $\{e \in \mathcal{E} \mid \nexists f \in \mathcal{E}(e \subset f)\}$.*



(a) ¬ conformal ∧ ¬ cycle-free

(b) ¬ conformal ∧ cycle-free

(c) conformal ∧ cycle-free

(d) conformal ∧ ¬ cycle-free

Figure 3.1: This figure exemplifies the properties of conformal and cycle-free hypergraphs (originally presented in [16]). Figure 3.1a shows a hypergraph that is ¬ conformal ∧ ¬ cycle-free. Figure 3.1c shows a hypergraph that is conformal ∧ cycle-free. Figure 3.1b shows a hypergraph that is ¬ conformal ∧ cycle-free. Figure 3.1d shows a hypergraph that is conformal ∧ ¬ cycle-free.

Definition 3.1.4 (Conformal Hypergraph [16])**.** *A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is conformal if there is no subset $S \subseteq \mathcal{V}$, with cardinality ≥ 3 such that*

$$\mathcal{M}(\mathcal{E}[S]) = \{S \setminus \{x\} \mid x \in S\}. \tag{3.1}$$

Definition 3.1.5 (Cycle-Free Hypergraph [16])**.** *A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is cycle-free if there is no subset $S = \{v^1, \ldots, v^n\} \subseteq \mathcal{V}$ with $n \geq 3$ such that:*

$$\mathcal{M}(\mathcal{E}[S]) = \{\{v^i, v^{i+1}\} \mid 1 \leq i < n\} \cup \{\{v^n, v^1\}\}. \tag{3.2}$$

Figure 3.1 illustrates four example hypergraphs which explain the properties of conformity and cycle-freedom. Both properties lead to a characterization for α-acyclicity, which is, a hypergraph is α-acyclic if it is both conformal and cycle-free. This characterization is due to Brault-Baron [16].

The Gyo reduction takes as an input a hypergraph and modifies it iteratively [28]. The definition of Gyo-reducible hypergraph is given next.

Definition 3.1.6 (GYO-reducible Hypergraph [16])**.** *A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is* GYO *reducible if by repeatedly applying the following steps it can be transformed to the empty hypergraph: (1) if a vertex $v \in \mathcal{V}$ occurs in only one hyperedge $e \in \mathcal{E}$, then remove v from e; and (2) if two distinct edges $e, f \in \mathcal{E}$ satisfy $e \subset f$, then remove e.*

The GYO reduction is used in another characterization for α-acyclicity, in particular, a hypergraph is α-acyclic if and only if it is GYO-reducible [16].

Another characterisation is the following: a hypergraph is α-acyclic CQ if and only if it has a join tree [1]. In the evaluation of a α-acyclic CQ, the join tree can be also used as a form of a join plan for the corresponding CQ [58]. The definition of the join tree of a hypergraph follows.

Definition 3.1.7 (Join Tree of a Query)**.** *A join tree of a CQ Q with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a tuple $(\mathcal{T}, \chi)$ where $\mathcal{T}$ is a tree and χ is a bijection of the form $\chi : V(\mathcal{T}) \to \mathcal{E}$, where for every vertex $v \in \mathcal{V}$, the set $\{t \mid t \in V(\mathcal{T}), v \in \chi(t)\}$ is a non-empty connected sub-tree of $\mathcal{T}$.*

The following are the characterizations of α-acyclicity mentioned above.

Definition 3.1.8 (α-acyclic Hypergraph [16, 1])**.** *Consider a hypergraph $\mathcal{H}$. The following statements are equivalent:*

- *The hypergraph $\mathcal{H}$ is α-acyclic;*
- *The hypergraph $\mathcal{H}$ has a join-tree (Definition 3.1.7);*
- *The hypergraph $\mathcal{H}$ is* GYO*-reducible.*
- *The hypergraph $\mathcal{H}$ is conformal and cycle-free (Definition 3.1.4 and 3.1.5).*

Free-connexity

Free-connexity is a property of α-acyclic CQs that is related to the existence of efficient evaluation algorithms that use linear time preprocessing and constant delay enumeration of the tuples in their output [52].

Definition 3.1.9 (Free-connex α-acyclic CQ [52]). *Consider an α-acyclic CQ Q over the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. The query Q is free-connex if the hypergraph whose set of hyperedges is $\mathcal{E} \cup \{\mathsf{free}(Q)\}$ is α-acyclic as well.*

Gamma

The notion of γ-acyclicity which is defined next, implies α-acyclicity, and is implied by Berge-acyclicity, cf. Figure 1.2.

Definition 3.1.10 (γ-acyclic Hypergraph [16]). *Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. $\mathcal{H}$ is γ-acyclic if and only if $\mathcal{H}$ is cycle-free and there exist no distinct vertices $x, y, z \in \mathcal{V}$ such that $\{\{x, y\}, \{x, z\}, \{x, y, z\}\} \subseteq \mathcal{E}[\{x, y, z\}]$.*

Berge

The class of Berge-acyclic hypergraphs is a subset of the class of γ-acyclic hypergraphs cf. Figure 1.2. Berge-acyclicity is defined next using the notion of a Berge cycle [25].

Definition 3.1.11 (Berge Cycle [25]). *Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. A Berge cycle in $\mathcal{H}$ is a sequence $(e^1, v^1, e^2, v^2, \ldots, e^n, v^n, e^{n+1})$ such that:*

- *$v^1, \ldots, v^n$ are distinct vertices in $\mathcal{V}$;*
- *$e^1, \ldots, e^n$ are distinct hyperedges in $\mathcal{E}$ and $e^{n+1} = e^1$;*
- *$n \geq 2$, that is, there are at least 2 hyperedges involved; and*
- *v^i is in e^i and e^{i+1} for each $i \in [1, n]$.*

Definition 3.1.12 (Berge Acyclic Hypergraph [25]). *A hypergraph is Berge-acyclic if it has no Berge cycle.*

3.1.2 Width Measures

This section presents two notions of width that are used to express the data complexity of CQs. Those are the fractional hypertree width [31], and the sub-modular width [43]. We start by introducing some initial concepts, and then proceed to formally define the notions of width mentioned above.

Definition 3.1.13 (Fractional Edge Cover Number [8])**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph. The fractional edge covers of $S \subseteq \mathcal{V}$ are the feasible solutions $(x_e)_{e\in\mathcal{E}}$ for the following linear program*

$$\begin{array}{lll} \textit{minimise} & \sum_{e\in\mathcal{E}} x_e & \\ \textit{subject to} & \sum_{e:v\in e} x_e \geq 1 & \forall\, v \in S \\ & x_e \geq 0 & \forall\, e \in \mathcal{E}. \end{array}$$

and the fractional edge cover number, $\rho^*_{\mathcal{E}}(S)$, is the cost $\sum_{e\in\mathcal{E}} x_e$ of the optimal solution. The minimum exists, and it is rational.

Definition 3.1.14 (Polymatroid [5])**.** *Consider the vertex set $\mathcal{V}$. A function $f : 2^{\mathcal{V}} \to \mathbb{R}^+$ is a (non-negative) set function on $\mathcal{V}$. A set function f on $\mathcal{V}$ is*

- *monotone if $f(X) \leq f(Y)$ whenever $X \subseteq Y$; and*
- *submodular if $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$ for all $X, Y \subseteq \mathcal{V}$.*

A monotone, submodular set function $h : 2^{\mathcal{V}} \to \mathbb{R}^+$ with $h(\emptyset) = 0$ is a polymatroid.

$\Gamma_{\mathcal{V}}$ denotes the set of all polymatroids $h : 2^{\mathcal{V}} \to \mathbb{R}^+$ over the set $\mathcal{V}$.

Definition 3.1.15 (Edge Dominated Set Functions [43, 5])**.** *Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. The set of edge-dominated set functions is defined as follows*

$$\mathsf{ED}(\mathcal{H}) = \{h \mid h : 2^{\mathcal{V}} \to \mathbb{R}^+, h(S) \leq 1, \forall S \in \mathcal{E}\}. \tag{3.3}$$

Definition 3.1.16 (Hypertree Decomposition [29]). *The hypertree decomposition of a hypergraph $\mathcal{H}$ is a pair $(\mathcal{T}, \chi)$, where $\mathcal{T}$ is a tree whose set of vertices is denoted by $V(\mathcal{T})$, and $\chi : V(\mathcal{T}) \to 2^{\mathcal{V}}$ maps each node t of the tree $\mathcal{T}$ to a subset $\chi(t)$ of vertices such that the following properties hold:*

1. *every hyperedge $e \in \mathcal{E}$ is a subset of a set $\chi(t)$ for some $t \in V(\mathcal{T})$, and*
2. *for every vertex $v \in \mathcal{V}$, the set $\{t \mid t \in V(\mathcal{T}), v \in \chi(t)\}$ is a non-empty connected subtree of $\mathcal{T}$.*

The sets $\chi(t)$ are called the *bags* of the hypertree decomposition. Let $\mathsf{TD}(\mathcal{H})$ denote the set of hypertree decompositions of a given hypergraph $\mathcal{H}$.

Fractional Hypertree Width

The fractional hypertree width was introduced by Grohe and Marx [31]. It can be computed by combining the concepts of the hypertree decomposition and the fractional edge cover number.

Definition 3.1.17 (Fractional Hypertree Width [31]). *Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. The fractional hypertree width of $\mathcal{H}$ is defined by:*

$$\mathsf{fhtw}(\mathcal{H}) = \min_{(\mathcal{T},\chi)\in\mathsf{TD}(\mathcal{H})} \max_{t\in V(\mathcal{T})} \rho^*_{\mathcal{E}}(\chi(t)). \tag{3.4}$$

The following is an alternative characterization for $\mathsf{fhtw}(\mathcal{H})$, originally derived in [5]:

$$\mathsf{fhtw}(\mathcal{H}) = \min_{(\mathcal{T},\chi)\in\mathsf{TD}(\mathcal{H})} \max_{h\in\mathsf{ED}(\mathcal{H})\cap\Gamma_{\mathcal{V}}} \max_{t\in V(\mathcal{T})} h(\chi(t)). \tag{3.5}$$

Sub-modular Width

The sub-modular width was introduced subsequently by Marx [43].

Definition 3.1.18 (Sub-modular Width [43]). *Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, the sub-modular width of $\mathcal{H}$ is defined by:*

$$\mathsf{subw}(\mathcal{H}) = \max_{h \in \mathsf{ED}(\mathcal{H}) \cap \Gamma_{\mathcal{V}}} \min_{(\mathcal{T}, \chi) \in \mathsf{TD}(\mathcal{H})} \max_{t \in V(\mathcal{T})} h(\chi(t)), \tag{3.6}$$

where $\mathsf{ED}(\mathcal{H})$ denotes the set of edge dominated set functions over $\mathcal{H}$, and $\Gamma_{\mathcal{V}}$ denotes the set of polymatroids over $\mathcal{V}$.

It has been proven that $\mathsf{subw}(\mathcal{H}) \leq \mathsf{fhtw}(\mathcal{H})$ for any hypergraph $\mathcal{H}$ [43, 5]. Additionally, there are classes of CQs with bounded sub-modular width and unbounded fractional hypertree width [43].

3.2 Conjunctive Queries

The class of Conjunctive queries (CQs) is fundamental in database theory. CQs are specific types of logical queries that can be expressed using a conjunction and existential quantification, which makes them highly expressive and useful for applications. They may also appear as parts of other more complex first order queries. The definition of CQs and in general the CQ evaluation problem is given implicitly in Section 2.2 by explaining that CQs are special cases of IJQs. More information about CQs can be found in the seminal article about CQs [19], and also in database theory textbooks [1]. This section reviews algorithms for the evaluation of CQs and the complexity bounds that are associated with them.

3.2.1 Complexity Bounds

The complexity of CQs is an important topic of research, as it has implications for the efficiency and scalability of query processing algorithms. CQs are NP-complete in terms of the combined complexity [19], whereas their data complexity is polynomial.

Concepts related to data complexity bounds for CQs are discussed in the following.

The AGM Bound

Consider a CQ Q with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. In the case where Q is a full CQ, the fractional edge cover number of $\mathcal{H}$ is associated with a bound on the worst-case answer size of Q for any database [8, 31]. In particular, for any database D, the size of $Q(D)$ is at most $\mathcal{O}(N^{\rho^*_{\mathcal{E}}(\mathcal{V})})$, where N is the size of the database. This bound is known as the AGM bound [8].

Fixed-parameter Tractability

The subclass of Boolean CQs is fixed-parameter tractable with parameter $|Q|$, if there exists an algorithm that computes every Boolean CQ in that subclass in time $f(|Q|) \cdot N^c$ for some fixed constant c, where f is any computable function, $|Q|$ is the number of symbols in Q, and N is the size of the database [43, 32]. A sub-class of Boolean CQ is fixed-parameter tractable with parameter $|Q|$, and thus can be solved in polynomial time if and only if every Boolean CQ in this sub-class has a bounded sub-modular width ($\mathsf{subw}(Q)$) [43].

3.2.2 Evaluation Algorithms

A review of known algorithms that match the above-mentioned bounds follows.

Yannakakis's Algorithm

An α-acyclic Boolean CQ can be evaluated in time linear in the size of the input database. Furthermore, a full CQ with the same schema can be evaluated in time linear in the input plus output size. The α-acyclic CQs, which are also free-connex, can be evaluated by an enumeration algorithm which uses linear time preprocessing in the size of the data, followed by constant delay enumeration of the tuples in the output. All the above can be achieved using Yannakakis's algorithm [58].

Proposition 3.2.1 (Combined Complexity for α-acyclic CQs [1])**.** *Consider an α-acyclic CQ Q, and a matching input database D. There exists an algorithm that evaluates Q on D in time polynomial in the size of the query, the input, and the output.*

Proposition 3.2.2 (Data Complexity for α-acyclic CQs [52])**.** *Consider a CQ Q over the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, and a matching input database D. If the 3SUM conjecture holds, then Q is α-acyclic free-connex if and only if Q can be evaluated by an enumeration algorithm that uses linear time preprocessing, and constant delay enumeration of the tuples in the output.*

Worst-case Optimal Join Algorithms

In the case of a full, not necessarily α-acyclic CQ, there exist join algorithms whose worst-case runtime matches the AGM bound up to a log factor [44, 55]. One of them is called Leapfrog Triejoin [55]. Those algorithms are called worst-case optimal because their runtime is linear in the size of the maximum output. Consider a Boolean CQ Q. By combining Yannakakis's algorithm with worst-case optimal algorithms one can obtain an algorithm with $\tilde{\mathcal{O}}(N^{\mathsf{fhtw}(\mathcal{H})})$ runtime. This algorithm first materializes each bag of the hypertree decomposition of Q (Definition 3.1.16) by computing the full CQ associated with the bag, using a worst-case optimal algorithm, and then, applies Yannakakis's algorithm [58] on the α-acyclic Boolean CQ, using the join tree obtained as a result of the materialisation of the bags.

The Panda algorithm, introduced by Khamis et al. [5], computes any Boolean CQ in time $\tilde{\mathcal{O}}(N^{\mathsf{subw}(\mathcal{H})})$. It has also been extended to handle CQs with inequalities (FAQ-AI) [2].

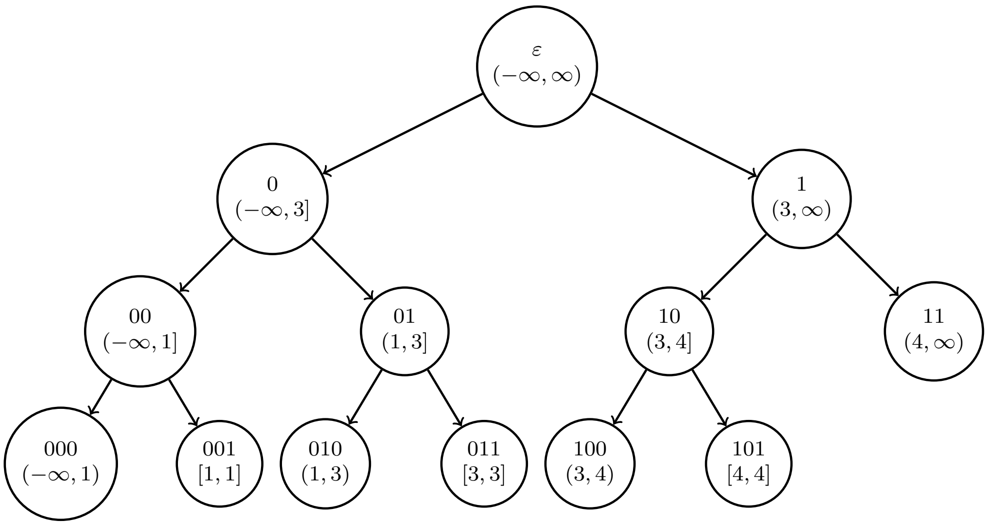


Figure 3.2: An example segment tree. This segment tree is built over a multi-set of intervals with endpoints $\{1, 3, 4\}$.

3.3 The Segment Tree

The segment tree is a data structure that serves as an index of a multi-set of intervals [11]. The following definition of a segment tree is similar to the definitions given in [15, 22].

Definition 3.3.1 (Segment Tree [22])**.** *Let $\mathcal{I}$ be a multi-set of n intervals. Let $p_1, \ldots, p_m$ be the sequence of the distinct endpoints of the intervals in ascending order ($m \leq 2n$). Consider the following disjoint intervals called elementary segments: $(-\infty, p_1)$, $[p_1, p_1]$, (p_1, p_2), $[p_2, p_2]$, $\ldots$, (p_{m-1}, p_m), $[p_m, p_m]$, $(p_m, +\infty)$. The segment tree $\mathfrak{T}_\mathcal{I}$ for $\mathcal{I}$ is a complete binary tree with the following properties.*

- *The leaves of $\mathfrak{T}_\mathcal{I}$ correspond to the elementary segments created by ordering the endpoints of the intervals in $\mathcal{I}$: the leftmost leaf corresponds to the leftmost elementary segment, and so on. The elementary segment corresponding to a leaf v is denoted by $\mathsf{seg}_\mathcal{I}(v)$.*
- *Each internal node u of $\mathfrak{T}_\mathcal{I}$ is associated with a set of segments which is the union of elementary segments of the leaves of the subtree rooted by u: the segment $\mathsf{seg}_\mathcal{I}(u)$*

corresponding to an internal node u is the union of the elementary segments $\mathsf{seg}_{\mathcal{I}}(v)$ at the leaves v in the subtree rooted at u.

- *Each node v is associated with a multi-set of intervals called canonical subset, defined as*

$$\mathcal{I}_v = \{i \in \mathcal{I} \mid \mathsf{seg}_{\mathcal{I}}(v) \subseteq i, \mathsf{seg}_{\mathcal{I}}(\mathsf{parent}(v)) \not\subseteq i\}. \tag{3.7}$$

- *Each node of a segment tree is uniquely identified by a bit-string. The root corresponds to the empty bit-string, its left child is the bit-string '0', its right child has the bit-string '1', and so on.*

Given an interval or scalar x, $\mathsf{leaf}_{\mathcal{I}}(x)$ denotes the leaf that corresponds to the left endpoint of the interval x, namely $x.\mathsf{start}$, or corresponds to x, in case x is a scalar. Since $\mathfrak{T}_{\mathcal{I}}$ is a complete binary tree with $2 \cdot m + 1 = \mathcal{O}(n)$ leaves, the size of $V(\mathfrak{T}_{\mathcal{I}})$ is $\mathcal{O}(n)$, and the height of the tree is $\mathcal{O}(\log n)$. The overall construction time of the segment tree is $\mathcal{O}(n \cdot \log n)$ [22].

Example 3.3.2 (Canonical Subsets)**.** *Consider the segment tree in Figure 3.2. The interval* $[1, 4]$ *is contained in the canonical subsets of the nodes* 001*,* 01*, and* 10*. The interval* $[3, 4]$ *is contained in the canonical subsets of the nodes* 011 *and* 10*.*

Property 3.3.3 (Ancestor Inclusion)**.** *Let $u, v \in V(\mathfrak{T}_{\mathcal{I}})$. It holds that $u \in \mathsf{anc}(v)$ if and only if $\mathsf{seg}_{\mathcal{I}}(u) \supseteq \mathsf{seg}_{\mathcal{I}}(v)$. Equivalently, it holds that $u \in \mathsf{anc}(v)$ if and only if u is a prefix of v.*

3.3.1 Canonical Partition

This subsection defines the notion of an interval's canonical partition [15].

Definition 3.3.4 (Canonical Partition of an Interval [15])**.** *Let $\mathcal{I}$ be a set of intervals* *and $x \in \mathcal{I}$. The Canonical Partition of x with respect to $\mathcal{I}$ is a set defined by:*

$$\mathsf{cp}_{\mathcal{I}}(x) = \{u \in V(\mathfrak{T}_{\mathcal{I}}) \mid x \in \mathcal{I}_u\}. \tag{3.8}$$

The nodes in $\mathsf{cp}_{\mathcal{I}}(x)$ are represented by bit-strings, hence $\mathsf{cp}_{\mathcal{I}}(x)$ can be considered as a set of bit-strings.

The canonical partition of an interval x is a set of nodes $\mathsf{cp}_{\mathcal{I}}(x)$ such that the segments associated to the nodes in $\mathsf{cp}_{\mathcal{I}}(x)$ are pairwise disjoint, and by taking the union of those segments one can "reconstruct" x. Furthermore, any path that starts from the root and ends up in a leaf that corresponds to a value in x contains exactly one node from $\mathsf{cp}_{\mathcal{I}}(x)$.

Example 3.3.5 (Canonical Partition)**.** *Consider the segment tree from Figure 3.2. The canonical partition of the interval $[1, 4]$ is the set of nodes $\{001, 01, 10\}$.*

Property 3.3.6. *For any interval $x \in \mathcal{I}$, there cannot be two distinct nodes in $\mathsf{cp}_{\mathcal{I}}(x)$ that are ancestors of one another.*

Proof. Assume that there exist two distinct nodes $u, v \in \mathsf{cp}_{\mathcal{I}}(x)$ such that u is an ancestor of v. By Property 3.3.3, we have $\mathsf{seg}_{\mathcal{I}}(u) \supseteq \mathsf{seg}_{\mathcal{I}}(v)$. This is a contradiction because the segments in $\{\mathsf{seg}_{\mathcal{I}}(v) \mid v \in \mathsf{cp}_{\mathcal{I}}(x)\}$ do not overlap. $\square$

Property 3.3.7 (Canonical Partition Size [15])**.** *For any interval $x \in \mathcal{I}$, its canonical partition $\mathsf{cp}_{\mathcal{I}}(x)$ has $\mathcal{O}(\log n)$ size.*

Proof. This proof is based on arguments originally introduced in [22]. Given an interval $x \in \mathcal{I}$, there are no three nodes in $\mathsf{cp}_{\mathcal{I}}(x)$ that are at the same depth. For contradiction, suppose that the nodes v_1, v_2, v_3 are at the same depth, numbered from left to right. Suppose v_1, v_2, $v_3 \in \mathsf{cp}_{\mathcal{I}}(x)$. It follows that x covers the entire interval from the left endpoint of $\mathsf{seg}_{\mathcal{I}}(v_1)$ to the right endpoint of $\mathsf{seg}_{\mathcal{I}}(v_3)$. Since v_2

is between v_1 and v_3, $\mathsf{seg}_{\mathcal{I}}(\mathsf{parent}(v_2))$ must be contained in x. Hence, $v_2 \notin \mathsf{cp}_{\mathcal{I}}(x)$. Contradiction. Since $\mathsf{cp}_{\mathcal{I}}(x)$ includes at most 2 nodes from each level of the segment tree, its size is $\mathcal{O}(\log n)$. □

Property 3.3.8 (Canonical Partition Construction Time [15]). *For any interval $x \in \mathcal{I}$, the canonical partition of x can be constructed in $\mathcal{O}(\log n)$ time.*

3.3.2 Query

Consider a point p and a segment tree $\mathfrak{T}_{\mathcal{I}}$. The procedure Query($\mathfrak{T}_{\mathcal{I}}, p$) (Algorithm 1) returns the leaf whose elementary segment contains the point p. The algorithm visits one node at each level of the segment tree, therefore, its runtime is $\mathcal{O}(\log n)$.

Algorithm 1 Segment Tree Query

```
1: procedure QUERY($\mathfrak{T}_{\mathcal{I}}$: segment tree, $p$: point)
2:     $v = \mathsf{root}(\mathfrak{T})$;
3:     while $v$ is not a leaf do
4:         if $p \in \mathrm{seg}_{\mathcal{I}}(\mathsf{lc}(v))$ then
5:             $v = \mathsf{lc}(v)$;
6:         else
7:             $v = \mathsf{rc}(v)$;
8:         end if
9:     end while
10:    return $v$;
11: end procedure
```

3.4 Literature Review

Algorithms for intersection joins have been developed in the context of temporal [27] and spatial databases [40, 34]. In temporal databases, the tuples are associated with intervals that represent the valid periods of the tuples. Temporal or interval joins are used to combine tuples that are valid at overlapping/intersecting time periods [27, 23]. In spatial databases, the tuples are associated with multi-dimensional objects that are approximated by intervals in the multi-dimensional space called Minimum Bounding Rectangles (MBRs). Spatial joins are used to find pairs of tuples associated with objects with overlapping MBRs as a filtering step, and then compute the pairs of tuples whose objects intersect as a refining step [40, 54, 34]. According to the definition of IJQs in Section 2.2, temporal and spatial joins in the stage of filtering are special cases of IJQs.

3.4.1 Two-way Joins

The nested-loop join is an algorithm for computing the join of two relations and can be generalised in the case of more than two relations in a simple way. It can be used with any predicates, for instance, equality, intersection, and distance predicates [51, 40]. Hence, it is suitable for the computation of IJQs. The drawback of the nested-loop join is that its runtime is upper bounded by the size of the Cartesian product of the two relations. For this reason, several attempts have been made to optimize its performance using indexes [51].

Consider the simplified problem of having two sets of axis-parallel rectangles, for example MBRs, and the goal of reporting all the pairs of rectangles, from the cross-product of the two sets that intersect. Algorithms that solve this problem in the context of spatiotemporal databases are usually classified as partition-based algorithms, and index-based algorithms [40, 34]. Examples of partition-based algorithms are the partition based spatial-merge join [47], the spatial hash join [39], the size separation spatial join [37], and the scalable sweeping-based spatial join [7]. Partition-based algorithms can be naturally used in parallel and distributed settings [54]. Examples of index-based algorithms are the R-tree join [17], the seeded-tree join [38], and relational interval tree join [24]. The majority of the work listed above is focused on external-memory evaluation, which is becoming less interesting because in-memory data management is becoming more available. However, by applying modifications, for example storing indexes in main memory for faster retrieval data, the external-memory algorithms under consideration in the following, can be used for in-memory computation [40, 34].

The plane-sweep approach, that stems from computational geometry, provides another way to solve the above problem [50]. It is based on the idea of employing a vertical line that moves across the plane; if two rectangles intersect with the sweep line at the same time, it suggests that they might intersect with each other. Several variations of the sweep-line were developed later on, such as the endpoint based intersection [49], and the scalable sweeping based spatial join [7], which are both focused on the join of two sets of intervals instead of rectangles. Those algorithms compute two-way interval intersection joins in $\mathcal{O}(N \cdot \log N + \textsc{out})$, where $\textsc{out}$ is the output size and N is the input size. Plane-sweep based algorithms are designed to run in main memory, hence, they are frequently used as building blocks to improve the main memory performance of algorithms which run in external memory. Modern main memory data management and the availability of distributed systems that process the data in main memory has motivated research to further optimise and parallelise the plane-sweep method [49, 13].

3.4.2 Multi-way Joins

Multi-way joins involve the join of more than two input tables. Algorithms for multi-way intersection joins have seen less progress than algorithms for two-way intersection joins. A simple approach to solve multi-way intersection joins is to employ two-way joins [41], and focus on the optimisation of the join ordering. However, this approach can produce intermediate results that are much larger than the final result on certain database instances, leading to sub-optimal running time [33]. As mentioned previously in Section 1.1, this is an issue for CQs as well [45]. Hu et al. [33] studied a special case of the IJQ problem; they considered CQs with one interval variable that occurs in all relation schemas in the query. That is, the query has a single multi-way intersection join. They proposed an approach based on worst-case optimal join algorithms [45], which avoids producing large intermediate results.

Chapter 4

Prelude: Using Segment Trees for Solving IJQs

This chapter presents novel methods for effectively solving IJQs by utilizing the segment tree data structure. Two key algorithms are introduced, (1) an enumeration algorithm, called SETIJ, that computes the intersection join of k sets of intervals of size N using $\mathcal{O}(k^2 \cdot N \cdot \log(k \cdot N))$ preprocessing time, and constant delay enumeration of the tuples in the output; and (2) a simple reduction from the problem of evaluating any IJQ to the problem of evaluating a union of CQs, called Intersection Join Decomposition (IJDEC). Interestingly, the IJDEC reduction shows that the Boolean Triangle IJQ can be computed in $\tilde{\mathcal{O}}(N^{3/2})$ time in data complexity, and the full Triangle IJQ can be computed in $\tilde{\mathcal{O}}(N^{3/2})$ pre-processing and constant delay enumeration of the tuples in the output. The IJDEC reduction is refined in Chapter 5, to lead to more efficient algorithms for the computation of Boolean IJQs, with respect to data complexity. It also leads to the derivation of lower bounds.

Organization. The SETIJ algorithm is described in Section 4.1. The IJDEC reduction is described in Section 4.2.

Notation. Given a sequence s of elements, the ith element in the sequence is denoted by s_i, that is $s = (s_1, \ldots, s_k)$ for some integer k. The sequence $(s_1, \ldots, s_k)$ is also denoted by $(s_1)_{1 \leq i \leq k}$, and the same sequence with the jth element omitted is denoted by $(s_i)_{1 \leq i \leq k, i \neq j}$. Given an integer k, the expression $[1, k]$ is used to denote the set of all integers from 1 to k.

Terminology. Given a variable v, the set of all v-values in a database consists of all the values of all columns named v in the database. Linked lists are used instead of sets or multi-sets to indicate that the elements of the set or multi-set can be structured in such a way using pointers so that given a pointer that points one element from the set, the rest of the elements can be accessed one after the other with constant delay. Given a hypertree decomposition of a CQ (Definition 3.1.16), the materialisation of a bag in the decomposition is a relation which is the result of the full CQ which arises by projecting all the relations of the input CQ on the variables which are included in the bag.

4.1 Set Intersection Join

This section describes an enumeration algorithm, called SetIJ, which computes the intersection join of N multi-sets of intervals of size N. This algorithm uses $\mathcal{O}(k^2 \cdot N \cdot \log(k \cdot N))$ preprocessing time, and constant delay enumeration of the output. It is inspired by Chazzelle's approach to counting the number of intersecting pairs between two sets of intervals in two dimensions [20]. The set intersection join problem is a special case of the general IJQ evaluation problem addressed by this thesis. As a result, the above complexity result is implied by the developments that follow in Section 4.2, and also in Chapter 5. Nonetheless, the description of this algorithm is included in this thesis because it is simple and contains key ideas that led to the results presented in the following.

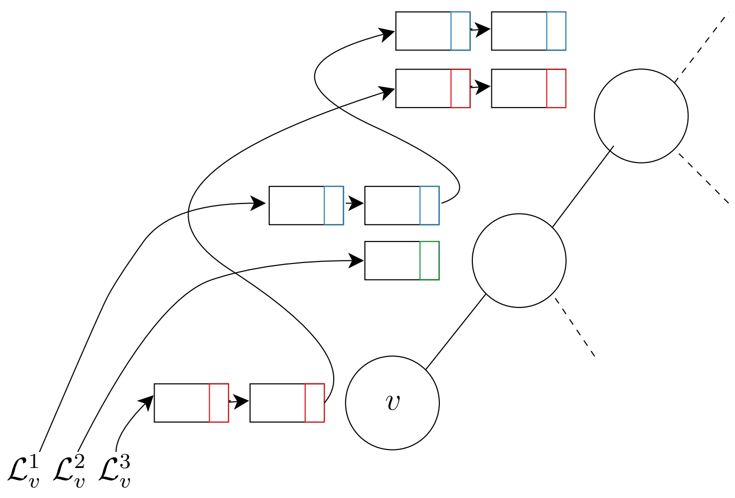


Figure 4.1: This figure illustrates a part of a segment tree over three sets of intervals: a red, a green, and a blue. The leaf v, then, is associated with three linked lists, one for each one of the sets. Each linked list consist of the intervals from the corresponding set, that cover the segment of v.

4.1.1 The Algorithm

Before moving on to the description of the algorithm, it is essential to make a few adjustments to the segment tree (Section 3.3). Consider k multi-sets of intervals $S_1, \dots, S_k$, and a segment tree over the multi-set $\mathcal{I} = \bigcup_{1 \leq i \leq k} S_i$. The first adjustment is that each node u is associated with k canonical subsets $\mathcal{I}_u^i$ for each $i \in [1, k]$, such that $\mathcal{I}_u^i \subseteq S_i$ and $\mathcal{I}_u = \bigcup_{1 \leq i \leq k} \mathcal{I}_u^i$. Recall that $\mathcal{I}_u$ is defined in Equation (3.7). The second adjustment is that each leaf v is associated with k linked lists $\mathcal{L}_v^i$, for each $i \in [1, k]$, where each linked list is constructed as follows: starting from the empty linked list, append to it all the linked lists that represent the canonical subsets of the nodes that lie along the path from v towards the root. Appending two linked lists takes $\mathcal{O}(1)$. Therefore, this can be done in $\mathcal{O}(k \cdot N \cdot \log(k \cdot N))$ by traversing all the paths that start from a leaf and end up at the root. The linked lists associated with the leaves do not occupy extra space; this becomes obvious by checking the illustration in Figure 4.1. Furthermore, by construction, the multi-set represented by $\mathcal{L}_v^i$ contains all the intervals from S_i that cover the segment $\mathsf{seg}_{\mathcal{I}}(v)$.

Algorithm 2 receives as an input k multi-sets $S_1, \dots, S_k$ of intervals. It returns as an output $\mathcal{S}$, which is a data structure that one can use to enumerate the tuples of

Algorithm 2 Set Intersection Join — Preprocessing

```
1: procedure SetIJ($S_1, \dots, S_k$: $k$ lists of intervals)
2:     Construct a segment tree $\mathfrak{T}_{\mathcal{I}}$ over $\mathcal{I} = \bigcup_{i \in [k]} S_i$.
3:     Set $\mathcal{S} = \{\}$;
4:     for each $1 \le i \le k$ do
5:         for each $s_i \in S_i$ do
6:             $u = $ Query(root($\mathfrak{T}_{\mathcal{I}}$), $s_i$.start)
7:             if $\mathcal{L}_u^j \neq \emptyset$ for each $1 \le j \le k, i \neq j$ then
8:                 Insert the tuple $(s_i, (\mathcal{L}_u^j)_{1 \le j \le k, j \neq i})$ to $\mathcal{S}$;
9:             end if
10:        end for
11:    end for
12:    return $\mathcal{S}$;
13: end procedure
```

intersecting intervals from $S_1 \times \cdots \times S_k$. The steps to construct $\mathcal{S}$ are described as follows. For each interval $s_i \in S_i$ collect all the intervals from all the multi-sets other than S_i that contain the left endpoint of s_i, which is s_i.start, using as a subroutine Algorithm 1. Let $\mathcal{L}_u^j$ be the linked list that consists of all the intervals in S_j that contain s_i.start, for each $j \in [1, k]$, $j \neq i$. If $\mathcal{L}_u^j \neq \emptyset$ for each $j \in [1, k]$, $j \neq i$, which means at least one interval from each list is collected, then the tuple $(s_i, (\mathcal{L}_u^j)_{1 \le j \le k, j \neq i})$ is inserted to $\mathcal{S}$. Note that in this context $\mathcal{L}_u^j$ represents a pointer to the corresponding linked list and not the entire list. The procedure described above is repeated for each $i \in [1, k]$. The algorithm returns the data structure $\mathcal{S}$.

Algorithm 3 Set Intersection Join — Enumeration

```
1: procedure Enumeration($\mathcal{S}$: output from Algorithm 2)
2:     for each tuple in $\mathcal{S}$ do
3:         Report the tuples from Equation (4.1);
4:     end for
5: end procedure
```

Using the data structure $\mathcal{S}$, which was constructed during the preprocessing phase (Algorithm 2), one can enumerate all the tuples of intersecting intervals in $S_1 \times \cdots \times S_k$ (Algorithm 3). This can be done by iterating through each tuple $(s_i, (\mathcal{L}_u^j)_{1 \le j \le k, j \neq i})$

in $\mathcal{S}$, and reporting all the tuples in

$$\left(\bigtimes_{1 \le j < i} \mathcal{L}_u^j \right) \times \{s_i\} \times \left(\bigtimes_{i < j \le k} \mathcal{L}_u^j \right). \tag{4.1}$$

For each s_i at least one tuple from the output is reported. Additionally, the next tuple can be accessed with a constant delay, enabling the enumeration of all tuples in the output one after the other with constant delay.

4.1.2 Correctness

This subsection provides a proof of the algorithm's correctness (Proposition 4.1.2), which relies on the following crucial property.

Property 4.1.1 (Starting Point of an Intersection)**.** *Given any tuple of intersecting intervals $(s_1, \ldots, s_k) \in S_1 \times \cdots \times S_k$, the starting endpoint of their intersection coincides with the starting endpoint $s_i.\mathsf{start}$ of the interval with the rightmost left endpoint among $s_1, \ldots, s_k$. The point $s_i.\mathsf{start}$ is called the starting point of the intersection of the intervals $s_1, \ldots, s_k$.*

Proposition 4.1.2 (Set Intersection Correctness)**.** *A tuple $(s_1, \ldots, s_k) \in S_1 \times \cdots \times S_k$ is reported by Algorithm 3 if and only if the intervals in the tuple intersect. Furthermore, each tuple of intersecting intervals is reported only once.*

Proof. "$\Longleftarrow$": Assume there exists a tuple of intersecting intervals $(s_1, \ldots, s_k) \in S_1 \times \cdots \times S_k$. According to Property 4.1.1, the starting endpoint of the interval $\bigcap_{1 \le i \le k} s_i$ coincides with the starting endpoint $s_i.\mathsf{start}$ for some $i \in [1, k]$. During the preprocessing phase, Algorithm 2, takes the interval $s_i \in S_i$ and collects all the subsets $\mathcal{L}_u^j \subseteq S_j$, such that if $s_i.\mathsf{start} \in s_j$ then $s_j \in \mathcal{L}_u^j$, for each $j \in [1, k], j \neq i$. In the enumeration phase (Algorithm 3), takes the interval s_i and reports all the tuples in the Cartesian product of Equation (4.1). Therefore, the tuple $(s_1, \ldots, s_k)$ will be reported by Algorithm 3.

In case any two intervals x_i, x_j in $(x_1, \ldots, x_k)$ have coinciding left endpoints, the tuple of intersecting intervals $(x_1, \ldots, x_k)$ will be found in two iterations, one that collects all intervals that cover x_i.start, and one that collects all the intervals that cover x_j.start. To ensure that each intersection is reported once, one can apply the following adjustment: on the i-th iteration, the left endpoints of the intervals of the lists S_j for each $j \in [1, k]$, $i < j$ are considered to be open. Assume that x_i, x_j in $(x_1, \ldots, x_k)$ have coinciding left endpoints and $i < j$, then the tuple $(x_1, \ldots, x_k)$ will be found only in the j-th iteration.

"$\implies$": Consider a tuple $(s_1, \ldots, s_k) \in S_1 \times \cdots \times S_k$ that is reported by Algorithm 3. Let s_i be the interval with the leftmost right endpoint. Since the tuple is reported, it means that $s_j \in \mathcal{L}_u^j$ for each $j \in [1, k], i \neq j$. All the intervals included in the linked lists $\mathcal{L}_u^j$ for each $j \in [1, k], i \neq j$ have a point in common, that is the point s_i.start. Hence, the intervals s_j for each $j \in [1, k], i \neq j$ have the point s_i.start in common. Therefore, it holds that $\left(\bigcap_{1 \leq i \leq k} s_i\right) \neq \emptyset$. □

4.1.3 Complexity Analysis

Proposition 4.1.3 (Set Intersection Join Complexity). *Algorithms 2 and 3 can be used to report all the tuples of intervals in $S_1 \times \cdots \times S_k$ that intersect, using $\mathcal{O}(k^2 \cdot N \cdot \log(k \cdot N))$ preprocessing time, and constant delay enumeration.*

Proof. The preprocessing and enumeration phases are considered separately. Consider the preprocessing phase. Consider an integer $i \in [1, k]$. For every interval $s_i \in S_i$ and for each $j \in [1, k]$ where $i \neq j$, Algorithm 2 collects all intervals in S_j such that they cover the start of s_i. This can be achieved in $\mathcal{O}(k \cdot \log(k \cdot N))$ time using Algorithm 1. Therefore, the time complexity for i will be $\mathcal{O}(k \cdot N \cdot \log(k \cdot N))$. The above procedure is repeated for each $i \in [1, k]$. Therefore, the overall time complexity will be $\mathcal{O}(k^2 \cdot N \cdot \log(k \cdot N))$.

Turning to the enumeration phase, the structure of $\mathcal{S}$ ensures that Algorithm 3 yields at least one tuple per iteration. Therefore, the output tuples can be enumerated with constant delay. □

4.2 Intersection Join Decomposition

This section introduces a simple reduction, called Intersection Join Decomposition (IJDec), that reduces the problem of evaluating an IJQ on a database to the equivalent problem of evaluating a union of CQs on a new database. This reduction makes it possible to use algorithms that were originally developed for CQs to solve IJQs. The derived bounds are based on the data complexity where it is assumed that the query size is fixed and that the data size is the only parameter in the complexity.

In order to make the analysis that follows clear, it is necessary to slightly modify the IJQ syntax. The modification of the syntax only applies to this chapter, hence, the original syntax is used in the next chapters. According to the syntax and semantics of IJQs, which are presented in Section 2.2, an IJQ Q is a conjunction of relation schemas, where the multiple occurrences of the same variable in different relation schemas imply an intersection join, or in other words, imply the existence of an intersection predicate over those variables. In the following, the syntax of an IJQ is modified in a way such that the intersection joins are expressed using intersection predicates explicitly. In particular, all the variables in $\mathsf{vars}(Q)$ are renamed to have unique names. For each relation schema $R_i(e_i)$ in $\mathsf{schema}(Q)$, for each variable v in e_i rename v to v_i. The intersection joins are then represented explicitly by using intersection predicates, as defined formally in the following.

Definition 4.2.1 (Intersection Predicate). *Consider a multi-set of intervals $S = \{x_1, \ldots, x_m\}$. An intersection predicate over S is the assertion $\left(\bigcap_{1 \leq i \leq m} x_i\right) \neq \emptyset$.*

The above modification results in an IJQ that can be expressed as a conjunction of relation schemas with disjoint sets of variables, and a conjunction of intersection predicates. In particular, it can be written as

$$Q = \langle e, \mathsf{conj} \wedge \mathsf{ij} \rangle, \tag{4.2}$$

where conj denotes a conjunction of relation schemas with disjoint sets of variables, and ij denotes a conjunction of intersection predicates. The set e, also denoted by $\mathsf{free}(Q)$, consists of the new variables that correspond to each free variable in the original syntax. The new set of variables in Q is denoted by $\mathsf{vars}(Q)$ and the new schema of Q is denoted by $\mathsf{atoms}(Q)$. It must be noted that the above syntax allows the same variable to occur in multiple intersection predicates in the query, whereas the original syntax of an IJQ does not (Section 2.2, Equation (2.1)). Since the purpose of the above modification is to establish an equivalent syntax, this restriction is also applied in Q from Equation (4.2). Every variable is restricted so that it occurs in at most one intersection predicate.

The following example explains the usage of predicates in the modified syntax.

Example 4.2.2 (The full Triangle IJQ with Predicates)**.** *Consider the full Triangle IJQ from Example 2.3.1. Following the syntax introduced in Equation* (4.2)*, the query* $Q_\triangle$ *is rewritten as*

$$\begin{aligned} Q_\triangle = \langle \{a_1, a_2, b_1, b_2, c_2, c_3\}, &R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3]) \\ &\wedge ((a_1 \cap a_3) \neq \emptyset) \wedge ((b_1 \cap b_2) \neq \emptyset) \wedge ((c_2 \cap c_3) \neq \emptyset)\rangle. \end{aligned}$$

The new set of free variables is $\mathsf{free}(Q_\triangle) = \{a_1, a_3, b_1, b_2, c_2, c_3\}$*. The new schema of the query is* $\mathsf{schema}(Q) = \{R_1([a_1],[b_1]),\ R_2([b_2],[c_2]),\ R_3([a_3],[c_3])\}$*.*

The key idea of the IJDec reduction is the efficient decomposition of the materialised predicates in the query. In reality, the reduction never replaces an intersection predicate by its materialisation because this would be inefficient. Instead, an intersection predicate will be replaced by the decomposition of its materialisation, which is efficient to compute and store. The concept of a predicate's materialisation is introduced in the following in order to effectively define its decomposition later.

Definition 4.2.3 (Materialised Intersection Predicate)**.** *Consider an IJQ Q following the syntax in Equation* (4.2) *and a database D. Consider also an intersection predicate $\left(\bigcap_{1\leq i\leq m} x_i\right) \neq \emptyset$ in ij, where $\{x_1, \dots, x_m\} \subseteq \mathsf{vars}(Q)$. Let $\mathcal{X}$ be the set of all x_i-intervals in D. The materialisation of the intersection predicate is a relation $\mathsf{pred}_{\mathcal{X}}([x_1], \dots, [x_m])$ which is defined as*

$$\left\{(t_1, \dots, t_m) \middle| (t_1, \dots, t_m) \in \mathcal{X}^m, \left(\bigcap_{1\leq i\leq m} t_i\right) \neq \emptyset\right\}. \tag{4.3}$$

Definition 4.2.4 (Rewriting with Materialised Predicates)**.** *Consider an IJQ Q following the syntax in Equation* (4.2) *and a database D. Define a query Q^+ as*

$$Q^+ = \langle e, \mathsf{conj} \wedge \mathsf{ij}^+\rangle, \tag{4.4}$$

where ij^+ consists of all the relation schemas that correspond to the materialisation of each intersection predicate in ij from Definition 4.2.3. Define a database D^+ as the union of D with the set of the relation instances that correspond to the materialised intersection predicates in the query Q^+.

The following example is a continuation of Example 4.2.2, which explains the usage of materialised predicates introduced above.

Example 4.2.5 (The full Triangle IJQ with Materialised Predicates)**.** *According to Definition 4.2.4 the query $Q_\triangle$ from Example 4.2.2 can be rewritten as*

$$Q^+_\triangle = \langle\{a_1, a_3, b_1, b_2, c_2, c_3\}, R_1([a_1], [b_1]) \wedge R_2([b_2], [c_2]) \wedge R_3([a_3], [c_3])$$
$$\wedge\, \mathsf{pred}_{\mathcal{A}}([a_1], [a_3]) \wedge \mathsf{pred}_{\mathcal{B}}([b_1], [b_2]) \wedge \mathsf{pred}_{\mathcal{C}}([c_2], [c_3])\rangle,$$

where $\mathcal{A}$ is the set of a_1 and a_3-intervals; $\mathcal{B}$ is the set of b_1 and b_2-intervals; and $\mathcal{C}$ is the set of c_2 and c_3-intervals in the database D. The set of free variables is the same as before, that is $\mathsf{free}(Q_\triangle) = \mathsf{free}(Q^+_\triangle)$. The database D^+ is defined by $D \cup \{\mathsf{pred}_\mathcal{X} \mid \mathcal{X} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}\}$, where the new relations that correspond to the materialised predicates are defined according to Definition 4.3.

4.2.1 Predicate Rewriting

Given that N is the size of each relation in D, a materialised intersection predicate (Definition 4.2.3) over m variables can be computed in $\mathcal{O}(N^m)$ time and requires analogous space, leading to high complexity. The key observation on which IJDec is based, is that the relation that represents a materialised intersection predicate admits an efficient decomposition into a set of binary relations of size $\mathcal{O}(N \cdot \log N)$. These "helper" relations are classified and referred to as index and seek relations (see Definitions 4.2.7 and 4.2.6). By "combining" those relations appropriately one can reconstruct the corresponding materialised predicate (Lemma 4.2.10).

Helper Relations

An index relation associates an interval from the set of intervals $\mathcal{X}$, to the nodes of the segment tree on $\mathcal{X}$, which maximally cover that interval (see Definition 3.3.4). As a remainder, $\mathcal{X}$ is the set of all x_i-intervals in D.

Definition 4.2.6 (Index Relation)**.** *Consider a set of intervals $\mathcal{X}$ and a segment tree on $\mathcal{X}$. Define a relation $\mathsf{idx}_\mathcal{X}([x], v)$, called index relation with respect to $\mathcal{X}$, as*

$$\mathsf{idx}_\mathcal{X} = \{(t_1, t_2) \mid t_1 \in \mathcal{X}, t_2 \in \mathsf{cp}_\mathcal{X}(t_1)\}.$$

A seek relation associates any interval from a set of intervals $\mathcal{X}$ to the nodes of a segment tree on $\mathcal{X}$ that belong to the root-to-leaf path that ends up in the leaf that contains the starting endpoint of that interval.

Definition 4.2.7 (Seek Relation)**.** *Consider a set of intervals $\mathcal{X}$ and a segment tree on $\mathcal{X}$. Define a relation $\mathsf{seek}_{\mathcal{X}}([x], v)$, called seek relation with respect to $\mathcal{X}$, as*

$$\mathsf{seek}_{\mathcal{X}} = \{(t_1, t_2) \mid t_1 \in \mathcal{X}, t_2 \in \mathsf{anc}(\mathsf{leaf}_{\mathcal{X}}(t_1))\}$$

Lemma 4.2.8 (Size of Index and Seek relations)**.** *The size and construction time of both $\mathsf{idx}_{\mathcal{X}}$ and $\mathsf{seek}_{\mathcal{X}}$ is $\mathcal{O}(N \cdot \log N)$.*

Proof. According to Definition 4.2.6, the relation $\mathsf{idx}_{\mathcal{X}}$ maps each interval x from $\mathcal{X}$ to each one of the nodes in the canonical partition $\mathsf{cp}_{\mathcal{X}}(x)$ (Definition 3.3.4). By Property 3.3.7, the size and construction time of $\mathsf{cp}_{\mathcal{X}}(x)$ is $\mathcal{O}(\log N)$. Therefore, the size and construction time of $\mathsf{idx}_{\mathcal{X}}$ is $\mathcal{O}(N \cdot \log N)$. According to Definition 4.2.6, the relation $\mathsf{seek}_{\mathcal{X}}$ maps each interval x from $\mathcal{X}$ to each one of the nodes in the path that starts from the root of the segment tree on $\mathcal{X}$ to the leaf that corresponds to the left endpoint of x. By construction, the segment tree on $\mathcal{X}$ has a height of $\mathcal{O}(\log N)$, and the nodes within any path from a leaf to the root can be retrieved in analogous time. Therefore, the size and construction time of $\mathsf{seek}_{\mathcal{X}}$ is $\mathcal{O}(N \cdot \log N)$. □

Each interval in the relation is paired with at most logarithmic, in the size of data, nodes from the segment tree for both the index and seek relations. As will be shown in Subsection 4.2.2, the above mentioned property is useful when arguing about the complexity of the reduced problem.

Property 4.2.9 (Degrees of Values)**.** *Consider any relation constructed by Definitions 4.2.7 and 4.2.6. The following statements hold.*

- *Every value in $\pi_{\{x\}}(\mathsf{seek}_{\mathcal{X}}([x], v))$ has degree of $\mathcal{O}(\log N)$.*
- *Every value in $\pi_{\{x\}}(\mathsf{idx}_{\mathcal{X}}([x], v))$ has degree of $\mathcal{O}(\log N)$.*

Proof. By Property 3.3.7, for each x-interval there can be at most $\mathcal{O}(\log N)$ nodes in $\mathsf{cp}_{\mathcal{X}}(x)$. Therefore, by the construction of $\mathsf{idx}_{\mathcal{X}}([x], v)$ each x-value corresponds to at most $\mathcal{O}(\log N)$ v-values. By Definition 3.3.1, a segment tree on $\mathcal{X}$ is a complete binary tree of height $\mathcal{O}(\log N)$. Hence, for each x-interval there can be at most $\mathcal{O}(\log N)$ nodes in the set $\mathsf{anc}(\mathsf{leaf}_{\mathcal{X}}(x))$. Therefore, by the construction of $\mathsf{seek}_{\mathcal{X}}([x], v)$ each x-value corresponds to at most $\mathcal{O}(\log N)$ v-values. $\square$

Predicate Rewriting

The expression in the following lemma yields the materialised intersection predicate from Definition 4.2.3

Lemma 4.2.10 (Rewriting of the Materialised Predicate)**.** *The materialised predicate* $\mathsf{pred}_{\mathcal{X}}([x_1], \ldots, [x_m])$ *from Definition 4.2.3 can be rewritten as*

$$\bigvee_{1 \leq i \leq m} \left[\bigwedge_{\substack{1 \leq j \leq m \\ i \neq j}} \exists x_{ij} \left(\mathsf{seek}_{\mathcal{X}}([x_i], x_{ij}) \wedge \mathsf{idx}_{\mathcal{X}}([x_j], x_{ij})\right) \right], \tag{4.5}$$

where the corresponding relation instances for $\mathsf{seek}_{\mathcal{X}}$ *and* $\mathsf{idx}_{\mathcal{X}}$ *are defined according to Definitions 4.2.7 and 4.2.6 respectively.*

Proof. "$\Longrightarrow$": Consider a tuple $(x_1, \ldots, x_m) \in \mathsf{pred}_{\mathcal{X}}$. It follows that the intervals in $(x_1, \ldots, x_m)$ intersect. Furthermore, the starting endpoint of their intersection coincides with the starting endpoint of the interval with the rightmost left endpoint among $x_1, \ldots, x_m$. Let x_i be the interval with the rightmost left endpoint. It holds that $x_i.\mathsf{start} \in x_j$ for each $j \in [1, m]$, $i \neq j$. By the construction of the segment tree, every interval x_j for each $j \in [1, m]$, $i \neq j$, covers a segment that corresponds to a node within the path that starts from the root and ends up at $\mathsf{leaf}_{\mathcal{X}}(x_i)$. Hence, for each x_j where $j \in [1, m]$, $i \neq j$, there exists a node $x_{ij} \in \mathsf{cp}_{\mathcal{X}}(x_j)$ such that $x_{ij} \in \mathsf{anc}(\mathsf{leaf}_{\mathcal{X}}(x_i))$. Therefore, by the Definitions 4.2.7 and 4.2.6, there exists an integer $i \in [1, m]$ and nodes x_{ij} for each $j \in [1, m]$, $j \neq i$ such that the inner conjunction of Equation (4.5) is satisfied.

"$\Longleftarrow$": Assume that there exists an integer $i \in [1, m]$, a tuple of intervals $(x_1, \ldots, x_m)$, and nodes x_{ij} for $j \in [1, m]$, $j \neq i$, that satisfy Equation (4.5). By the Definitions 4.2.7 and 4.2.6, it follows that the intervals x_j for $j \in [1, m]$, $i \neq j$, cover the segment of some node that falls within the path $\mathsf{anc}(\mathsf{leaf}_{\mathcal{X}}(x_i))$. Hence, by the construction of the segment tree, the intervals in $(x_1, \ldots, x_m)$ have a point in common, which is $x_i.\mathsf{start}$. Therefore, $(x_1, \ldots, x_m) \in \mathsf{pred}_{\mathcal{X}}$. □

Remark 4.2.11. *An intersection join query can be rewritten equivalently as a disjunction of CQs with inequality joins* ($\leq, <, \geq, >$)*. Consider an IJQ following the syntax with intersection predicates described previously. Each intersection predicate can be replaced by its equivalent predicate that includes inequalities. Consider a multi-set* $\{x_1, \ldots, x_m\}$ *of intervals. The intersection predicate from Definition 4.2.1 is equivalent to*

$$\bigvee_{1 \leq i \leq m} \left[\bigwedge_{\substack{1 \leq j \leq m \\ j \neq i}} (x_j.\mathsf{start} \leq x_i.\mathsf{start} \leq x_j.\mathsf{end}) \right]. \tag{4.6}$$

Notice that there is a bijection between the conjuncts in the intersection predicate decomposition (Lemma 4.2.10) and the equivalent predicate using inequalities in Equation 4.6. This gives an insight for the correspondence between the conjuncts in Equation (4.5)*, with the conjuncts in Equation* (4.6)*.*

Disjointness

For the correct computation of the tuples in output, it is essential to make an adjustment so that for each tuple of intersecting intervals $(x_1, \ldots, x_m)$, there exists exactly one $i \in [1, m]$ that satisfies the right-hand side in Lemma 4.2.10. Otherwise, the same tuple will be reported multiple times. In other words, it must hold that the intervals $(x_1, \ldots, x_m)$ intersect if and only if exactly one of the conjuncts is true. Notice that currently, this is not guaranteed. Consider the case where two intervals x_a, x_b, where $a, b \in [1, m]$ and $a < b$, have coinciding left endpoints then the expression on the right-hand side is true for both $i = a$ and $i = b$.

Definition 4.2.12 (Index Relation with Left-open Intervals). *Consider a multi-set of intervals $\mathcal{X}$, and a segment tree on $\mathcal{X}$. Define a relation $\mathsf{idx}'_{\mathcal{X}}([x], v)$, called the index relation (with left-open intervals) over the set of variables $\mathcal{X}$ as*

$$\mathsf{idx}'_{\mathcal{X}} = \{(t_1, t_2) \mid t_1 \in \mathcal{X}, t_2 \in \mathsf{cp}_{\mathcal{X}}((t_1.\mathsf{start}, t_1.\mathsf{end}])\}.$$

To ensure that at most one conjunct is true, one can apply the following adjustment: on the i-th iteration if $i < j$, the relation $\mathsf{idx}_{\mathcal{X}}([x_j], x_{ij})$ is replaced with the corresponding relation $\mathsf{idx}'_{\mathcal{X}}([x_j], x_{ij})$.

Property 4.2.13 (Decomposition Functional Dependency). *Consider the materialized predicate decomposition from Definition 4.2.10, and a tuple of intervals $(x_1, \ldots, x_m)$ that intersect. Given an $i \in [1, m]$, there is precisely one tuple $(x_{i1}, \ldots, x_{im})$ that satisfies the predicate.*

Proof. We prove that for any $j \in [1, m], j \neq i$, there exists no $x'_{ij} \neq x_{ij}$ such that both $\mathsf{seek}_{\mathcal{X}}(x_i, x_{ij}) \wedge \mathsf{idx}_{\mathcal{X}}(x_j, x_{ij})$ and $\mathsf{seek}_{\mathcal{X}}(x_i, x'_{ij}) \wedge \mathsf{idx}_{\mathcal{X}}(x_j, x'_{ij})$ are true. Assume for contradiction that for some $j \in [1, m]$, $j \neq i$, there exists such nodes $x'_{ij} \neq x_{ij}$. Then, by Definition 4.2.6, we have that $x_{ij}, x'_{ij} \in \mathsf{cp}_{\mathcal{X}}(x_j)$. By Definition 4.2.7, we have that both x_{ij} and x'_{ij} belong to the path from the root to $\mathsf{leaf}_{\mathcal{X}}(x_i)$, which means that they are ancestors of one another. By Property 3.3.6, this is a contradiction. □

4.2.2 Reduction and Evaluation

By Lemma 4.2.10, a materialised intersection predicate can be equivalently rewritten as a disjunction of expressions, where each expression is a conjunction of seek and index relations. This equivalence is fundamental as it enables us to transform the problem of evaluating an IJQ to the problem of evaluating a union of CQs.

Definition 4.2.14 (IJDec Reduction)**.** *Consider an IJQ with materialised predicates Q^+ (Definition 4.2.4) and a database D. Define Q^{dec} as*

$$Q^{dec} = \langle e, \mathsf{conj} \wedge \mathsf{ij}^{dec} \rangle,$$

where ij^{dec} consists of all the equivalent predicates that correspond to the intersection predicates in ij^+ according to Definition 4.2.10. By writing the query Q^{dec} in disjunctive normal form, one can obtain a union of CQs. The set of free variables is $\mathsf{free}(Q^{dec}) = \mathsf{free}(Q^+)$. Define D^{dec} as the union of D and the set of the helper relation instances corresponding to the intersection predicates.

Proposition 4.2.15 (Correctness)**.** *Consider an IJQ Q and a database D. It holds that $Q(D) = Q^{dec}(D^{dec})$.*

Proof. The proof follows immediately from Lemma 4.2.10. □

Proposition 4.2.16. *Consider an IJQ Q and a database D. The following statements hold.*

- *The size of Q^{dec} is within a polynomial factor from the size of Q.*
- *Given that each relation in D is of size N, the size of each relation in D^{dec} is $\mathcal{O}(N \cdot \log N)$ and its construction time is proportional to its size.*

Proof. Each $m$ry materialised intersection predicate in Q^+ generates m conjuncts — one for each variable in the predicate (Lemma 4.2.10). The number m is at most $|\mathsf{atoms}(Q)|$. Furthermore, there are at most $|\mathsf{vars}(Q)|$ predicates in the query. Therefore, the number of generated CQs are $|\mathsf{rels}(Q)| \cdot |\mathsf{vars}(Q)|$. Each CQ in the reduction consists of at most $2 \cdot |\mathsf{rels}(Q)| \cdot |\mathsf{vars}(Q)|$ additional relations. The second statement in Proposition 4.2.16 is justified by Lemma 4.2.8. □

The rewriting eliminates all intersection join predicates in the input query. Furthermore, given any query in the union Q^{dec}, all the joins in the query are equality joins. This is justified by the fact that only the newly added scalar variables are join variables. Therefore, the query Q^{dec} is a union of CQs. Notice that the set of free variables in the query is the same, i.e. $\mathsf{free}(Q^+) = \mathsf{free}(Q^{dec})$, and also that, if the the set of free variables is empty, then this union becomes a disjunction, and hence the queries in Q^{dec} are Boolean CQs. The key takeaway is that the reduction breaks down the computation required to obtain the result of the query into disjoint sub-tasks that are expressible as CQs. Each CQ "contributes" independently to the computation of different parts of the output. This enables the concurrent computation of the CQs.

Upper Bounds

The reduction of any IJQ to a union of CQs provides a way for obtaining upper bounds for the runtime of the IJQ, using known upper bounds for the runtime of CQs. In particular, the upper bound for the runtime an IJQ is the same as the upper bound for the CQ with the maximum runtime in the union constructed by the reduction.

4.2.3 The Triangle

The IJDec reduction leads to an interesting result in data complexity: an algorithm that computes the Boolean Triangle IJQ on any database of size N in $\tilde{\mathcal{O}}(N^{3/2})$ time, and any full IJQ, in $\tilde{\mathcal{O}}(N^{3/2})$ preprocessing time and constant delay enumeration of the tuples in the output.

Consider the Triangle $Q_\triangle$ introduced in Example 2.3.1 and its rewriting obtained in Examples 4.2.2 and 4.2.5. By replacing each materialised intersection predicate in the modified query by its decomposition (Lemma 4.2.10), the query is rewritten as

$$\begin{aligned} Q_{\triangle}^{dec} = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, & R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3]) \\ & \wedge (\mathsf{seek}_{\mathcal{A}}([a_1], a_{13}) \wedge \mathsf{idx}_{\mathcal{A}}([a_3], a_{13})) \vee (\mathsf{seek}_{\mathcal{A}}([a_3], a_{31}) \wedge \mathsf{idx}_{\mathcal{A}}([a_1], a_{31})) \\ & \wedge (\mathsf{seek}_{\mathcal{B}}([b_1], b_{12}) \wedge \mathsf{idx}_{\mathcal{B}}([b_2], b_{12})) \vee (\mathsf{seek}_{\mathcal{B}}([b_2], b_{21}) \wedge \mathsf{idx}_{\mathcal{B}}([b_1], b_{21})) \\ & \wedge (\mathsf{seek}_{\mathcal{C}}([c_2], c_{23}) \wedge \mathsf{idx}_{\mathcal{C}}([c_3], c_{23})) \vee (\mathsf{seek}_{\mathcal{C}}([c_3], c_{32}) \wedge \mathsf{idx}_{\mathcal{C}}([c_2], c_{32}))\rangle. \end{aligned}$$

The above query can be further rewritten in disjunctive normal form as $Q_{\triangle}^{dec} = \bigvee_{1 \le i \le 8} Q_i$, where

$$\begin{aligned} Q_1 = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, & R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3]) \\ & \wedge (\mathsf{seek}_{\mathcal{A}}([a_1], a_{13}) \wedge \mathsf{idx}_{\mathcal{A}}([a_3], a_{13})) \wedge (\mathsf{seek}_{\mathcal{B}}([b_1], b_{12}) \wedge \mathsf{idx}_{\mathcal{B}}([b_2], b_{12})) \\ & \wedge (\mathsf{seek}_{\mathcal{C}}([c_2], c_{23}) \wedge \mathsf{idx}_{\mathcal{C}}([c_3], c_{23}))\rangle \end{aligned}$$

$$\begin{aligned} Q_2 = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, & R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3]) \\ & \wedge (\mathsf{seek}_{\mathcal{A}}([a_1], a_{13}) \wedge \mathsf{idx}_{\mathcal{A}}([a_3], a_{13})) \wedge (\mathsf{seek}_{\mathcal{B}}([b_1], b_{12}) \wedge \mathsf{idx}_{\mathcal{B}}([b_2], b_{12})) \\ & \wedge (\mathsf{seek}_{\mathcal{C}}([c_3], c_{32}) \wedge \mathsf{idx}_{\mathcal{C}}([c_2], c_{32}))\rangle \end{aligned}$$

$$\begin{aligned} Q_3 = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, & R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3]) \\ & \wedge (\mathsf{seek}_{\mathcal{A}}([a_1], a_{13}) \wedge \mathsf{idx}_{\mathcal{A}}([a_3], a_{13})) \wedge (\mathsf{seek}_{\mathcal{B}}([b_2], b_{21}) \wedge \mathsf{idx}_{\mathcal{B}}([b_1], b_{21})) \\ & \wedge (\mathsf{seek}_{\mathcal{C}}([c_2], c_{23}) \wedge \mathsf{idx}_{\mathcal{C}}([c_3], c_{23}))\rangle \end{aligned}$$

$$Q_4 = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3])$$
$$\wedge\ (\mathsf{seek}_{\mathcal{A}}([a_1], a_{13}) \wedge \mathsf{idx}_{\mathcal{A}}([a_3], a_{13})) \wedge (\mathsf{seek}_{\mathcal{B}}([b_2], b_{21}) \wedge \mathsf{idx}_{\mathcal{B}}([b_1], b_{21}))$$
$$\wedge\ (\mathsf{seek}_{\mathcal{C}}([c_3], c_{32}) \wedge \mathsf{idx}_{\mathcal{C}}([c_2], c_{32}))\rangle$$

$$Q_5 = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3])$$
$$\wedge\ (\mathsf{seek}_{\mathcal{A}}([a_3], a_{31}) \wedge \mathsf{idx}_{\mathcal{A}}([a_1], a_{31})) \wedge (\mathsf{seek}_{\mathcal{B}}([b_1], b_{12}) \wedge \mathsf{idx}_{\mathcal{B}}([b_2], b_{12}))$$
$$\wedge\ (\mathsf{seek}_{\mathcal{C}}([c_2], c_{23}) \wedge \mathsf{idx}_{\mathcal{C}}([c_3], c_{23}))\rangle$$

$$Q_6 = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3])$$
$$\wedge\ (\mathsf{seek}_{\mathcal{A}}([a_3], a_{31}) \wedge \mathsf{idx}_{\mathcal{A}}([a_1], a_{31})) \wedge (\mathsf{seek}_{\mathcal{B}}([b_1], b_{12}) \wedge \mathsf{idx}_{\mathcal{B}}([b_2], b_{12}))$$
$$\wedge\ (\mathsf{seek}_{\mathcal{C}}([c_3], c_{32}) \wedge \mathsf{idx}_{\mathcal{C}}([c_2], c_{32}))\rangle$$

$$Q_7 = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3])$$
$$\wedge\ (\mathsf{seek}_{\mathcal{A}}([a_3], a_{31}) \wedge \mathsf{idx}_{\mathcal{A}}([a_1], a_{31})) \wedge (\mathsf{seek}_{\mathcal{B}}([b_2], b_{21}) \wedge \mathsf{idx}_{\mathcal{B}}([b_1], b_{21}))$$
$$\wedge\ (\mathsf{seek}_{\mathcal{C}}([c_2], c_{23}) \wedge \mathsf{idx}_{\mathcal{C}}([c_3], c_{23}))\rangle$$

$$Q_8 = \langle \{a_1, a_3, b_1, b_2, c_2, c_3\}, R_1([a_1],[b_1]) \wedge R_2([b_2],[c_2]) \wedge R_3([a_3],[c_3])$$
$$\wedge\ (\mathsf{seek}_{\mathcal{A}}([a_3], a_{31}) \wedge \mathsf{idx}_{\mathcal{A}}([a_1], a_{31})) \wedge (\mathsf{seek}_{\mathcal{B}}([b_2], b_{21}) \wedge \mathsf{idx}_{\mathcal{B}}([b_1], b_{21}))$$
$$\wedge\ (\mathsf{seek}_{\mathcal{C}}([c_3], c_{32}) \wedge \mathsf{idx}_{\mathcal{C}}([c_2], c_{32}))\rangle$$

Analysis of Q_4

Consider the Q_4 conjunct. Due to symmetry, the analysis of the rest of the conjuncts is similar. Construct a relation $S_1([a_1], a_{13}, [b_1], b_{21})$ as

$$S_1 = \{(t_1, t_2, t_3, t_4) \mid (t_1, t_3) \in R_1, (t_1, t_2) \in \mathsf{seek}_{\mathcal{A}}, (t_3, t_4) \in \mathsf{idx}_{\mathcal{B}}\}. \tag{4.7}$$

By Property 4.2.9 for each a_1-value in S_1 there are at most $\mathcal{O}(\log N)$ a_{13}-values in $\mathsf{seek}_{\mathcal{A}}$, and for each b_1-value in S_1 there are at most $\mathcal{O}(\log N)$ b_{21}-values $\mathsf{idx}_{\mathcal{B}}$. Hence, the size of S_1 is $\mathcal{O}(N \log^2 N)$. Construct a relation $S_2([b_2], b_{21}, [c_2], c_{32})$ as

$$S_2 = \{(t_1, t_2, t_3, t_4) \mid (t_1, t_3) \in R_2, (t_1, t_2) \in \mathsf{seek}_{\mathcal{B}}, (t_3, t_4) \in \mathsf{idx}_{\mathcal{C}}\}. \tag{4.8}$$

By Property 4.2.9 for each b_2-value in S_2 there are at most $\mathcal{O}(\log N)$ b_{21}-values in $\mathsf{seek}_{\mathcal{B}}$, and for each c_2-value and there are at most $\mathcal{O}(\log n)$ c_{32}-values $\mathsf{idx}_{\mathcal{C}}$. Hence, the size of S_2 is $\mathcal{O}(N \log^2 N)$. Construct a relation $S_3([a_3], a_{13}, [c_3], c_{23})$ as

$$S_3 = \{(t_1, t_2, t_3, t_4) \mid (t_1, t_3) \in R_3, (t_1, t_2) \in \mathsf{idx}_{\mathcal{A}}, (t_3, t_4) \in \mathsf{seek}_{\mathcal{C}}\}. \tag{4.9}$$

By Property 4.2.9, for each a_3-value in S_3 there are at most $\mathcal{O}(\log N)$ a_{13}-values in $\mathsf{idx}_{\mathcal{A}}$, and for each c_3-value and there are at most $\mathcal{O}(\log N)$ c_{23} values $\mathsf{seek}_{\mathcal{C}}$. Hence, the size of S_3 is $\mathcal{O}(N \log^2 N)$. Define the query

$$\begin{aligned} W_4 = \langle \{&a_1, a_3, b_1, b_2, c_2, c_3\}, \\ &S_1([a_1], a_{13}, [b_1], b_{21}) \wedge S_2([b_2], b_{21}, [c_2], c_{32}) \wedge S_3([a_3], a_{13}, [c_3], c_{32})\rangle, \end{aligned}$$

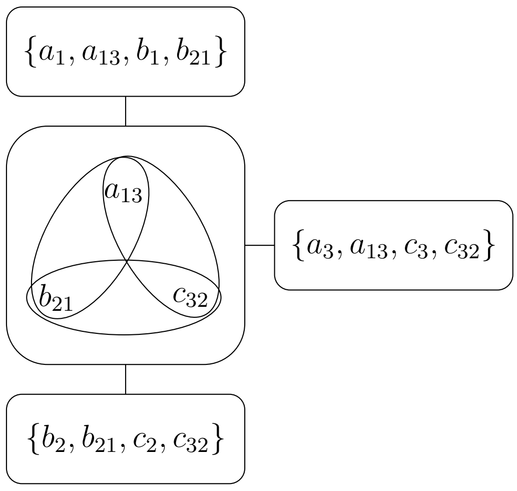
where $\mathsf{free}(W_4) = \mathsf{free}(Q_4)$. Define a database that contains the relations S_1, S_2, and S_3 defined by Equations (4.7), (4.8), and (4.9) respectively.

Boolean Case. Assume that the input query has no free variables. It holds that $\mathsf{free}(W_4) = \emptyset$. Hence, the non-join variables (i.e., a_1, a_3, b_1, b_2, c_2, and c_3) are ignored by taking into account the projections of S_1, S_2, and S_3 on the join variables (i.e., a_{13}, b_{21}, and c_{23}). The projections are $S'_1 = \pi_{\{a_{13},b_{21}\}}(S_1)$, $S'_2 = \pi_{\{b_{21},c_{32}\}}(S_2)$, and $S'_3 = \pi_{\{a_{13},c_{23}\}}(S_3)$ respectively. The result of the query can be obtained by computing the query

$$W'_4 = S'_1(a_{13}, b_{21}) \wedge S'_2(b_{21}, c_{32}) \wedge S'_3(a_{13}, c_{23}). \tag{4.10}$$

This is the Triangle CQ, which can be computed in $\mathcal{O}((N \cdot \log^2 N)^{3/2}) = \mathcal{O}(N^{3/2} \cdot \log^3 N)$.

Full case. Assume that all the variables in the input query are free. In practice, this means that all the tuples that satisfy the input query must be reported in full (Subsection 2.2). By Definition 4.2.14 and the construction of W_4, we have that $\mathsf{free}(W_4) = \{a_1,\ a_3,\ b_1,\ b_2,\ c_2,\ c_3\}$. Consider the hypertree decomposition of W_4 with the following bags $\{a_{13}, b_{21}, c_{32}\}$, $\{a_1, a_{13}, b_1, b_{21}\}$, $\{b_2, b_{21}, c_2, c_{32}\}$, and $\{a_3, a_{13}, c_3, c_{32}\}$, which are visually represented as follows.



$\{a_1, a_{13}, b_1, b_{21}\}$
a_{13}
b_{21}
c_{32}
$\{a_3, a_{13}, c_3, c_{32}\}$
$\{b_2, b_{21}, c_2, c_{32}\}$

More details about hypertree decompositions can be found in Section 3.1. The above hypertree decomposition consists of four bags — a central bag consisting of the variables $\{a_{13}, b_{21}, c_{32}\}$, and three more bags, where each one of them is connected to the central bag only. The central bag is covered by projections $S'_1 = \pi_{\{a_{13}, b_{21}\}}(S_1)$, $S'_2 = \pi_{\{b_{21}, c_{32}\}}(S_2)$ and $S'_3 = \pi_{\{a_{13}, c_{23}\}}(S_3)$. Each one of those projections has size of $\mathcal{O}(N \cdot \log^2 N)$ and can be computed in time analogous to its size. Hence, the materialisation of the central bag requires the computation of the full version of the Triangle CQ W'_4 (Equation 4.10), which is

$$W''_4 = \langle \{a_{13}, b_{21}, c_{32}\}, S'_1(a_{13}, b_{21}) \wedge S'_2(b_{21}, c_{32}) \wedge S'_3(a_{13}, c_{23}) \rangle.$$

This can be done in $\mathcal{O}((N \cdot \log^2 N)^{3/2}) = \mathcal{O}(N^{3/2} \cdot \log^3 N)$, by employing a worst-case optimal algorithm. Hence, the upper bound for the runtime of its computation is $\tilde{\mathcal{O}}(N^{3/2})$. The resulting query, which is the query where the central bag is materialised is α-acyclic. According to Property 4.2.13, each pair of intervals a_1 and a_3 that intersect, functionally determine the scalar a_{13}; each pair of intervals b_1 and b_2 that intersect, functionally determine the scalar b_{21}; and each pair of intervals c_2 and c_3 that intersect, functionally determine the scalar c_{32}. Therefore, the variables a_{13}, b_{21}, and c_{32} can be treated as if they were free. Hence, one can evaluate the full W_4, and then remove the values for those variables from the result, without affecting the size of the output. The full W_4 is α-acyclic free-connex, and thus admits constant delay enumeration of the tuples in the output [9]. The rest of the CQ conjuncts admit a similar analysis.

In conclusion, the above analysis showed that the Boolean Triangle IJQ can be computed in $\tilde{\mathcal{O}}(N^{3/2})$ time, and the full Triangle IJQ can be computed using $\tilde{\mathcal{O}}(N^{3/2})$ preprocessing time and constant delay enumeration of the tuples in the output.

4.3 Discussion

This chapter is called prelude because it presents concepts and preliminary research that directly led to the results reported in the following chapters. Two contributions are made. The first is an enumeration algorithm that computes the intersection join of k sets of intervals of size n using $\mathcal{O}(k^2 \cdot N \cdot \log(k \cdot N))$ preprocessing time, and constant delay enumeration of the tuples in the output. The second is a simple reduction from the problem of evaluating any IJQ to the problem of evaluating a union of CQs, called Intersection Join Decomposition (IJDEC). Interestingly, this reduction shows that the Boolean Triangle IJQ can be evaluated in time $\tilde{\mathcal{O}}(N^{3/2})$ in data complexity, and the full Triangle IJQ can be evaluated in $\tilde{\mathcal{O}}(N^{3/2})$ preprocessing time, followed by constant delay enumeration of the tuples in the output.

Both IJDEC and the reduction introduced in the following chapter are based on the segment tree and the equivalent rewriting of the intersection predicate into a predicate involving equalities. The primary difference is that the reduction from the following chapter captures the intersection of a set of intervals using more cases, by accounting for all potential interval permutations. Although this method produces additional conjuncts in the reduction, it also results in refined upper bounds and the derivation of lower bounds. Consider for example the Boolean IJQ $Q = R_1([a],[b]) \wedge R_2([a],[b]) \wedge R_3([a],[b])$. Using the reduction presented in the next chapter one can obtain an upper bound of $\tilde{\mathcal{O}}(N)$ on any input database, whereas using IJDEC this is not possible. On the other hand, while IJDEC lags behind in terms of asymptotic complexity, it is significantly simpler for certain queries and has a greater chance of becoming practical.

Chapter 5

The Complexity of Boolean IJQs

This chapter establishes the data complexity of Boolean IJQs, using a forward and a backward reduction.

The forward reduction takes as input a Boolean IJQ Q and a database D. It constructs a disjunction $\tilde{Q}$ of Boolean CQs, and a new database $\tilde{D}$ that matches the structure of $\tilde{Q}$ where the size of $\tilde{D}$ is within a poly-logarithmic factor from the size of D. Furthermore, it holds that $Q(D)$ if and only if $\tilde{Q}(\tilde{D})$. The forward reduction shows that one can use existing runtime upper bounds for Boolean CQs to derive an upper bound on the runtime of any IJQ. Specifically, the runtime of Q is upper bounded by the runtime of the Boolean CQ in $\tilde{Q}$ with the maximum runtime.

The backward reduction does the reverse; it takes as input any Boolean CQ $\tilde{Q}^{(i)}$ from the disjunction $\tilde{Q}$ constructed by the forward reduction of Q, and an arbitrary database $\tilde{B}$ that matches the structure of $\tilde{Q}^{(i)}$. It shows that there exists a bijection from the tuples in $\tilde{B}$ to the tuples of a database B, where B matches the structure of Q, such that $\tilde{Q}^{(i)}(\tilde{B})$ if and only if $Q(D)$. The backward reduction shows that the lower bound for $Q^{(i)}$ can be used as a lower bound for Q.

Taking into account both reductions (Figure 1.1) one can conclude that any Boolean IJQ is as difficult as the most difficult Boolean CQ in the disjunction constructed by the forward reduction.

The results presented in this chapter are based on the results reported in [36] and its full version [35]. This chapter extends the work mentioned above by considering queries with intersection joins that include also variables that range over scalars.

Organisation. Section 5.1 introduces an equivalent rewriting for the intersection join predicate from Section 2.2. This equivalence is required to establish the correctness of the forward reduction. The forward reduction is established in Section 5.2. In particular, Section 5.2.1 introduces the one-step forward reduction, which is a building-block of the full forward reduction, introduced immediately after, in Section 5.2.2. The backward reduction is established in Section 5.3. Section 5.4 examines the extension of the forward reduction to non-boolean IJQs. Section 5.5 illustrates the performance of the forward reduction in practice. All omitted proofs are included in Section 5.7.

Notation. Given a multi-set S, a permutation of S is an ordered sequence that consists of the elements in S. The set of all the permutations of S is denoted by $\mathsf{perms}(S)$. Considering the bit-strings $b_1, \ldots, b_k$, their concatenation is denoted by $b_1 \circ \cdots \circ b_k$. Given two bit-strings, or in other words sequences of bits, a and b, the expression $a \preceq b$ means that a is a prefix of b. Consider a multi-set of intervals S. The subset of S which consists of the intervals in S with coincident left and right endpoints, i.e., scalars, is denoted by $\dot{S}$. Respectively, the subset of S which consists of the intervals in S with non-coincident left and right endpoints is denoted by $\bar{S}$.

Consider an IJQ and its hypergraph $\mathcal{H}(Q) = (\mathcal{V}(Q), \mathcal{E}(QS))$. Given a vertex $u \in \mathcal{V}(Q)$, define the subset $\mathcal{E}_u(Q) = \{e \mid e \in \mathcal{E}(Q), u \in e\} \subseteq \mathcal{E}(Q)$. In addition, define two sets $\dot{\mathcal{E}}_u(Q) \subseteq \mathcal{E}_u(Q)$ and $\bar{\mathcal{E}}_u(Q) \subseteq \mathcal{E}_u(Q)$, where $\dot{\mathcal{E}}_u(Q)$ is the set of all the hyperedges which correspond to relation schemas in which the variable u is a scalar variable, and $\bar{\mathcal{E}}_u(Q)$ is the set of all the hyperedges which correspond to relation schemas in which the variable u is an interval variable. Recall that there is a one to one correspondence between the hyperedges in $\mathcal{E}(Q)$ and the relation schemas in $\mathsf{schema}(Q)$. The subsets of hyperedges defined above are used in the analysis that follows to keep the extra information about the types of the variables. For a variable $u \in \mathcal{V}(Q)$, let $n_u = |\mathcal{E}_u(Q)|$ be the number of relation schemas that contain u, and $\bar{n}_u = |\bar{\mathcal{E}}_u(Q)|$ be the number of relation schemas in which u has interval type. Let $\mu : \mathcal{E}(Q) \to [1, n]$ be a mapping that maps each hyperedge $e \in \mathcal{E}(Q)$ to an integer in $i \in [1, n]$ that corresponds to the subscript of the relation that corresponds to that hyperedge e.

It is said that a database matches the structure of the query when the schema of the database is superset or has the same schema with the query. Essentially this means that the query is valid and can be executed on this particular database.

5.1 Intersection Predicate Rewriting

Consider a multi-set of intervals $S = \{x_1, \dots, x_k\} \subseteq \mathcal{I}$. The intersection predicate over S is an assertion of the following form

$$\left(\bigcap_{1 \le i \le k} x_i \right) \neq \emptyset. \tag{5.1}$$

The above predicate is equivalent to the predicate in Equation (2.2), Section 2.2, which constitutes the core of IJQ evaluation. The goal is to further rewrite it into an equivalent predicate, which is expressed as a disjunction of equality conditions. This rewriting will aid the forward reduction in two ways: it will aid in the understanding of the forward reduction, and it will be used as an argument in the proof of the forward reduction's correctness.

A first step towards obtaining a rewriting with equalities is to consider two cases, depending on whether the set $\dot{S}$ is empty or not. If $\dot{S}$ is not empty, then one can obtain a refined condition that checks whether the elements in $\dot{S}$ are equal, and then check if they are contained in the intervals in $\bar{S}$. On the other hand, if the set $\dot{S}$ is empty, then the intersection predicate remains unchanged.

Lemma 5.1.1 (Intersection Predicate)**.** *Consider the multi-set of intervals $S = \{x_1, \ldots, x_k\}$. Let $\dot{S} = \{x_{c+1}, \ldots, x_k\}$, where $0 \leq c \leq k$, and respectively $\bar{S} = \{x_1, \ldots, x_c\}$*[1]*. The predicate of Definition 4.2.1 can be equivalently rewritten as follows.*

- *Case $\dot{S} \neq \emptyset$:*
$$(x_{c+1} = \cdots = x_k) \wedge \left[x_{c+1} \in \left(\bigcap_{1 \leq i \leq c} x_i \right) \right] \tag{5.2}$$

- *Case $\dot{S} = \emptyset$:*
$$\left(\bigcap_{1 \leq i \leq k} x_i \right) \neq \emptyset. \tag{5.3}$$

Consider a segment tree $\mathfrak{T}_{\mathcal{I}}$ on $\mathcal{I} \supseteq S$ (Definition 3.3.1). Recall the following two properties: (1) since the elementary segments corresponding to the leaves form a partition of the interval that starts from the leftmost endpoint and ends at the rightmost endpoint in $\mathcal{I}$, for any point p that falls in that interval, there is precisely one leaf v such that $p \in \mathsf{seg}_{\mathcal{I}}(v)$; and (2) all the segments that correspond to the nodes in the path from v to the root contain p. The second property is implied by Property 3.3.3. It follows that checking the truth of the predicate in Equation (5.2), is equivalent to finding a tuple of nodes $(n_1, \ldots, n_{c+1}) \in \mathsf{anc}(\mathsf{leaf}_{\mathcal{I}}(x_{c+1}))^{c+1}$ such that $n_{c+1} = \mathsf{leaf}_{\mathcal{I}}(x_{c+1})$ and $n_i \in \mathsf{cp}_{\mathcal{I}}(x_i)$ for each $1 \leq i \leq c$ (Definition 3.3.4). Furthermore, checking the truth of the predicate in Equation (5.3), is the same as finding an interval $x_i \in S$, and a tuple of nodes $(n_1, \ldots, n_c) \in \mathsf{anc}(\mathsf{leaf}_{\mathcal{I}}(x_i))^c$ such that $n_i = \mathsf{leaf}_{\mathcal{I}}(x_i)$, and $n_j \in \mathsf{cp}_{\mathcal{I}}(x_j)$ for each $1 \leq j \leq c$, $i \neq j$. In other words, the intervals in S intersect if and only if there is an interval $x_i \in S$ such that the canonical partition of each other interval in S contains an ancestor of $\mathsf{leaf}_{\mathcal{I}}(x_i)$.

[1]If $c = 0$, then $\dot{S} = S$ and $\bar{S} = \emptyset$. If $c = k$, then $\dot{S} = \emptyset$ and $\bar{S} = S$.

Lemma 5.1.2 (Predicate Rewriting 1)**.** *The predicate in Lemma 5.1.1 can be equivalently rewritten as follows.*

- *Case $\dot{S} \neq \emptyset$:*

$$(x_{c+1} = \cdots = x_k) \wedge \left[\bigvee_{\substack{(n_1,\ldots,n_{c+1}) \in \mathsf{anc}(n_{c+1})^{c+1} \\ n_{c+1} = \mathsf{leaf}_{\mathcal{I}}(x_{c+1})}} \left(\bigwedge_{1 \leq j \leq c} n_j \in \mathsf{cp}_{\mathcal{I}}(x_j) \right) \right] \tag{5.4}$$

- *Case $\dot{S} = \emptyset$:*

$$\bigvee_{1 \leq i \leq c} \left[\bigvee_{\substack{(n_1,\ldots,n_c) \in \mathsf{anc}(n_i)^c \\ n_i = \mathsf{leaf}_{\mathcal{I}}(x_i)}} \left(\bigwedge_{\substack{1 \leq j \leq c \\ j \neq i}} n_j \in \mathsf{cp}_{\mathcal{I}}(x_j) \right) \right] \tag{5.5}$$

In the following it is shown next that the predicate introduced by Lemma 5.1.4 can be satisfied by at most one tuple of nodes.

Property 5.1.3 (Unique Tuple of nodes)**.** *Consider a set of intervals that intersect and the predicate introduced in Lemma 5.1.2. There is precisely one tuple of nodes that satisfies the predicate.*

Proof. Assume that $\dot{S} \neq \emptyset$ and there exist two distinct tuples of nodes $(n_1, \ldots, n_{c+1})$ and $(n'_1, \ldots, n'_{c+1})$ that satisfy the predicate in Lemma 5.1.2. Since x_{c+1} is fixed, the nodes from both tuples are within the path from the root to $\mathsf{leaf}_{\mathcal{I}}(x_{c+1}) = n_{c+1} = n'_{c+1}$. Therefore, there exists some $j \in [1, c]$ such that $n_j \neq n'_j$ and $n_j, n'_j \in \mathsf{cp}_{\mathcal{I}}(x_j)$. By Property 3.3.6, there cannot be distinct nodes in $\mathsf{cp}_{\mathcal{I}}(x_j)$ that belong to the same root-to-leaf path. Contradiction. Assume that $\dot{S} = \emptyset$. Since x_i is fixed, the nodes from the two tuples are within the same path from the root to $\mathsf{leaf}_{\mathcal{I}}(x_i) = n_i = n'_i$. Therefore, there exists some $1 \leq j \in c, j \neq i$ such that $n_j \neq n'_j$ and $n_j, n'_j \in \mathsf{cp}_{\mathcal{I}}(x_j)$. Similarly as above, by Property 3.3.6 there cannot be distinct nodes in $\mathsf{cp}_{\mathcal{I}}(x_j)$ that belong to the same root-to-leaf path. Contradiction. $\square$

The expression in Lemma 5.1.2 can be further rewritten by considering an ordering of the nodes that satisfy the predicate (Lemma 5.1.4). Consider the set of permutations $\mathsf{perms}(\{x_1, \dots, x_c\})$. There exists at least one permutation $\sigma \in \mathsf{perms}(\{x_1, \dots, x_c\})$ that corresponds to the ordering of the nodes $n_1, \dots, n_c$. This ordering suggests that $n_1 \in \mathsf{cp}_{\mathcal{I}}(\sigma_1), \dots, n_c \in \mathsf{cp}_{\mathcal{I}}(\sigma_c)$ where n_1 is ancestor of n_2, $\dots$, and n_{c-1} is ancestor of n_c.

Lemma 5.1.4 (Predicate Rewriting 2)**.** *The predicate in Lemma 5.1.1 can be equivalently rewritten as follows.*

- *Case $\dot{S} \neq \emptyset$:*

$$(x_{c+1} = \cdots = x_k) \wedge \left[\bigvee_{\sigma \in \mathsf{perms}(\bar{S})} \left[\bigvee_{\substack{(n_1, \dots, n_{c+1}) \in \mathsf{anc}(n_{c+1})^{c+1} \\ n_{c+1} = \mathsf{leaf}_{\mathcal{I}}(x_{c+1}) \\ \forall i \in [c] : n_i \in \mathsf{anc}(n_{i+1})}} \left(\bigwedge_{j \in [c]} n_j \in \mathsf{cp}_{\mathcal{I}}(\sigma_j) \right) \right] \right] \tag{5.6}$$

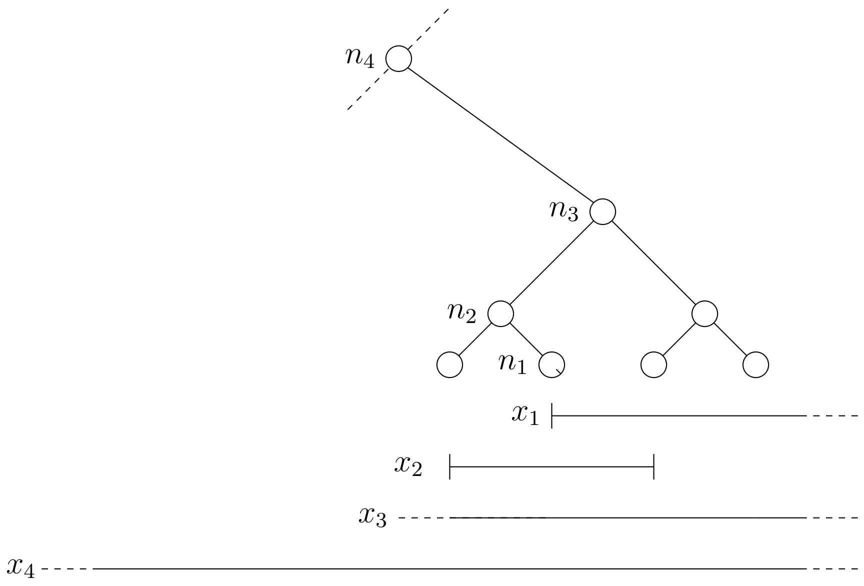
- *Case $\dot{S} = \emptyset$:*

$$\bigvee_{\sigma \in \mathsf{perms}(\bar{S})} \left[\bigvee_{\substack{(n_1, \dots, n_c) \in \mathsf{anc}(n_c)^c \\ n_c = \mathsf{leaf}_{\mathcal{I}}(\sigma_c) \\ \forall i \in [c-1] : n_i \in \mathsf{anc}(n_{i+1})}} \left(\bigwedge_{j \in [c-1]} n_j \in \mathsf{cp}_{\mathcal{I}}(\sigma_j) \right) \right] \tag{5.7}$$

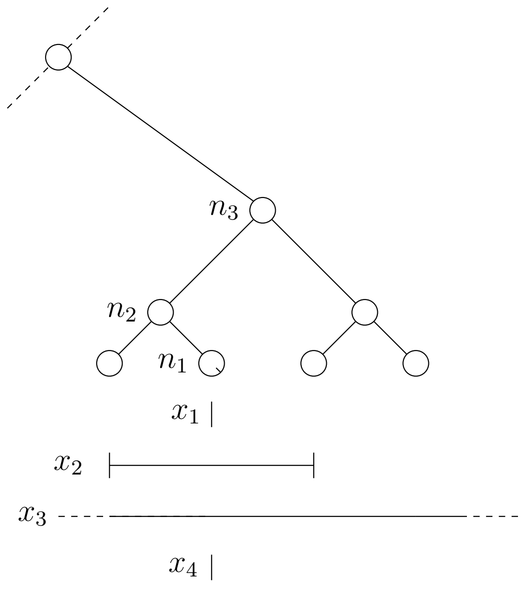
Notice that disjunction over $\sigma \in \mathsf{perms}(\{x_1, \dots, x_c\})$ in Equation (5.7), subsumes the disjunction over $1 \leq i \leq c$ in Equation (5.5).

Example 5.1.5. *The following examples explain the rewriting obtained by Lemmas 5.1.2 and 5.1.4.*

- *Consider the multi-set of intervals $S = \{x_1, x_2, x_3, x_4\} \subseteq \mathcal{I}$ illustrated in Figure 5.1a. The predicate in Equation* (5.5) *is satisfied by the nodes in* (n_1, n_2, n_3,



(a) Figure 5.1a illustrates four intersecting intervals $\{x_1, x_2, x_4, x_4\}$. The leaf $n_1 = \mathsf{leaf}_{\mathcal{I}}(x_1)$ corresponds to the left endpoint of x_1, which is the interval with the rightmost left endpoint among $\{x_1, x_2, x_4, x_4\}$. The nodes n_2, n_3 and n_4 belong to the canonical partition of the x_2, x_3 and x_4 interval respectively.



(b) Figure 5.1b illustrates four intersecting intervals $\{x_1, x_2, x_4\}$ where x_1 and x_4 have the same right and left endpoints, i.e., they are scalars. The leaf $n_1 = \mathsf{leaf}_{\mathcal{I}}(x_1) = \mathsf{leaf}_{\mathcal{I}}(x_4)$ corresponds to the left endpoint of x_1, which is the interval with the rightmost left endpoint among $\{x_1, x_2, x_4\}$. The nodes n_2, n_3 belong to the canonical partition of the x_2, and x_3 interval respectively.

Figure 5.1: This figure illustrates how to use the segment tree to check the non-empty intersection of four intervals.

n_4), where $n_1 \in \mathsf{leaf}_{\mathcal{I}}(x_1)$ and $n_i \in \mathsf{cp}_{\mathcal{I}}(x_i)$ for each $i \in \{2, 3, 4\}$. The predicate in Equation (5.7) is satisfied by the permutation (x_1, x_2, x_3, x_4) of the intervals and the corresponding tuple (n_1, n_2, n_3, n_4) of nodes, where $n_1 \in \mathsf{leaf}_{\mathcal{I}}(x_1)$, and $n_i \in cp_{\mathcal{I}}(x_i)$ for each $i \in \{2, 3, 4\}$.

- *Consider the multi-set of intervals $S = \{x_1, x_2, x_3, x_4\} \subseteq \mathcal{I}$, illustrated in Figure 5.1b. In fact, both x_1 and x_4 have the same right and left endpoints, hence, they are scalars. The predicate in Equation (5.4) is satisfied by the nodes in (n_1, n_2, n_3), where $n_1 \in \mathsf{leaf}_{\mathcal{I}}(x_1) = \mathsf{leaf}_{\mathcal{I}}(x_4)$ and $n_i \in \mathsf{cp}_{\mathcal{I}}(x_i)$ for each $i \in \{2, 3\}$. The predicate in Equation (5.7) is satisfied by the permutation (x_2, x_3) of the non-scalar intervals, and the tuple of nodes (n_1, n_2, n_3), where $n_1 \in \mathsf{leaf}_{\mathcal{I}}(x_1) = \mathsf{leaf}_{\mathcal{I}}(x_4)$, and $n_i \in \mathsf{cp}_{\mathcal{I}}(x_i)$ for each $i \in \{2, 3\}$.*

5.1.1 Disjointness

Consider a set of intersecting intervals $S = \{x_1, \ldots, x_k\} \subseteq \mathcal{I}$. Since the intervals in S intersect, they satisfy the predicate in Lemma 5.1.4. By Property 5.1.3, given a permutation $\sigma \in \mathsf{perms}(\{x_1, \ldots, x_c\})$ that satisfies the predicate in Lemma 5.1.4, there exists precisely one tuple $(n_1, \ldots, n_c)$ (respectively $(n_1, \ldots, n_{c+1})$) that satisfies the conjunction in Equation (5.7) (respectively Equation (5.6)). Suppose that there exists j such that $n_j = n_{j+1}$. That is, a different the permutation σ', obtained by swapping σ_j and σ_{j+1} in σ, together with $(n_1, \ldots, n_c)$ (respectively $(n_1, \ldots, n_{c+1})$) satisfy the predicate as well.

It is possible to restrict the permutations further such that each tuple of segment tree nodes that satisfies the predicate in Lemma 5.1.4, corresponds to precisely one permutation. This can be done by modifying the bit-strings that encode the nodes of the segment tree as follows: the root is the empty bit-string, its left child is the bit-string 0^k (i.e. 0 repeated k times), its right child has the bit-string 1^k, (i.e. 1 repeated k times), the left child of the node 0^k is the bit-string 0^k0^k, its right child is the bit-string 0^k1^k, and so on. Then, consider the following modification in the definition of the canonical partition of an interval x_i where $1 \le i \le k$ (Definition 3.3.4):

$$\mathsf{cp}_{\mathcal{I}}(x_i) = \{u' \in V(\mathfrak{T}_{\mathcal{I}}) \mid x_i \in \mathcal{I}_u\},$$

where u' is the bit-string obtained by removing the last $k-i$ bits from the bit-string u. It can be shown that the statements in Properties 3.3.3, and 3.3.6, and Lemma 5.1.4 still hold after the modification. The following property also holds.

Property 5.1.6 (Unique Permutation)**.** *After applying the modification mentioned above, there exists precisely one permutation satisfying the predicate in Lemma 5.1.4.*

Proof. Consider any permutation σ of the intervals in $\{x_1, \ldots, x_c\}$ and a tuple $(n_1, \ldots, n_c)$ (respectively $(n_1, \ldots, n_{c+1})$) that satisfy the predicate in Lemma 5.1.4. By the above modification, all the elements in the tuple are unique. Hence, one cannot obtain another permutation σ' that also satisfies the predicate in Lemma 5.1.4, by swapping n_j with n_{j+1} for some j. □

5.1.2 From Intersections to Equalities

The predicate in Lemma 5.1.4 can be further rewritten to an equivalent predicate, which is a disjunction of equalities, using the bit-string encodings of the nodes in the segment tree. They key to achieve this is Property 3.3.3, which states that $n_j \in \mathsf{anc}(n_{j+1})$ if and only if "n_j is a prefix of n_{j+1}".

Lemma 5.1.7 (Predicate with Equalities)**.** *The predicate in Lemma 5.1.1 can be equivalently rewritten as follows.*

- *Case $\dot{S} \neq \emptyset$: There exists a permutation $\sigma \in \mathsf{perms}(\bar{S})$ and a tuple of bit-strings $(b_1, \ldots, b_{c+1})$ such that:*

$$(b_1 \circ \cdots \circ b_i) \in \mathsf{cp}_{\mathcal{I}}(\sigma_i), \text{ for each } 1 \le i \le c$$

$$(b_1 \circ \cdots \circ b_{c+1}) = \mathsf{leaf}_{\mathcal{I}}(x_i), \; \textit{for each } c < i \leq k$$

- *Case $\dot{S} = \emptyset$: There exists a permutation $\sigma \in \mathsf{perms}(\bar{S})$ and a tuple of bit-strings $(b_1, \ldots, b_c)$ such that:*

$$(b_1 \circ \cdots \circ b_i) \in \mathsf{cp}_{\mathcal{I}}(\sigma_i), \; \textit{for each } 1 \leq i < c$$

$$(b_1 \circ \cdots \circ b_c) = \mathsf{leaf}_{\mathcal{I}}(\sigma_c)$$

The following property states that given a multi-set of intersecting intervals, there is precisely one permutation and one tuple of bit-strings that satisfy the predicate in Lemma 5.1.7.

Property 5.1.8 (Unique Permutation and Tuple of Bit-strings)**.** *Consider a multi-set of intervals $S = \{x_1, \ldots, x_k\} \subseteq \mathcal{I}$ that intersect. There is precisely one permutation $\sigma \in \mathsf{perms}(\bar{S})$ and one tuple of bit-strings $(b_1, \ldots, b_c)$ (respectively $(b_1, \ldots, b_{c+1})$) that satisfy the predicate in Lemma 5.1.7.*

Proof. The property follows directly from Properties 5.1.3 and 5.1.6. □

The above property is crucial because it guarantees that the forward reduction presented in the next section produces a disjunction of queries with disjoint solutions. This is a necessary condition for efficiently enumerating the tuples in the output or for correctly computing the number of tuples in the output. However, it is not a mandatory requirement for the Boolean case. Thus, it is not necessary to include the adjustment suggested in Subsection 5.1.1 in the Boolean case, as its omission does not impact the accurate computation of the output.

5.2 Forward Reduction

This section describes the forward reduction algorithm. The forward reduction algorithm takes a Boolean IJQ Q and a database D as input. It outputs a disjunction of Boolean CQs denoted by $\tilde{Q}$, and a new database $\tilde{D}$ that matches the structure of $\tilde{Q}$. This reduction guarantees that $Q(D)$ is equivalent to $\tilde{Q}(\tilde{D})$, allowing the use of existing algorithms for solving Boolean CQs to solve Q. The most difficult CQ conjunct in $\tilde{Q}$ determines the upper bound for the computation of Q.

5.2.1 One-step Forward Reduction

The forward reduction is described as an algorithm that iterates through each join variable in the query. Each iteration corresponds to a reduction, called one-step forward reduction. Consider a join variable $v \in \mathsf{vars}(Q)$. The computation of the join on v requires the computation of the predicate in Equation (5.1), over a multi-set of intervals, i.e., one input interval for each relation involved in the join. Lemma 5.1.7 introduces an equivalent rewriting of the intersection predicate into a disjunction of equalities. The one-step forward reduction is driven by the equivalence introduced in Lemma 5.1.7 to construct a new query $\tilde{Q}^{(v)}$ and a new matching database $\tilde{D}^{(v)}$, such that $\tilde{Q}^{(v)}(\tilde{D}^{(v)})$ if and only if $Q(D)$.

One-step Query Transformation

Consider the Boolean IJQ Q and its hypergraph $\mathcal{H}(Q) = (\mathcal{V}(Q), \mathcal{E}(Q))$. Given a permutation $\sigma = (\sigma_1, \dots, \sigma_{\bar{n}_v}) \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$, each relation schema $R_{\mu(\sigma_i)}(\sigma_i)$ induces a new relation schema $R_{\mu(\sigma_i)}(\tilde{\sigma}_i^{(v,\sigma)})$ in which the variable v is replaced by the new interval variable $\tilde{v}^{(\mu(\sigma_i))}$, and the new scalar variables $\tilde{v}_1, \dots, \tilde{v}_i$. The set $\tilde{\sigma}_i^{(v,\sigma)}$ is defined by

$$\tilde{\sigma}_i^{(v,\sigma)} = \{\sigma_i \setminus \{v\}\} \cup \{\tilde{v}_1, \dots, \tilde{v}_i\} \cup \{\tilde{v}^{(\mu(\sigma_i))}\}. \tag{5.8}$$

Furthermore, each relation schema $R_{\mu(e)}(e)$ where $e \in \dot{\mathcal{E}}_v(Q)$ is replaced by a new relation schema $R_{\mu(e)}(\tilde{e}^{(v,\sigma)})$, in which the variable v is replaced by the new scalar variable $\tilde{v}^{(\mu(e))}$, and the new scalar variables $\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v+1}$. The set $\tilde{e}^{(v,\sigma)}$ is defined by

$$\tilde{e}^{(v,\sigma)} = \{e \setminus \{v\}\} \cup \{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v+1}\} \cup \{\tilde{v}^{(\mu(e))}\}. \tag{5.9}$$

The variables $\tilde{v}_1, \ldots, \tilde{v}_i$ are scalar variables in all of their occurrences, whereas the variables in $\{\tilde{v}^{(\mu(e))} \mid e \in \mathcal{E}_v(Q)\}$ are singleton variables whose type is the same as the type of the variable that they replace in the corresponding relation schema. Finally, each relation schema $R_{\mu(f)}(f)$ where $f \in \mathcal{E}(Q) \setminus \mathcal{E}_v(Q)$ remains unchanged.

The input Boolean IJQ is transformed into a disjunction $\tilde{Q}^{(v)}$ of Boolean IJQs. In particular, for each permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$ the one-step forward reduction algorithm generates a new Boolean IJQ $\tilde{Q}^{(v,\sigma)}$. The new query has equality joins and possibly remaining intersection joins. The query $\tilde{Q}^{(v)}$ is defined as their disjunction. The formal definition of $\tilde{Q}^{(v,\sigma)}$ and $\tilde{Q}^{(v)}$ is as follows.

Definition 5.2.1 (One-step Query Trasformation)**.** *Consider a Boolean IJQ Q, a permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$, and a variable $v \in \mathcal{V}(Q)$. Define $\tilde{Q}^{(v,\sigma)}$ as*

$$\tilde{Q}^{(v,\sigma)} = \left(\bigwedge_{1 \le i \le \bar{n}_v} \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i^{(v,\sigma)}) \right) \wedge \left(\bigwedge_{e \in \dot{\mathcal{E}}_v(Q)} \tilde{R}_{\mu(e)}(\tilde{e}^{(v,\sigma)}) \right) \wedge \left(\bigwedge_{f \in \mathcal{E}(Q) \setminus \mathcal{E}_v(Q)} \tilde{R}_{\mu(f)}(f) \right).$$

Define $\tilde{Q}^{(v)}$ as

$$\tilde{Q}^{(v)} = \bigvee_{\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))} \tilde{Q}^{(v,\sigma)}.$$

One can make some general observations about the newly constructed IJQs. First of all, the variables of the form $\tilde{v}^{(i)}$ do not affect the Boolean IJQ evaluation because they are not join variables in $\tilde{Q}^{(v)}$. They are only applicable in the computation of non-Boolean IJQs, which are discussed in Section 5.4. In addition, the relation schemas in $\tilde{Q}^{(v,\sigma)}$ are constructed by modifying the relation schemas in Q, establishing

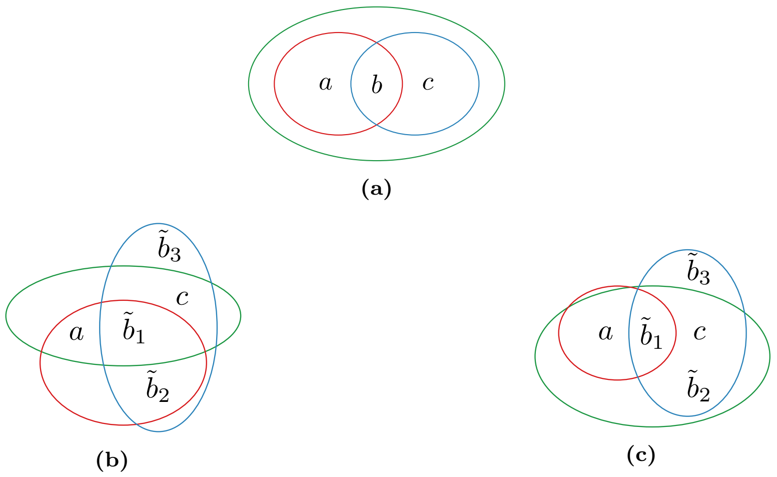


Figure 5.2: Figures 5.2b and Figure 5.2c illustrate the hypergraphs that correspond to the Boolean IJQs $\tilde{Q}^{(v,\sigma)}$ and $\tilde{Q}^{(v,\sigma')}$, respectively, from the Example 5.2.2.

a one-to-one correspondence between the relation schemas in $\tilde{Q}^{(v,\sigma)}$ and those in Q. Furthermore, the number of conjuncts in the disjunction $\tilde{Q}^{(v)}$ is the same as the number of permutations of the hyperedges in $\bar{\mathcal{E}}_v(Q)$, hence it is $\bar{n}_v!$.

The following example explains the one-step query transformation for the Boolean IJQ from Example 2.3.2.

Example 5.2.2. *Consider the Boolean IJQ from Example 2.3.2. Consider the one-step query transformation for the intersection join variable b (Definition 5.2.1). Consider the permutations $\sigma = (\{a, b, c\}, \{a, b\})$ and $\sigma' = (\{a, b\}, \{a, b, c\})$. The query $\tilde{Q}^{(b)}$ is defined as the disjunction of the following Boolean IJQs.*

$$\tilde{Q}^{(b,\sigma)} = \tilde{R}_1(a, \tilde{b}_1, [\tilde{b}^{(1)}], [c]) \wedge \tilde{R}_2(a, \tilde{b}_1, \tilde{b}_2, [\tilde{b}^{(2)}]) \wedge \tilde{R}_3(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \tilde{b}^{(3)}, [c])$$

$$\tilde{Q}^{(b,\sigma')} = \tilde{R}_1(a, \tilde{b}_1, \tilde{b}_2, [\tilde{b}^{(1)}], [c]) \wedge \tilde{R}_2(a, \tilde{b}_1, [\tilde{b}^{(2)}]) \wedge \tilde{R}_3(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \tilde{b}^{(3)}, [c])$$

The variables $\tilde{b}_1, \tilde{b}_2, \tilde{b}_3$ are scalar variables in all of their occurrences, and the type of the variables $\tilde{b}^{(1)}$, $\tilde{b}^{(2)}$ and $\tilde{b}^{(3)}$ is the same as the type of the original variable b in the corresponding relation. Because the queries are Boolean, the variables $\tilde{b}^{(1)}$, $\tilde{b}^{(2)}$ and $\tilde{b}^{(3)}$, which occur in only one relation, do not affect the evaluation.

One-step Hypergraph Transformation

The subsequent analysis presents a definition of the hypergraphs that correspond to the Boolean IJQs in the disjunction resulting from the one-step query transformation. Although this definition is not necessary, because the hypergraphs can be computed directly from the Boolean IJQs, it can provide insights into the structure of the reduction. Additionally, Chapter 6 highly depends on the arising hypergraphs to study the properties of the corresponding queries.

The one-step hypergraph transformation with respect to the join variable v is defined in the following.

Definition 5.2.3 (One-step Hypergraph Transformation)**.** *Consider the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, a vertex $v \in \mathcal{V}$, and a permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v)$. The hypergraph $\tilde{\mathcal{H}}^{(v,\sigma)} = (\tilde{\mathcal{V}}^{(v)}, \tilde{\mathcal{E}}^{(v,\sigma)})$ is defined as follows.*

- *If $\dot{\mathcal{E}}_v \neq \emptyset$*

$$\tilde{\mathcal{V}}^{(v)} = (\mathcal{V} \setminus \{v\}) \cup \{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v+1}\} \cup \{\tilde{v}^{(\mu(e))} \mid e \in \mathcal{E}_v\}$$

$$\tilde{\mathcal{E}}^{(v,\sigma)} = (\mathcal{E} \setminus \mathcal{E}_v) \cup \{\tilde{\sigma}_i^{(v,\sigma)} \mid 1 \leq i \leq \bar{n}_v\} \cup \{\tilde{e}^{(v,\sigma)} \mid e \in \dot{\mathcal{E}}_v\}$$

- *If $\dot{\mathcal{E}}_v = \emptyset$*

$$\tilde{\mathcal{V}}^{(v)} = (\mathcal{V} \setminus \{v\}) \cup \{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v}\} \cup \{\tilde{v}^{(\mu(e))} \mid e \in \mathcal{E}_v\}$$

$$\tilde{\mathcal{E}}^{(v,\sigma)} = (\mathcal{E} \setminus \mathcal{E}_v) \cup \{\tilde{\sigma}_i^{(v,\sigma)} \mid 1 \leq i \leq n_v\}$$

where $\tilde{\sigma}_i^{(v,\sigma)}$ and $\tilde{e}^{(v,\sigma)}$ are defined in Equations (5.8) *and* (5.9) *respectively. Define the set of hypergraphs $\tilde{\mathcal{H}}^{(v)} = \{\tilde{\mathcal{H}}^{(v,\sigma)} \mid \sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v)\}$.*

The following example explains the one-step hypergraph transformation using the Boolean IJQ from Example 5.2.4.

Example 5.2.4. *Consider the Boolean IJQ from Example 2.3.2, which is associated with the hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ *where* $\mathcal{V} = \{a, b, c\}$, $\mathcal{E} = \{\{a, b, c\}, \{a, b\}, \{b, c\}\}$, $\dot{\mathcal{E}}_a = \{\{a, b, c\}, \{a, b\}\}$, $\dot{\mathcal{E}}_b = \{\{b, c\}\}$, $\bar{\mathcal{E}}_b = \{\{a, b, c\}, \{a, b\}\}$ *and* $\bar{\mathcal{E}}_c = \{\{a, b, c\}, \{b, c\}\}$. *Consider the one-step hypergraph transformation with respect to the join variable b (Definition 5.2.3). Let* $\mathsf{perms}(\bar{\mathcal{E}}_b) = \{\sigma = (\{a, b, c\}, \{a, b\}), \sigma' = (\{a, b\}, \{a, b, c\})\}$. *Let* $e = \{b, c\}$. *The set* $\tilde{\mathcal{H}}^{(b)}$ *consists of the hypergraphs* $\tilde{\mathcal{H}}^{(b,\sigma)}$, *and* $\tilde{\mathcal{H}}^{(b,\sigma')}$, *which are defined as follows:*

- $\tilde{\mathcal{H}}^{(b,\sigma)} = (\tilde{\mathcal{V}}^{(b)}, \tilde{\mathcal{E}}^{(b,\sigma)})$ *where* $\tilde{\mathcal{V}}^{(b)} = \{a, \tilde{b}_1, \tilde{b}_2, \tilde{b}_2, \tilde{b}^{(1)}, \tilde{b}^{(2)}, \tilde{b}^{(3)}, c\}$ *and* $\tilde{\mathcal{E}}^{(b,\sigma)} = \{\{a, \tilde{b}_1, \tilde{b}^{(1)} c\}, \{a, \tilde{b}_1, \tilde{b}_2, \tilde{b}^{(2)}\}, \{\tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \tilde{b}^{(3)}, c\}\}$; *and*
- $\tilde{\mathcal{H}}^{(b,\sigma')} = (\tilde{\mathcal{V}}^{(b)}, \tilde{\mathcal{E}}^{(b,\sigma')})$ *where* $\tilde{\mathcal{V}}^{(b)} = \{a, \tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \tilde{b}^{(1)}, \tilde{b}^{(2)}, \tilde{b}^{(3)}, c\}$ *and* $\tilde{\mathcal{E}}^{(b,\sigma')} = \{\{a, \tilde{b}_1, \tilde{b}_2, \tilde{b}^{(1)}, c\}, \{a, \tilde{b}_1, \tilde{b}^{(2)}\}, \{\tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \tilde{b}^{(3)}, c\}\}$.

The hypergraphs defined above are illustrated in Figures 5.2b and 5.2c alongside the input hypergraph in Figure 5.2a. In the figures, the vertices that correspond to singleton variables are omitted for simplicity.

One-step Database Transformation

The one-step forward reduction requires the transformation of both the query and the database. In the following, the focus shifts on the transformation of the database. The one-step forward reduction transforms the database D to a new database $\tilde{D}^{(v)}$, which can be expressed as a union of databases

$$\bigcup_{\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))} \tilde{D}^{(v,\sigma)},$$

in a way such that the schema of each new database $\tilde{D}^{(v,\sigma)}$ matches the schema of the query $\tilde{Q}^{(v,\sigma)}$. Consider a segment tree over the set of v-values in the active domain of the variable v. Let

$$\mathcal{I} = \{t(v) \mid t \in R_{\mu(e)}(e), e \in \mathcal{E}_v(Q)\},$$

and $\mathfrak{T}_{\mathcal{I}}$ be a segment tree on $\mathcal{I}$. The definition of the one-step database transformation is the following.

Definition 5.2.5 (One-step Database Transformation)**.** *Consider an IJQ Q and a database D. Consider also a vertex $v \in \mathcal{V}(Q)$, and a permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$. The database $\tilde{D}^{(v,\sigma)}$ is constructed from the database D as follows:*

- *Case $\dot{\mathcal{E}}_v(Q) \neq \emptyset$*
  - *$\forall i \in [1, \bar{n}_v]$ $\forall t \in R_{\mu(\sigma_i)}(\sigma_i)$ construct the tuples $\tilde{t} \in \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$, where $\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i) \in \tilde{D}^{(v,\sigma)}$, such that:*
    - *$t[\sigma_i \setminus \{v\}] = \tilde{t}[\tilde{\sigma}_i \setminus \{\tilde{v}_1, \ldots, \tilde{v}_i\}]$;*
    - *$t(\tilde{v}^{(\mu(\sigma_i))}) = \tilde{t}(v)$; and*
    - *$\tilde{t}(\tilde{v}_1) \circ \cdots \circ \tilde{t}(\tilde{v}_i) \in \mathsf{cp}_{\mathcal{I}}(t(v))$.*
  - *$\forall e \in \dot{\mathcal{E}}_v(Q)$ $\forall t \in R_{\mu(e)}(e)$ construct the tuples $\tilde{t} \in \tilde{R}_{\mu(e)}(\tilde{e})$, where $\tilde{R}_{\mu(e)}(\tilde{e}) \in \tilde{D}^{(v,\sigma)}$, such that:*
    - *$t[e \setminus \{v\}] = \tilde{t}[\tilde{e} \setminus \{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v+1}\}]$;*
    - *$t(\tilde{v}^{(\mu(e))}) = t(v)$; and*
    - *$\tilde{t}(\tilde{v}_1) \circ \cdots \circ \tilde{t}(\tilde{v}_{\bar{n}_v+1}) = \mathsf{leaf}_{\mathcal{I}}(t(v))$.*
- *Case $\dot{\mathcal{E}}_v(Q) = \emptyset$*
  - *$\forall i \in [1, \bar{n}_v - 1]$ $\forall t \in R_{\mu(\sigma_i)}(\sigma_i)$, construct the tuples $\tilde{t} \in \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$, where $\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i) \in \tilde{D}^{(v,\sigma)}$, such that:*
    - *$t[\sigma_i \setminus \{v\}] = \tilde{t}[\tilde{\sigma}_i \setminus \{\tilde{v}_1, \ldots, \tilde{v}_i\}]$;*
    - *$t(\tilde{v}^{(\mu(\sigma_i))}) = \tilde{t}(v)$; and*
    - *$\tilde{t}(\tilde{v}_1) \circ \cdots \circ \tilde{t}(\tilde{v}_i) \in \mathsf{cp}_{\mathcal{I}}(t(v))$.*
  - *$\forall t \in R_{\mu(\sigma_{\bar{n}_v})}(\sigma_{\bar{n}_v})$, construct the tuples $\tilde{t} \in \tilde{R}_{\mu(\sigma_{\bar{n}_v})}(\tilde{\sigma}_{\bar{n}_v})$, where $\tilde{R}_{\mu(\sigma_{\bar{n}_v})}(\tilde{\sigma}_{\bar{n}_v}) \in \tilde{D}^{(v,\sigma)}$, such that:*

* $t[\sigma_{\bar{n}_v} \setminus \{v\}] = \tilde{t}[\tilde{\sigma}_{\bar{n}_v} \setminus \{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v}\}]$;
* $t(\tilde{v}^{(\mu(\sigma_{\bar{n}_v}))}) = \tilde{t}(v)$; *and*
* $\tilde{t}(\tilde{v}_1) \circ \cdots \circ \tilde{t}(\tilde{v}_{\bar{n}_v}) = \mathsf{leaf}_{\mathcal{I}}(t(v))$.

Each relation whose schema does not contain v is copied from D to $\tilde{D}^{(v,\sigma)}$. The database $\tilde{D}^{(v)}$ is defined by

$$\tilde{D}^{(v)} = \bigcup_{\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))} \tilde{D}^{(v,\sigma)}. \tag{5.10}$$

In Definition 5.2.5 the same relation name can be used for a relation in $\tilde{D}^{(v,\sigma)}$ and $\tilde{D}^{(v,\sigma')}$ for $\sigma, \sigma' \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$, $\sigma \neq \sigma'$. Hence, $\tilde{D}^{(v)}$ may contain two relation instances with the same name, i.e. $\tilde{R}_{\mu(\sigma_i)}$ and $\tilde{R}_{\mu(\sigma'_j)}$ for some $i \neq j$. However, by construction, those relations are over different schemas (Definition 5.2.1). Recall that a relation is uniquely identified by its schema, which consists of a relation name and a set of variables (Section 2.2). As a result, it is guaranteed that the relation instances in $\tilde{D}^{(v)}$ are distinct. This above can be observed in Example 5.2.2.

Example 5.2.6 gives the details of the one-step database transformation that corresponds to the one-step query transformation from Example 5.2.2.

Example 5.2.6. *Consider the Boolean IJQ from Example 5.2.2. Consider the one-step forward reduction on the intersection join variable b. The following is an explanation of how the one-step database transformation is derived.*

Construct a segment tree on the b-values coming from R_1, R_2 and R_3, following Definition 3.3.1. A segment tree on $\mathcal{O}(n)$ values can be constructed in $\mathcal{O}(n \cdot \log n)$ time and has a height $\mathcal{O}(\log n)$. The nodes of the segment tree represent intervals called segments. Recall that the segment associated with a node, includes the segments associated with its descendants and is partitioned by the segments associated with its children. A property of a segment tree is that each input interval can be expressed as the disjoint union of at most $\mathcal{O}(\log n)$ segments. In particular, by Definition 3.3.4, the *canonical partition of an interval consists of a set of segments, corresponding to nodes of the segment tree, whose disjoint union covers the interval. Consider three tuples $r \in R_1$, $s \in R_2$ and $t \in R_3$. The problem of checking whether a point $t(b)$ and two intervals $r(b), s(b)$ intersect becomes the problem of checking if the point is contained in both intervals. This relationship can be captured, by three nodes n_r, n_s and n_t on the segment tree, and either of the following: (1) n_t is the leaf that corresponds to the point, n_s is ancestor of n_t and belongs to the canonical partition of $s(b)$ and n_r is ancestor of n_s and belongs to the canonical partition of $r(b)$; (2) n_t is the leaf that corresponds to the point, n_r is ancestor of n_t and belongs to the canonical partition of the interval $r(b)$ and n_s is ancestor of n_r and belongs to the canonical partition of the interval $s(b)$.*

The two conditions described above, can be expressed using equality joins. By Property 3.3.3, given three nodes n_i and n_j, where n_i is an ancestor of n_j, the bit-string for n_i, is a prefix of that for n_j. Based on that, one can use three variables b_1, b_2, b_3 and express the two conditions as follows: (1) b_1 represents the bit-string of n_r and is also a prefix of the bit-string of n_s and n_t; $b_1 \circ b_2$ represents the bit-string of n_s and is also a prefix of n_t, and $b_1 \circ b_2 \circ b_3$ represents the bit-string of n_t; and (2) b_1 represents the bit-string of n_s and is also a prefix of the bit-string of n_r and n_t; $b_1 \circ b_2$ represents the bit-string of n_r and is also a prefix of n_t, and $b_1 \circ b_2 \circ b_3$ represents the bit-string of n_t. Each such case can be expressed using the Boolean IJQs $\tilde{Q}^{(b,\sigma)}$ and $\tilde{Q}^{(b,\sigma')}$ from Example 5.2.2. Each one of the queries is evaluated on a database $\tilde{D}^{(v,\sigma)}$ and $\tilde{D}^{(v,\sigma')}$ respectively, where:

$$\begin{aligned}
\tilde{D}^{(b,\sigma')} &= \{\tilde{R}_1(a, \tilde{b}_1, \tilde{b}^{(1)}, c), \tilde{R}_2(a, \tilde{b}_1, \tilde{b}_2, \tilde{b}^{(2)}, c), \tilde{R}_3(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \tilde{b}^{(1)}, c)\}; \\
\tilde{D}^{(b,\sigma)} &= \{\tilde{R}_1(a, \tilde{b}_1, \tilde{b}_2, \tilde{b}^{(1)}, c), \tilde{R}_2(a, \tilde{b}_1, \tilde{b}^{(2)}), \tilde{R}_3(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \tilde{b}^{(3)}, c)\}.
\end{aligned}$$

It holds that $Q(D)$ is satisfied if and only if at least one of the following is satisfied

$\tilde{Q}^{(b,\sigma)}(\tilde{D}^{(b,\sigma)})$ or $\tilde{Q}^{(b,\sigma')}(\tilde{D}^{(b,\sigma')})$. The correctness of the latter statement is formally stated in the following.

Correctness

The transformations in Definitions 5.2.1 and 5.2.5 preserve the equivalence to the original evaluation problem.

Lemma 5.2.7 (One-step Forward Reduction Correctness)**.** *Consider a Boolean IJQ Q and a database D. Consider also a variable $v \in \mathcal{V}(Q)$. It holds that $Q(D)$ is equivalent to $\tilde{Q}^{(v)}(\tilde{D}^{(v)})$, where $\tilde{Q}^{(v)}$ and $\tilde{D}^{(v)}$ are constructed by Definitions 5.2.1*

Complexity

The following lemma states the size and construction time of each relation in $\tilde{D}^{(v)}$, as well as their number. Consequently, by employing it one can determine the size and construction time of the new database $\tilde{D}^{(v)}$.

Lemma 5.2.8 (One-step Forward Reduction Complexity)**.** *The size of the query $\tilde{Q}^{(v)}$ which is constructed by Definition 5.2.1 is $\mathcal{O}(1)$. The size of each new relation in the database $\tilde{D}^{(v)}$ which is constructed by Definition 5.2.5 is the following.*

- *Case $\dot{\mathcal{E}}_v(Q) \neq \emptyset$*
  - $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^i |D|) \qquad \forall i \in [1, \bar{n}_v]$;
  - $|\tilde{R}_{\mu(e)}(\tilde{e})| = \mathcal{O}(|R_{\mu(e)}(e)| \cdot \log^{i-1} |D|) \qquad \forall e \in \mathcal{E}_v(Q)$.
- *Case $\dot{\mathcal{E}}_v(Q) = \emptyset$*
  - $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^i |D|) \qquad \forall i \in [1, \bar{n}_v - 1]$;
  - $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^{i-1} |D|) \qquad i = \bar{n}_v$.

The construction time of each new relation is proportional to its size.

5.2.2 Full Forward Reduction

The forward reduction transforms any Boolean IJQ into a disjunction of Boolean CQs. Hence, all the intersection joins are replaced by equality joins. Furthermore, the database is transformed accordingly. Those transformations are achieved by applying the one-step forward reduction, described in Section 5.2.1, on a join variable and then apply it iteratively to the result of the previous step for the remaining join variables. The algorithm presented below describes the full forward reduction procedure, which involves transforming both the query and the database.

Algorithm 4 Full Forward Reduction

```
1:  procedure FORWARD-REDUCTION($Q$ : query, $D$ : database)
2:      $\mathbf{Q} = \{Q\}$;
3:      for each join variable $v$ in $\mathcal{V}$ do
4:          $\mathbf{Q}_0 = \mathbf{Q}; \mathbf{Q} = \emptyset$;
5:          for each $Q \in \mathbf{Q}_0$; do
6:              for each $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$ do
7:                  $\mathbf{Q} = \mathbf{Q} \cup \{\tilde{Q}^{(v,\sigma)}\}$;          ▷ Definition 5.2.1
8:              end for
9:          end for
10:         $\tilde{D} = \tilde{D}^{(v)}; D = \tilde{D}$;          ▷ Definition 5.2.5
11:     end for
12:     return $(\mathbf{Q}, \tilde{D})$
13: end procedure
```

Definition of the Forward Reduction

The output of Algorithm 4 is a tuple consisting of a set of Boolean CQs $\mathbf{Q}$, and a database $\tilde{D}$. The full forward reduction of the problem of evaluating a Boolean IJQ Q on D is formally defined as follows.

Definition 5.2.9 (Full Forward Reduction)**.** *Consider a Boolean IJQ Q and a database D. Define the query*

$$\tilde{Q} = \bigvee_{Q \in \mathbf{Q}} Q, \tag{5.11}$$

where $\boldsymbol{Q}$ is the set of Boolean CQs returned by Algorithm 4. Define $\tilde{D}$ be the database returned by Algorithm 4.

Example 5.2.10 explains the full forward reduction for the Boolean Triangle IJQ (Example 2.3.1).

Example 5.2.10 (The Boolean Triangle IJQ — Forward Reduction)**.** *Consider the Boolean Triangle IJQ*

$$Q_\triangle = R_1([a],[b]) \wedge R_2([b],[c]) \wedge R_3([a],[c]),$$

where $\bar{\mathcal{E}}_a(Q_\triangle) = \{\{a,b\},\ \{a,c\}\}$, $\bar{\mathcal{E}}_b(Q_\triangle) = \{\{a,b\},\{b,c\}\}$ and $\bar{\mathcal{E}}_c(Q_\triangle) = \{\{b,c\}, \{a,c\}\}$. The query has three intersection joins on a, b and c. The forward reduction transforms the Boolean IJQ $Q_\triangle$ to the disjunction of the following Boolean CQs.

$$\begin{aligned}
\tilde{Q}_1 &= \tilde{R}_1(\tilde{a}_1,\tilde{a}_2,\tilde{a}^{(1)},\tilde{b}_1,\tilde{b}_2,\tilde{b}^{(1)}) \wedge \tilde{R}_2(\tilde{b}_1,\tilde{b}^{(2)},\tilde{c}_1,\tilde{c}_2,\tilde{c}^{(2)}) \wedge \tilde{R}_3(\tilde{a}_1,\tilde{a}^{(3)},\tilde{c}_1,\tilde{c}^{(3)})\\
\tilde{Q}_2 &= \tilde{R}_1(\tilde{a}_1,\tilde{a}_2,\tilde{a}^{(1)},\tilde{b}_1,\tilde{b}_2,\tilde{b}^{(1)}) \wedge \tilde{R}_2(\tilde{b}_1,\tilde{b}^{(2)}\tilde{c}_1,\tilde{c}^{(2)}) \wedge \tilde{R}_3(\tilde{a}_1,\tilde{a}^{(3)},\tilde{c}_1,\tilde{c}_2,\tilde{c}^{(3)})\\
\tilde{Q}_3 &= \tilde{R}_1(\tilde{a}_1,\tilde{a}_2,\tilde{a}^{(1)},\tilde{b}_1,\tilde{b}^{(1)}) \wedge \tilde{R}_2(\tilde{b}_1,\tilde{b}_2,\tilde{b}^{(2)},\tilde{c}_1,\tilde{c}_2,\tilde{c}^{(2)}) \wedge \tilde{R}_3(\tilde{a}_1,\tilde{a}^{(3)}\tilde{c}_1,\tilde{c}^{(3)})\\
\tilde{Q}_4 &= \tilde{R}_1(\tilde{a}_1,\tilde{a}_2,\tilde{a}^{(1)},\tilde{b}_1,\tilde{b}^{(1)}) \wedge \tilde{R}_2(\tilde{b}_1,\tilde{b}_2,\tilde{b}^{(2)},\tilde{c}_1,\tilde{c}^{(2)}) \wedge \tilde{R}_3(\tilde{a}_1,\tilde{a}^{(3)},\tilde{c}_1,\tilde{c}_2,\tilde{c}^{(3)})\\
\tilde{Q}_5 &= \tilde{R}_1(\tilde{a}_1,\tilde{a}^{(1)},\tilde{b}_1,\tilde{b}_2,\tilde{b}^{(2)}) \wedge \tilde{R}_2(\tilde{b}_1,\tilde{b}^{(2)},\tilde{c}_1,\tilde{c}_2,\tilde{c}^{(2)}) \wedge \tilde{R}_3(\tilde{a}_1,\tilde{a}_2,\tilde{a}^{(3)},\tilde{c}_1,\tilde{c}^{(3)})\\
\tilde{Q}_6 &= \tilde{R}_1(\tilde{a}_1,\tilde{a}^{(1)},\tilde{b}_1,\tilde{b}_2,\tilde{b}^{(1)}) \wedge \tilde{R}_2(\tilde{b}_1,\tilde{b}^{(2)},\tilde{c}_1,\tilde{c}^{(2)}) \wedge \tilde{R}_3(\tilde{a}_1,\tilde{a}_2,\tilde{a}^{(3)},\tilde{c}_1,\tilde{c}_2,\tilde{c}^{(3)})\\
\tilde{Q}_7 &= \tilde{R}_1(\tilde{a}_1,\tilde{a}^{(1)},\tilde{b}_1,\tilde{b}^{(1)}) \wedge \tilde{R}_2(\tilde{b}_1,\tilde{b}_2,\tilde{b}^{(2)},\tilde{c}_1,\tilde{c}_2,\tilde{c}^{(2)}) \wedge \tilde{R}_3(\tilde{a}_1,\tilde{a}_2,\tilde{a}^{(3)},\tilde{c}_1,\tilde{c}^{(3)})\\
\tilde{Q}_8 &= \tilde{R}_1(\tilde{a}_1,\tilde{a}^{(1)},\tilde{b}_1,\tilde{b}^{(1)}) \wedge \tilde{R}_2(\tilde{b}_1,\tilde{b}_2,\tilde{b}^{(2)},\tilde{c}_1,\tilde{c}^{(2)}) \wedge \tilde{R}_3(\tilde{a}_1,\tilde{a}_2,\tilde{a}^{(3)},\tilde{c}_1,\tilde{c}_2,\tilde{c}^{(3)})
\end{aligned}$$

Complexity

The forward reduction's complexity is measured by the time and space required to transform both the query and the database. Recall that this transformation is accomplished by Algorithm 4. The complexity of the forward reduction is formally stated in the following.

Proposition 5.2.11 (Forward Reduction Complexity)**.** *Consider a Boolean IJQ Q and a database D. The size of $\tilde{Q}$ is $\mathcal{O}(1)$. Furthermore, the new database $\tilde{D}$ has size $\tilde{\mathcal{O}}(|D|)$, and can be constructed in time proportional to its size.*

Proof. The number of the CQ conjuncts in $\tilde{Q}$ is $\prod_{v \in \mathcal{V}(Q)} \bar{n}_v!$, hence, it only depends on the size of the input query. Moreover, the size of each one of the CQ conjuncts depends only on the size of the input query. Therefore, we have that $|\tilde{Q}| = \mathcal{O}(1)$.

Let us analyse the construction time of the relation $\tilde{R}_{\mu(e)}$, which is based on the relation $R_{\mu(e)}$. According to Lemma 5.2.8, by replacing a variable $v \in e$, one ends up with a relation whose size and construction time is $\mathcal{O}(|R_e| \cdot \log^{\bar{n}_v}|D|)$. Hence, the replacement of the rest of the variables, namely the variables in $e \setminus \{v\}$, will result in a size and computation time of

$$\mathcal{O}(|R_e| \cdot \log^{\bar{n}_v}|D| \cdot \prod_{u \in e \setminus \{v\}} \log^{\bar{n}_u}|D|) = \tilde{\mathcal{O}}(|D|).$$

□

The number of the CQ conjuncts in the disjunction $\tilde{Q}$ is at most $|\mathcal{E}(Q)|!^{|\mathcal{V}(Q)|}$. The worst case occurs when all the hyperedges include all the variables, and all the variables are of interval type wherever they occur.

Correctness

The correctness of the forward reduction is formally stated in the following.

Proposition 5.2.12 (Correctness)**.** *Consider a Boolean IJQ Q and a database D. It holds that $Q(D)$ is equivalent to $\tilde{Q}(\tilde{D})$, where $\tilde{Q}$ and $\tilde{D}$ are defined by Definition 5.2.9.*

5.2.3 Upper Bounds

The upper bounds for the computation of Boolean IJQs are established using known upper bounds for the computation of Boolean CQs. In particular, the upper bound for the computation of an IJQ Q, is the upper bound for the computation of the most difficult CQ in the forward reduction of Q. Existing upper bounds for Boolean CQs [4, 8] use the sub-modular width (Definition 3.1.18) or the fractional hypertree width (Definition 3.1.17) of the query as an exponent. In the following, those two notions of width are extended to be used as width measures for Boolean IJQs.

Definition 5.2.13 (Extended Width Measures)**.** *Consider a Boolean IJQ Q. Let $\mathbf{Q}$ be the set that consists of the Boolean CQs returned by Algorithm 4. Define*

$$\mathsf{subw}^{ij}(Q) = \max_{q \in \mathbf{Q}} \mathsf{subw}(q),$$

$$\mathsf{fhtw}^{ij}(Q) = \max_{q \in \mathbf{Q}} \mathsf{fhtw}(q),$$

where $\mathsf{subw}(q)$ is the sub-modular width of q (Definition 3.1.18), and $\mathsf{fhtw}(q)$ is the fractional hypertree width of q (Definition 3.1.17).

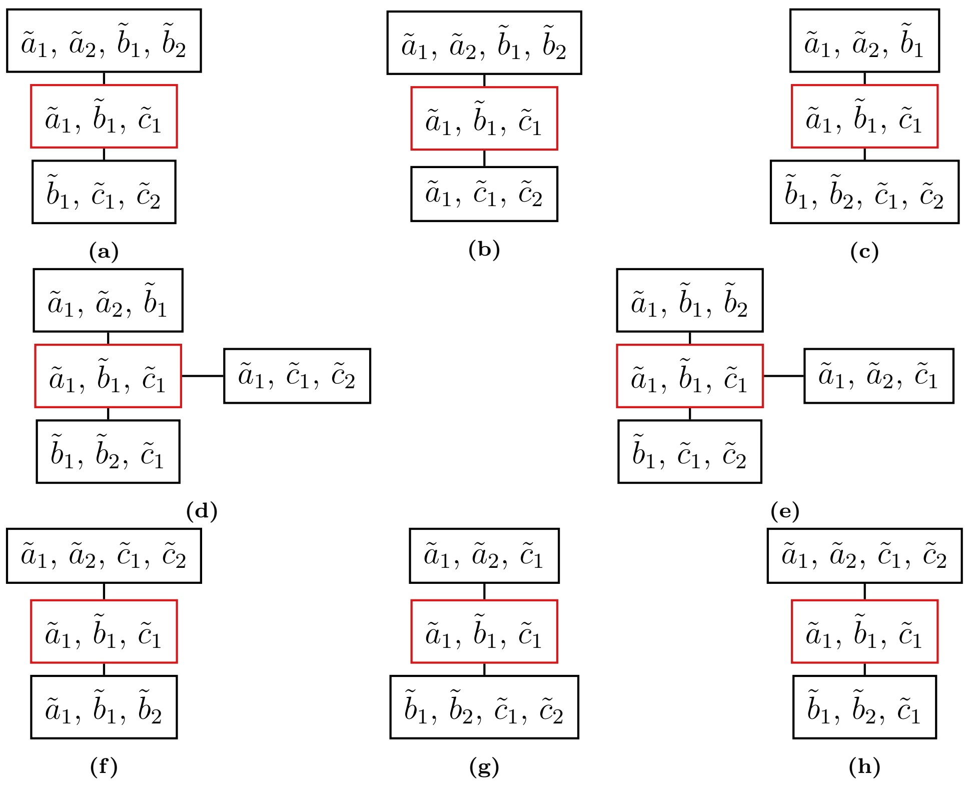
The following statement provides the upper bounds on the runtime for computing Boolean IJQs, based on the extended width measures introduced above.

Theorem 5.2.14 (Upper Bounds for Boolean IJQs)**.** *Consider a Boolean IJQ Q and a database D. The runtime upper bound of Q on D is $\tilde{\mathcal{O}}(|D|^{\mathsf{subw}^{ij}(Q)})$.*

Proof. By Proposition 5.2.12, we have $Q(D)$ if and only if $\bigvee_{q \in \mathbf{Q}} q(\tilde{D})$, where

$$(\mathbf{Q}, \tilde{D}) = \textsc{Forward-Reduction}(Q, D).$$

Therefore, the upper bound for the computation of $Q(D)$ is given by the upper bound of the query with the maximum upper bound among the CQ conjuncts in the



$\tilde{a}_1, \tilde{a}_2, \tilde{b}_1, \tilde{b}_2$
$\tilde{a}_1, \tilde{b}_1, \tilde{c}_1$
$\tilde{b}_1, \tilde{c}_1, \tilde{c}_2$
(a)
$\tilde{a}_1, \tilde{a}_2, \tilde{b}_1, \tilde{b}_2$
$\tilde{a}_1, \tilde{b}_1, \tilde{c}_1$
$\tilde{a}_1, \tilde{c}_1, \tilde{c}_2$
(b)
$\tilde{a}_1, \tilde{a}_2, \tilde{b}_1$
$\tilde{a}_1, \tilde{b}_1, \tilde{c}_1$
$\tilde{b}_1, \tilde{b}_2, \tilde{c}_1, \tilde{c}_2$
(c)
$\tilde{a}_1, \tilde{a}_2, \tilde{b}_1$
$\tilde{a}_1, \tilde{b}_1, \tilde{c}_1$
$\tilde{a}_1, \tilde{c}_1, \tilde{c}_2$
$\tilde{b}_1, \tilde{b}_2, \tilde{c}_1$
(d)
$\tilde{a}_1, \tilde{b}_1, \tilde{b}_2$
$\tilde{a}_1, \tilde{b}_1, \tilde{c}_1$
$\tilde{a}_1, \tilde{a}_2, \tilde{c}_1$
$\tilde{b}_1, \tilde{c}_1, \tilde{c}_2$
(e)
$\tilde{a}_1, \tilde{a}_2, \tilde{c}_1, \tilde{c}_2$
$\tilde{a}_1, \tilde{b}_1, \tilde{c}_1$
$\tilde{a}_1, \tilde{b}_1, \tilde{b}_2$
(f)
$\tilde{a}_1, \tilde{a}_2, \tilde{c}_1$
$\tilde{a}_1, \tilde{b}_1, \tilde{c}_1$
$\tilde{b}_1, \tilde{b}_2, \tilde{c}_1, \tilde{c}_2$
(g)
$\tilde{a}_1, \tilde{a}_2, \tilde{c}_1, \tilde{c}_2$
$\tilde{a}_1, \tilde{b}_1, \tilde{c}_1$
$\tilde{b}_1, \tilde{b}_2, \tilde{c}_1$
(h)

Figure 5.3: The hypertree decompositions of the eight queries in $\tilde{Q}_{\triangle}$. All decompositions have a bag $\{\tilde{a}_1, \tilde{b}_1, \tilde{c}_1\}$, whose materialisation requires the computation of a full Triangle CQ.

disjunction of Equation (5.11) (Definition 5.2.9). That is the one whose hypergraph has the maximum sub-modular width [5]. Hence, the runtime of evaluating Q on D is upper bounded by $\tilde{\mathcal{O}}((|D|)^{\max_{q \in \mathbf{Q}} \mathsf{subw}(q)})$. By Definition 5.2.13 we have

$$\tilde{\mathcal{O}}((|D|)^{\max_{q \in \mathbf{Q}} \mathsf{subw}(q)}) = \tilde{\mathcal{O}}((|D|)^{\mathsf{subw}^{ij}(Q)}).$$

Hence, $Q(D)$ can be computed in time $\tilde{\mathcal{O}}(|D|^{\mathsf{subw}^{ij}(Q)})$. $\square$

The sub-modular width of a CQ is less or equal than its fractional hypertree width [5]. Therefore, another valid upper bound for the runtime of the Boolean IJQ computation is $\tilde{\mathcal{O}}(|D|^{\mathsf{fhtw}^{\mathsf{ij}}(Q)})$.

Example 5.2.15 revisits Example 5.2.10 and analyze the upper bound for the Boolean Triangle IJQ. This is done by utilizing the upper bounds for the Boolean CQs obtained during its reduction.

Example 5.2.15 (The Complexity of the Boolean Triangle IJQ)**.** *The hypergraph of each one of the eight Boolean CQs from Example 5.2.10 admits a hypertree decomposition in the form of a star with a central bag* $\{a_1, b_1, c_1\}$ *(see Figure 5.3). In each of these decompositions, the materialisation of the central bag requires solving the full Triangle CQ*

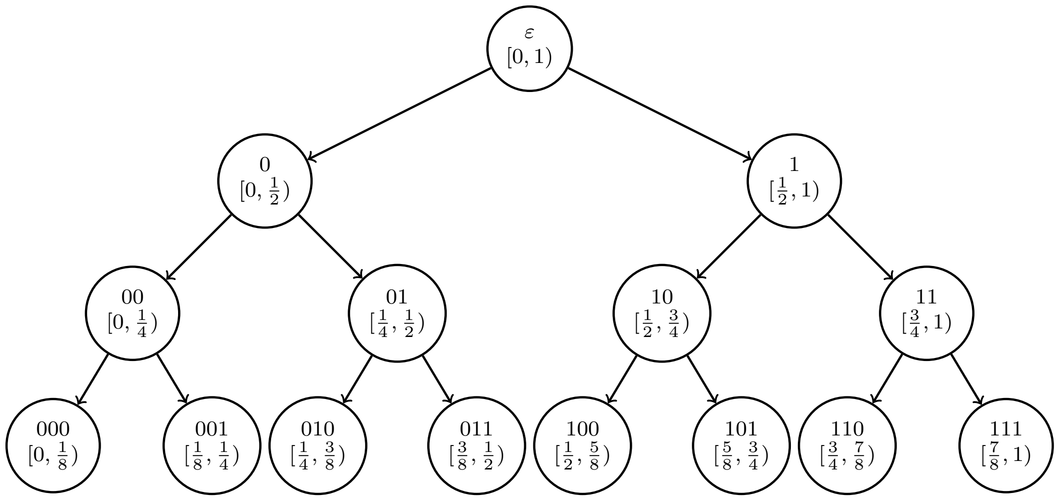
$$\langle \{a_1, b_1, c_1\}, \tilde{S}_1(a_1, b_1) \wedge \tilde{S}_2(b_1, c_1) \wedge \tilde{S}_3(a_1, c_1) \rangle,$$

where $\tilde{S}_1$ *is the projection of* $\tilde{R}_1$ *on* a_1, b_1*,* $\tilde{S}_2$ *is the projection of* $\tilde{R}_2$ *on* b_1, c_1*, and* $\tilde{S}_3$ *is the projection of* $\tilde{R}_3$ *on* c_1, a_1*. The new relations and their projections have size* $\mathcal{O}(n \cdot \log^2 n)$*. Hence, the materialisation of the join takes time* $\mathcal{O}((n \cdot \log^2 n)^{3/2}) = \mathcal{O}(n^{3/2} \cdot \log^3 n)$ *using existing worst-case optimal join algorithms [44]. Checking if any of the eight CQs is true takes time linear in the maximum size of the bags of its decomposition. This gives an overall computation time* $\mathcal{O}(n^{3/2} \cdot \log^3 n)$ *for* $\tilde{Q}_\triangle$ *and as a result for* $Q_\triangle$*.*

5.3 Backward Reduction

The previous section showed that any Boolean IJQ has as an upper bound the maximum runtime upper bound of the queries in its forward reduction up to a polylogarithmic factor in the size of data. This section does the reverse; it shows that the runtime of any Boolean IJQ has as a lower bound the maximum lower bound of the Boolean CQs in its forward reduction. This is achieved using a backward reduction, cf. Figure 1.1.

In computational complexity, a reduction is an algorithm that transforms one problem into another. By showing that there exists a sufficiently efficient reduction from one problem to another, one can prove that the latter problem is at least as hard as the first. Consider a Boolean IJQ Q, a Boolean CQ $\tilde{Q}^{(i)}$, whose schema is the same as the schema of one of the conjuncts in $\tilde{Q}$, and a database $\tilde{B}$, which matches



ε $[0,1)$
0 $[0,\frac{1}{2})$
1 $[\frac{1}{2},1)$
00 $[0,\frac{1}{4})$
01 $[\frac{1}{4},\frac{1}{2})$
10 $[\frac{1}{2},\frac{3}{4})$
11 $[\frac{3}{4},1)$
000 $[0,\frac{1}{8})$
001 $[\frac{1}{8},\frac{1}{4})$
010 $[\frac{1}{4},\frac{3}{8})$
011 $[\frac{3}{8},\frac{1}{2})$
100 $[\frac{1}{2},\frac{5}{8})$
101 $[\frac{5}{8},\frac{3}{4})$
110 $[\frac{3}{4},\frac{7}{8})$
111 $[\frac{7}{8},1)$

Figure 5.4: This figure illustrates an example of the segment tree that visualises the backward mapping function.

the schema of $\tilde{Q}^{(i)}$. The backward reduction reduces the decision problem $\tilde{Q}^{(i)}(\tilde{B})$ to the decision problem $Q(B)$, where B is a database that matches the schema of Q. The reduction is based on a bijection from the tuples of $\tilde{B}$ to the tuples of B.

Remark 5.3.1. *By the above description, it follows that the database $\tilde{B}$ contains scalars only, whereas B contains intervals (and scalars).*

Proposition 5.3.2 (Backward Reduction Correctness)**.** *Consider a Boolean IJQ Q, a Boolean CQ $\tilde{Q}^{(i)}$ from the forward reduction of Q, and a database $\tilde{B}$ that matches the structure of $\tilde{Q}^{(i)}$. There exists a bijection that maps each tuple from $\tilde{B}$ to a tuple from a database B that matches the structure of Q, such that $\tilde{Q}^{(i)}(\tilde{B})$ is equivalent to $Q(B)$.*

The following example explains the backward reduction for the Boolean Triangle Intersection Join.

Example 5.3.3 (The Boolean Triangle IJQ — Backward Reduction)**.** *Consider the Boolean IJQ $Q_\triangle$ from Example 5.2.10, and the Boolean CQ $\tilde{Q}^{(3)}_\triangle$ resulting from the reduction of $Q_\triangle$:*

$$\tilde{Q}^{(3)}_\triangle = \tilde{R}_1(\tilde{a}_1, \tilde{a}_2, \tilde{b}_1) \wedge \tilde{R}_2(\tilde{b}_1, \tilde{b}_2, \tilde{c}_1, \tilde{c}_2) \wedge \tilde{R}_3(\tilde{a}_1, \tilde{c}_1)$$

Let $\tilde{B} = \{\tilde{R}_1, \tilde{R}_2, \tilde{R}_3\}$ be a database with a matching schema. Define a bijective function F, called backward mapping, which maps each bit-string in $\bigcup_{i=0}^{m} \{0,1\}^i$, where m is a constant, to an interval $[x, y)$ where $x, y \in [0,1]$, as follows: $F(\varepsilon) = [0,1)$, $F(\text{"0"}) = [0, 1/2)$, $F(\text{"1"}) = [1/2, 1)$, $F(\text{"00"}) = [0, 1/4)$ and so on. For any given bit-string b, the intervals $F(b \circ \text{"0"})$ and $F(b \circ \text{"1"})$ correspond to the first and second half of the interval $F(b)$ respectively. The function F can be explained graphically using a segment tree $\mathfrak{T}_{\mathcal{I}}$, as defined in Definition 3.3.1, build over the elementary segments:

$$\mathcal{I} = \{[0, 1/2^m), [1/2^m, 2/2^m), \ldots, [(2^m - 1)/2^m, 1)\}.$$

The function F maps each node to its corresponding segment, hence, $F = \mathsf{seg}_{\mathcal{I}}$. Figure 5.4 depicts this segment tree for $m = 3$. Using the segment tree one can easily see that two bit-strings are ancestors of each other if and only if their corresponding segments intersect.

The decision problem $\tilde{Q}_{\triangle}^{(3)}(\tilde{B})$ can be reduced to the decision problem $Q_{\triangle}(B)$, where $B = \{R_1, R_2, R_3\}$ constructed as follows:

$$\begin{aligned}
R_1 &= \{(F(a_1 \circ a_2), F(b_1)) \mid (a_1, a_2, b_1) \in \tilde{R}_1\}, \\
R_2 &= \{(F(b_1 \circ b_2), F(c_1 \circ c_2)) \mid (b_1, b_2, c_1, c_2) \in \tilde{R}_2\}, \\
R_3 &= \{(F(a_1), F(c_1)) \mid (a_1, c_1) \in \tilde{R}_3\}.
\end{aligned}$$

It holds that $Q_{\triangle}(B)$ is equivalent to $\tilde{Q}_{\triangle}^{(3)}(\tilde{B})$. Furthermore, it holds that $|\tilde{B}| = |B|$. Therefore, solving $Q_{\triangle}$ is at least as difficult as solving $\tilde{Q}_3$. In similar way, one can obtain that for every $\tilde{Q}_i \in \{\tilde{Q}_1, \ldots, \tilde{Q}_8\}$ it holds that $Q_{\triangle}(b)$ is at least as difficult as solving $\tilde{Q}_i$. Notice that the forward reduction shows that $Q_{\triangle}$ is at most as difficult as solving the most difficult query among $\{\tilde{Q}_1, \ldots, \tilde{Q}_8\}$. Hence, overall, it holds that $Q_{\triangle}$ is as difficult as the most difficult query among $\tilde{Q}_1, \ldots$, and $\tilde{Q}_8$, meaning that given an optimal algorithm for solving $\tilde{Q}_1, \tilde{Q}_2, \ldots$, and $\tilde{Q}_8$, one can obtain an optimal *algorithm for solving $Q_{\triangle}$.*

5.3.1 Lower Bounds

Theorem 5.3.4. *Consider a Boolean IJQ Q. Let $\tilde{Q}^{(i)}$ be any Boolean CQ that corresponds to a conjunct in $\tilde{Q}$. For any database $\tilde{B}$, let $\Omega(T(|B|))$ be a lower bound on the time complexity for computing the query $\tilde{Q}^{(i)}(\tilde{B})$. Then, there cannot be an algorithm that computes $Q(B)$ in time $o(T(|B|))$ for any database B.*

Proof. For contradiction, assume that there is such an algorithm $\mathcal{A}_Q$. That is, there exists an algorithm $\mathcal{A}_Q$ that decides $Q(B)$ in time $o(T(|B|))$ for any database B. By Proposition 5.3.2, one can construct a bijection from the tuples in $\tilde{B}$ to the tuples of a database B, that matches the structure of Q. Furthermore, it holds that $Q(D)$ is equivalent to $\tilde{Q}(\tilde{B})$. Notice that it holds $|B| = |\tilde{B}|$. As a result, one can use the algorithm $\mathcal{A}_Q$ as an oracle to compute $\tilde{Q}^{(i)}(\tilde{B})$ in time $o(T(|\tilde{B}|))$. This is a contradiction since $\tilde{Q}(\tilde{B})$ has a lower bound of $\Omega(T(|\tilde{B}|))$. □

5.4 Beyond Boolean IJQs

This section explains how to compute the answer of a non-Boolean IJQ using the previously described forward reduction (Subsection 5.4.1). This observation might be useful for investigating the enumeration complexity of non-Boolean IJQs in the future. In light of this, a complexity statement is made about the enumeration complexity for full IJQs is made in this thesis (Subsection 5.4.2). In particular, it is stated that there exists an enumeration algorithm that computes $Q(D)$ by using $\tilde{\mathcal{O}}(|D|^{\mathsf{fhtw}^{ij}(Q)})$ preprocessing and constant delay enumeration of the tuples in the output.

5.4.1 Non-Boolean IJQs

Consider an IJQ Q and a matching database D. Assume that the IJQ is non-Boolean, therefore, it holds that $\mathsf{free}(Q) \neq \emptyset$. According to Section 2.2, the answer $Q(D)$ is a set of tuples

$$\left\{ (t_1[e], \ldots, t_n[e]) \middle| t_i \in R_i(e_i) \text{ for each } 1 \leq i \leq n, \text{ and} \right.$$
$$\left. \text{for each } u \in \mathsf{vars}(Q) : \left(\bigcap_{i \in \mathsf{ind}(Q,u)} t_i(u) \right) \neq \emptyset \right\}.$$

The forward reduction replaces a variable $v \in \mathcal{V}(Q)$ in the query with the variables $\{\tilde{v}_1, \ldots, \tilde{v}_i\} \cup \{\tilde{v}^{(\mu(e))} \mid e \in \mathcal{E}_v(Q)\}$, where the former variables in the union are new join variables, and the latter variables in the union are singleton variables, whose purpose is to preserve the original data when evaluating the reduced problem, i.e., when evaluating the new query $\tilde{Q}$ on the new database $\tilde{D}$. Consider any conjunct $\tilde{Q}^{(j)}$ in $\tilde{Q}$. It is natural to define the free variables of $\tilde{Q}^{(j)}$ as a set $\mathsf{free}(\tilde{Q}^{(j)})$ obtained from the set $\mathsf{free}(Q)$, by replacing each variable $v \in \mathsf{free}(Q)$ with its corresponding newly created singleton variables $\tilde{v}^{(\mu(e))}$ for each $e \in \mathcal{E}_v(Q)$. The result $Q(D)$ can be obtained by the result $\tilde{Q}(\tilde{D})$, by associating each tuple in the latter result to a tuple in the former result. In particular, by the construction of the query $\tilde{Q}$, and the database $\tilde{D}$ (Algorithm 4, and Proposition 5.2.12), given a tuple $(\tilde{t}_1, \ldots, \tilde{t}_n)$ which belongs to $\tilde{Q}(\tilde{D})$, one can construct a tuple $(t_1, \ldots . t_n)$ which belongs to $Q(D)$ by assigning $t_{\mu(e)}(v) = \tilde{t}_{\mu(e)}(\tilde{v}^{(\mu(e))})$ for each $v \in \mathsf{free}(Q)$ for each $e \in \mathcal{E}_v$. By applying the above procedure to all the tuples in the result of the reduced problem one can obtain all the tuples in the result of the input problem.

5.4.2 Full IJQs and Constant Delay Enumeration

The CQ evaluation has been extensively studied in the enumeration setting, where the tuples in the output of a query are reported one after the other and maximum delay between reporting two tuples is measured. In this case the target is to achieve enumeration with constant delay, which means that the time between reporting two consecutive tuples depends solely on the query size and not on the data size [12]. Let us consider full CQs. It is well understood that α-acyclic full CQs can be computed in linear time and with constant delay enumeration (Proposition 3.2.2). As a result, for full CQs in general, one can build a hypertree decomposition of the query (Definition 3.1.16), and materialise its bags with the appropriate amount of preprocessing time. Therefore, the full CQ is transformed to an α-acyclic CQ, hence, it is possible to enumerate the tuples in its output with constant delay (Subsection 3.2.2).

Full IJQs are special cases of non-Boolean IJQs, where all the variables in the query belong to the set of free variables. For the full IJQs, aside from how to compute the result of the input problem using the result of the reduced problem (Subsection 5.4.1), a specific statement is made for the amount of preprocessing needed for the enumeration of the tuples in the output in constant delay (Proposition 5.4.1). We have that $\mathsf{free}(Q) = \mathsf{vars}(Q)$, which means that the input query is a full IJQ. Recall that the set of the free variables of $\tilde{Q}^{(j)}$, which is a CQ conjunct in $\tilde{Q}$, consists of the singleton variables $\{\tilde{v}^{(\mu(e))} \mid e \in \mathcal{E}_v(Q)\}$ for each $v \in \mathsf{vars}(Q)$. The result $\tilde{Q}(\tilde{D})$ can be computed as explained in Subsection 5.4.2. For ease of description, since $\tilde{Q}^{(i)}$ is a CQ, the relational representation of the result $\tilde{Q}^{(i)}(\tilde{D})$ is adopted; that is the result is a relation over $\mathsf{free}(\tilde{Q}^{(i)})$ (see Section 2.2). By Property 5.1.8, it holds that for each $v \in \mathsf{vars}(Q)$, and for each satisfying assignment of values for the variables $\{\tilde{v}^{(\mu(e))} \mid e \in \mathcal{E}_v(Q)\}$ there is a uniquely satisfying assignment of values for the variables $\tilde{v}_1, \dots, \tilde{v}_i$. This is important for the efficient enumeration of the tuples in the result $Q(D)$, as one can enumerate the tuples in the result of the full query $\tilde{Q}^{(i)}$ and then remove the values assigned to the the variables $\tilde{v}_1, \ldots, \tilde{v}_i$ without affecting the size of the output. In other words, one can add the variables $\tilde{v}_1, \ldots, \tilde{v}_i$ in the set of free variables of $\tilde{Q}^{(i)}$, and compute the query as all the variables are free without affecting the asymptotic complexity of the preprocessing and the enumeration.

Proposition 5.4.1. *Consider a full IJQ Q and a database D. There exists an enumeration algorithm that computes $Q(D)$ using $\tilde{\mathcal{O}}(|D|^{\mathsf{fhtw}^{ij}(Q)})$ preprocessing and constant delay enumeration of the tuples in the output.*

The above complexity is justified by the fact that, by Definition 5.2.13, $\mathsf{fhtw}^{ij}(Q)$ is the fractional hypertree width (Definition: 3.1.17) of the CQ with the maximum fractional hypertree width among the CQs in the union generated by the reduction. The tuples of the result of that particular full CQ can be enumerated with constant delay after $\tilde{\mathcal{O}}(|D|^{\mathsf{fhtw}^{ij}(Q)})$ of preprocessing [12]. The preprocessing for this conjunct dominates the entire preprocessing phase for $\tilde{Q}$. After this amount of preprocessing the tuples of Q can be enumerated with constant delay.

5.5 Illustrative Experimental Evaluation

This section illustrates the forward reduction's performance on synthetic databases. The goal is not to provide an in-depth experimental evaluation, but to highlight the efficiency of the reduction in practice as well as its limitations.

5.5.1 Setup

The following experiments were run on a machine with an Intel Xeon Platinum 8268 CPU @2.90 GHz, with 48 cores, 392 GB of RAM, and CentOS Linux 8.1 operating system. This machine belongs to the Advanced Research Computing (ARC) cluster, which is a collection of high-performance computing resources, available to researchers within the University of Oxford. All the datasets are synthetic. The original implementation of the forward reduction (Algorithm 4) in C++, is developed in [21], which is also based on [35, 36]. This is an in-memory implementation of the forward reduction. The code is adapted appropriately in the context of this thesis to support also variables that range over scalars (Section 2.2).

5.5.2 Benchmarks

The databases used for the experiments are randomly generated. All tables within a database consist of the same number of tuples. The tables consist of columns (three at maximum), where each column is of interval or scalar type. The columns within a table are independent, and intervals are randomly generated using two alternative methods: (1) both the endpoints of the intervals are chosen uniformly at random from a domain; or (2) the left endpoint of each interval is chosen uniformly at random from a domain, and the length of the interval is chosen with respect to the Zipfian distribution with skew parameter $\alpha = 1.6$. The right endpoint must be within the domain. The first method is referred to as *random*, and the second as *Zipf-based*. The Zipf-based method may produce more realistic database instances in comparison to the random method, because the intervals do not span a wide range in the domain.

Three full IJQs are considered, namely $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$, which are detailed in the following. Intersection joins produce large results; hence, reporting the tuples in the output dominates the computation time. For this reason, it is meaningful to consider counting the tuples in the output rather than listing them. The evaluation algorithm, called IJQ, runs in main memory and consists of two stages, the reduction and the join evaluation. The former stage is based on Algorithm 4; it generates a union of CQs, and a matching database. It must be noted that for the IJQs $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$, all the CQs in their reduction are α-acyclic. The latter stage is based on Yannakaki's algorithm (Subsection 3.2.2). The IJQ algroritm is compared with two PostgreSQL

#Tuples	Time (secs)			Red. Time (secs)	Count
	IJQ	RANGESS	INEQUALITIES		
5K	3.7	3.2	3.4	0.4	∼11M
10K	9.1	11.8	12.5	0.8	∼44M
50K	71.1	361.7	403.7	5.8	∼1B
100K	182.2	656.4	692.7	13.0	∼4B

Table 5.1: This table summarises the experimental findings for the running time of the query $\mathcal{Q}_1$ on five databases with a varied number of tuples per table. The reported running times correspond to three different approaches, namely IJQ, RANGES and INEQUALITIES. The table also includes the reduction time for IJQ, as well as the number of tuples in the output.

implementations; one that uses inequality conditions (e.g. $a.\mathsf{start} \leq b \leq a.\mathsf{end}$), and one that uses range conditions (e.g., check if the point b is within the the interval a). Both implementations are based on the rewriting from Remark 4.2.11, hence, they are similar to each other. The PostgreSQL queries are written in disjunctive form and the conjuncts are executed in parallel, using multiple threads. For applying the queries with inequalities the intervals are stored in two columns, one for the left endpoint and one for the right endpoint. All columns are indexed by B-trees (similarly to [23]). For applying the queries with ranges the intervals are stored in one column of type range, which is provided by the PostGIS extension for PostgreSQL. All such columns are indexed using spatial indexes. Note that PostgreSQL is disk-based engine; although there is enough main memory so that the query is executed in main memory, there may exist overhead caused it implements advanced features and extra data structures to support disk-based computation.

5.5.3 Runtime Experiments

The performance of the IJQs $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$ is tested in the following.

Query $\mathcal{Q}_1$

Consider the IJQ $\mathcal{Q}_1 = R_1([a],[b]) \wedge R_2([a],[b])$. In this query, the variables a and b in all their occurrences, both in R_1 and R_2 range over intervals. To understand what

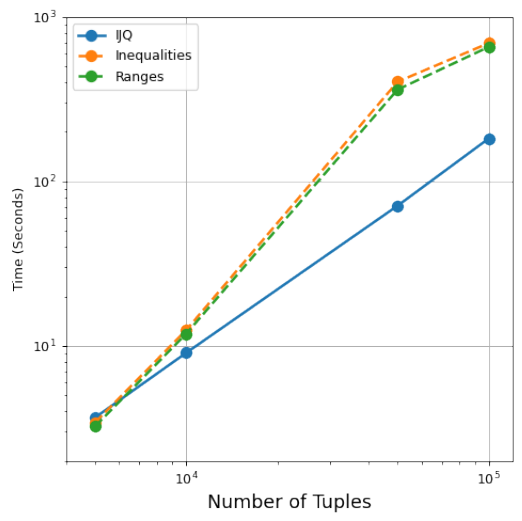


Figure 5.5: This figure graphs the experimental findings for the running time of the query $\mathcal{Q}_1$ on five databases with a varied number of tuples per table. The reported running times correspond to three different approaches, namely IJQ, RANGES and INEQUALITIES. The result are also summarised in Table 5.1.

this query asks in practice, consider two sets of rectangles in two dimensions, and the task of counting pairs of the overlapping rectangles in the cross-product of the two sets. For this particular query, the reduction generates a union of 4 CQs.

This set of experiments uses 5 databases with a varied number of tuples per relation ranging from 5000 to 100000. Recall that every relation the database has the same number of tuples. The data are generated according to the *random* method. The running times are summarised in Table 5.1, together with the reduction time for the IJQ approach and the number of tuples in the output. The times are also illustrated in Figure 5.5, showing that for $\mathcal{Q}_1$, the IJQ approach outperforms both PostgreSQL approaches. In particular, the increase in the performance gap is noticeable, at least for the sizes from 5000 to 50000.

Query $\mathcal{Q}_2$

Consider the IJQ $\mathcal{Q}_2 = R_1([a]) \wedge R_2([a]) \wedge R_3([a])$. This query is a set intersection join over 3 sets of intervals, and it is the simplest query that can be used to demonstrate

Table 5.2: This table summarises the experimental findings for $\mathcal{Q}_2$'s running time on four databases with varying numbers of tuples per relation, where the intervals are generated using both the random and the Zipf-based method. The results correspond to four different approaches, namely IJQ, SETIJ, RANGES and INEQUALITIES. The table includes the reduction time for the IJQ approach, as well as the count of the tuples in the output.

(a) Random

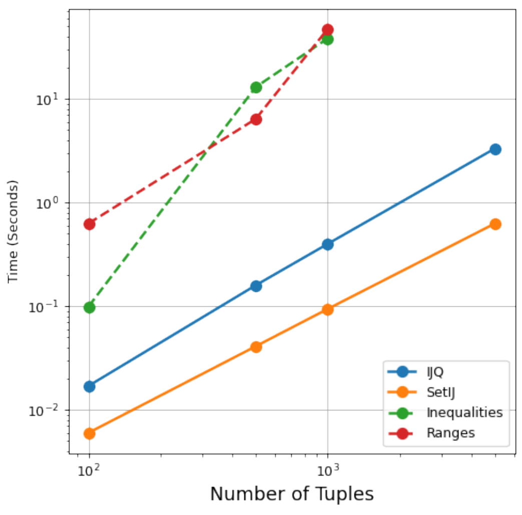
#Tuples	Time (secs)				Red. Time (secs)	Count
	IJQ	SETIJ	RANGES	INEQUALITIES		
100	0.02	0.006	0.1	0.01	0.003	∼393K
500	0.2	0.04	6.4	13.0	0.02	∼48M
1K	0.4	0.09	47.3	38.3	0.04	∼399M
5K	3.3	0.6	>600	>600	0.3	∼50B

(b) Zipf-based

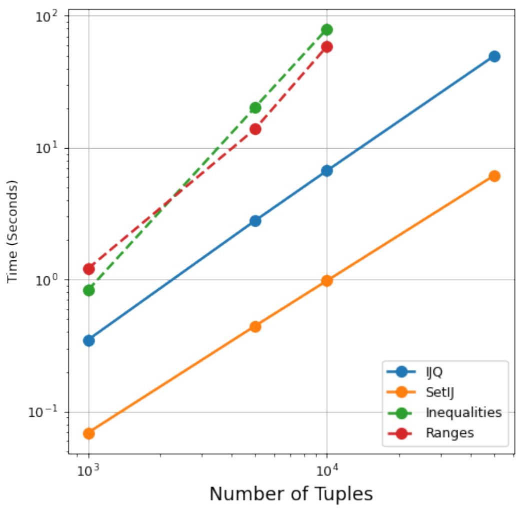
#Tuples	Time (secs)				Red. Time (secs)	Count
	IJQ	SETIJ	RANGES	INEQUALITIES		
1K	0.4	0.07	1.2	0.8	0.04	∼767K
5K	2.8	0.4	13.9	20.2	0.2	∼12M
10K	6.7	1.0	58.6	79.5	0.4	∼37M
50K	49.6	6.2	>600	>600	2.6	∼772M

the effectiveness of the approaches introduced by this thesis for queries with multi-way intersection joins. For this query, the reduction generates a union of 6 CQs.

Two sets of experiments are conducted. The first set uses 5 databases with varied tuple counts per relation, ranging from 100 to 1000, generated in the random method. A graphic representation of the running times is shown in Figure 5.6a and summarised in Table 5.2b. The results show that SETIJ and IJQ outperform both PostgreSQL approaches. The second set of experiments uses 5 databases, with a varied number of tuples per relation, ranging from 100 to 1000, created using the Zipf-based method. A graphic representation of the running times is shown in Figure 5.6b and summarised in Table 5.2b. The same observation holds; however, the performance gap with the PostgreSQL approaches is reduced. This happens because the intervals do not cover a wide range of the domain; therefore the tuples produced by joining any two out of the three tables are less than if the intervals were generated randomly. The performance gap between SETIJ and the IJQ approach is justified by the fact that the latter



(a)



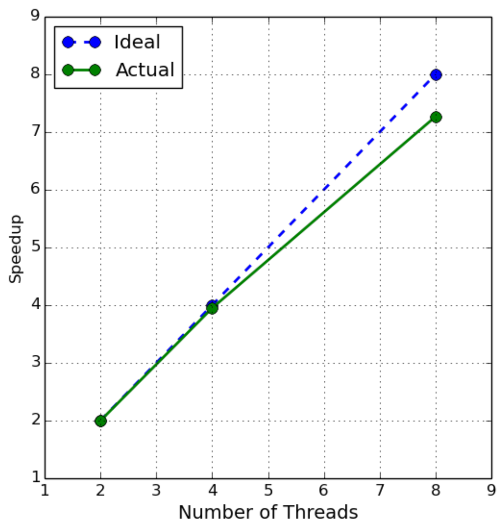
(b)

Figure 5.6: This figure graphs the running times that correspond to three different approaches, namely SetIJ, IJQ, Ranges, and Inequalities, against the number of tuples per relation. Figure 5.6a illustrates the running times for databases where the intervals are constructed using the random method, whereas Figure 5.6a illustrates the running times for databases where the intervals are constructed using the Zipf-based method. The results are also summarised in Table 5.2b.

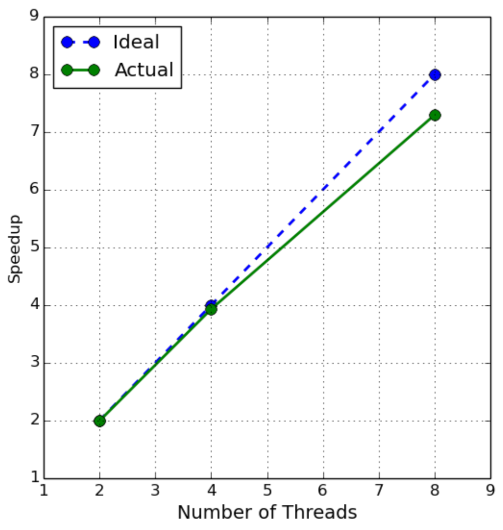
approach incurs extra logarithmic factors in the data size.

Query $\mathcal{Q}_3$

Consider the IJQ $\mathcal{Q}_3 = R_1(a, b, c) \wedge R_2([a], [b]) \wedge R_3([b], [c]) \wedge R_4([a], [c])$, where all the variables in R_1 range over scalars, whereas the variables in the rest of the relations range over intervals. The databases are randomly generated using the Zipf-based method, and the Random methods and each relation consists of 100 tuples. As mentioned earlier, the reduction transforms an IJQ evaluation problem to a union of CQ evaluation problems, which are independent of each other, i.e., the input of one CQ does not require the output from another CQ in the union. In this case, the number of CQs generated by the reduction is 8. Hence, the union of CQs can be computed in parallel. This set of experiments is carried out to demonstrate the speedup of the IJQ when varying the number of parallel threads from 2 to 8. Figure 5.7 illustrates the speedup ratio of the computation time for the sequential algorithm to



(a) Random



(b) Zipf-based

Figure 5.7: This figure graphs the ratio of the compute time for the sequential algorithm to the time for the parallel algorithm.

the time for the parallel algorithm. The speedup runtime scales with the number of threads.

5.6 Discussion

The data complexity of Boolean IJQs is established in this chapter by employing a reduction to Boolean CQs. The complexity of a Boolean IJQ, in particular, is precisely that of the most difficult Boolean CQ in the disjunction generated by the reduction. Hence, given optimal algorithms for Boolean CQs, one can acquire optimal algorithms for Boolean IJQs. Furthermore, it is shown this reduction can be extended so that it can be applied to non-Boolean IJQs. Last but not least, the overall approach, including the reduction and the CQ evaluation, is compared with standard PostgreSQL implementations on synthetic databases, showing the potential to be practical.

Experimental Evaluation

Section 5.5 provides us with some illustrative performance experiments for IJQs applied on synthetic databases. It illustrates the efficiency of the IJQ reduction, however, it does not provide an in-depth evaluation and comparison. In the future, it would be useful to test the performance of the IJQ reduction on realistic use cases, where the data distributions differ from the ones used in the context of this thesis. Further, an in-depth evaluation, taking into account the space requirements would help us to assess the performance more accurately. Furthermore, existing database systems rely on statistics and specialised algorithms to generate efficient query plans for queries; thus, outperforming such algorithms may be difficult [51]. Such a task would require further optimisations on the reduction and, ideally removing the arising polylogarithmic factors in the data size. Last but not least, another aspect of this thesis that could be improved is the collection of existing in-memory systems and implementations to serve as competitors; a as mentioned in Section 5.5 the IJQ algorithm is compared to a disk-based database systems and this comparison may be unfair due to the overhead caused by the implementation of advanced features.

Theoretical Aspects

Section 5.4 explains briefly how the forward reduction can be used to solve non-Boolean IJQs. It would be interesting to investigate the enumeration complexity of IJQs in the presence of free variables in greater depth [12]. In particular, the intriguing question would be to identify the class of IJQs that admit quasi-linear time preprocessing, followed by constant delay enumeration of the tuples in the output. This can be accomplished by utilising existing results for the enumeration of CQs and unions of CQs [18, 9, 52].

As a future work, it would also be interesting to consider queries with inequalities and try to understand their complexity by utilising reductions to queries with equality joins, as in the case of intersection joins. Can such an approach recover or even improve the results obtained by [56, 2]? For instance, the Boolean query with inequalities

$$Q = R(a,b) \wedge S(b,c) \wedge T(c,d) \wedge a \leq c \tag{5.12}$$

can be evaluated in $\tilde{\mathcal{O}}(N^{\frac{3}{2}})$ [2]. The same complexity can be achieved by rewriting the query from Equation (5.12) in the form of an IJQ as follows:

$$Q' = R(a,b) \wedge S(b,c) \wedge T(c,[a]) \tag{5.13}$$

where the variable d is replaced by $[a]$ which ranges ove the intervals $(-\infty, d]$. Note that the database is modified to accommodate the interval $(-\infty, d]$ for each value d. The problem of evaluating the query in Equation (5.13) on the corresponding database can be further reduced to the problem of evaluating the Boolean CQ

$$Q'' = R(\tilde{a}_1, \tilde{a}_2, b) \wedge S(b,c) \wedge T(c, \tilde{a}_1),$$

on a new database, as suggested by Definition 5.2.9. The complexity of the latter evaluation problem is $\tilde{\mathcal{O}}(N^{\frac{3}{2}})$. It is therefore interesting to identify queries with inequalities for which the above approach has lower asymptotic complexity than commonly used techniques.

5.7 Proofs

5.7.1 Proof of Lemma 5.1.7

Lemma 5.1.7 (Predicate Rewriting — Equalities). *The predicate in Lemma 5.1.1 can be equivalently rewritten as follows.*

- *Case $\dot{S} \neq \emptyset$: There exists a permutation $\sigma \in \mathsf{perms}(\bar{S})$ and a tuple of bit-strings*

$(b_1, \ldots, b_{c+1})$ *such that:*

$$(b_1 \circ \cdots \circ b_i) \in \mathsf{cp}_{\mathcal{I}}(\sigma_i), \textit{ for each } 1 \leq i \leq c$$

$$(b_1 \circ \cdots \circ b_{c+1}) = \mathsf{leaf}_{\mathcal{I}}(x_i), \textit{ for each } c < i \leq k$$

- *Case $\dot{S} = \emptyset$: There exists a permutation $\sigma \in \mathsf{perms}(\bar{S})$ and a tuple of bit-strings $(b_1, \ldots, b_c)$ such that:*

$$(b_1 \circ \cdots \circ b_i) \in \mathsf{cp}_{\mathcal{I}}(\sigma_i), \textit{ for each } 1 \leq i < c$$

$$(b_1 \circ \cdots \circ b_c) = \mathsf{leaf}_{\mathcal{I}}(\sigma_c)$$

Proof. "$\Longrightarrow$": Assume that $\dot{S} \neq \emptyset$. Assume that the predicate in Equation (5.2) is true. By Lemma 5.1.2 and Lemma 5.1.4 Equation (5.2) is equivalent to Equation (5.7). Hence, the predicate in Equation (5.7) is true. Therefore, there exists a permutation $\sigma \in \mathsf{perms}(\bar{S})$ and a tuple of nodes $(n_1, \ldots, n_{c+1})$ such that: $n_1 \in \mathsf{anc}_{\mathcal{I}}(n_2), \ldots, n_c \in \mathsf{anc}_{\mathcal{I}}(n_{c+1})$, $n_{c+1} = \mathsf{leaf}_{\mathcal{I}}(x_{c+1})$, $x_{c+1} = \cdots = x_k$ and $n_j \in \mathsf{cp}_{\mathcal{I}}(\sigma_j)$ for each $1 \leq j \leq c$. By Property 3.3.3, it holds that n_1 is a prefix of n_2 is a prefix of n_3 and so on. Hence, there exists a tuple of bit-strings $(b_1, \ldots, b_{c+1})$ such that $b_1 \circ \cdots \circ b_i \in \mathsf{cp}_{\mathcal{I}}(\sigma_i)$ for each $1 \leq i \leq c$ and $b_1 \circ \cdots \circ b_{c+1} = \mathsf{leaf}_{\mathcal{I}}(x_{c+1}) = \cdots = \mathsf{leaf}_{\mathcal{I}}(x_k)$. Assume that $\dot{S} = \emptyset$. Assume that the predicate in Equation (5.3) is true. By Lemma 5.1.4, Equation (5.3) is equivalent to Equation 5.7. Hence, there exists a permutation $\sigma \in \mathsf{perms}(\bar{S})$ and a tuple of nodes $(n_1, \ldots, n_c)$ such that: $n_1 \in \text{anc}(n_2), \ldots, n_{c-1} \in \mathsf{anc}(n_c)$, $n_c = \mathsf{leaf}_{\mathcal{I}}(\sigma_c)$ and $n_j \in \mathsf{cp}_{\mathcal{I}}(\sigma_j)$ for each $1 \leq j \leq c$. By Property 3.3.3, it holds that n_1 is a prefix of n_2 is a prefix of n_3 and so on. Therefore, there exists a tuple of bit-strings $(b_1, \ldots, b_c)$ such that $b_1 \circ \cdots \circ b_i \in \mathsf{cp}_{\mathcal{I}}(\sigma_i)$ for each $1 \leq i < c$ and $b_1 \circ \cdots \circ b_c = \mathsf{leaf}_{\mathcal{I}}(\sigma_c)$.

"$\Longleftarrow$": Assume that $\dot{S} \neq \emptyset$. Assume there exists a permutation $\sigma \in \mathsf{perms}(\bar{S})$ and a tuple of bit-strings $(b_1, \ldots, b_{c+1})$ such that for each $1 \leq i \leq c$ we have $b_1 \circ \cdots \circ b_i \in$

$\mathsf{cp}_{\mathcal{I}}(\sigma_i)$ and $b_1 \circ \cdots \circ b_{c+1} = \mathsf{leaf}_{\mathcal{I}}(x_{c+1}) = \cdots = \mathsf{leaf}_{\mathcal{I}}(x_k)$. Let $n_i = b_1 \circ \cdots \circ b_i$ for each $1 \leq i \leq c$ and $n_{c+1} = b_1 \circ \cdots \circ b_{c+1}$. It holds that $n_i \in \mathsf{cp}_{\mathcal{I}}(\sigma_i)$ for each $1 \leq i \leq c$ and $n_{c+1} = \mathsf{leaf}_{\mathcal{I}}(x_{c+1}) = \cdots = \mathsf{leaf}_{\mathcal{I}}(x_k)$. By Property 3.3.3, n_1 is prefix of n_2, n_2 is prefix of n_3 and so on. Hence, $n_1 \in \mathsf{anc}(n_2)$, $n_2 \in \mathsf{anc}(n_3)$, $\ldots$, $n_{c-1} \in \mathsf{anc}(\mathsf{leaf}_{\mathcal{I}}(x_{c+1}))$. By the construction of the segment tree, this implies that Equation (5.2) is true. Assume that $\dot{S} = \emptyset$. Assume there exists a permutation $\sigma \in \mathsf{perms}(\bar{S})$ and a tuple of bit-strings $(b_1, \ldots, b_c)$ such that for each $1 \leq i < c$ we have $b_1 \circ \cdots \circ b_i \in \mathsf{cp}_{\mathcal{I}}(\sigma_i)$ and $b_1 \circ \cdots \circ b_c = \mathsf{leaf}_{\mathcal{I}}(\sigma_c)$. Let $n_i = b_1 \circ \cdots \circ b_i$ for each $1 \leq i \leq c$. It holds that $n_i \in \mathsf{cp}_{\mathcal{I}}(\sigma_i)$ for each $1 \leq i < c$ and $n_c = \mathsf{leaf}_{\mathcal{I}}(\sigma_c)$. By Property 3.3.3, n_1 is prefix of n_2, n_2 is prefix of n_3 and so on. Hence, $n_1 \in \mathsf{anc}(n_2)$, $n_2 \in \mathsf{anc}(n_3)$, $\ldots$, $n_{c-1} \in \mathsf{anc}(n_c)$. Given the structure of the segment tree, this implies that Equation (5.3) is true. □

5.7.2 Proof of Lemma 5.2.7

Lemma 5.2.7 (One-step Forward Reduction Correctness). *Consider a Boolean IJQ Q and a database D. Consider also a variable $v \in \mathcal{V}(Q)$. It holds that $Q(D)$ is equivalent to $\tilde{Q}^{(v)}(\tilde{D}^{(v)})$, where $\tilde{Q}^{(v)}$ and $\tilde{D}^{(v)}$ are constructed by Definitions 5.2.1.*

Proof. "$\implies$": Assume that $Q(D)$ is true. By the semantics of IJQs, there exist tuples $(t_{\mu(e)})_{e\in\mathcal{E}(Q)} \in \prod_{e\in\mathcal{E}(Q)} R_{\mu(e)}$, that satisfy $\left(\bigcap_{e\in\mathcal{E}_v(Q)} t_{\mu(e)}\right) \neq \emptyset$, and also satisfy the rest of the join conditions in the query Q.

- Case $\dot{\mathcal{E}}_v(Q) \neq \emptyset$: By Lemma 5.1.1 it holds that $t_{\mu(e)}(v) = \xi$ for some scalar ξ for each $e \in \dot{\mathcal{E}}_v(Q)$, and $\xi \in \left(\bigcap_{e\in\bar{\mathcal{E}}_v(Q)} t_{\mu(e)}(v)\right)$. Hence, by Lemma 5.1.7 there exist a permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$ and a tuple of bit-strings $(b_1, \ldots, b_{\bar{n}_v+1})$ such that:

$$b_1 \circ \cdots \circ b_i \in \mathsf{cp}_{\mathcal{I}}(t_{\mu(\sigma_i)}(v)) \qquad \forall\, i \in [1, \bar{n}_v]$$
$$b_1 \circ \cdots \circ b_{\bar{n}_v+1} = \mathsf{leaf}_{\mathcal{I}}(t_{\mu(e)}(v)) \qquad \forall\, e \in \dot{\mathcal{E}}_v(Q)$$

By Definition 5.2.5, there exist tuples:

$$\tilde{t}_{\mu(\sigma_i)} \in \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i), \text{ where } \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i) \in \tilde{D}^{(v,\sigma)} \qquad \forall\ i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(e)} \in \tilde{R}_{\mu(e)}(\tilde{e}), \text{ where } \tilde{R}_{\mu(e)}(\tilde{e}) \in \tilde{D}^{(v,\sigma)} \qquad \forall\ e \in\ \dot{\mathcal{E}}_v(Q)$$

$$\tilde{t}_{\mu(f)} \in \tilde{R}_{\mu(f)}(f), \text{ where } \tilde{R}_{\mu(f)}(f) \in \tilde{D}^{(v,\sigma)} \qquad \forall\ f \in\ \mathcal{E}(Q) \setminus \mathcal{E}_v(Q)$$

where $\tilde{\sigma}_i$ and $\tilde{e}$ are defined by Equation (5.8) and (5.9) respectively, such that:

$$\tilde{t}_{\mu(\sigma_i)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(\sigma_i)}(\tilde{v}_i) = b_i \qquad \forall\ i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(e)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(e)}(\tilde{v}_{\bar{n}_v+1}) = b_{\bar{n}_v+1} \qquad \forall\ e \in\ \dot{\mathcal{E}}_v(Q)$$

Hence, the above tuples satisfy the equality joins on the variables $\tilde{v}_1, \cdots, \tilde{v}_{\bar{n}_v+1}$ in the query $\tilde{Q}^{(v,\sigma)}$. Furthermore, we have:

$$\tilde{t}_{\mu(\sigma_i)}[\tilde{\sigma}_i \setminus (\{\tilde{v}_1, \ldots, \tilde{v}_i\} \cup \{\tilde{v}^{(\mu(\sigma_i))}\})] = t_{\mu(\sigma_i)}[\sigma_i \setminus \{v\}] \qquad \forall\ i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(e)}[\tilde{e} \setminus (\{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v+1}\} \cup \{\tilde{v}^{(\mu(e))}\})] = t_{\mu(e)}[e \setminus \{v\}] \qquad \forall\ e \in\ \dot{\mathcal{E}}_v(Q)$$

$$\tilde{t}_{\mu(f)} = t_{\mu(f)} \qquad \forall\ f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)$$

Hence, the same tuples satisfy the rest of the join conditions in the query $\tilde{Q}^{(v,\sigma)}$. Therefore, it holds that $\tilde{Q}^{(v)}(\tilde{D}^{(v)})$ is true.

- Case $\dot{\mathcal{E}}_v(Q) = \emptyset$: By Lemma 5.1.1 it holds that $\left(\bigcap_{e \in \bar{\mathcal{E}}_v(Q)} t_{\mu(e)}(v)\right) \neq \emptyset$. Hence, by Lemma 5.1.7, there exists a permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$ and a tuple of bit-strings $(b_1, \ldots, b_{\bar{n}_v})$ such that:

$$b_1 \circ \cdots \circ b_i \in \mathsf{cp}_{\mathcal{I}}(t_{\mu(\sigma_i)}(v)) \qquad \forall\ i \in [1, \bar{n}_v - 1]$$

$$b_1 \circ \cdots \circ b_{\bar{n}_v+1} = \mathsf{leaf}_{\mathcal{I}}(t_{\mu(e)}(v)) \qquad \forall\ i = \bar{n}_v$$

By Definition 5.2.5, there exist tuples:

$$\tilde{t}_{\mu(\sigma_i)} \in \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i), \text{ where } \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i) \in \tilde{D}^{(v,\sigma)} \qquad \forall\, i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(f)} \in \tilde{R}_{\mu(f)}(f), \text{ where } \tilde{R}_{\mu(f)}(f) \in \tilde{D}^{(v,\sigma)} \qquad \forall\, f \in\ \mathcal{E}(Q) \setminus \mathcal{E}_v(Q)$$

where $\tilde{\sigma}_i$ and $\tilde{e}$ are defined by Equations (5.8), and (5.9) respectively, such that:

$$\tilde{t}_{\mu(\sigma_i)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(\sigma_i)}(\tilde{v}_i) = b_i \qquad \forall\, i \in\ [1, \bar{n}_v]$$

Hence, the above tuples satisfy the equality joins on the variables $\tilde{v}_1, \cdots, \tilde{v}_{\bar{n}_v+1}$ in the query $\tilde{Q}^{(v,\sigma)}$. Furthermore, we have:

$$\tilde{t}_{\mu(\sigma_i)}[\tilde{\sigma}_i \setminus (\{\tilde{v}_1, \ldots, \tilde{v}_i\} \cup \{\tilde{v}^{(\mu(\sigma_i))}\})] = t_{\mu(\sigma_i)}[\sigma_i \setminus (\{v\})] \qquad \forall\, i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(f)} = t_{\mu(f)} \qquad \forall\, f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)$$

Hence, the same tuples satisfy the rest of the join conditions in the query $\tilde{Q}^{(v,\sigma)}$. Therefore, it holds that $\tilde{Q}^{(v)}(\tilde{D}^{(v)})$ is true.

" $\Longleftarrow$ ": Assume that $\tilde{Q}^{(v)}(\tilde{D}^{(v)})$ is true. That means that there exists a permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v)$ and there exist tuples $(\tilde{t}_{\mu(e)})_{e\in\mathcal{E}(Q)} \in \prod_{e\in\mathcal{E}(Q)} \tilde{R}_{\mu(e)}(\tilde{e}^{(v,\sigma)})$ that satisfy all the join conditions in the query $\tilde{Q}^{(v,\sigma)}$.

- Case $\dot{\mathcal{E}}_v(Q) \neq \emptyset$: By the semantics of IJQs, it holds that:

$$\tilde{t}_{\mu(\sigma_i)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(\sigma_i)}(\tilde{v}_i) = b_i \qquad \forall\, i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(e)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(e)}(\tilde{v}_{\bar{n}_v+1}) = b_{\bar{n}_v+1} \qquad \forall\, e \in\ \dot{\mathcal{E}}_v(Q)$$

By Definition 5.2.5, there exist tuples:

$$\begin{aligned}
&t_{\mu(\sigma_i)} \in R_{\mu(\sigma_i)}(\sigma_i), \text{ where } R_{\mu(\sigma_i)}(\sigma_i) \in D && \forall\, i \in\ [1, \bar{n}_v] \\
&t_{\mu(e)} \in R_{\mu(e)}(e), \text{ where } R_{\mu(e)}(e) \in D && \forall\, e \in\ \dot{\mathcal{E}}_v(Q) \\
&t_{\mu(f)} \in R_{\mu(f)}(f) && \forall\, f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)
\end{aligned}$$

such that:

$$\begin{aligned}
&b_1 \circ \cdots \circ b_i \in \mathsf{cp}_{\mathcal{I}}(t_{\mu(\sigma_i)}(v)) && \forall\, i \in\ [1, \bar{n}_v] \\
&b_1 \circ \cdots \circ b_{\bar{n}_v+1} = \mathsf{leaf}_{\mathcal{I}}(t_{\mu(e)}(v)) && \forall\, e \in\ \dot{\mathcal{E}}_v(Q)
\end{aligned}$$

Hence, the above tuples satisfy the intersection join on variable v in the query Q. Furthermore, we have:

$$\begin{aligned}
t_{\mu(\sigma_i)}[\sigma_i \setminus (\{v\})] &= \tilde{t}_{\mu(\sigma_i)}[\tilde{\sigma}_i \setminus (\{\tilde{v}_1, \ldots, \tilde{v}_i\} \cup \{\tilde{v}^{(\mu(\sigma_i))}\})] && \forall\, i \in\ [1, \bar{n}_v] \\
t_{\mu(e)}[\sigma_i \setminus (\{v\})] &= \tilde{t}_{\mu(e)}[\tilde{\sigma}_i \setminus (\{\tilde{v}_1, \ldots, \tilde{v}_i\} \cup \{\tilde{v}^{(\mu(e))}\})] && \forall\, e \in\ \dot{\mathcal{E}}_v(Q) \\
\tilde{t}_{\mu(f)} &= t_{\mu(f)} && \forall\, f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)
\end{aligned}$$

Hence, the above tuples satisfy the rest of the join conditions in the query Q. Therefore, $Q(D)$ is true.

- Case $\dot{\mathcal{E}}_v(Q) = \emptyset$: By the semantics of IJQs, it holds that:

$$\begin{aligned}
&\tilde{t}_{\mu(\sigma_i)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(\sigma_i)}(\tilde{v}_i) = b_i && \forall\, i \in\ [1, \bar{n}_v - 1] \\
&\tilde{t}_{\mu(e)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(e)}(\tilde{v}_i) = b_{\bar{n}_v+1} && i = \bar{n}_v
\end{aligned}$$

By Definition 5.2.5, there exist tuples:

$$t_{\mu(\sigma_i)} \in R_{\mu(\sigma_i)}(\sigma_i) \text{ where } R_{\mu(\sigma_i)}(\sigma_i) \in D \qquad \forall\, i \in\ [1, \bar{n}_v]$$

$$t_{\mu(f)} \in R_{\mu(f)}(f) \qquad \forall\, f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)$$

such that:

$$b_1 \circ \cdots \circ b_i \in \mathsf{cp}_{\mathcal{I}}(t_{\mu(\sigma_i)}(v)) \qquad \forall\, i \in\ [1, \bar{n}_v - 1]$$

$$b_1 \circ \cdots \circ b_i = \mathsf{leaf}_{\mathcal{I}}(t_{\mu(e)}(v)) \qquad i = \bar{n}_v$$

Hence, the above tuples satisfy the intersection join on variable v in the query Q. Furthermore, we have:

$$t_{\mu(\sigma_i)}[\sigma_i \setminus (\{v\})] = \tilde{t}_{\mu(\sigma_i)}[\tilde{\sigma}_i \setminus (\{\tilde{v}_1, \dots, \tilde{v}_i\} \cup \{\tilde{v}^{(\mu(\sigma_i))}\})] \qquad \forall\, i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(f)} = t_{\mu(f)} \qquad \forall\, f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)$$

Hence, the same tuples satisfy the rest of the join conditions in the query Q. Therefore, $Q(D)$.

□

5.7.3 Proof of Proposition 5.2.1

Lemma 5.2.8 (One-step Forward Reduction Complexity). *The size of the query $\tilde{Q}^{(v)}$ which is constructed by Definition 5.2.1 is $\mathcal{O}(1)$. The size of each new relation in the database $\tilde{D}^{(v)}$ which is constructed by Definition 5.2.5 is the following.*

- *Case $\dot{\mathcal{E}}_v(Q) \neq \emptyset$*
  - $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^i |D|) \qquad \forall i \in [1, \bar{n}_v]$*;*

  - $|\tilde{R}_{\mu(e)}(\tilde{e})| = \mathcal{O}(|R_{\mu(e)}(e)| \cdot \log^{i-1}|D|)$ $\qquad \forall e \in \mathcal{E}_v(Q)$.

- *Case* $\dot{\mathcal{E}}_v(Q) = \emptyset$

  - $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^{i}|D|)$ $\qquad \forall i \in [1, \bar{n}_v - 1]$;

  - $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^{i-1}|D|)$ $\qquad i = \bar{n}_v$.

The construction time of each new relation is proportional to its size.

Proof. By Definition 5.2.1, the number of queries in the disjunction $\tilde{Q}^{(v)}$ is $\bar{n}_v!$, which is considered to be a constant because it depends only on the query size. Furthermore, the size of each conjunct only depends on the query size, hence, it is considered to be a constant as well. Therefore, $|\tilde{Q}^{(v)}| = \mathcal{O}(1)$.

The size and construction time of the relations in $\tilde{D}^{(v)}$ is analyzed next. Given an node $u \in V(\mathfrak{T}_{\mathcal{I}})$ and an integer i, let $\mathsf{break}(u, i) = \{(x_1, \ldots, x_i) \mid x_1 \circ \cdots \circ x_i = u\}$.

Claim 5.7.1. *It holds that* $|\mathsf{break}(u, i)| = \mathcal{O}(|u|^{i-1})$.

Proof. The number of i-tuples generated by splitting u in i non-empty substrings is:

$$\begin{aligned}
\binom{|u|-1}{i-1} &= \frac{(|u|-1)!}{(i-1)! \cdot (|u|-i)!} = \frac{1 \cdot 2 \cdot \cdots \cdot (|u|-i)}{(i-1)!} \\
&= \frac{(1 \cdot \cdots \cdot (|u|-i)) \cdot ((|u|-i+1) \cdot \cdots \cdot (|u|-1))}{(1 \cdot \cdots \cdot (|u|-i)) \cdot (i-1)!} \\
&= \frac{(|u|-i+1) \cdot \cdots \cdot (|u|-1)}{(i-1)!} \\
&\leq \frac{|u|^{i-1}}{(i-1)!} = \mathcal{O}(|u|^{i-1})
\end{aligned}$$

□

Since $u \in V(\mathfrak{T}_{\mathcal{I}})$, the bit-string u consists of $\mathcal{O}(\log|\mathcal{I}|) = \mathcal{O}(\log|D|)$ bits. Hence, $|\mathsf{break}(u, i)| = \mathcal{O}(\log^{i-1}|D|)$. Consider each case separately:

- Case $\dot{\mathcal{E}}_v(Q) \neq \emptyset$.

  - Let $i \in [1, \bar{n}_v]$. The relation $\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$ can be constructed by using the following procedure: $\forall t \in R_{\mu(\sigma_i)}(\sigma_i)\ \forall u \in \mathsf{cp}_{\mathcal{I}}(t(v))\ \forall (x_1, \ldots, x_i) \in \mathsf{break}(u, i)$, construct the tuple $\tilde{t}$ over the schema $\tilde{\sigma}_i$ (Equation 5.8) such that: $\tilde{t}[\tilde{\sigma}_i \setminus (\{\tilde{v}_1, \ldots, \tilde{v}_i\} \cup \{\tilde{v}^{(\sigma_i)}\})] = t[\sigma_i \setminus \{v\}]$, $\tilde{t}(\tilde{v}^{(\sigma_i)}) = t(v)$, and $\tilde{t}[\{\tilde{v}_1, \ldots, \tilde{v}_i\}] = (x_1, \ldots, x_i)$. Then, insert the tuple $\tilde{t}$ in the relation $\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$.

    By Property 3.3.7, it holds that $|\mathsf{cp}_{\mathcal{I}}(t(v))| = \mathcal{O}(\log|\mathcal{I}|) = \mathcal{O}(\log|D|)$. Furthermore, by Claim 5.7.1 it holds that $|\mathsf{break}(u, i)| = \mathcal{O}(\log^{i-1}|D|)$. Therefore, $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log|D| \cdot \log^{i-1}|D|) = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^{i}|D|)$. Moreover, the above justifies that its construction time is proportional to its size.

  - Let $e \in \dot{\mathcal{E}}_v(Q)$. The relation $\tilde{R}_{\mu(e)}(\tilde{e})$ can be constructed by using the following procedure: $\forall t \in R_{\mu(e)}(e)\ \forall (x_1, \ldots, x_{\bar{n}_u+1}) \in \mathsf{break}(u, i)$, where $u = \mathsf{leaf}_{\mathcal{I}}(t(v))$, construct the tuple $\tilde{t}$ over the schema $\tilde{e}$ (Equation 5.9) such that: $\tilde{t}[\tilde{e} \setminus (\{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v+1}\} \cup \{\tilde{v}^{(e)}\})] = t[e \setminus \{v\}]$, $\tilde{t}(\tilde{v}^{(e)}) = t(v)$, and $\tilde{t}[\{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v+1}\}] = (x_1, \ldots, x_{\bar{n}_v+1})$. Then, insert the tuple $\tilde{t}$ into $\tilde{R}_{\mu(e)}(\tilde{e})$.

    By Claim 5.7.1, it holds that $|\mathsf{break}(u, \bar{n}_v + 1)| = \mathcal{O}(\log^{\bar{n}_v}|D|)$. Therefore, $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^{\bar{n}_v}|D|)$. Moreover, the above justifies that its construction time is proportional to its size.

- Case $\dot{\mathcal{E}}_v(Q) = \emptyset$.

  - Let $i \in [1, \bar{n}_v - 1]$. The relation $\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$ can be constructed using the following procedure: $\forall t \in R_{\mu(\sigma_i)}(\sigma_i)\ \forall u \in \mathsf{cp}_{\mathcal{I}}(t(v))\ \forall (x_1, \ldots, x_i) \in \mathsf{break}(u, i)$ construct the tuple $\tilde{t}$ over the schema $\tilde{\sigma}_i$ (Equation 5.8), such that $\tilde{t}[\tilde{\sigma}_i \setminus (\{\tilde{v}_1, \ldots, \tilde{v}_i\} \cup \{\tilde{v}^{(\sigma_i)}\})] = t[\sigma_i \setminus \{v\}]$, $\tilde{t}(\tilde{v}^{(\sigma_i)}) = t(v)$ and $\tilde{t}(\{\tilde{v}_1, \ldots, \tilde{v}_{\bar{n}_v-1}\}) = (x_1, \ldots, x_{\bar{n}_v-1})$. Then, insert the tuple $\tilde{t}$ into $\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$.

    By Property 3.3.7, it holds that $|\mathsf{cp}_{\mathcal{I}}(t(v))| = \mathcal{O}(\log|\mathcal{I}|) = \mathcal{O}(\log|D|)$. Furthermore, by Claim 5.7.1, it holds that $|\mathsf{break}(u, i)| = \mathcal{O}(\log^{i-1}|D|)$.

Therefore, $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log|D| \cdot \log^{i-1}|D|) = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^{i}|D|)$. Moreover, the above justifies that its construction time is proportional to its size.

– Let $i = \bar{n}_u$. The relation $\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$ is constructed using the following procedure: for each tuple in $t \in R_{\mu(\sigma_i)}(\sigma_i)$, for each $(x_1, \dots, x_i) \in \mathsf{break}(u, i)$, where $u = \mathsf{leaf}_{\mathcal{I}}(t(v))$, construct the tuple $\tilde{t}$ over the schema $\tilde{\sigma}_i$ (Equation 5.8) such that: $\tilde{t}[\tilde{\sigma}_i \setminus (\{\tilde{v}_1, \dots, \tilde{v}_i\} \cup \{\tilde{v}^{(\sigma_i)}\})] = t[\sigma_i \setminus \{v\}]$, $t(\tilde{v}^{(\sigma_i)}) = t(v)$ and $\tilde{t}[\tilde{v}_1, \dots, \tilde{v}_i] = (x_1, \dots, x_i)$. Then, insert the tuple $\tilde{t}$ into $\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$.

  By Claim 5.7.1, it holds that $|\mathsf{break}(u, i)| = \mathcal{O}(\log^{i-1}|D|)$. Therefore, $|\tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)| = \mathcal{O}(|R_{\mu(\sigma_i)}(\sigma_i)| \cdot \log^{i-1}|D|)$. Moreover, the above justifies that its construction time is proportional to its size.

□

5.7.4 Proof of Proposition 5.2.12

Proposition 5.2.12 (Correctness). *Consider a Boolean IJQ Q and a database D. It holds that $Q(D)$ is equivalent to $\tilde{Q}(\tilde{D})$, where $\tilde{Q}$ and $\tilde{D}$ are defined by Definition 5.2.9.*

Proof. Without loss of generality, assume that the IJQ Q includes the join variables $\mathsf{vars}(Q) = \{v_1, \dots, v_n\}$, and that Algorithm 4, iterates over them in the listed order. This is a proof by induction. Let $P(j)$ denote the statement:

$$Q(D) \equiv \left(\bigvee_{Q \in \mathbf{Q}} Q(\tilde{D}) \right) \qquad \text{after the } j\text{-th iteration}$$

Base case. We prove that $P(1)$ is true. The statement $P(1)$ is equivalent to the statement:

$$Q(D) \equiv \left(\bigvee_{Q \in \mathbf{Q}} Q(\tilde{D}) \right) \qquad \text{after the 1-st iteration}$$

Upon the beginning of the 1-st iteration, it holds that $\mathbf{Q}_0 = \{Q\}, \mathbf{Q} = \emptyset$. Hence, after the 1-st iteration it holds that:

$$\mathbf{Q} = \bigcup_{Q \in \mathbf{Q}_0} \left(\bigcup_{\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_{v_1}(Q))} \{\tilde{Q}^{(v_1,\sigma)}\} \right) = \bigcup_{Q \in \{Q\}} \left(\bigcup_{\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_{v_1}(Q))} \{\tilde{Q}^{(v_1,\sigma)}\} \right)$$
$$= \bigcup_{\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_{v_1}(Q))} \{\tilde{Q}^{(v_1,\sigma)}\}$$

where $\tilde{Q}^{(v_1,\sigma)}$ is defined by Definition 5.2.1, and $\tilde{D} = \tilde{D}^{(v_1)}$, where $\tilde{D}^{(v_1)}$ is defined by Definition 5.2.5. Hence, it holds that:

$$\bigvee_{Q \in \mathbf{Q}} Q(\tilde{D}) \equiv \left(\bigvee_{\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_{v_1}(Q))} \tilde{Q}^{(v_1,\sigma)}(\tilde{D}^{(v_1)}) \right) \equiv \tilde{Q}^{(v_1)}(\tilde{D}^{(v_1)})$$

The second equality is due to Definition 5.2.1. By Lemma 5.2.7, it holds that $\tilde{Q}^{(v_1)}(\tilde{D}^{(v_1)}) \equiv Q(D)$. Hence, $P(1)$ is true.

Inductive step. We prove that:

$$(P(j) \implies P(j+1)) \qquad\qquad \forall\, j \in [n-1]$$

Assume $P(j)$ is true. Let $\underline{\mathbf{Q}}_0$ denote $\mathbf{Q}_0$, $\underline{\mathbf{Q}}$ denote $\mathbf{Q}$ and $\underline{D}$ denote D upon the beginning of the j-th iteration. After the j-th iteration, we have:

$$\underline{\mathbf{Q}} = \bigcup_{\underline{Q} \in \underline{\mathbf{Q}}_0} \left(\bigcup_{\underline{\sigma} \in \mathsf{perms}(\underline{\bar{\mathcal{E}}}_{v_j}(\underline{Q}))} \{\underline{\tilde{Q}}^{(v_j,\underline{\sigma})}\} \right)$$

where $\underline{\tilde{Q}}^{(v_j,\underline{\sigma})}$ is defined by Definition 5.2.1, and $\underline{\mathcal{E}}_{v_j}(\underline{Q})$ denotes the set of hyperedges of the hypergraph associated to $\underline{Q}$. We also have $\underline{\tilde{D}} = \underline{\tilde{D}}^{(v_j)}$, where $\underline{\tilde{D}}^{(v_j)}$ is defined by Definition 5.2.5. Since $P(j)$ is true, we have:

$$\left(\bigvee_{\underline{Q}\in\underline{\mathbf{Q}}} \underline{Q}(\tilde{\underline{D}})\right) \equiv Q(D) \tag{5.14}$$

Upon the $j+1$-th iteration, we have $\mathbf{Q}_0 = \underline{\mathbf{Q}}$, $\mathbf{Q} = \emptyset$. Hence, after the $j+1$-th iteration it holds that:

$$\begin{aligned}\mathbf{Q} &= \bigcup_{Q\in\mathbf{Q}_0}\left(\bigcup_{\sigma\in\mathsf{perms}(\bar{\mathcal{E}}_{v_{j+1}}(Q))} \{\tilde{Q}^{(v_{j+1},\sigma)}\}\right)\\ &= \bigcup_{\underline{Q}\in\underline{\mathbf{Q}}_0}\left[\bigcup_{\underline{\sigma}\in\mathsf{perms}(\bar{\underline{\mathcal{E}}}_{v_j}(Q))}\left(\bigcup_{\sigma\in\mathsf{perms}(\bar{\mathcal{E}}_{v_{j+1}}(Q))} \{\tilde{J}^{(v_{j+1},\sigma)}\}\right)\right]\end{aligned} \tag{5.15}$$

where $J = \tilde{\underline{Q}}^{(v_j,\underline{\sigma})}$, and $\tilde{J}^{(v_{j+1},\sigma)}$ is defined by Definition 5.2.1. We also have $\tilde{D} = \tilde{D}^{(v_{j+1})}$, where $\tilde{D}^{(v_{j+1})}$ is defined by Definition 5.2.5. Hence, by Equation (5.15) it holds that:

$$\begin{aligned}\bigvee_{Q\in\mathbf{Q}} Q(D) &\equiv \bigvee_{\underline{Q}\in\underline{\mathbf{Q}}_0}\left[\bigvee_{\underline{\sigma}\in\mathsf{perms}(\bar{\underline{\mathcal{E}}}_{v_j}(Q))}\left(\bigvee_{\sigma\in\mathsf{perms}(\bar{\mathcal{E}}_{v_{j+1}}(Q))} \tilde{J}^{(v_{j+1},\sigma)}(\tilde{D}^{(v_{j+1})})\right)\right]\\ &\equiv \bigvee_{\underline{Q}\in\underline{\mathbf{Q}}_{\mathbf{0}}}\left(\bigvee_{\underline{\sigma}\in\mathsf{perms}(\bar{\underline{\mathcal{E}}}_{v_j}(Q))} \tilde{J}^{(v_{j+1})}(\tilde{D}^{(v_{j+1})})\right)\end{aligned}$$

The second equivalence is obtained by Definition 5.2.1. Because it holds that $J = \tilde{\underline{Q}}^{(v_j,\underline{\sigma})}$, by Lemma 5.2.7, it also holds that $\tilde{J}^{(v_{j+1})}(\tilde{D}^{(v_{j+1})}) \equiv \tilde{\underline{Q}}^{(v_j,\underline{\sigma})}(\tilde{\underline{D}}^{(v_j)})$. Hence, we have:

$$\bigvee_{Q\in\mathbf{Q}} Q(D) \equiv \bigvee_{\underline{Q}\in\underline{\mathbf{Q}}_0}\left(\bigvee_{\underline{\sigma}\in\mathsf{perms}(\bar{\underline{\mathcal{E}}}_{v_j}(Q))} \tilde{\underline{Q}}^{(v_j,\underline{\sigma})}(\tilde{\underline{D}}^{(v_j)})\right) \equiv \left(\bigvee_{\underline{Q}\in\underline{\mathbf{Q}}} \underline{Q}(\tilde{\underline{D}})\right) \equiv Q(D)$$

The third equivalence is due to Equation (5.14). Therefore, the statement $P(j+1)$

is true.

Conclusion. Since both the base case and the inductive step have been proven true, by induction, the statement $P(n)$ is true. $\square$

5.7.5 Proof of Proposition 5.3.2

Definition 5.7.2 (Backward Mapping)**.** *Consider a finite set of bit-strings $\mathcal{B}$. Let m be the length of the bit-string in $\mathcal{B}$ with maximum length. Define a function F that maps each bit-string in $\bigcup_{i=0}^{m}\{0,1\}^i$ to an interval $[x, y)$ where $x, y \in [0, 1]$, as follows: $F(\varepsilon) = [0, 1)$, $F(\text{“0”}) = [0, 1/2)$, $F(\text{“1”}) = [1/2, 1)$, $F(\text{“00”}) = [0, 1/4)$ and so on. Note that for any given bit-string b, the intervals $F(b \circ \text{“0”})$ and $F(b \circ \text{“1”})$ correspond to the first and second half of the interval $F(b)$ respectively.*

Property 5.7.3 (Backward Mapping Property)**.** *Consider a finite set of bit-strings $\mathcal{B}$ and, the function F (Definition 5.7.2). Given two bit-strings $u, v \in \bigcup_{i=0}^{m}\{0,1\}^i$, it holds that:*

$$(u \preceq v) \vee (v \preceq u) \iff (F(u) \cap F(v)) \neq \emptyset,$$

where $a \preceq b$ means that a is a prefix of b.

Proof. “ $\Longrightarrow$ ”: Assume that u is a prefix of v. By Definition 5.7.2, $F(u)$ and $F(v)$ are two right-open intervals. Since u is prefix of v it holds that $F(u) \supseteq F(v)$. Therefore, $(F(u) \cap F(v)) \neq \emptyset$. Assume that v is a prefix of u. Using similar arguments, one can conclude that $(F(u) \cap F(v)) \neq \emptyset$ in this case as well.

“ $\Longleftarrow$ ”: Assume that $(F(u) \cap F(v)) \neq \emptyset$. By the construction of the function F (see Definition 5.7.2), it holds that for any two intervals in the co-domain of F are either contained in each other, or they are disjoint. Therefore, we have that $F(u) \supseteq F(v)$ or $F(v) \supseteq F(u)$. Hence, by Definition 5.7.2 it holds that $u \preceq v$ or $v \preceq u$. $\square$

Similar to the forward reduction, the backward reduction has as a building block, the one-step backward reduction. The one-step backward reduction is defined as follows.

Definition 5.7.4 (One-step Backward Reduction). *Consider a Boolean IJQ Q, a variable $v \in \mathcal{V}$ and a permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$. Let $\tilde{Q}^{(v,\sigma)}$ be the query resulting from the one-step query rewriting (Definition 5.2.1), and let $\tilde{B}$ be a database that matches the structure of the latter query. Define a database B as follows:*

- *Case $\dot{\mathcal{E}}_v(Q) \neq \emptyset$*
  - $\forall i \in [1, \bar{n}_v]$ $\forall \tilde{t} \in \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$, *construct a tuple $t \in R_{\mu(\sigma_i)}(\sigma_i)$ such that:*
    - $t[\sigma_i \setminus \{v\}] = \tilde{t}[\tilde{\sigma}_i \setminus \{v_1, \ldots, v_i\}]$; *and*
    - $t(v) = F(\tilde{t}(v_1) \circ \cdots \circ \tilde{t}(v_i))$
  - $\forall \tilde{e} \in \dot{\mathcal{E}}_v(Q)$ $\forall \tilde{t} \in \tilde{R}_{\mu(e)}(\tilde{e})$, *construct a tuple $t \in R_{\mu(e)}(e)$ such that:*
    - $t[e \setminus \{v\}] = \tilde{t}[\tilde{e} \setminus \{v_1, \ldots, v_{\bar{n}_v+1}\}]$; *and*
    - $t(v) = F(\tilde{t}(v_1) \circ \cdots \circ \tilde{t}(v_{\bar{n}_v+1})).\mathsf{start}$
- *Case $\dot{\mathcal{E}}_v(Q) = \emptyset$*
  - $\forall i \in [1, \bar{n}_v]$ $\forall \tilde{t} \in \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i)$, *construct a tuple $t \in R_{\sigma_i}(\sigma_i)$ such that:*
    - $t[\sigma_i \setminus \{v\}] = \tilde{t}[\tilde{\sigma}_i \setminus \{v_1, \ldots, v_i\}]$; *and*
    - $t(v) = F(\tilde{t}(v_1) \circ \cdots \circ \tilde{t}(v_i))$

The rest of the relations in $\tilde{B}$ are copied directly to the database B.

The correctness of the one-step backward reduction is proven as follows.

Lemma 5.7.5 (One-step Backward Reduction Correctness). *Consider a Boolean IJQ Q. Let $v \in \mathcal{V}(Q)$ and $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$. Let $\tilde{Q}^{(v,\sigma)}$ be the query resulting from the one-step query rewriting (Definition 5.2.1), and let $\tilde{B}$ be a database that matches the structure of the latter query. It holds that $Q(B)$ is equivalent to $\tilde{Q}^{(v,\sigma)}(\tilde{B})$.*

Proof. " $\Longrightarrow$ ": Assume that there exists a tuple t that satisfies $Q(B)$. By the semantics of IJQs we have that there exists a tuple $(t_{\mu(e)})_{e\in\mathcal{E}(Q)} \in \prod_{e\in\mathcal{E}(Q)} R_e(e)$, that satisfy all the join conditions in the query Q.

- Case $\dot{\mathcal{E}}_v(Q) \neq \emptyset$: By Lemma 5.1.1 it holds that $t_{\mu(e)} = \xi$ for each $e \in \dot{\mathcal{E}}_v(Q)$ and $\xi \in \left(\bigcap_{e\in\bar{\mathcal{E}}_v(Q)} t_{\mu(e)}\right)$. By Property 5.7.3, there exists a tuple of bit-strings $(b_1, \dots, b_{\bar{n}_v+1})$ such that:

$$b_1 \circ \cdots \circ b_i = F(t_{\mu(\sigma_i)}(v)) \qquad \forall\, i \in\ [1, \bar{n}_v]$$

$$b_1 \circ \cdots \circ b_{\bar{n}_v+1} = F(t_{\mu(e)}(v)).\mathsf{start} \qquad \forall\, e \in\ \dot{\mathcal{E}}_v(Q)$$

  By Definition 5.7.4 there exists a set of tuples:

$$\tilde{t}_{\mu(\sigma_i)} \in \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i), \text{ where } \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i) \in \tilde{B} \qquad \forall i \in [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(e)} \in \tilde{R}_{\mu(e)}(\tilde{e}), \text{ where } \tilde{R}_{\mu(e)}(\tilde{e}) \in \tilde{B} \qquad \forall\, e \in\ \dot{\mathcal{E}}_v(Q)$$

$$\tilde{t}_{\mu(f)} \in \tilde{R}_{\mu(f)}(f), \text{ where } \tilde{R}_{\mu(f)}(f) \in \tilde{B} \qquad \forall\, f \in\ \mathcal{E}\ \setminus\ \mathcal{E}_v(Q)$$

  where $\tilde{\sigma}_i$ and $\tilde{e}$ are defined in Equations (5.8) and (5.9) respectively, such that:

$$\tilde{t}_{\mu(\sigma_i)}(\tilde{v}_1) = b_1, \dots, \tilde{t}_{\mu(\sigma_i)}(\tilde{v}_i) = b_i \qquad \forall\, i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(e)}(\tilde{v}_1) = b_1, \dots, \tilde{t}_{\mu(e)}(\tilde{v}_{\bar{n}_v+1}) = b_{\bar{n}_v+1} \qquad \forall\, e \in\ \dot{\mathcal{E}}_v(Q)$$

  Hence, the above tuples satisfy the equality joins on the variables $\tilde{v}_1, \dots, \tilde{v}_{\bar{n}_v+1}$ in the query $\tilde{Q}^{(v,\sigma)}$. Furthermore, we have:

$$\tilde{t}_{\mu(\sigma_i)}[\tilde{\sigma}_i \setminus \{\tilde{v}_1, \dots, \tilde{v}_i\}] = t_{\mu(\sigma_i)}[\sigma_i \setminus \{v\}] \qquad \forall\, i \in\ [1, \bar{n}_v]$$

$$\tilde{t}_{\mu(e)}[\tilde{e} \setminus \{\tilde{v}_1, \dots, \tilde{v}_{\bar{n}_v+1}\}] = t_{\mu(e)}[e \setminus \{v\}] \qquad \forall\, e \in\ \dot{\mathcal{E}}_v(Q)$$

$$\tilde{t}_{\mu(f)} = t_{\mu(f)} \qquad \forall\, f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)$$

Hence, the same tuples satisfy the rest of the join conditions in the query $\tilde{Q}^{(v,\sigma)}$. Therefore, there exists $\tilde{t}$ that satisfies the query $\tilde{Q}^{(v,\sigma)}(\tilde{B})$.

- Case $\bar{\mathcal{E}}_v(Q) = \emptyset$: By Lemma 5.1.1 it holds that $\left(\bigcap_{e \in \bar{\mathcal{E}}_v(Q)} t_{\mu(e)}\right) \neq \emptyset$. By Property 5.7.3, there exists a tuple of bit-strings $(b_1, \dots, b_{\bar{n}_v})$ such that:

  $$b_1 \circ \cdots \circ b_i = F(t_{\mu(\sigma_i)}(v)) \qquad \forall\ i \in\ [1, \bar{n}_v]$$

  By Definition 5.7.4 there exists a set of tuples:

  $$\begin{aligned} \tilde{t}_{\mu(\sigma_i)} \in \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i), \text{ where } \tilde{R}_{\mu(\sigma_i)}(\tilde{\sigma}_i) \in \tilde{B} && \forall i \in [1, \bar{n}_v] \\ \tilde{t}_{\mu(f)} \in \tilde{R}_{\mu(f)}(f), \text{ where } \tilde{R}_{\mu(f)}(f) \in \tilde{B} && \forall f \in \mathcal{E}(Q) \setminus \mathcal{E}_v(Q) \end{aligned}$$

  where $\tilde{\sigma}_i$ and $\tilde{e}$ are defined in Equations (5.8) and (5.9) respectively, such that:

  $$\tilde{t}_{\mu(\sigma_i)}(\tilde{v}_1) = b_1, \dots, \tilde{t}_{\mu(\sigma_i)}(\tilde{v}_i) = b_i \qquad \forall i \in [1, \bar{n}_v]$$

  Hence, the above tuples satisfy the equality joins on the variables $\tilde{v}_1, \dots, \tilde{v}_{\bar{n}_v+1}$ in the query $\tilde{Q}^{(v,\sigma)}$. Furthermore, we have:

  $$\begin{aligned} \tilde{t}_{\mu(\sigma_i)}[\tilde{\sigma}_i \setminus \{\tilde{v}_1, \dots, \tilde{v}_i\}] = t_{\mu(\sigma_i)}[\sigma_i \setminus \{v\}] && \forall i \in [1, \bar{n}_v] \\ \tilde{t}_{\mu(f)} = t_{\mu(f)} && \forall\ f \in\ \mathcal{E}\ \setminus\ \mathcal{E}_v(Q) \end{aligned}$$

  Hence, the same tuples satisfy the rest of the join conditions in the query $\tilde{Q}^{(v,\sigma)}$. Therefore, there exists $\tilde{t}$ that satisfies the query $\tilde{Q}^{(v,\sigma)}(\tilde{B})$.

" $\Longleftarrow$ ": Assume that there exists a permutation $\sigma \in \mathsf{perms}(\bar{\mathcal{E}}_v(Q))$ and tuples $(\tilde{t}_{\mu(e)})_{e \in \mathcal{E}(Q)} \in \prod_{e \in \mathcal{E}} \tilde{R}_{\mu(e)}(\tilde{e}^{(v,\sigma)})$ that satisfy all the join conditions in the query $\tilde{Q}^{(v,\sigma)}$.

- Case $\mathcal{E}_v(Q) \neq \emptyset$: By the semantics of IJQs, it holds that:

$$\tilde{t}_{\mu(\sigma_i)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(\sigma_i)}(\tilde{v}_i) = b_i \qquad \forall\ i \in\ [1, \bar{n}_v]$$
$$\tilde{t}_{\mu(e)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(e)}(\tilde{v}_{\bar{n}_v+1}) = b_{\bar{n}_v+1} \qquad \forall\ e \in\ \dot{\mathcal{E}}_v(Q)$$

By Definition 5.7.4, there exist tuples:

$$t_{\mu(\sigma_i)} \in R_{\mu(\sigma_i)}(\sigma_i) \text{ where } R_{\mu(\sigma_i)}(\sigma_i) \in B \qquad \forall\ i \in\ [1, \bar{n}_v]$$
$$t_{\mu(e)} \in R_{\mu(e)}(e) \text{ where } R_{\mu(\sigma_i)}(e) \in B \qquad \forall\ e \in\ \dot{\mathcal{E}}_v(Q)$$
$$t_{\mu(f)} \in R_{\mu(f)}(f) \text{ where } R_{\mu(f)}(f) \in B \qquad \forall\ e \in\ \mathcal{E} \setminus \mathcal{E}_v(Q)$$

such that:

$$b_1 \circ \cdots \circ b_i = F(t_{\mu(\sigma_i)}(v)) \qquad \forall\ i \in\ [1, \bar{n}_v]$$
$$b_1 \circ \cdots \circ b_{\bar{n}_v+1} = F(t_{\mu(e)}(v)).\mathsf{start} \qquad \forall\ e \in\ \dot{\mathcal{E}}(Q)$$

Hence, by Lemma 5.1.7 the above tuples satisfy the intersection join on variable v in the query Q. Furthermore, we have:

$$t_{\mu(\sigma_i)}[\sigma_i \setminus (\{v\})] = \tilde{t}_{\mu(\sigma_i)}[\tilde{\sigma}_i \setminus \{\tilde{v}_1, \ldots, \tilde{v}_i\}] \qquad \forall\ i \in\ [1, \bar{n}_v]$$
$$t_{\mu(e)}[\sigma_i \setminus (\{v\})] = \tilde{t}_{\mu(e)}[\tilde{\sigma}_i \setminus \{\tilde{v}_1, \ldots, \tilde{v}_i\}] \qquad \forall\ e \in\ \dot{\mathcal{E}}_v(Q)$$
$$t_{\mu(f)} = \tilde{t}_{\mu(f)} \qquad \forall\ f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)$$

Hence, there exists a tuple $\tilde{t}$ that satisfies the query Q.

- Case $\mathcal{E}_v(Q) = \emptyset$: By the semantics of IJQs, it holds that:

$$\tilde{t}_{\mu(\sigma_i)}(\tilde{v}_1) = b_1, \ldots, \tilde{t}_{\mu(\sigma_i)}(\tilde{v}_i) = b_i \qquad \forall i \in [1, \bar{n}_v]$$

By Definition 5.7.4, there exist tuples:

$$t_{\mu(\sigma_i)} \in R_{\mu(\sigma_i)}(\sigma_i) \text{ where } R_{\mu(\sigma_i)}(\sigma_i) \in B \qquad \forall i \in [1, \bar{n}_v]$$

$$t_{\mu(f)} \in R_{\mu(f)}(f) \text{ where } R_{\mu(f)}(f) \in B \qquad \forall\, e \in\ \mathcal{E}(Q) \setminus \mathcal{E}_v(Q)$$

such that:

$$b_1 \circ \cdots \circ b_i = F(t_{\mu(\sigma_i)}(v)) \qquad \forall\ i \in\ [1, \bar{n}_v]$$

Hence, by Lemma 5.1.7 the above tuples satisfy the intersection join on variable v in the query Q. Furthermore, we have:

$$t_{\mu(\sigma_i)}[\sigma_i \setminus (\{v\})] = \tilde{t}_{\mu(\sigma_i)}[\tilde{\sigma}_i \setminus \{\tilde{v}_1, \ldots, \tilde{v}_i\}] \qquad \forall\ i \in\ [1, \bar{n}_v]$$

$$t_{\mu(f)} = \tilde{t}_{\mu(f)} \qquad \forall\ f \in\ \mathcal{E}(Q)\ \setminus\ \mathcal{E}_v(Q)$$

Hence, there exists a tuple $\tilde{t}$ that satisfies the query Q.

□

Proposition 5.3.2 (Backward Reduction Correctness). *Consider a Boolean IJQ Q, a Boolean CQ $\tilde{Q}^{(i)}$ in the forward reduction of Q and a database $\tilde{B}$ that matches the structure of $\tilde{Q}^{(i)}$. There exists a bijection that maps each tuple from $\tilde{B}$ to a tuple from a database B, that matches the structure of Q, and $\tilde{Q}^{(i)}(\tilde{B})$ is equivalent to $Q(B)$.*

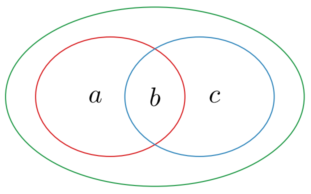
Proof. By repeatedly applying the one-step backward reduction (Definition 5.7.4) on the database $\tilde{B}$, one can construct a database B with scalar values only, such that $Q(B)$ is equivalent to $\tilde{Q}(\tilde{B})$. Notice that the reduction implies the existence of a bijection g from the tuples in $\tilde{B}$ to the tuples in B. Hence, we also have $|B| = |\tilde{B}|$. □

Chapter 6

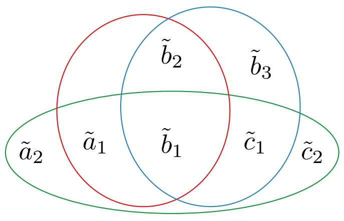
Acyclicity

Hypergraphs can describe conjunctive Queries (CQs) and their corresponding database schemas. The CQs that are associated with α-acyclic hypergraphs have some computation advantages. One of the advantages is that, in data complexity, an α-acyclic Boolean CQ can be computed in linear time [58]. Consider the class of Boolean IJQs where all the variables in all of their occurrences are interval variables. Such queries are denoted as $\overline{\text{IJQ}}s$ (see Chapter 2). Because all the variables, in all of their occurrences have the same type, $\overline{\text{IJQ}}s$ are described by hypergraphs as well. This Chapter introduces a new notion of acyclicity, namely ι-acyclcity, that characterizes the Boolean $\overline{\text{IJQ}}s$ that can be computed in quasilinear time; an $\overline{\text{IJQ}}$ is ι-acyclic if and only if its associated hypergraph includes no Berge cycle of length greater than or equal to 3.

The findings reported by this Chapter are those reported in [36] and its full version [35]. This Chapter extends the work mentioned above by introducing a way to determine ι-acyclicity in polynomial in the size of the query time (Section 6.3).



(a) $\mathcal{H} = (\mathcal{V}, \mathcal{E})$



(b) $(\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})) \in \tilde{\mathbf{H}}$

Figure 6.1: Figure 6.1a illustrates a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ corresponding to an IJQ. Figure 6.1a illustrates a hypergraph which is a member of $\tilde{\mathbf{H}}$.

Organization

Section 6.1 defines ι-acylicity and introduces a syntactic characterization of ι-acyclicity. Section 6.2 shows that it is unlikely that a non-ι-acyclic Boolean $\overline{\text{IJQ}}$ can be computed in quasilinear time. Section 6.3 shows that ι-acyclicity can be checked in polynomial in the size of query time. Section 6.4 discusses the results.

Notations and Terminology

Consider Boolean $\overline{\text{IJQ}}$ Q with associated hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Let $\tilde{\mathbf{H}}$ denote the set of hypergraphs corresponding to the CQ conjuncts in $\tilde{Q}$, which is the disjunction of CQs generated by the forward reduction (Definition 5.2.9). The vertices in $\mathcal{V}$ are denoted by lowercase letters (e.g. u). Let $\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ be any member of $\tilde{\mathbf{H}}$. The vertices in $\tilde{\mathcal{V}}$ are denoted by lower case, possibly subscripted letters with tilde (e.g. $\tilde{u}, \tilde{u}_i$).

6.1 Iota Acyclicity

There is a simple way to tell if a Boolean $\overline{\text{IJQ}}$ Q can be computed in quasilinear time: take the Boolean CQ conjuncts in the disjunction $\tilde{Q}$, which is constructed by the forward reduction of Q (Definition 5.2.9). If all of them are α-acyclic, then Q can be computed in quasilinear time [58]. Notice that the set $\tilde{\mathbf{H}}$ consists of the corresponding hypergraphs; hence, one can equivalently say that, if every hypergraph in $\tilde{\mathbf{H}}$ is α-acyclic, then Q can be computed in quasilinear time.

6.1.1 Definition

Definition 6.1.1 (Definition of Iota Acyclicity). *A hypergraph $\mathcal{H}$ is ι-acyclic if and only if every hypergraph in $\tilde{\mathbf{H}}$ is α-acyclic.*

To check if $\mathcal{H}$ is ι-acyclic, one has to compute the forward reduction of the input query in advance and then, check α-acyclicity on each hypergraph in the froward reduction. The size of $\tilde{\mathbf{H}}$ is independent of the input database and only depends on the size of the query.

6.1.2 Syntactic Characterization

Theorem 6.1.2 presents a syntactic characterisation for ι-acyclicity. This characterisation shows that ι-acyclcity depends on the structure of $\mathcal{H}$, therefore, there is no need to compute the forward reduction in advance.

Theorem 6.1.2 (Iota Acyclicity Characterisation). *A hypergraph is ι-acyclic if and only if it has no Berge cycle of length greater than or equal to* 3.

The notion of a Berge cycle is defined in Definition 3.1.11. The minor Berge cycle that makes a hypergraph $\mathcal{H}$ not ι-acyclic has length three. It is a sequence $(e^1, v^1, e^2, v^2, e^3, v^3, e^4)$, where v^1, v^2, v^3 are distinct vertices in the hypergraph, e^1, e^2, e^3 are distinct hyperedges in the hypergraph, $e^4 = e^1$, and $v^i \in e^i \cap e^{i+1}$ for $1 \leq i \leq 3$.

The notion of ι-acyclicity is between γ- and Berge-acyclicity in the hierarchy of the known notions of acyclicity for hypergraphs [25, 16], cf. Figure 1.2.

Corollary 6.1.3 (Iota Implies Gamma and is Implied by Berge). *The class of ι-acyclic hypergraphs is:*

- *strict superset of the class of Berge-acyclic hypergraphs; and*

- *strict subset of the class of γ-acyclic hypergraphs.*

Proof. The statement that Berge-acyclicity strictly implies ι-acyclicity follows immediately from Theorem 6.1.2.

Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Next, it is proven that if $\mathcal{H}$ is ι-acyclic then $\mathcal{H}$ is γ-acyclic. Assume for a contradiction that $\mathcal{H}$ is not γ-acyclic. By Definition 3.1.10, either $\mathcal{H}$ is not cycle-free, or there exist three distinct vertices $x, y, z \in \mathcal{V}$ such that $\{\{x,y\},\{y,z\},\{x,y,z\}\} \subseteq \mathcal{E}[\{x,y,z\}]$. In the former case, by Definition 3.1.5, there exists a subset $S = \{v^1, \ldots, v^n\} \subseteq \mathcal{V}$ with $n \geq 3$ such that: $\mathcal{M}(\mathcal{E}[S]) = \{\{v^i, v^{i+1}\} \mid 1 \leq i < n\} \cup \{\{v^n, v^1\}\}$. Hence, the hypergraph has a Berge cycle $(e_1, v_1, \ldots, v_{n-1}, e_n, v_n, e_1)$, which has a length greater than or equal to 3. In the latter case, the hypergraph $\mathcal{H}$ has a Berge cycle $(e_1, x, e_2, y, e_3, z, e_1)$, which has a length of 3. Therefore, the hypergraph $\mathcal{H}$ is not ι-acyclic in both cases. Contradiction.

To confirm the strictness of the inclusion, consider the hypergraph with hyperedges: $e_1 = \{x,y,z\}, e_2 = \{x,y,z\}, e_3 = \{x,y,z\}$. This hypergraph is not ι-acyclic, since it contains a Berge cycle $(e_1, x, e_2, y, e_3, z, e_1)$ of length 3, but it is γ-acyclic, since it is cycle-free, and the three vertices x, y, z do not satisfy the condition $\{\{x,y\},\{y,z\},\{x,y,z\}\} \subseteq \mathcal{E}[\{x,y,z\}]$ (Definition 3.1.10). $\square$

A result that immediately follows from Corollary 6.1.3 is the following.

Corollary 6.1.4 (Iota Implies Alpha)**.** *The class of ι-acyclic hypergraphs is a strict subset of the class of α-acyclic hypergraphs.*

Example 6.1.5. *Consider the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ depicted in Figure 6.1a. This hypergraph includes the Berge cycle (a, e_1, b, e_2, c, e_3), assuming that $e_1 = \{a,b\}, e_2 = \{a,b,c\}$ and $e_3 = \{b,c\}$. Since the Berge cycle is of length 3, the hypergraph $\mathcal{H}$ is not ι-acyclic. The hypergraph $\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$, depicted in Figure 6.1b is a member of the set $\tilde{\mathbf{H}}$ (Definition 5.2.9). Notice that this hypergraph is not α-acyclic, and there*

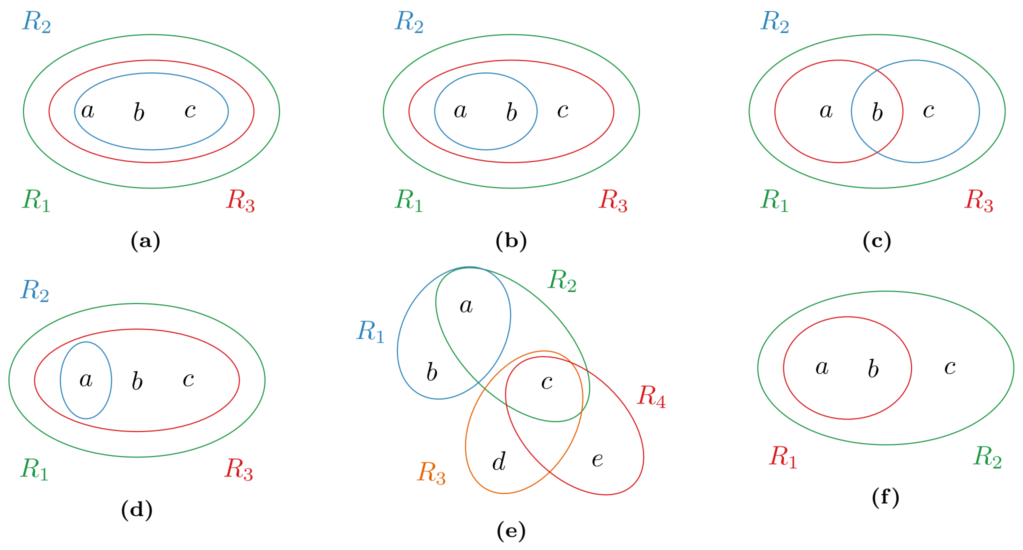


Figure 6.2: Example hypergraphs. Hypergraphs 6.2a - 6.2c are α-acyclic but not ι-acyclic. Hypergraphs 6.2d - 6.2f are ι-acyclic.

are several ways to see that. The first one is to try to apply the GYO reduction to it (Definition 3.1.6); the procedure will get stuck in the hypergraph with hyperedges $\{\{\tilde{a}_1, \tilde{b}_1, \tilde{b}_2\}, \{\tilde{a}_1, \tilde{b}_1, \tilde{c}_1\}, \{\tilde{b}_1, \tilde{b}_2, \tilde{c}_1\}\}$*. The second one is to check conformity and cycle-freedom properties (Definition 3.1.8). Consider the subset* $\{\tilde{a}_1, \tilde{b}_2, \tilde{c}_1\}$*. We have that* $\tilde{\mathcal{E}}[\{\tilde{a}_1, \tilde{b}_2, \tilde{c}_1\}] = \{\{\tilde{a}_1, \tilde{b}_2\}, \{\tilde{a}_1, \tilde{c}_1\}, \{\tilde{b}_2, \tilde{c}_1\}\}$*, which means that the hypergraph is neither conformal nor cycle-free. Hence, it is not* α*-acyclic.*

Example 6.1.6. *According to Corollary 6.1.3, the class of* ι*-acyclic hypergraphs is a strict subset of the class of* γ*-hypergraphs. Figure 6.2 depicts six* α*-acyclic hypergraphs. The hypergraph in Figure 6.2c is* α*-acyclic but not* γ*-acyclic. The hypergraphs in Figures 6.2a- 6.2b are* γ*-acyclic but not* ι*-acyclic. The hypergraphs in Figures 6.2d-6.2f are* ι*-acyclic.*

6.2 Dichotomy

This section shows that the Boolean IJQs which are not ι-acyclic are at least as hard as the Triangle CQ, which cannot be solved in linear time, unless the 3SUM conjecture does not hold [48].

Theorem 6.2.1 (Dichotomy)**.** *Let Q be any Boolean IJQ with hypergraph $\mathcal{H}$ and D be any matching database.*

- *If $\mathcal{H}$ is ι-acyclic, then Q can be computed in $\mathcal{O}(|D|)$ time.*
- *If $\mathcal{H}$ is not ι-acyclic, then there is no algorithm that can compute Q in $O(|D|^{4/3-\epsilon})$ time for $\epsilon > 0$, unless the 3SUM conjecture fails.*

Proof. The linear-time complexity in case $\mathcal{H}$ is ι-acyclic follows immediately: Since each hypergraph in $\tilde{\mathbf{H}}$ is α-acyclic, its corresponding CQ can be computed in linear time using Yannakakis's algorithm [58]. Furthermore, the size of $\tilde{\mathbf{H}}$ is independent of the size of the input database D.

We next prove the hardness in case $\mathcal{H}$ is not ι-acyclic. Assume for a contradiction that there exists an algorithm $\mathcal{A}_Q$ that can decide $Q(D)$ in time $\mathcal{O}(|D|^{4/3-\epsilon})$, for some $\epsilon > 0$, for any database D. Since the hypergraph $\mathcal{H}$ is not ι-acyclic, by the characterisation given in Theorem 6.1.2, it has a Berge cycle of length $k \geq 3$. This means, according to Definition 3.1.11, that there exists a sequence of alternating hyperedges and vertices $(e_1, v_1, e_2, v_2, \dots, e_k, v_k, e_{k+1} = e_1)$ such that $v_1, \dots, v_k \in \mathcal{V}$ are pairwise distinct vertices, $e_1, \dots, e_k \in \mathcal{E}$ are pairwise distinct hyperedges, $k \geq 3$, and for each $1 \leq i \leq k$ we have that $v_i \in e_i \cap e_{i+1}$. Let

$$Q' = S_1(w_k, w_1) \wedge S_2(w_1, w_2) \wedge \cdots \wedge S_k(w_{k-1}, w_k),$$

be the k-cyclic CQ query; all the variables in all of their occurrences range over scalars. We can construct an algorithm $\mathcal{A}_{Q'}$ based on $\mathcal{A}_Q$ that can solve $Q'(D')$ in time $\mathcal{O}(|D'|^{4/3-\epsilon})$, for any input database instance $D' = \{S_1, \dots, S_k\}$. Construct the following database instance D for the IJQ query Q. Let $D = \{R_1, \dots, R_k\}$, where the relations corresponding to the hyperedges $e_1, \dots, e_k$ are denoted by $R_1, \dots, R_k$ respectively. Furthermore,

- for every $1 \leq i \leq k$ and for every tuple $(a, b) \in S_i$, where $S_i \in D'$, include in the relation $R_i \in D$ the tuple where the value of v_{i-1} is the interval with coincident endpoints $[a, a]$, the value of v_i is the interval with coincident endpoints $[b, b]$, and the value of each other variable in R_i is the interval $(-\infty, +\infty)$; and
- every relation R other than $R_1, \ldots, R_k$ in D consists of one tuple; the tuple where the value of each variable from R is the interval $(-\infty, +\infty)$.

Hence, by construction, it holds that $|D| = \mathcal{O}(|D'|)$. Since the interval $(-\infty, +\infty)$ intersects with any other interval and the intervals $[a, a]$ and $[b, b]$ intersect if and only if $a = b$, we have that there is a bijection that maps each satisfying assignment of the variables $w_1, \ldots, w_k$ for the CQ query $Q'(D')$ to a satisfying assignment of the variables in $\mathcal{V}$ for the IJQ query $Q(D)$. Therefore, the answer of $Q'(D')$ is equal to the answer of $Q(D)$. The algorithm $\mathcal{A}_{Q'}$ first constructs the database D in time $\mathcal{O}(|D'|)$ and then, calls the algorithm $\mathcal{A}_Q$ on the input D. Since $D = \mathcal{O}(|D'|)$, the runtime of $\mathcal{A}_Q$ is $\mathcal{O}(|D|^{4/3-\epsilon}) = \mathcal{O}(|D'|^{4/3-\epsilon}|)$. Therefore, $\mathcal{A}_{Q'}$ solves $Q'(D')$ in time $\mathcal{O}(|D'|^{4/3-\epsilon})$. This is a contradiction, because the k-cyclic CQ Q' cannot be computed in time $O(|D'|^{4/3-\epsilon})$ for $\epsilon > 0$ [3], unless the 3SUM conjecture fails [48]. □

6.3 Determining Iota-acyclicity in Polynomial Time

This section shows how to derive a polynomial-time algorithm for determining ι-acyclicity. The goal of finding such an algorithm is met by achieving another characterisation of ι-acyclicity, which reveals a property in the structure of ι-acyclic hypergraphs that can be checked using a polynomial number of steps in the hypergraph's size. It should be noted that the main goal is to find a polynomial time algorithm, not necessarily the best algorithm. Finding more efficient, or even optimal, algorithms is an open question.

Theorem 6.3.1 (Another Characterisation for ι-acyclicity)**.** *The hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is ι-acyclic if and only if $\mathcal{H}$ is α-acyclic and there exist no three vertices $\{x, y, z\} \subseteq \mathcal{V}$ such that either of the following hold:*

- $\mathcal{E}[\{x, y, z\}] \supseteq \{\{x, y, z\}, \{x, y, z\}, \{x, y, z\}\}$ *(C_1);*
- $\mathcal{E}[\{x, y, z\}] \supseteq \{\{x, y, z\}, \{x, y\}, \{y, z\}\}$ *(C_2);*
- $\mathcal{E}[\{x, y, z\}] \supseteq \{\{x, y, z\}, \{x, y, z\}, \{x, y\}\}$ *(C_3).*

Proof. "$\implies$": Assume that $\mathcal{H}$ is ι-acyclic. We prove that $\mathcal{H}$ is α-acyclic, and there exist no $\{x, y, z\} \subseteq \mathcal{V}$ such that C_1 or C_2 or C_3. Assume that there exist such a set $\{x, y, z\} \subseteq \mathcal{H}$ such that C_1 or C_2 or C_3. In any of those three cases $\mathcal{H}$ has a Berge cycle of length 3 (Definition 3.1.11). For example, if C_1 is true, by Definition 3.1.2, there exists three hyperedges e_1, e_2, $e_3 \in \mathcal{E}$ such that $e_1, e_2, e_3 \subseteq \{x, y, z\}$, which form the following Berge cycle of length 3: $(e_1, x, e_2, y, e_3, z, e_1)$. In both the cases C_2 and C_3 a Berge cycle of length 3 can be found similarly. Therefore, $\mathcal{H}$ it is not ι-acyclic. Contradiction. Assume that $\mathcal{H}$ is not α-acyclic. By Corollary 6.1.4, the hypergraph $\mathcal{H}$ is not ι-acyclic. Contradiction.

"$\impliedby$": Assume that $\mathcal{H}$ is α-acyclic and there exist no $\{x, y, z\} \subseteq \mathcal{V}$ such that C_1 or C_2 or C_3. We prove that $\mathcal{H}$ has no Berge cycle with length ≥ 3. We use strong induction in the length of the Berge-cycle. Let us call $P(n)$ the statement:

The hypergraph $\mathcal{H}$ has no Berge-cycle with length n.

Next, we prove $P(n)$ for every $n \geq 3$.

Base case. We prove $P(3)$ that is:

The hypergraph $\mathcal{H}$ has no Berge-cycle with length 3.

For contradiction, assume that $\mathcal{H}$ has a Berge cycle of length 3. Considering any 3 vertices $x, y, z \in \mathcal{V}$, there are four different ways to obtain a Berge cycle of length precisely

3 using those three vertices: having $\mathcal{E}[\{x,y,z\}] \supseteq \{\{x,y,z\}, \{x,y,z\}, \{x,y,z\}\}$; or $\mathcal{E}[\{x,y,z\}] \supseteq \{\{x,y,z\}, \{x,y\}, \{y,z\}\}$; or $\mathcal{E}[\{x,y,z\}] \supseteq \{\{x,y,z\}, \{x,y,z\}, \{y,z\}\}$; or $\mathcal{E}[\{x,y,z\}] \supseteq \{\{x,y\}, \{y,z\}, \{z,x\}\}$. It is straightforward that the first three cases lead to a contradiction. As for the fourth case, since $\mathcal{H}$ is α-acyclic, there should exist a hyperedge $\{x,y,z\} \in \mathcal{E}[\{x,y,z\}]$ — otherwise, the hypergraph would include a "Triangle", hence, it would not be cycle-free. Hence, there exists x,y,z such that $\{\{x,y,z\}, \{x,y\}, \{y,z\}, \{z,x\}\} \subseteq \mathcal{H}[\{x,y,z\}]$. This leads to a contradiction as well.

Inductive step. We prove that $P(i)$ for each $3 \leq i < n$ implies $P(n)$. Assume $P(i)$ is true for each $3 \leq i < n$. For contradiction, assume $\neg P(n)$, that is:

The hypergraph $\mathcal{H}$ has a Berge cycle of length n.

Hence, there exists a sequence $(e_1, v_1, e_2, v_2, \ldots, e_n, v_n, e_1)$, where $e_1, \ldots, e_n \in \mathcal{E}$ and $v_1, \ldots, v_n \in \mathcal{V}$. Let $S = \{v_1, \ldots, v_n\}$. Next, we consider three disjoint cases and prove that each contradicts the inductive hypothesis.

- Assume that $e_i = \{v_i, v_{i+1}\}$ for each $1 \leq i \leq n$, assuming $v_1 = v_{n+1}$, and that $\mathcal{M}(\mathcal{E}[S]) = \{e_1, \cdots e_n\}$. The latter condition means that there is no distinct hyperedge f in $\mathcal{E}[S]$ that includes two non consecutive vertices from $(v_1, \ldots, v_n)$, otherwise it would belong to $\mathcal{M}(\mathcal{E}[S])$. Hence, by Definition 3.1.5, the hypergraph $\mathcal{H}$ is not cycle-free. Therefore, by Definition 3.1.8, the hypergraph $\mathcal{H}$ is not α-acyclic. Contradiction.

- Assume that $e_i = \{v_i, v_{i+1}\}$ for each $1 \leq i \leq n$, assuming $v_1 = v_{n+1}$, and that $\mathcal{M}(\mathcal{E}[S])$ contains a distinct hyperedge f that includes two non consecutive vertices from $(v_1, \ldots, v_n)$. This creates a smaller Berge cycle in $\mathcal{H}$. Assume without loss of generality that $f \subseteq \{v_k, v_l\}$, where $1 \leq k < l \leq n$. The smaller Berge cycle will be the following sequence $(f, v_k, e_{k+1}, \ldots, v_{l-1}, e_{l-1}, v_l, f)$. Contradiction.

- Assume that there exists a hyperedge e_i for some $1 \leq i \leq n$ on the Berge cycle, such that $e_i \supseteq \{v_i, v_{i+1}, v_j\}$ for some $1 \leq j \leq n$. This creates a smaller Berge cycle in $\mathcal{H}$. Assume without loss of generality that $j > i + 1$. The smaller Berge cycle will be the following sequence $(e_i,\ v_j,\ e_j,\ \ldots,\ v_{i-1},\ e^{i-1},\ v_i,\ e_i)$. Contradiction.

In conclusion, we have proven that $P(n)$ holds for every n. $\square$

Corollary 6.3.2. *The hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is ι-acyclic if and only if $\mathcal{H}$ is cycle-free and there exist no three vertices $\{x, y, z\} \subseteq \mathcal{V}$ such either of the following hold:*

- $\mathcal{H}[\{x, y, z\}] \supseteq \{\{x, y, z\}, \{x, y, z\}, \{x, y, z\}\}$ *(C_1);*
- $\mathcal{H}[\{x, y, z\}] \supseteq \{\{x, y, z\}, \{x, y\}, \{y, z\}\}$ *(C_2);*
- $\mathcal{H}[\{x, y, z\}] \supseteq \{\{x, y, z\}, \{x, y, z\}, \{x, y\}\}$ *(C_3).*

Proof. The proof of this corollary is similar to that of Theorem 6.3.1. $\square$

Theorem 6.3.3 (Polynomial Time Algorithm)**.** *ι-acyclicity is polynomial-time decidable.*

Proof. An algorithm that checks ι-acyclicity on an input hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ can be build using two building-blocks: (1) an algorithm that checks if the hypergraph $\mathcal{H}$ is α-acyclic; and (2) an algorithm that checks if there exists a structure of the form C_1, or C_2, or C_3 in the hypergraph $\mathcal{H}$. There exists an algorithm that performs step (1) in time linear in the size of the hypergraph [53]. Let us analyse step (2). There are $\mathcal{O}(|\mathcal{V}|^3)$ ways to choose three distinct vertices x, y, z from $\mathcal{V}$. For each triple: (i) compute $\mathcal{E}[\{x, y, z\}]$ in $\mathcal{O}(|\mathcal{V}| \cdot |\mathcal{E}|)$, and (ii) check C_i for each $1 \leq i \leq 3$ in $\mathcal{O}(|\mathcal{E}|)$. $\square$

6.4 Discussion

This chapter introduced a novel syntactic notion of acyclicity, called ι-acyclicity, which characterizes the class of Boolean $\overline{\text{IJQ}}s$ that are quasilinear-time decidable. Hence,

ι-acyclicity for Boolean $\overline{\text{IJQ}}s$ is analogous to the notion of α-acyclicity for Boolean CQs. Iota-acyclicity is strictly implied by Berge-acyclicity and strictly implies γ-acyclicity. This helps determine how restrictive this class is in comparison to other acyclicity classes. Notably, the non ι-acyclic Boolean $\overline{\text{IJQ}}s$ are at least as hard as the Boolean triangle CQ, which is widely considered not decidable in linear time. Last but not least, given a hypergraph, its ι-acyclicity can be checked in polynomial time.

Future Directions

A natural extension of this work is to refine the acyclicity notion for the class of IJQs, by removing the assumption that all the variables in all of their occurrences are interval variables. Such a notion of acyclicity would necessarily lie between α-acyclicity and ι-acyclicity; it would be the former when the IJQ is a CQ, and it would be the latter when the IJQ is an $\overline{\text{IJQ}}$.

In previous work, another notion of width known as hypertree width, was developed to generalise acyclcity; the larger the hypertree, the closer the hypergraph is to being acyclic [29, 30]. It would be natural to study and extend the hypertree width for IJQs as well. Investigating the complexity of non ι-acyclic IJQs that satisfy another definition of acyclicity might also be intriguing. Are all the $\overline{\text{IJQ}}s$ that are not ι-acyclic but are α-acyclic decidable in $\tilde{\mathcal{O}}(N^{3/2})$, for instance?

6.5 Proofs

Two helper functions are defined in the following. A surjective function that maps each vertex in $\tilde{\mathcal{V}}$ to some vertex in $\mathcal{V}$; and a bijective function that maps each hyperedge in $\tilde{\mathcal{E}}$ to some hyperedge in $\mathcal{E}$. Both functions follow immediately from Definitions 5.2.3 and 5.2.9.

Definition 6.5.1 (Help Functions)**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph and $\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$*

be any member of the set $\tilde{\mathbf{H}}$. Define the following functions:

1. *a surjective function $\nu_{\tilde{\mathcal{H}},\mathcal{H}} : \tilde{\mathcal{V}} \to \mathcal{V}$ that maps each vertex $\tilde{u}_i \in \tilde{\mathcal{V}}$ to the corresponding vertex $u \in \mathcal{V}$; and*

2. *a bijective function $\epsilon_{\tilde{\mathcal{H}},\mathcal{H}} : \tilde{\mathcal{E}} \to \mathcal{E}$ that maps each hyperedge $\tilde{e} \in \tilde{\mathcal{E}}$ to the corresponding hyperedge $e = \{\nu_{\tilde{\mathcal{H}},\mathcal{H}}(\tilde{u}) \mid \tilde{u} \in \tilde{e}\} \in \mathcal{E}$.*

Whenever it is clear from the context, the subscript in the names of the functions is omitted.

The following properties follow immediately from Definitions 5.2.3 and 5.2.9 as well. They are based on the specific structure of the hypergraphs constructed by the forward reduction. The first property states that any node in $\tilde{\mathcal{H}}$ is mapped back to precisely one node in $\mathcal{H}$ and there is a bijection between the hyperedges to which these two nodes belong respectively. The second property makes the first property stronger; there is always a node $\tilde{u}_1$ in $\tilde{\mathcal{H}}$ for every node u in $\mathcal{H}$, and there is a bijection between the hyperedges to which these two nodes belong, respectively. Finally, the third property states that whenever we have a node $\tilde{u}_j$ in a hyperedge e in $\tilde{\mathcal{H}}$, which corresponds to a node u in $\mathcal{H}$, we also have all nodes $\tilde{u}_1, \ldots, \tilde{u}_{j-1}$ in the hyperedge e.

Property 6.5.2 (Reduction Properties)**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a mixed hypergraph and $\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ be any member of $\tilde{\mathbf{H}}$. The following properties hold.*

1. *For each hyperedge $\tilde{e} \in \tilde{\mathcal{E}}$ and each vertex $\tilde{u} \in \tilde{\mathcal{V}}$, if $\tilde{u} \in \tilde{e}$ in $\tilde{\mathcal{H}}$ then $\nu_{\tilde{\mathcal{H}},\mathcal{H}}(\tilde{u}) \in \epsilon_{\tilde{\mathcal{H}},\mathcal{H}}(\tilde{e})$ in $\mathcal{H}$.*

2. *For each hyperedge $\tilde{e} \in \tilde{\mathcal{E}}$ and each vertex $u \in \mathcal{V}$, $\tilde{u}_1 \in \tilde{e}$ in $\tilde{\mathcal{H}}$ if and only if $u \in \epsilon_{\tilde{\mathcal{H}},\mathcal{H}}(\tilde{e})$ in $\mathcal{H}$.*

3. *For any two vertices $\tilde{u}_i, \tilde{u}_j \in \tilde{\mathcal{V}}$ with $i < j$ and $\tilde{e} \in \tilde{\mathcal{E}}$, we have that if $\tilde{u}_j \in \tilde{e}$ then $\tilde{u}_i \in \tilde{e}$.*

Lemma 6.5.3. *Consider a hypergraph $\mathcal{H}$ and any hypergraph $\tilde{\mathcal{H}} \in \tilde{\mathbf{H}}$. If $\tilde{\mathcal{H}}$ has a Berge cycle $(\tilde{e}^1, \tilde{v}^1, \tilde{e}^2, \tilde{v}^2, \ldots, \tilde{e}^k, \tilde{v}^k, \tilde{e}^{k+1} = \tilde{e}^1)$ of length k such that $\nu(\tilde{v}^1), \ldots, \nu(\tilde{v}^k)$ are distinct pairwise vertices from $\mathcal{V}$, then $\mathcal{H}$ also has a Berge cycle of length k.*

Proof. This is a proof by construction. Assume that the statement is true. Since $(\tilde{e}^1, \tilde{v}^1, \tilde{e}^2, \tilde{v}^2, \ldots, \tilde{e}^k, \tilde{v}^k, \tilde{e}^{k+1} = \tilde{e}^1)$ is a Berge cycle, for each $i \in [k]$, we have $\tilde{v}^i \in \tilde{e}^i$ and $\tilde{v}^i \in \tilde{e}^{i+1}$. Hence, by Property 6.5.2, part (1), for each $i \in [k]$, we get that $\nu(\tilde{v}^i) \in \epsilon(\tilde{e}^i)$ and $\nu(\tilde{v}^i) \in \epsilon(\tilde{e}^{i+1})$. Since ϵ is a bijection and $\tilde{e}^1, \ldots, \tilde{e}^k$ are pairwise distinct hyperedges of $\tilde{\mathcal{E}}$, we get that $\epsilon(\tilde{e}^1), \ldots, \epsilon(\tilde{e}^k) \in \mathcal{E}$ are distinct hyperedges. Therefore, the sequence $(\epsilon(\tilde{e}^1), \nu(\tilde{v}^1), \ldots, \epsilon(\tilde{e}^k), \nu(\tilde{v}^k), \epsilon(\tilde{e}^{k+1}) = \epsilon(\tilde{e}^1))$ is a Berge cycle in $\mathcal{H}$, and it is of length k. □

6.5.1 Proof of Theorem 6.1.2

Theorem 6.1.2 (Iota Acyclicity Characterisation). *A hypergraph is ι-acyclic if and only if it has no Berge cycle of length greater than or equal to 3.*

Proof. "$\implies$": This is proof by contradiction. Assume $\mathcal{H}$ has a Berge cycle of length strictly greater than two, or equivalently, at least three. Hence, there exists a cyclic sequence $(e^1, v^1, e^2, v^2, \ldots, e^k, v^k, e^{k+1} = e^1)$ such that: $k \geq 3$; $v^1, \ldots, v^k$ are pairwise distinct vertices from $\mathcal{V}$; $e^1, \ldots, e^k$ are pairwise distinct hyperedges in $\mathcal{E}$; and for each $i \in [k]$, we have $v^i \in e^i$ and $v^i \in e^{i+1}$.

By the structure of the hypergraphs in $\tilde{\mathbf{H}}$ (Algorithm 4), there exists a hypergraph $(\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})) \in \tilde{\mathbf{H}}$ such that for each $i \in [k]$ we have:

$$\{\tilde{v}^i_1, \ldots, \tilde{v}^i_{n_{v^i}-1}\} \subseteq \epsilon^{-1}(e^i) \text{ and } \tilde{v}^i_{n_{v^i}} \notin \epsilon^{-1}(e^i)$$

and

$$\{\tilde{v}^i_1, \ldots, \tilde{v}^i_{n_{v^i}}\} \subseteq \epsilon^{-1}(e^{i+1}).$$

Since $k \geq 3$, the hypergraph $\tilde{\mathcal{H}}$ has the following three properties:

1. For each $i \in [k]$ the vertex $\tilde{v}^i_{n_{v^i}-1}$ belongs to precisely two hyperedges from $\tilde{\mathcal{E}}$. These hyperedges are $\epsilon^{-1}(e^i)$ and $\epsilon^{-1}(e^{i+1})$;

2. For each $1 \leq i, j \leq k$ with $|i-j| \leq 2$, the hyperedge $\epsilon^{-1}(e^i)$ cannot be contained in the hyperedge $\epsilon^{-1}(e^j)$ because, by (1) the vertex $\tilde{v}^i_{n_{v^i}-1}$ belongs to $\epsilon^{-1}(e^i)$ but cannot belong to $\epsilon^{-1}(e^j)$;

3. For each $i \in [k]$, the hyperedges $\epsilon^{-1}(e^i)$ and $\epsilon^{-1}(e^{i+1})$ cannot be subset of each other because by (1) we have:

   - $\tilde{v}^{i-1}_{n_{v^{i-1}}-1}$ belongs to $\epsilon^{-1}(e^i)$ but cannot belong to $\epsilon^{-1}(e^{i+1})$, and
   - $\tilde{v}^{i+1}_{n_{v^{i+1}}-1}$ belongs to $\epsilon^{-1}(e^{i+1})$ but cannot belong to $\epsilon^{-1}(e^i)$.

Let $\tilde{\mathcal{V}}' = \{\tilde{v}^i_{n_{v^i}-1} \mid 1 \leq i \leq k\}$ and $\tilde{\mathcal{E}}' = \{\epsilon^{-1}(e^i) \mid 1 \leq i \leq k\}$. Note that $\tilde{\mathcal{V}}' \subseteq \tilde{\mathcal{V}}$ and $\tilde{\mathcal{E}}' \subseteq \tilde{\mathcal{E}}$. Therefore, no matter what other steps are taken during the runtime of the GYO reduction on $\tilde{\mathcal{H}}$, by (1), no vertex from $\tilde{\mathcal{V}}'$ will become candidate for removal, and by (2) and (3), no hyperedge from $\tilde{\mathcal{E}}'$ will become a candidate for removal. Hence, $\tilde{\mathcal{H}}$ cannot be GYO reducible to the empty hypergraph. In other words, the hypergraph $\tilde{\mathcal{H}}$ includes the cycle

$$\{\{\tilde{v}^i_{n_{v^i}-1}, \tilde{v}^{i+1}_{n_{v^{i+1}}-1}\} \mid 1 \leq i < k\} \cup \{\{\tilde{v}^k_{n_{v^k}-1}, \tilde{v}^1_{n_{v^1}-1}\}\}, \tag{6.1}$$

where, by the above property (1), the edge $\{\tilde{v}^i_{n_{v^i}-1}, \tilde{v}^{i+1}_{n_{v^{i+1}}-1}\}$ for each $1 \leq i < k$ is included in precisely one hyperedge from $\tilde{\mathcal{E}}$, that is $\epsilon^{-1}(e^{i+1})$, and $\{\tilde{v}^k_{n_{v^k}-1}, \tilde{v}^1_{n_{v^1}-1}\}$ is included in precisely one hyperedge from $\tilde{\mathcal{E}}$, that is $\epsilon^{-1}(e^1)(= \epsilon^{-1}(e^{k+1}))$. Thus, $\tilde{\mathcal{H}}$ is not α-acyclic. Therefore, by Definition 6.1.1 of ι-acyclicity, $\mathcal{H}$ is not ι-acyclic. This contradicts the initial assumption.

" $\Longleftarrow$ ": Assume for a contradiction that $\mathcal{H}$ is not ι-acyclic. Hence, by Definition 6.1.1, there exists $\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}}) \in \mathfrak{H}(\mathcal{H})$ that is not α-acyclic. Therefore, by

Definition 3.1.8, $\tilde{\mathcal{H}}$ is not conformal or not cycle-free. Next, we prove that each of the two cases leads to a contradiction.

Case 1. The hypergraph $\tilde{\mathcal{H}}$ is not conformal. Therefore, there exists a subset $S \subseteq \tilde{\mathcal{V}}$ with $|S| \geq 3$ such that

$$\mathcal{M}(\tilde{\mathcal{E}}[S]) = \{S \setminus \{x\} \mid x \in S\}.$$

According to Definition 3.1.2, we have $\tilde{\mathcal{E}}[S] = \{e \cap S \mid e \in \tilde{\mathcal{E}}\} \setminus \emptyset$. According to Definition 3.1.3, we have $\mathcal{M}(\tilde{\mathcal{E}}[S]) = \{e \in \tilde{\mathcal{E}}[S] \mid \nexists f \in \tilde{\mathcal{E}}[S], e \subset f\}$. Let $\tilde{x}, \tilde{y}, \tilde{z} \in S$ be distinct vertices from S. Let $\tilde{e}_{\tilde{x}} = S \setminus \{\tilde{x}\}$, $\tilde{e}_{\tilde{y}} = S \setminus \{\tilde{y}\}$, and $\tilde{e}_{\tilde{z}} = S \setminus \{\tilde{z}\}$ be distinct hyperedges from $\mathcal{M}(\tilde{\mathcal{E}}[S])$. Since $S \subseteq \tilde{\mathcal{V}}$, we also have that $\tilde{x}, \tilde{y}, \tilde{z} \in \tilde{\mathcal{V}}$.

We now need the following claim at this point in the proof; its own proof is given at the end of this section.

Claim 6.5.4. *The vertices $\nu(\tilde{x})$, $\nu(\tilde{y})$, and $\nu(\tilde{z})$ are pairwise distinct vertices of $\mathcal{V}$.*

By Definition 3.1.3, we have $\mathcal{M}(\tilde{\mathcal{E}}[S]) \subseteq \tilde{\mathcal{E}}[S]$, hence, $e_{\tilde{x}}$, $e_{\tilde{y}}$, and $e_{\tilde{z}}$ belong also to $\tilde{\mathcal{E}}[S]$. By Definition 3.1.2 of the induced set, this means that there exist three distinct hyperedges $\tilde{c}_{\tilde{x}}, \tilde{c}_{\tilde{y}}, \tilde{c}_{\tilde{z}} \in \tilde{\mathcal{E}}$ such that $\tilde{e}_{\tilde{x}} \subseteq \tilde{c}_{\tilde{x}}$, $e_{\tilde{y}} \subseteq \tilde{c}_{\tilde{y}}$, and $e_{\tilde{z}} \subseteq \tilde{c}_{\tilde{z}}$. Therefore, the sequence $(\tilde{c}_{\tilde{x}}, \tilde{z}, \tilde{c}_{\tilde{y}}, \tilde{x}, \tilde{c}_{\tilde{z}}, \tilde{y}, \tilde{e}_{\tilde{x}})$ is a Berge cycle of length 3 in $\tilde{\mathcal{H}}$ where, by Claim 6.5.4, $\nu(\tilde{x}), \nu(\tilde{y}), \nu(\tilde{z})$ are pairwise distinct vertices of $\mathcal{H}$. Therefore, by Lemma 6.5.3, $\mathcal{H}$ also has a Berge cycle of length 3. This statement contradicts the initial assumption that $\mathcal{H}$ is ι-acyclic.

Case 2. The hypergraph $\tilde{\mathcal{H}}$ is non-cycle-free. Hence, there exist $S = \{\tilde{v}^1, \ldots, \tilde{v}^k\} \subseteq \tilde{\mathcal{V}}$ where $k \geq 3$ of pairwise distinct vertices such that

$$\mathcal{M}(\tilde{\mathcal{E}}[S]) = \{\{\tilde{v}^i, \tilde{v}^{i+1}\} \mid 1 \leq i < k\} \cup \{\{\tilde{v}^k, \tilde{v}^1\}\}.$$

Let $\tilde{e}^{i+1} = \{\tilde{v}^i, \tilde{v}^{i+1}\}$ for each $1 \leq i < k$, and $\tilde{e}^1 = \{\tilde{v}^k, \tilde{v}^1\}$.

We now need the following claim at this point in the proof; its own proof is given at the end of this section.

Claim 6.5.5. *The vertices $\nu(\tilde{v}^1), \ldots, \nu(\tilde{v}^k)$ are pairwise distinct vertices of $\mathcal{V}$.*

By Definition 3.1.3, we have $\mathcal{M}(\tilde{\mathcal{E}}[S]) \subseteq \tilde{\mathcal{E}}[S]$, this means $\tilde{e}^i \in \tilde{\mathcal{E}}[S]$ for each $1 \leq i \leq k$. By Definition 3.1.2, there exist k pairwise distinct hyperedges $\tilde{c}^1, \ldots, \tilde{c}^k \in \tilde{\mathcal{E}}$ such that $\tilde{e}^i \subseteq \tilde{c}^i$ for each $1 \leq i \leq k$. Since $\tilde{c}^1, \ldots, \tilde{c}^k$ are distinct hyperedges in $\tilde{\mathcal{E}}$ and $\tilde{v}^1, \ldots, \tilde{v}^k$ are distinct vertices in $\tilde{\mathcal{V}}$, the sequence $(\tilde{c}^1, \tilde{v}^1, \tilde{c}^2, \tilde{c}^2, \ldots, \tilde{c}^k, \tilde{v}^k, \tilde{c}^1)$ is a Berge cycle of length $k \geq 3$ in $\tilde{\mathcal{H}}$. Moreover, by Claim 6.5.5, $\nu(\tilde{v}^1), \ldots, \nu(\tilde{v}^k)$ are pairwise distinct vertices of $\mathcal{H}$. Therefore, by Lemma 6.5.3, $\mathcal{H}$ has a Berge-cycle of length $k \geq 3$. This statement contradicts the initial assumption that $\mathcal{H}$ is ι-acyclic.

Finally, we give the proofs of Claims 6.5.4 and 6.5.5.

Proof of Claim 6.5.4. Assume for contradiction that there are two distinct vertices $\tilde{u}, \tilde{v} \in \{\tilde{x}, \tilde{y}, \tilde{z}\}$ such that $\nu(\tilde{u}) = \nu(\tilde{v})$. By Property 6.5.2 (3), this means that any hyperedge $\tilde{e} \in \tilde{\mathcal{E}}$ that contains $\tilde{u}$ contains also vertex $\tilde{v}$ (or vice versa). Note that, since $\mathcal{M}(\tilde{\mathcal{E}}[S]) \subseteq \tilde{\mathcal{E}}[S]$ and since $\tilde{u}, \tilde{v} \in S$, the property of the previous statement holds also for the hyperedges of $\mathcal{M}(\tilde{\mathcal{E}}[S])$. This violates the condition that $\mathcal{M}(\tilde{\mathcal{E}}[S]) = \{S \setminus \{x\} \mid x \in S\}$ since in this case the hyperedge $S \setminus \{\tilde{v}\}$ (which contains vertex $\tilde{u}$) would actually need to include vertex $\tilde{v}$ as well. The reverse case is analogous due to symmetry. Contradiction.

Proof of Claim 6.5.5. Assume for contradiction that there are $1 \leq i < j \leq k$ such that $\nu(\tilde{v}^i) = \nu(\tilde{v}^j)$. By Property 6.5.2 (3), this means that any hyperedge $\tilde{e} \in \tilde{\mathcal{E}}$ that contains vertex $\tilde{v}^i$ also contains vertex $\tilde{v}^j$ (or vice versa). Since $\mathcal{M}(\tilde{\mathcal{E}}[S]) \subseteq \tilde{\mathcal{E}}[S]$ and since $\tilde{v}^i, \tilde{v}^j \in S$, we get that the hyperedges from $\mathcal{M}(\tilde{\mathcal{E}}[S])$ satisfy this property too. That is, any hyperedge $\tilde{e} \in \mathcal{M}(\tilde{\mathcal{E}}[S])$ that contains vertex $\tilde{v}^i$ also contains vertex $\tilde{v}^j$ (or vice versa). This violates the condition that $\mathcal{M}(\tilde{\mathcal{E}}[S]) = \{\tilde{e}^1, \ldots, \tilde{e}^k\}$ since, in this case, the hyperedge $\tilde{e}^i = \{\tilde{v}^{i-1}, \tilde{v}^i\}$ (in case $\tilde{v}^{i-1} \neq \tilde{v}^j$) or the hyperedge $\tilde{e}^{i+1} = \{\tilde{v}^i, \tilde{v}^{i+1}\}$ (in case $\tilde{v}^{i+1} \neq \tilde{v}^j$) would also include the vertex $\tilde{v}^j$, creating a chord in the cycle. The reverse case is analogous due to symmetry. Contradiction. □

6.5.2 Proof of Corollary 6.1.4

It is essential to remark that the statement follows immediately from the following argument. According to Corollary 6.1.3 ι-acyclicity implies γ-acyclicity. Furthermore, γ-acyclicity implies α-acyclicity. Therefore, ι-acyclicity implies α-acyclicity. Next, a proof that is based on the join-tree structure is presented.

Corollary 6.1.4 (Iota Implies Alpha). *The class of ι-acyclic hypergraphs is a strict subset of the class of α-acyclic hypergraphs.*

Proof. We prove by construction that $\mathcal{H}$ has a join tree, and hence, by Definition 3.1.7, $\mathcal{H}$ is α-acyclic.

Assume that $\mathcal{H}$ is ι-acyclic. By Definition 6.1.1 this means that all members of $\tau(\mathcal{H})$ are α-acyclic. Let $\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ be a member of $\tau(\mathcal{H})$. Since all members of $\tau(\mathcal{H})$ are α-acyclic, then $\tilde{\mathcal{H}}$ is α-acyclic, and hence, it has a join tree $(\tilde{\mathcal{T}}, \tilde{\chi})$ where $\tilde{\mathcal{T}}$ is a tree and $\tilde{\chi}$ is a bijection $\tilde{\chi} : V(\tilde{\mathcal{T}}) \to \tilde{\mathcal{E}}$ such that the connectivity property holds (see Definition 3.1.7).

It is possible to construct a join tree $(\mathcal{T}, \chi)$ for $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ as follows: assign $\mathcal{T} = \tilde{\mathcal{T}}$ and for each node $t \in \mathcal{T}$ assign $\chi(t) = \epsilon(\tilde{\chi}(t))$. Next, we show that $(\mathcal{T}, \chi)$ is a valid join tree, that is (1) χ is a bijection of the from $\chi : V(\mathcal{T}) \to \mathcal{E}$ and (2) connectivity property holds (see Definition 3.1.7).

1. χ is the composition of the bijections $\tilde{\chi} : V(\tilde{\mathcal{T}}) \to \tilde{\mathcal{E}}$ and $\epsilon : \tilde{\mathcal{E}} \to \mathcal{E}$. Hence, it is a bijection of the form $\chi : V(\tilde{\mathcal{T}}) \to \mathcal{E}$. Since $\mathcal{T} = \tilde{\mathcal{T}}$, we get that χ is a bijection of the form $\chi : V(\mathcal{T}) \to \mathcal{E}$.

2. Let $v \in \mathcal{V}$ be any vertex of $\mathcal{H}$. By Property (2) of 6.5.2, for each hyperedge $\tilde{e} \in \tilde{\mathcal{E}}$ we have $v_1 \in \tilde{e}$ if and only if $v \in \epsilon(\tilde{e})$. Since $\mathcal{T} = \tilde{\mathcal{T}}$ and $\chi(t) = \epsilon(\tilde{\chi}(t))$ for each node $t \in \mathcal{T}$, the set of nodes $\{t \in \tilde{\mathcal{T}} \mid v_1 \in \tilde{\chi}(t)\}$ is equal to the set of

nodes $\{t \in \mathcal{T} \mid v \in \epsilon(\tilde{\chi}(t)) = \chi(t)\}$. Since $(\tilde{\mathcal{T}}, \tilde{\chi})$ is a join tree, the former set of nodes is a non-empty connected subtree of $\tilde{\mathcal{T}}$ (by Definition 3.1.7). Therefore, the latter set of nodes is also a non-empty connected subtree of $\mathcal{T}$.

$\square$

Chapter 7

Conclusions

This Chapter summarises the overall contributions and identifies the further research directions related to this thesis.

7.1 Summary

Chapter 4 made two fundamental contributions. The first is an algorithm that computes the intersection join of k sets of intervals of size n using $\mathcal{O}(k^2 \cdot n \cdot \log(k \cdot n))$ preprocessing time and constant delay enumeration of the tuples in the output. The second is a simple reduction from the problem of evaluating any IJQ to the problem of evaluating a union of CQs, called Intersection Join Decomposition (IJDec). Interestingly, the IJDec reduction reveals that the Boolean Triangle IJQ can be evaluated in time $\tilde{\mathcal{O}}(n^{3/2})$ in data complexity, and the full Triangle IJQ can be evaluated in $\tilde{\mathcal{O}}(n^{3/2})$ preprocessing time, followed by constant delay enumeration of the tuples in the output. This observation led to an in-depth investigation of the data complexity of IJQs, which is covered in Chapter 5.

Chapter 5 focused on the data complexity of Boolean IJQs. It established the data complexity of Boolean IJQs by employing a refined reduction to a disjunction of Boolean CQs. Specifically, it proved that the complexity of any Boolean IJQ is that of the most difficult Boolean CQ in the disjunction created by the reduction. This implies that one can obtain optimal algorithms for the evaluation of Boolean IJQs, given optimal algorithms for the evaluation of Boolean CQs. Furthermore, it showed that the reduction can be also applied in the case of non-Boolean IJQs. Last but not least, the effectiveness of the proposed reduction was illustrated experimentally using synthetic databases. s Chapter 6 introduced a new notion of acyclicity, called ι-acyclicity, which is the class of Boolean $\overline{\text{IJQ}}s$ that can be computed in quasilinear-time. Hence, the notion of ι-acyclicity for Boolean IJQs is analogous to the notion of α-acyclicity for Boolean CQs. It is proven that ι-acyclicity is strictly implied by Berge-acyclicity and strictly implies γ-acyclicity. This is helpful for determining how restrictive this class is, in comparison to other acyclicity classes, cf. Figure 1.2. It was also shown that the Boolean $\overline{\text{IJQ}}s$ that are not ι-acyclic are at least as difficult as the Boolean Triangle CQ, which is widely considered not to be decidable in linear time [3]. Last but not least, it was proven that given a hypergraph, its ι-acyclicity can be checked in polynomial time.

7.2 Future Research Directions

This section suggests a number of directions for future research. Several of them are related to aspects of the IJQ computation that are left uncovered, whereas others propose the utilisation of the techniques introduced by this thesis to solve various computational problems.

Practical Aspects

In spatio-temporal databases, the various complex objects can be either represented, or approximated by intervals [40]. In the future, it would be useful to test the performance of the reduction in realistic use cases involving spatiotemporal data, where the data distributions differ from those used in this thesis. Existing database systems rely on statistics and specialised algorithms to generate efficient query plans for queries; thus, improving the performance of such plans may be difficult. Further optimisations to the reduction and, ideally, the removal of the arising polylogarithmic factors in the data size are required for such a task.

The Complexity of non-Boolean IJQs

Chapter 5 is focused on the complexity of Boolean IJQs, and towards the end, in Section 5.4, it is discussed how one can utilise the forward reduction to solve non-Boolean IJQs. Furthermore, a complexity statement is made about the preprocessing time needed to achieve constant delay enumeration of the tuples in the output in the case of full IJQs (Subsection 5.4.2). It would be interesting to investigate the enumeration complexity of IJQs in the presence of free variables in greater depth, and in particular, identify the class of IJQs that admit quasi-linear time preprocessing followed by constant delay enumeration of the tuples in the output. This can be accomplished by studying and utilising existing results for the enumeration of CQs and unions of CQs [18, 9, 52].

Queries with Inequalities

Another idea for future research is to consider queries with inequality joins and try to understand their complexity by transforming them into queries with equality joins, possibly by using techniques introduced by this thesis. A useful observation is that an inequality condition, $a < b$ for example, can be rewritten to an intersection condition by representing a as an interval $a' = (-\infty, a)$ and asking whether $b \in a'$, or by representing b with the interval $b' = (b, +\infty)$ and asking whether $a \in b'$. It is worth exploring if by systematically applying such an approach, one can recover or even improve results obtained by previous work [56, 2]. Moreover, it would be interesting to check if above-mentioned approach leads to lower bounds for Boolean queries with inequality joins.

Refined Acyclicity

The notion of ι-acyclicity is defined for Boolean $\overline{\text{IJQ}}s$, which is the subset of Boolean IJQs, where all variables in all of their occurrences range over intervals. As mentioned before, it is the counterpart of α-acylicity for Boolean $\overline{\text{IJQ}}s$. A natural extension of Chapter 6 is to refine ι-acyclicity for the class of IJQs, by dropping the assumption that all variables in all of their occurrences are interval variables. The refined ι-acyclicity would have to fall somewhere between α-acyclicity and ι-acyclicity; it would coincide with the former when the IJQ is a CQ, and it would coincide with the latter when the IJQ is an $\overline{\text{IJQ}}$.

Hypertree Width

Another notion of width, known as hypertree width, was previously developed to generalise acyclicity; the larger the hypertree width gets, the closer the hypergraph is to being α-acyclic [29, 30]. Apart from the counterpart of submodular width and fractional hypertree width for IJQs (Definition 5.2.13), it would also be interesting to study the counterpart of the hypertree width for IJQs, and its properties. This notion of width could be defined as the maximum hypertree width among the CQs generated by the reduction of the input IJQ. How would the extended width be related to ι-acyclicity?

Further Complexity Classes

Investigating the complexity of non ι-acyclic IJQs that satisfy another definition of acyclicity might also be intriguing. Are all the $\overline{\text{IJQ}}s$ that are non ι-acyclic but are α-acyclic decidable in $\tilde{\mathcal{O}}(N^{3/2})$, for instance? The reason why this question is being asked is because certain representative queries in this class have this property, for example, the queries in Figures 6.2a, 6.2b, and 6.2c.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, and Maximilian Schleich. Functional aggregate queries with additive inequalities. *ACM Trans. Database Syst.*, 45(4), 2020.

[3] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Trans. Database Syst.*, 41(4):22:1–22:45, 2016.

[4] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *PODS*, pages 13–28, 2016.

[5] 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 *PODS*, pages 429–444, 2017.

[6] Tamas Abraham and John F. Roddick. Survey of spatio-temporal databases. *GeoInformatica*, 3(1):61–99, 1999.

[7] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *VLDB*, pages 570–581, 1998.

[8] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[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] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.

[11] Jon Louis Bentley and Derick Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29(07):571–577, 1980.

[12] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020.

[13] Panagiotis Bouros and Nikos Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *Proc. VLDB Endow.*, 10(11):1346–1357, 2017.

[14] Panagiotis Bouros and Nikos Mamoulis. Spatial joins: what's next? *ACM SIGSPATIAL Special*, 11(1):13–21, 2019.

[15] Peter Brass. *Advanced data structures*, volume 193. Cambridge University Press Cambridge, 2008.

[16] Johann Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3):54:1–54:26, 2016.

[17] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, pages 237–246, 1993.

[18] 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.

[19] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[20] Bernard Chazelle, Herbert Edelsbrunner, Leonidas J Guibas, and Micha Sharir. Algorithms for bichromatic line-segment problems and polyhedral terrains. *Algorithmica*, 11(2):116–132, 1994.

[21] George Chichirim. Efficient computation of intersection join queries. Master's thesis, University of Oxford, 2021.

[22] Mark de Berg. *Computational geometry: algorithms and applications, 2nd Edition*. Springer, 2000.

[23] Anton Dignös, Michael H Böhlen, Johann Gamper, Christian S Jensen, and Peter Moser. Leveraging range joins for the computation of overlap joins. *The VLDB Journal*, 31(1):75–99, 2022.

[24] Jost Enderle, Matthias Hampel, and Thomas Seidl. Joining interval data in relational databases. In *SIGMOD*, pages 683–694, 2004.

[25] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[26] Johann Gamper, Matteo Ceccarello, and Anton Dignös. What's new in temporal databases? In *Advances in Databases and Information Systems - 26th European Conference, ADBIS 2022, Turin, Italy, September 5-8, 2022, Proceedings*, volume 13389 of *Lecture Notes in Computer Science*, pages 45–58. Springer, 2022.

[27] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join operations in temporal databases. *VLDB J.*, 14(1):2–29, 2005.

[28] Nathan Goodman, Oded Shmueli, and Yong Chiang Tay. Gyo reductions, canonical connections, tree and cyclic schemas and tree projections. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 267–278, 1983.

[29] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 57–74, 2016.

[30] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. Complexity analysis of generalized and fractional hypertree decompositions. *Journal of the ACM (JACM)*, 68(5):1–50, 2021.

[31] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, 2014.

[32] Martin Grohe, Thomas Schwentick, and Luc Segoufin. When is the evaluation of conjunctive queries tractable? In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 657–666. ACM, 2001.

[33] Xiao Hu, Stavros Sintos, Junyang Gao, and Pankaj K Agarwal. Computing complex temporal join queries efficiently. Technical report, 2020.

[34] Edwin H. Jacox and Hanan Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7, 2007.

[35] Mahmoud Abo Khamis, George Chichirim, Antonia Kormpa, and Dan Olteanu. The complexity of boolean conjunctive queries with intersection joins. *CoRR*, abs/2106.13342, 2021.

[36] Mahmoud Abo Khamis, George Chichirim, Antonia Kormpa, and Dan Olteanu. The complexity of boolean conjunctive queries with intersection joins. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 53–65. ACM, 2022.

[37] Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.

[38] Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *SIGMOD*, pages 209–220, 1994.

[39] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *SIGMOD*, pages 247–258, 1996.

[40] Nikos Mamoulis. *Spatial Data Management.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[41] Nikos Mamoulis and Dimitris Papadias. Multiway spatial joins. *ACM Trans. Database Syst.*, 26(4):424–475, 2001.

[42] Nikos Mamoulis and Dimitris Papadias. Slot index spatial join. *IEEE Trans. Knowl. Data Eng.*, 15(1):211–231, 2003.

[43] Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013.

[44] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.

[45] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

[46] Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *J. Comput. Syst. Sci.*, 58(3):407–427, 1999.

[47] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[48] Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *STOC*, pages 603–610, 2010.

[49] Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. In *ICDE*, pages 1098–1109, 2016.

[50] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction.* Texts and Monographs in Computer Science. Springer, 1985.

[51] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

[52] Luc Segoufin. Constant delay enumeration for conjunctive queries. *ACM SIGMOD Record*, 44(1):10–17, 2015.

[53] Robert Endre Tarjan and Mihalis Yannakakis. Addendum: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 14(1):254–255, 1985.

[54] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. Parallel in-memory evaluation of spatial joins. In *SIGSPATIAL*, pages 516–519, 2019.

[55] Todd L. Veldhuizen. Leapfrog Triejoin: A worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[56] Qichen Wang and Ke Yi. Conjunctive queries with comparisons. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 108–121. ACM, 2022.

[57] Michael F. Worboys and Matt Duckham. *GIS - a computing perspective (2. ed.)*. Taylor & Francis, 2004.

[58] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.