Université de Lille

Doctoral School **MADIS**

University Department **CRIStAL**

Thesis defended by **Oliver Irwin**

Defended on **March 11, 2026**

In order to become Doctor from Université de Lille

Academic Field **Computer Science**

Speciality **Theoretical Computer Science**

Branching and Circuits: Algorithmic Techniques for Efficient Query Evaluation

Thesis supervised by	Sylvain Salvati	Supervisor
	Florent Capelli	Co-Supervisor

Committee members

Referees	Diego Figueira	Senior Researcher at CNRS - LaBRI
	Benny Kimelfeld	Professor at Technion
Examiners	Nofar Carmeli	Junior Researcher at INRIA - LIRMM
	Cristina Sirangelo	Professor at Université Paris Cité
Supervisors	Sylvain Salvati	Professor at Université de Lille
	Florent Capelli	Junior Professor at Université d'Artois

Colophon

Doctoral dissertation entitled “Branching and Circuits: Algorithmic Techniques for Efficient Query Evaluation”, written by Oliver Irwin, completed on March 3, 2026, typeset with the document preparation system LaTeX and the yathesis class dedicated to theses prepared in France.

Université de Lille

Doctoral School **MADIS**

University Department **CRIStAL**

Thesis defended by **Oliver Irwin**

Defended on **March 11, 2026**

In order to become Doctor from Université de Lille

Academic Field **Computer Science**

Speciality **Theoretical Computer Science**

Branching and Circuits: Algorithmic Techniques for Efficient Query Evaluation

Thesis supervised by	Sylvain Salvati	Supervisor
	Florent Capelli	Co-Supervisor

Committee members

Referees	Diego Figueira	Senior Researcher at CNRS - LaBRI
	Benny Kimelfeld	Professor at Technion
Examiners	Nofar Carmeli	Junior Researcher at INRIA - LIRMM
	Cristina Sirangelo	Professor at Université Paris Cité
Supervisors	Sylvain Salvati	Professor at Université de Lille
	Florent Capelli	Junior Professor at Université d'Artois

Université de Lille

École doctorale **MADIS**

Unité de recherche **CRIStAL**

Thèse présentée par **Oliver Irwin**

Soutenue le **11 mars 2026**

En vue de l'obtention du grade de docteur de l'Université de Lille

Discipline **Informatique**

Spécialité **Informatique Théorique**

Branchements et Circuits : techniques algorithmiques pour l'évaluation efficace de requêtes

Thèse dirigée par Sylvain Salvati directeur
Florent Capelli co-directeur

Composition du jury

Rapporteurs	Diego Figueira	directeur de recherche au CNRS - LaBRI
	Benny Kimelfeld	professeur au Technion
Examinatrices	Nofar Carmeli	chargée de recherche à l'INRIA - LIRMM
	Cristina Sirangelo	professeure à l'Université Paris Cité
Directeurs de thèse	Sylvain Salvati	professeur à l'Université de Lille
	Florent Capelli	professeur junior à l'Université d'Artois

This thesis has been prepared at the following research units.

CRIStAL
Université de Lille - Campus scientifique
Bâtiment ESPRIT
Avenue Henri Poincaré
59655 Villeneuve d'Ascq - France

03 28 77 85 82
https://cristal.univ-lille.fr

INRIA
Parc scientifique de la Haute-Borne
40, avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq - France

03 59 57 78 00
contact-lille@inria.fr
https://www.inria.fr

This thesis has been funded by:

ANR KCODA (ANR-20-CE48-0004)

To Dad, you would've really loved reading this

Branching and Circuits: Algorithmic Techniques for Efficient Query Evaluation

Abstract

Efficiently evaluating and accessing the answers of queries over databases is a central problem in data management, particularly as data volumes continue to grow and queries become more complex. In the relational database model, query evaluation is complicated by the presence of joins, which may induce large intermediate results and lead to prohibitive computational costs. This thesis explores several aspects of query evaluation and their respective computational complexity.

The first aspect we consider is that of worst-case optimal join algorithms. These algorithms aim to return the answer set of a query over a database, while providing strong guarantees ensuring that the evaluation runs in time proportional to the maximum possible output size. In this manuscript, we present a join algorithm that can be shown to be worst-case optimal in a simple and modular way. One of the advantages of our approach is the implicit data representation it induces. By improving on this representation of the answers of a query, we are able to work on finer algorithmic problems.

The two main evaluation tasks we consider in this thesis are direct access and uniform sampling. For uniform sampling, we show how the execution trace of our worst-case optimal join algorithm can be leveraged to sample query answers according to a uniform distribution without enumerating the full result set. For direct access, we exploit relational circuits to navigate the answer space efficiently and retrieve answers at arbitrary positions in a fixed order, extending existing approaches to broader classes of queries, including queries with negation.

In both cases, the aim of our algorithms is to build the answer set of queries incrementally. We also use the structure of relational circuits to improve on the complexity of these methods.

Keywords: databases, algorithms, aggregation, compilation, join queries, conjunctive queries

---

Branchements et Circuits : techniques algorithmiques pour l'évaluation efficace de requêtes

Résumé

L'évaluation et l'accès efficaces aux réponses des requêtes sur les bases de données constituent un enjeu majeur dans le cadre de la gestion des données. Cette problématique revêt une importance croissante avec l'augmentation constante des volumes de données et la complexité croissante des requêtes plus complexes. Dans le modèle des bases de données relationnelles, l'évaluation des requêtes s'avère complexe en raison de la présence des jointures. Ces dernières peuvent engendrer des résultats intermédiaires de grande taille et occasionner des frais de calcul substantiels. Dans le cadre de cette thèse, une exploration approfondie de divers aspects inhérents à l'évaluation des requêtes et à leur complexité computationnelle respective est entreprise.

L'analyse se concentre en premier lieu sur les algorithmes de jointure optimaux dans le pire des cas. Ces algorithmes ont pour objectif de renvoyer l'ensemble des réponses à une requête sur une base de données, tout en offrant des garanties solides quant au temps d'évaluation, qui doit rester proportionnel à la taille maximale de la sortie. Dans le présent manuscrit, nous proposons une exposition d'un algorithme de jointure simple, qui peut être démontré comme étant optimal dans le pire des cas, de manière simple et modulaire.

Dans cette thèse, nous avons exploré deux tâches d'évaluation principales : l'accès direct et l'échantillonnage uniforme. Dans le cadre de l'échantillonnage uniforme, nous montrons comment utiliser la trace d'exécution de notre algorithme de jointure pour échantilloner uniformément, sans pour autant énumérer tous les résultats. Afin d'assurer l'accès direct, nous avons exploité une structure de circuits relationnels. Cette méthode permet une navigation efficace dans l'espace des réponses et la récupération des réponses à des positions arbitraires pour un ordre fixé. Nous montrons également que notre méthode permet d'étendre les résultats présents dans la littérature à des classes plus larges de requêtes, et notamment les requêtes avec négation.

Dans les deux cas, l'objectif des algorithmes développés est de construire l'ensemble des réponses aux requêtes de manière incrémentale. Les structures employées, à base de circuits et de branchements, nous permettent de réduire la complexité des méthodes employées.

Mots clés : bases de données, algorithmes, agrégation, compilation, requêtes de jointure, requêtes conjonctives

---

CRIStAL

Université de Lille - Campus scientifique – Bâtiment ESPRIT – Avenue Henri Poincaré – 59655 Villeneuve d'Ascq - France

Remerciements

Pour paraphraser une personne d'une grande sagesse : si lire les remerciements d'une thèse est toujours un bon moment, c'est autrement plus compliqué quand il s'agit de les écrire soi-même.

Ce travail de thèse n'a pas été une aventure solitaire, loin de là, et j'ai pu compter sur de nombreuses personnes avec qui ce fut un plaisir de travailler et de discuter.

Pour la recherche

I first want to specially thank Diego Figueira and Benny Kimelfeld for agreeing to be referees for this manuscript. The time they spent meticulously reading and their notes enabled me to greatly improve on this manuscript. I also thank Nofar Carmeli and Cristina Sirangela, who honour me by being part of the defense committee.

L'équipe D-DAL[1] (RIP LINKS) a été un environnement incroyable pour découvrir la recherche et l'enseignement. Merci à Antoine, Aurélien, Charles, Iovka, Mikael, Sara, Sophie et Sylvain pour l'accueil et l'ambiance chaleureuse de l'équipe. J'ai pu tous vous embêter avec des tonnes de questions et de discussions aléatoires et je suis reconnaissant d'avoir pu compter sur vous pour m'apporter des réponses. De plus, tout ce travail si sérieux n'aurait pu voir le jour sans l'ambiance éminemment studieuse du bureau B213. Merci aux doctorants du passé : Antonio, Claire, Corentin et Nico et du présent : Arthur, Bastien, Loup et Sébastien de m'avoir accompagné.

Merci à tous les membres de SISE pour leurs encouragements et les bons moments partagés lors des journées GT ou des SISE-au-vert. Un remerciement particulier à Julien et Raphaël pour leur organisation de tous ces moments collectifs. Merci à CRIStAL et INRIA de m'avoir accueilli pendant 3 ans. En particulier, merci à tous leurs membres qui font de ces lieux des endroits si agréables pour travailler (et bien rire).

Je remercie également tous les enseignants du département informatique de l'Université, qui m'ont inspiré et encouragé à faire mes premiers pas de l'autre côté des copies à corriger. Notamment, un grand merci à Charles, François, Jean-Stéphane, Patrice, Romain, Sara et Sylvain pour m'avoir fait confiance avec des groupes de TD, et à l'équipe de la salle 14 : Alexandre, Claire, Émilie, Jean-Christophe, Jean-François, Marius et tous les autres. Merci aussi à tous les enseignants que j'ai eu au fil de mon parcours qui m'ont donné goût à l'informatique.

J'ai eu la chance de pouvoir aller à beaucoup d'endroits différents pendant ma thèse et je suis reconnaissant à toutes les personnes avec qui j'ai pu travailler ou discuter pendant ces déplacements, que ce soit en visite, en conférences ou en écoles d'été (vive les centres de vacances CNRS). Special thanks to Nofar for welcoming us so well in Montpellier, which led to the discovery of the rare species of "Urchin Queries", a nice paper, and my learning of the word רוֹעִ (someday, I'll probably learn more).

Ce qui a rendu toutes ces choses possibles pendant la thèse, c'est aussi le soutien des équipes administratives, sans qui les méandres des différents services auraient été beaucoup plus rudes à traverser. Un merci particulier à Aurore et Nathalie à INRIA, Alexandre à CRIStAL et Jessica au département.

Parce qu'on garde toujours le(s) meilleur(s) pour la fin, et parce que je sais que j'ai eu énormément de chance, je veux bien sûr remercier du fond du cœur ma direction de thèse, sans qui rien de tout ça n'aurait pu être imaginable.

[1] dans nos coeurs TRACtOPL ou ZELDA bien sûr

Merci Florent, pour tout ce que tu as fait pendant cette thèse. Moi qui n'avais jamais fait de théorie avant, je n'aurais pas pu imaginer être aussi bien accompagné sur ce chemin. J'ai beaucoup appris à tes côtés, et ce dans tous les domaines liés de près ou de loin à la recherche et l'enseignement : des bonnes pratiques quand on écrit un papier à comment faire des présentations qui *glissent* (promis, je passe sur monoski bientôt). Tu m'avais dit de voir la thèse comme un shônen : au final, tu n'as pas été le boss de fin, mais le *sensei* avisé tout du long.

Merci Sylvain, vénérable Léviathan du Lambda, pour tes conseils, ton expérience et ton exigence, qui n'ont fait que me tirer vers le haut pendant cette thèse. Merci d'avoir partagé une partie de ton inépuisable connaissance avec moi malgré mon penchant avéré pour la moquerie. Malgré toutes tes responsabilités (et lourde est la tête qui porte la couronne), tu as toujours su trouver du temps pour m'accompagner dans ce périple et je t'en suis reconnaissant.

C'est en très grande partie grâce à vous deux et vos conseils que je peux envisager sereinement de poursuivre une carrière académique. Merci.

Et au-delà

En dehors de ces activités passionnantes, si j'ai pu aller au bout de cette thèse, c'est aussi grâce à de nombreuses personnes. Merci à mes parents, d'abord, pour leur soutien et l'éducation qu'ils m'ont transmises. Merci à mon frère Andrew, qui est une source d'inspiration du fait de ses talents et qualités.

Merci encore à Florent, pour tout les à-côtés de la thèse : les tacos ont plus de goût quand ils sont choisis avec soin. Merci à toi et Aurélia pour les séances de pose de terrasse, les GIF de TayTay en soutien et la découverte de l'arrière-pays lapinvillais. Merci encore à Sylvain, pour les discussions passionnantes sur le vélo, le rugby ou la meilleure traduction de Dostoïevski. J'espère que tu pourras continuer à insuffler la passion de l'ovalie dans les sujets d'examen. Charles, merci pour ton enthousiasme permanent et ta bonne humeur contagieuse, ça fait toujours énormément de bien.

Je remercie également toutes les personnes avec qui je me suis lié d'amitié pendant ce parcours. Merci Corentin, co-bureau parfait. Ton sens de l'humour et de la camaraderie m'a beaucoup manqué sur la fin.

Merci Marie-Éva pour ce joyeux désordre assumé, ce tourbillon de gaffes et de charme qui rend les choses plus vivantes. Ta force et ta détermination forcent l'admiration.

Sara, ti ringrazio infinitamente per tutto. Sei un'amica incredibile e un modello da seguire. I tuoi consigli sono sempre preziosi e sai sempre farmi ridere. Grazie mille anche a Pietro per tutte le battute e i pettegolezzi (e la focaccia, ovviamente).

Merci à tous mes amis qui m'ont soutenu tout du long, notamment Ethan, Florian, Jean-Baptiste, Mathieu, Pauline et Solène. Vous êtes incroyables.

Merci aux anciens de la Bourse aux Livres et de ses aventures : Catherine, Fatine et Jean-Charles. Vos conseils et votre soutien sont inestimables.

On ne peut pas oublier la joyeuse bande de Fakeremo pour des souvenirs inoubliables : se reconnaîtront en vrac une personne incognito, le héros du FabLab, le bioinformaticien dansant, l'expatrié et bien d'autres. Maille après maille, après le PhD Journey, il va être temps de laisser les stagiaires en détresse pour une nouvelle ère.

Un grand merci aussi à Thomas, Quentin, Tanguy et les autres membres des 18P : vous m'avez appris à marcher vers la défaite avec panache et ça n'a pas de prix.

Finalement, merci à vous tous qui êtes venus me voir gesticuler en présentant cette thèse.

Contents

Introduction

Storing and accessing information has been a central concern of human societies throughout history. From the earliest clay tablets and paper scrolls to the meticulously archived documents in modern libraries, methods of preserving information have continually evolved. In the second half of the twentieth century, the introduction of digital storage capacities marked a turning point in information systems. However, finding efficient ways to store and access data remained a central problem.

Accessing the data is indeed just as important as storing it, since, if one has no way of efficiently retrieving some specific information, then even the most efficient storage method will have little to no practical use. In his seminal paper, Codd proposed a data model to organise information: the *relational database model* [Cod70]. This model stores the data as relations (that can be thought of as *tables*), each relation consisting in tuples (or *rows* in the table) and attributes (or variables, the *columns* of the table). Codd's work not only introduced this data structure, but also a family of query languages to allow users to retrieve specific information from the tables. These *queries* represent questions a user asks a database, where the answer is contained in its stored information. In this model, multiple tables could also be linked by logical keys, such as common variables, which opens up the possibility of querying information from multiple rows connected across multiple tables. While conceptually simple, such combinations may quickly give rise to large intermediate or final result sets, making efficient evaluation and access to a query's answers a central algorithmic challenge.

In fact, evaluating queries over relational databases is a fundamental and extensively studied problem. As data volumes continue to grow and play an increasingly central role in modern information systems, it becomes crucial to identify which queries support efficient evaluation and access to their answers. To estimate the complexity of evaluating a query over a database, one may use two different measures: *combined complexity*, where both the query and the database are considered to be the input variables, and *data complexity*, where the query is considered to be fixed.

In the relational model, the source of most of the complexity of query evaluation comes from the *joins*, that is, the combinations of multiple relations. Indeed, although joins are syntactically simple, they might induce a substantial computational overhead. Very early in the theoretical study of relational databases, it was observed that evaluating general relational queries may require computational resources that grow exponentially in the size of the input. This added complexity is due to the nature of a join: if we join two relations each composed of N tuples, then the join can have a size of N^2 in the worst possible case, since each tuple from the first relation could be linked to every tuple of the second. For k such relations, the size of the join could be up to N^k and therefore, joining these relations might cause an exponential blowup in the size of the answer set. In a seminal result, Chandra and Merlin showed that the problem of evaluating queries is NP-complete with respect to combined complexity [CM77]. This hardness result therefore rules out the existence of a single evaluation strategy that would be efficient across the full spectrum of queries and databases. This observation led to new research directions that aimed to find "special cases" of queries or databases that allowed for efficient processing. These cases mostly rely on exploiting additional information, whether about the structure of the query itself, the properties of the data, or the nature of the task to be performed. Over the years, these research directions have given rise to a wide range of algorithmic techniques and evaluation paradigms. As a result, query evaluation is better understood not as a single monolithic problem,

but as a collection of computational tasks and classes of queries, each admitting its own notions of tractability.

More precisely, beyond the "classical" aspect of computing all of the answers to a query over a given database, one may be interested in different, more specific tasks. Two of the tasks closest to the original query evaluation problem might consist in merely finding out whether a query has an answer or not, which is often called *Boolean evaluation*, or *counting* the number of answers. However, more specific problems exist. The following problems generally aim to avoid materialising the full answer set of a query over a database. For instance, in practical applications, it is often useful to iterate over the answers of a query. To this end, one does not necessarily need the full answer set to be computed at a given time, but can work with the answers being generated one by one, without repetition, and with low delay between them. This problem is often referred to as *enumeration* and consists of a (ideally short) preprocessing phase followed by an efficient enumeration phase for the answers with limited delay. If we were to consider an order over the answers of the query, it could also be interesting in practice to access an individual answer based on its index in this order, without having to compute all of the other answers. This specific problem is known as *direct access* and also generally consists of two phases, one that preprocesses the data and the actual access phase. The problem of selecting an answer uniformly at random from the answer set, without materialising the full answer set is also a task that has been well studied. Although being closely related, these tasks exhibit different computational behaviours and often require fundamentally different algorithmic techniques.

Crucially, the feasibility of these tasks depends both on the chosen evaluation problem and on the type of queries or databases under consideration. Queries and databases are usually sorted into *classes*, that group them according to their properties, whether on their structure or their contents. While some classes remain intractable even for very weak tasks, others admit highly efficient algorithms if they consist of elements with suitable restrictions. Identifying and characterising such tractable fragments has therefore become a central theme in the theoretical study of query evaluation. One of the first tractable cases was introduced by Yannakakis as the class of *acyclic queries* [Yan81]. These queries support efficient evaluation, but are a very specific subset. While this class is based on a structural property of the queries that compose it, subsequent work has introduced broader classes defined through more refined structural parameters.

In this manuscript, we focus primarily on *join queries* and on *conjunctive queries*, which form a core fragment of relational query languages and serve as a common abstraction for joins. Our interest lies in understanding how the structural properties of such queries can be exploited to design efficient algorithms for a range of different query answering tasks. We will discuss the specific framework of "worst-case optimal joins". Most join algorithms work by computing intermediate joins and using those to iteratively build the answer set. In the worst-case optimal join setting, the aim is to never have an intermediate join whose size is larger than the largest possible answer set for the query over any database.

While worst-case optimal join algorithms provide a strong baseline for query evaluation, materialising the full answer set is often neither necessary nor desirable once such guarantees are achieved. Amongst the various query answering tasks discussed above, this manuscript is primarily concerned with tasks that go beyond worst-case optimal evaluation by avoiding full materialisation of query answers. We will mostly consider the problems of *uniformly sampling* from and *direct access* to the answers of a query over a given database. A unifying theme throughout the manuscript is that the feasibility of these tasks is based on exploiting structural properties of the queries and databases, rather than on algorithmic optimisations. We will also strive to present algorithms that are simple both in their conception and their complexity analysis, but allow us to recover known results from the literature.

Formal definitions and basic notions related to relational databases and query evaluation are

recalled in the preliminary chapters (see Chapters 1 and 2). In this introduction, we deliberately focus on presenting the conceptual landscape and the main research directions addressed in this manuscript, postponing technical details to the subsequent chapters. The remainder of this introduction provides a more detailed overview of the main problems addressed in this work and explains how they are organised across the different chapters of the manuscript.

Worst-Case Optimal Join Algorithms

As discussed above, the main source of complexity while evaluating relational queries lies in the presence of joins, whose execution may introduce large intermediate or final result sets. In implemented database management systems, traditional query evaluation strategies typically decompose a query in order to build a *query plan*. This plan can be seen as a sequence of binary joins, combining relations two at a time. While this approach is natural and widely used in practice, it may lead to intermediate results that might become exponentially larger than the final answer set, even when the query itself admits a relatively small output. This is notably the case when the query contains *cycles*, that is, when multiple relations are joined together in a way that creates circular dependencies between variables.

Worst-Case Optimal Join (or WCOJ for short) algorithms were introduced to address this phenomenon by taking a global view of the query structure. Informally, an algorithm is worst-case optimal if it can output the answers of a query in time bounded by the size of the largest possible answer set of the query over any database. Using such algorithms therefore typically avoids blowups in the size of intermediate joins. This is a rather recent research direction, with the first algorithm being introduced by Ngo, Porat, Ré, and Rudra [Ngo+12].

Beyond their theoretical optimality guarantees, WCOJ algorithms provide a new framework for understanding the role of the structural properties of queries during the join evaluation. The most recent algorithms do not join relations pairwise, but use branching techniques over the variables to determine which tuples constitute answers to the query. Their performance thus critically depends on how the variables are shared across the relations and on the order in which these variables are processed. As a result, WCOJ algorithms form a natural bridge between worst-case complexity bounds, structural measures of queries, and practical evaluation strategies.

In this manuscript, worst-case optimal joins serve as a foundational algorithmic paradigm. They provide the backbone upon which more refined query answering tasks are built. The algorithms introduced in this manuscript play a central role in several of the subsequent chapters. We introduce the framework more formally in Chapter 2 (see Section 2.2.1) and, in Chapter 3, we introduce a very simple algorithm for query evaluation that can be shown to be worst-case optimal. In Chapter 4, we build on this algorithm, and more precisely, on the *trace* of this algorithm, to answer uniform sampling tasks over join queries.

Relational Circuits as Factorised Representations

While worst-case optimal join algorithms provide a different way of evaluating join queries, the explicit materialisation of query answers often remains impractical for large result sets. In many applications, it is neither necessary nor desirable to store the entire set of answers in a flat relational form. The most naive such representation is a table, which, while practical to use, may incur a prohibitive memory cost. Aside from the purely algorithmic elements of join evaluation, another research direction consists in finding more compact representations that exploit the internal structure of the query. These representations can then be used to support efficient evaluation of query answering tasks, such as enumeration and direct access.

To this end, we use *relational circuits*, which offer a formal framework for representing the answers of relational queries in a structured and factorised manner. This approach is closely related to research on SAT solving, which concerns the problem of determining whether a logical formula is satisfiable. It also links to the *factorised database* research direction, introduced by Olteanu and Závodný [OZ15]. Rather than enumerating all tuples explicitly, factorised representations encode query answers as the result of combining simple components according to the structure of the query. This approach makes it possible to capture properties and shared substructures within the result set, often leading to representations that are exponentially more succinct than their flat counterparts.

Beyond their compactness, relational circuits provide a versatile abstraction for query answering. Their circuit-like structure naturally supports a range of operations on query results, including enumeration, uniform sampling, and direct access to individual answers. Crucially, these operations can often be performed without materialising the full answer set, thereby avoiding the prohibitive costs associated with explicit storage.

From an algorithmic perspective, relational circuits can be viewed as a bridge between query evaluation and query access. They decouple the process of computing a representation of the answer set from the tasks performed on that representation, allowing preprocessing costs to be amortised across multiple subsequent operations. This separation of concerns is particularly useful with enumeration tasks, where we want every answer returned one by one. However, reusing the same structure also happens to be useful in the case where we want to repeat uniform sampling or direct access tasks for several separate answers.

In this manuscript, relational circuits play a central role as an intermediate representation of query answers. They are formally introduced in Chapter 5, but are also closely linked with the results of Chapter 3. Chapter 5 also shows how this structure handles negation in queries with relative ease. In Chapter 6, we build upon these structures to design an efficient direct access algorithm that matches existing results from the literature while extending their scope to join queries with negation.

Answering Queries

The main contribution of this thesis lies in leveraging the properties of worst-case optimal join algorithms and relational circuits to efficiently support different query answering tasks. These two ingredients provide complementary tools: worst-case optimal join algorithms control the complexity of join evaluation, while relational circuits enable compact representations that support efficient access to query answers. In this work, we focus primarily on two such tasks: uniform sampling and direct access.

▶ **Uniform sampling.** Uniformly sampling an answer from the result set of a query without materialising the full answer set is an important task in applications ranging from approximate query answering to data analysis or even machine learning. The challenge lies in generating answers according to a uniform distribution, that is, ensuring that all answers have an equal probability of being produced, while maintaining strong guarantees on processing and sampling complexity. In this thesis, we show how our worst-case optimal join algorithm from Chapter 2 can be used to construct a structured representation of the query's answer set. This is done by exploiting the trace of the algorithm, which exhibits a structure that can be exploited for sampling. These ideas are more formally introduced and developed in Chapter 4, where we present a sampling algorithm that avoids enumerating the full answer set and matches known worst-case complexity bounds.

▶ **Direct access.** As we mentioned earlier, direct access aims to retrieve the answer at a given position in a fixed ordering of the query result, without enumerating all preceding answers. This task is particularly demanding, as it requires navigating through the answer space in a controlled and indexable manner. In this manuscript, we will mostly exploit the properties of relational circuits to guide this navigation and to isolate the contribution of individual components of the query result. Building on these representations, we present a direct access algorithm with efficient preprocessing and access phases, which matches existing results and extends to a broader class of join queries, including queries with negation. These contributions are presented in Chapter 6.

How to Read this Manuscript

This manuscript is organised around several interdependent themes. While the chapters follow a logical progression, not all of them need to be read sequentially, and different reading paths are possible depending on the reader's interests and background.

Chapters 1 and 2 are general introductory chapters. They are targeted to anyone with a keen interest in theoretical computer science wishing to understand more deeply the fundamental concepts developed in this thesis. Chapter 1 introduces the key base notions of graph and database theory that will be useful in the subsequent chapters. It is written with an emphasis on clarity and pedagogy, and aims to provide a self-contained introduction to the necessary background. Chapter 2 is a presentation of different query answering tasks that are mentioned in the thesis. It serves as a survey of existing results to consolidate the contributions from subsequent chapters.

The contributions of this thesis are exposed in Chapters 3 to 6. Chapter 3 proposes the base algorithm for the rest of the manuscript, a simple worst-case optimal join algorithm whose trace will serve as foundation for the relational circuits introduced in Chapter 5 and the specific algorithms for uniform sampling (Chapter 4) and direct access (Chapter 6). It is advisable to read Chapter 3 before Chapter 4, since there is a strong link between the two. Similarly, it is advisable to peruse through Chapter 5 before Chapter 6 to have a better understanding of the structure we use for direct access.

A simple illustration of a recommended reading order is presented in Figure 1. The "most recommended" links are represented with solid arrows (shown in green) and the more "optional" links are represented with dashed arrows (shown in orange).

A Note on Typesetting

This manuscript adopts a number of typesetting conventions intended to improve both readability and navigation. A global table of contents is provided at the beginning of the document, and each chapter is preceded by a more detailed local table of contents. Similarly, while a consolidated bibliography appears at the end of the manuscript, each chapter concludes with its own list of references.

Elements that are central to the development of the results, such as theorems, lemmas, and examples, are visually distinguished from the main text using framed environments. Numbering for these elements is consistent within each chapter and is reset at the beginning of every chapter.

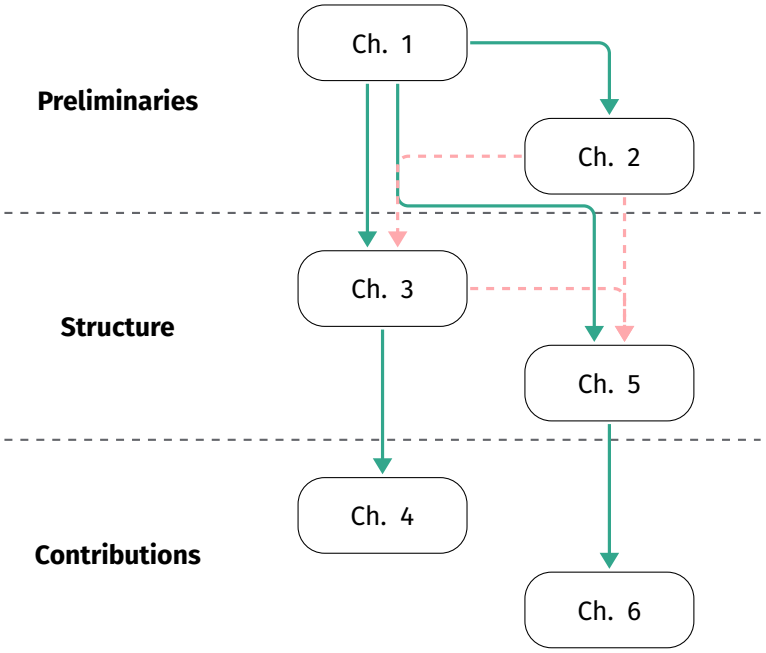


Figure 1: Suggested chapter reading order

Current chapter references

[CM77] Ashok K. **Chandra** and Philip M. **Merlin**. *Optimal Implementation of Conjunctive Queries in Relational Data Bases*. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, May 1977, pp. 77–90. doi 10.1145/800105.803397.

[Cod70] E. F. **Codd**. *A relational model of data for large shared data banks*. In *Communications of the ACM* 13.6 (June 1970), pp. 377–387. doi 10.1145/362384.362685.

[Ngo+12] Hung Q. **Ngo**, Ely **Porat**, Christopher **Ré**, and Atri **Rudra**. *Worst-Case Optimal Join Algorithms: [Extended Abstract]*. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '12: International Conference on Management of Data. Scottsdale Arizona USA: ACM, May 2012, pp. 37–48. doi 10.1145/2213556.2213565.

[OZ15] Dan **Olteanu** and Jakub **Závodný**. *Size Bounds for Factorised Representations of Query Results*. en. In *ACM Transactions on Database Systems* 40.1 (Mar. 2015), pp. 1–44. doi 10.1145/2656335.

[Yan81] Mihalis **Yannakakis**. *Algorithms for Acyclic Database Schemes*. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. Sept. 1981, pp. 82–94. doi 10.5555/1286831.1286840.

Chapter 1

Preliminaries

Foundations are like socks: you only notice them when they're missing.

Unknown

Before delving into the core contributions of this manuscript, we begin with a self-contained chapter that lays out the definitions and conventions upon which the remainder of this work relies. This chapter is not meant to be exhaustive, but aims to help equip the reader with a working vocabulary and the necessary mathematical tools to understand the following results. Readers familiar with the basic theory of databases, graphs, and computational complexity should most probably skim through this chapter and return if and when needed, while others might benefit from reading it sequentially.

We start by introducing general notations, then turn in Section 1.2 to graphs, hypergraphs, and associated structural measures, which will form the basis for structural tractability results in later chapters. These notions are introduced with an emphasis on intuition and examples, to help ground the more abstract definitions.

The next sections are devoted to the relational model of data. In Section 1.3, we define tuples, relations, and databases, before formalising queries. These will be central to the problems we study.

Outline of the current chapter

1.1 General Notations

We will often use sets in this manuscript. A *set* is a collection of different elements. If an object x belongs to a set S, we write $x \in S$. For a finite set S, we denote by $|S|$ its cardinality, that is the number of elements belonging to S. The *empty set* is the only set with no elements, and is denoted $\emptyset$. Given $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, \dots, n\}$. Given two sets S and T, their union is denoted $S \cup T$ and their intersection is denoted by $S \cap T$. If the elements of S all appear in T, then we call S a *subset* of T and write $S \subseteq T$. If S does not contain all of the elements of T, then it is denoted by $S \subset T$ and is a proper subset. Given two sets S and T, we denote by $S \setminus T$ the set obtained by removing from S the elements from T. When considering a set S, the *power set* of S, denoted by $\mathcal{P}(S)$ is the set of all subsets of S. For two sets S and T, we define the *Cartesian product* of S and T, denoted $S \times T$ by the set composed of all possible pairs of elements from S and T. Formally, $S \times T = \{(i, j) \mid i \in S, j \in T\}$.

In this thesis, we will use standard complexity notations. We only remind the main notations that we use here, but the interested reader might refer to the very complete textbooks by Perifel [Per14] or by Arora and Barak [AB16]. If f and g are two functions from $\mathbb{N}$ to $\mathbb{N}$, we say that $f = \mathcal{O}(g)$ if there exists a constant c such that $f(n) \leqslant c \cdot g(n)$ for any sufficiently large n. When writing down complexity statements, we use $\mathsf{poly}(n)$ to denote that the complexity is polynomial in n, $\mathsf{poly}_k(n)$ when the complexity is polynomial in n if k is considered to be constant and $\mathsf{polylog}(n)$ to denote that the complexity is polynomial in $\mathsf{log}(n)$. We also use the $\tilde{\mathcal{O}}(\cdot)$ notation to indicate that polylogarithmic factors are ignored, that is, if the complexity is $\mathcal{O}(n\mathsf{polylog}(n))$, then we write $\tilde{\mathcal{O}}(n)$.

Model of computation. In this thesis, we will often work in the word-RAM model of computation, first introduced by Fredman and Willard [FW90], with $\mathcal{O}(\mathsf{log}(n))$-bit words and unit-cost operations. Here n is the size of the input, which will mainly consist in the size of the database. The main consequence of this choice is the following: we can perform arithmetic operations on integers of size at most n^k (thus encoded over $\mathcal{O}(k\mathsf{log}(n))$ bits) in time polynomial in k only. In particular, it is known that addition can be performed in time $\mathcal{O}(k)$ (using the usual addition algorithm on numbers represented in base $\mathsf{log}(n)$) and both multiplication and division can be performed in time $\mathcal{O}(k\mathsf{log}(k))$ using Harvey and Van Der Hoeven's algorithm [HV21]. We will heavily use this fact in the complexity analysis of our algorithms since we will need to manipulate integers representing the size of relations on domain D and k variables, hence, of size at most $|\mathsf{D}|^k$. We will therefore assume that any arithmetic operations here can be performed in time $\mathcal{O}(k\mathsf{log}(k))$.

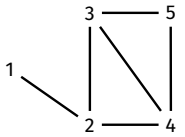
1.2 Graphs, Hypergraphs, Decompositions

Graphs, and more generally hypergraphs, are amongst the most common structures in computer science. Because of their generality and expressive power, they appear in a wide range of domains, from formal language theory to constraint satisfaction and beyond. Moreover, they provide a natural way to model relationships between objects, whether it is pairwise in the case of graphs or between sets of elements in the case of hypergraphs, which makes them a very useful tool in the study of databases. In this section, we introduce the basic notions surrounding graphs and hypergraphs, together with the decomposition techniques that allow us to analyse and reason about their structure. These notions will serve as essential tools throughout the rest of this thesis.

1.2.1 Graphs

A *graph* is a pair $G = (V, E)$, where V is a finite set and $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$. The elements of V are called the *vertices* or *nodes* of G and the elements of E are called the *edges* of G. A graph is often represented as in Example 1.1.

▶ **Example 1.1**
A simple example of a graph could be:



In this case, $V = \{1, 2, 3, 4, 5\}$ and $E = \{\{1, 2\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$.

With this definition, graphs are said to be *undirected*. Edges are considered as sets $\{u, v\}$ of vertices of cardinality 2. It is natural to then introduce a variant of this definition where the edges are *directed*, that is, they are represented as pairs (u, v) in which the first component (here, u) is the source of the edge and the second (here, v) is its destination. A graph defined in this way is said to be *directed*. Notice that this implies an orientation of the graph.

The *neighbourhood* of a vertex $v \in V$ in a graph G, denoted $\mathcal{N}_G(v)$ is defined as the set of nodes that are connected to v by an edge $e \in E$. When the graph is made obvious from the context, we abuse the notation slightly and denote $\mathcal{N}(v)$. Formally, $\mathcal{N}_G(v) = \{u \in V \mid \{v, u\} \in E\}$. This notion is also referred to as the *open neighbourhood* of a vertex. We can lift this notation to sets of vertices, that is $\mathcal{N}_G(W) = \bigcup_{w \in W} \mathcal{N}_G(w)$. However, when considering a set of vertices W, this neighbourhood could contain elements of W. Thus, for a set of vertices W, the open neighbourhood is defined as the set of neighbours of elements of W that are outside of W, that is: $\mathcal{N}^*_G(W) = \bigcup_{w \in W} \mathcal{N}_G(w) \setminus W$. The *degree* of a vertex v, denoted by $\mathsf{deg}(v)$, is the size of its open neighbourhood. The degree of a graph G is the maximal degree of its vertices. For directed graphs, we can define similar notions of neighbours: *out-neighbours* and *in-neighbours*. A vertex v is said to be an out-neighbour (respectively in-neighbour) of a vertex u if there exists an edge (u, v) (respectively an edge (v, u)).

A *path* of length k from a vertex u to a vertex v, is a sequence of distinct vertices $(x_1, \ldots, x_{k+1})$ with $x_1 = u, x_{k+1} = v$ and $\{x_i, x_{i+1}\} \in E$ for all $i \leqslant k$. In directed graphs, a *directed path* of length k from a vertex u to a vertex v is a sequence of distinct vertices $(x_1, \ldots, x_{k+1})$ with $x_1 = u, x_{k+1} = v$ and $(x_i, x_{i+1}) \in E$ for all $i \leqslant k$. A *cycle* of length k is a sequence $(x_1, x_2 \ldots, x_k, x_1)$ of vertices where $x_2 \neq x_k$ (that is, there are at least 3 distinct vertices in the cycle), $(x_1, x_2 \ldots, x_k)$ is a path of length $k-1$ and there exists an edge between x_k and x_1. Cycles can also be seen as paths of length $k > 2$ between u and u. A graph G is said to be *acyclic* if it contains no cycle. In Example 1.1, the sequences $(1, 2, 4), (1, 2, 3, 4)$ and $(1, 2, 3, 5, 4)$ are all paths from 1 to 4 of different lengths and the sequences $(3, 4, 5, 3)$ and $(2, 3, 5, 4, 2)$ are both cycles, of lengths 3 and 4 respectively. A *clique* is a graph such that for any pair $(u, v) \in V^2, u \neq v$, $\{u, v\} \in E$, that is, any pair of vertices is connected by an edge.

A graph is said to be *connected* if for every pair $(u, v) \in V^2$, there exists a path from u to v.

A *connected component* of a graph is subset of vertices $W \subseteq V$ such that for every $u \in W$ and $v \in V$, there exists a path from u to v if and only if $v \in W$. Again, we can consider the same notions for directed graphs by imposing that the edges between any u and v are directed as such. Moreover, in directed graphs, we can introduce the notion of *strongly connected components* as a subset of vertices forming a connected component, but where we also have that every vertex is connected to any other vertex. More formally, a subset $W \subseteq V$ is a strongly connected component if there is a directed path in each direction between each pair of vertices $(u, v) \in W^2$.

For a subset $W \in V$ of vertices, the *subgraph of G induced by W* is the graph defined by $G[W] = (W, \{\{u, v\} | u \in W, v \in W, \{u, v\} \in E\})$, that is, the graph obtained by keeping only the vertices from W and the edges from the original graph that connect vertices from W.

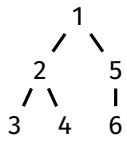
1.2.2 Trees

Trees are a class of graphs with interesting additional properties. A *tree* is defined as a connected and acyclic graph. In a tree, we call a vertex of degree 1 a *leaf*. The set of leaves of a tree T is denoted by $\mathsf{leaves}(T)$. Since a tree T is acyclic and connected, for any pair of vertices (u, v), there exists one and only one path from u to v such that all the vertices composing it are distinct.

A *rooted tree* is a pair between a tree T and a special vertex of $r \in T$, called *root*. For any vertex v that is not the root, the *parent* of v is the vertex that appears immediately before v on the path from the root to v. Similarly, for any vertex v, the *children* of v are the neighbours of v that are not their parent. If no vertex from T has more than 2 children, then T is said to be a *binary tree*. This notion of parenthood between the vertices of the tree induces a partial order on the vertices of the tree. If v is a descendant of u (that is, v is a child of u or a descendant of one of the children of u), then we write $u \prec v$. This induced order is not total since, without additional information about the ordering over the children of a same vertex, we cannot compare between vertices that are not on a same path from the root to the leaf. Also, from the connectedness and acyclicity of the tree, there always exists a unique path from the root to any other node of the tree. Contrary to real life, trees are often represented growing down from the root, such as in Example 1.2.

A *subtree* of a tree T is defined as a subgraph of T that is also a tree. For a node $v \in T$, the *subtree of T rooted in v*, denoted T_v is the subtree induced by v and its descendants, where v is the new root. A rooted tree T can therefore be seen as the union of its root and the subtrees rooted in each child of the root.

▶ **Example 1.2**
An example of a tree T, rooted in 1:



In this example, the set of leaves is $\mathsf{leaves}(T) = \{3, 4, 6\}$. Node 2 is the parent of nodes 3 and 4, and nodes 2 and 5 are the children of the root 1. Notice that, since no node in T has more than two children, T is also an example of a binary tree.

The tree structure induces a partial order such that $1 \prec 6$ or $1 \prec 2$ for example but where

we cannot compare 2 and 6 directly.

1.2.3 Measures on graphs

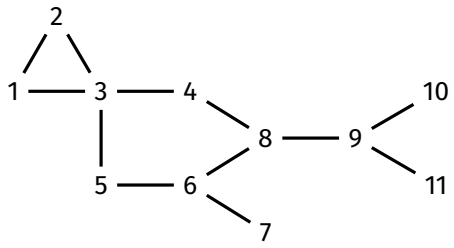
Graphs can vary widely in structure, from simple or sparse trees such as in Example 1.2 to denser, more complex graphs with many connections. One could assume that the complexity of working with a graph is closely linked to the number of vertices and edges. While this is true to some extent, the complexity of a graph has more to do with its structure, that is, the way these vertices and edges are connected to one another. We therefore need a way to reason about this structure in a comprehensive way. Typically, this is done by introducing a numerical measure of the graph, known as its *width*. In this section, we review these classical notions and highlight their relevance as structural complexity parameters for graphs.

Trees are graphs with restrictions that make them structurally simple, since they are acyclic and connected. A starting point to understanding the general complexity of a graph's structure could thus be to find out how *different* it is from a tree. Enter the notion of treewidth, capturing just this concept, which was introduced by Robertson and Seymour [RS83; RS86] and later by Bodlaender [Bod93]. This notion is based on the tree decomposition of the graph, which can be defined as:

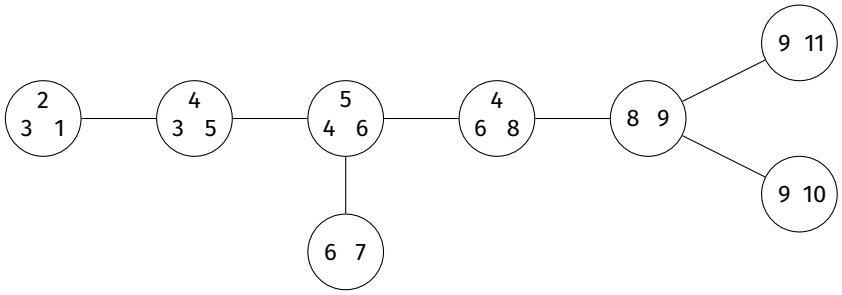
▶ **Definition 1.3** (Tree decomposition, [RS83; RS86; Bod93])
A *tree decomposition* of a graph $G = (V, E)$ is a pair (T, λ), where T is a tree and λ is a function that labels each vertex $t \in T$ by a subset $\lambda(t)$ of vertices from V, called a *bag*. The set union of all the bags is equal to V. The tree and the bags also respect the following conditions:

1. **Connectedness:** for any $v \in V$, the vertices $t \in T$ such that $v \in \lambda(t)$ form a subtree.
2. **Completeness:** for every edge $e \in E$, there exists a bag covering e. Formally, there exists $t \in T$ such that $e \subseteq \lambda(t)$.

▶ **Example 1.4** (A tree decomposition [Bod93])
In this example, we show an example of a (moderately) more complex graph and one of the possible tree decompositions for this graph.



(a) A graph G



(b) A tree decomposition of G

Note that every graph has at least one tree decomposition: the tree containing one node labelled by V. However, they may be multiple different possible decompositions, and we need a way to compare them. From this definition of tree decomposition, we can define the treewidth.

▶ **Definition 1.5** (Treewidth)
For a graph $G = (V, E)$ and a tree decomposition of (T, λ) of G, the *treewidth* of T, denoted by $\mathsf{tw}(T, \lambda)$, is the size of the largest bag of T minus one. Formally, $\mathsf{tw}(T, \lambda) = \mathsf{max}_{t \in T} |\lambda(t)| - 1$.

The treewidth of a graph G, denoted $\mathsf{tw}(G)$, is the minimal treewidth over all the possible tree decompositions of G.

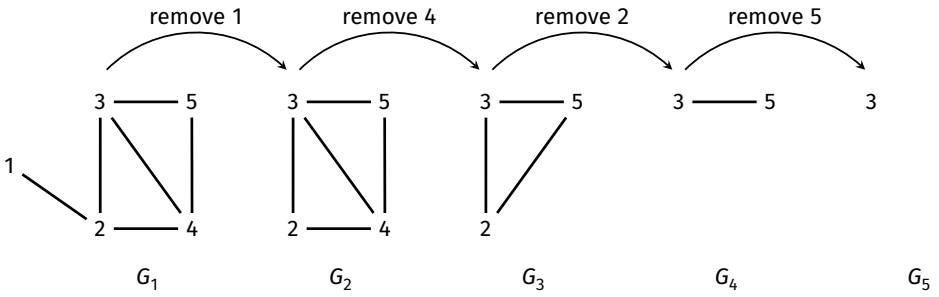
From this definition, trees will have treewidth 1, since we can build a tree decomposition where each node contains an edge of the tree, thus having a bag size of 2. The tree decomposition presented in Example 1.4 **(b)** has a maximum bag size of 3 and thus, a width of 2. Cycles are *close* to being trees: by removing only one edge, we get a path, which has treewidth 1. In fact, cycles have treewidth 2 because of this.

▶ **Remark 1.6**
Notice that, on the other hand, cliques are highly connected: a k-clique has treewidth $k-1$, it is therefore *far* from being a tree. This is due to the fact that, from the completeness and connectedness properties, every clique has to be contained in one bag. In Example 1.4, this is the case for the $1 — 2 — 3$ triangle for example.

Tree decompositions are a standard way of defining various graph parameters. However, we can also define most of these parameters from elimination orders on the vertices. For a graph $G = (V, E)$ and a vertex $v \in V$, we define the graph G/v as the graph obtained by removing v and connecting its neighbourhood by a clique. Elimination orders were defined by Bodlaender [Bod06], but we will prefer the following definition from [Cap16] which also includes the notion of width for the order.

▶ **Definition 1.7** (Elimination Order, [Bod06; Cap16])
Let $G = (V, E)$ be a graph. An *elimination order* of width k for G is an ordering $(x_1, \ldots, x_n)$ of the vertices of G such that the degree of x_i in G_i is at most k where $G_1 = G$ and $G_{i+1} = (G_i/x_i)$ for all $i \leqslant n-1$.

▶ **Example 1.8** (Elimination order of a graph)
The following is an example of the elimination order $\pi = (1, 4, 2, 5, 3)$ applied over the graph from Example 1.1.



Notice how the edge $(2, 5)$ is created between G_2 and G_3 when we remove 4 and connect its neighbourhood by a clique.

The degrees of the vertices when removed are: $\deg_{G_1}(1) = 1$, $\deg_{G_2}(4) = 3$, $\deg_{G_3}(2) = 2$, $\deg_{G_4}(5) = 1$, $\deg_{G_5}(3) = 0$. This elimination order is therefore of width $k = 3$, since the degree of any x_i in G_i is at most 3.

Bodlaender has proved the correspondence between the treewidth and the width of an elimination order [Bod06].

▶ **Theorem 1.9** ([Bod06, Theorem 1])
Let G be a graph. G has an elimination order of width k if and only if G has a tree decomposition of treewidth k.

Proof. Let T be a tree decomposition of G of width k. We now want to show that this implies that G has an elimination order of width k. In order to do this, we show that there exists a vertex v_1 in G of degree at most k such that G/v_1 has a treewidth of at most k. Consider a leaf t of T: if the bag $\lambda(t)$ at t is contained in the bag of its parent, then we can remove t and T remains a valid tree decomposition. This can be repeated while possible. At some point, we will obtain a new tree decomposition T' of G where there is a leaf t such that $\lambda(t)$ is not contained in the bag of the parent of t. This implies that there is a vertex $v_1 \in \lambda(t)$ that does not appear in the preceding nodes, and thus, by the connectedness property, nowhere else in T. By the completeness property, we know that every edge is covered by some bag. This implies that all the neighbours of v_1 are also included in $\lambda(t)$. However, the tree decomposition is of width at most k, so $|\lambda(t)| \leqslant k+1$, which in turns implies that v_1 is of degree at most k. Furthermore, we have that since $\mathcal{N}(v_1) \subseteq \lambda(t)$, the tree decomposition obtained by removing v_1 from $\lambda(t)$ is valid for G/v_1. The elimination order of width k can be shown by induction on the number of vertices, since we now search for an elimination order of width k in G/v_1 and so forth.

The second part of the proof consists of building a tree decomposition of width k from an elimination order $(v_1, \ldots, v_n)$. Again, we can work inductively. Suppose T_{i+1} is a tree decomposition of width k of G_{i+1}. G_{i+1} is built from G_i by connecting the neighbourhood of v_i with a clique, which implies that there exists a node in $t \in T_{i+1}$ such that $\lambda(t)$ contains every vertex from $\mathcal{N}_{v_i}(G_i)$. T_i is then built by connecting to t in T_{i+1} a new leaf t' with $\lambda(t') = \mathcal{N}_{G_i}(v_i) \cup \{v_i\}$. We must now show that T_i is indeed a tree decomposition of G_i and has a treewidth of at most k. Each bag of T_{i+1} contained at most $k+1$ vertices, thus each bag in T_i except for $\lambda(t')$ contains at most $k+1$ vertices also. We know from the width of the elimination order that v_i is of degree at most k in G_i, therefore $\lambda(t')$ is also of size at most $k+1$. We now show that all the edges of G_i are covered in T_i. All edges (u, v) of G_i where both u and v are different from v_i were covered in T_{i+1} and still are. All edges containing v_i are by definition covered by $\lambda(t')$. Finally, we show that T_i is connected: v_i is clearly connected since it is only present in $\lambda(t')$. If a vertex v is not in the neighbourhood of v_i, then it stays connected since it was connected in T_{i+1}. If $v \in \mathcal{N}_{G_i}(v_i)$, then it was connected in T_{i+1} and appears in $\lambda(t)$ and $\lambda(t')$. Since t and t' are neighbours, all vertices remain connected. □

Note that for any n-cycle, removing one of vertices and replacing its neighbourhood by a clique induces an $(n-1)$-cycle. Since the degree of all the vertices in a cycle is 2, all elimination orders over a cycle have width 2, and therefore we have a treewidth of 2.

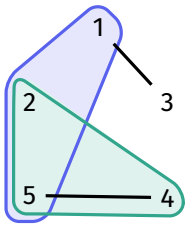
1.2.4 Hypergraphs

Graphs are a useful structure to represent pairwise connections between elements, such as a friend network for example. However, sometimes one would like to represent connections between more than two elements. Suppose for example that we want to model the relationships between authors of academic papers. If a paper could only have been written by a single author or a pair of authors, then we could use a graph, where the vertices would represent the authors and the edges the papers. However, sometimes more than two people collaborate on a given paper. In

this case, we need to be able to link multiple authors (vertices) to one paper (edge)[1].

In order to solve this kind of problem, Berge introduced the notion of hypergraph as a generalisation of graphs [Ber73; Ber87]. A *hypergraph* $\mathcal{H} = (V, E)$ is a pairing between a set of vertices V and a set of sets E. A set $e \in E$ is called a *hyperedge*, or when the hypergraph context is clear, simply an edge. The set of vertices of $\mathcal{H}$ is defined as the set union of the edges, that is $V(\mathcal{H}) = \bigcup_{e \in \mathcal{H}} e$. We define the *rank* of a hyperedge e as the number of vertices in e. We show a simple example of a hypergraph in Example 1.10.

▶ **Example 1.10**
Let $\mathcal{H}$ be a hypergraph such that $\mathcal{H} = (\{1,2,3,4,5\}, \{\{1,3\},\{1,2,5\},\{2,4,5\},\{4,5\}\})$. $\mathcal{H}$ has two hyperedges of rank 3 and two edges of rank 2. $\mathcal{H}$ can be represented graphically as such:



In this example, the $\{1,2,5\}$ edge is represented coloured in blue and the $\{2,4,5\}$ edge is represented coloured in green. Regular edges (between two nodes) are represented in the same way as in Example 1.1.

Some of the notions we defined over graphs translate quite naturally in the context of hypergraphs. The *neighbourhood* of a vertex v in a hypergraph $\mathcal{H}$, denoted by $\mathcal{N}_{\mathcal{H}}(v)$ is defined as the set of vertices that are connected to v by a hyperedge $e \in \mathcal{H}$, $\mathcal{N}_{\mathcal{H}}(v) = \{w \in V(\mathcal{H}) \mid \exists e \in \mathcal{H}, \{v, w\} \subseteq e\}$. The *open neighbourhood* is defined similarly for hypergraphs. A *path* of length k from u to v is a sequence of distinct vertices $(x_1, \ldots, x_{k+1})$, with $x_1 = u, x_{k+1} = v$ and such that, for all $1 \leqslant i \leqslant k$, there exists an edge $e_i \in \mathcal{H}$ with $x_i \in e_i, x_{i+1} \in e_i$. A hypergraph is said to be *connected* if for every pair $(u, v) \in V(\mathcal{H})^2$, there exists a path from u to v. A *cycle* can still be seen as a path of length $k > 2$ from u to u, but in the context of hypergraphs, more relevant definitions exist, as we will see in Section 1.2.5. A hypergraph $\mathcal{H}'$ is a *subhypergraph* of $\mathcal{H}$ if $\mathcal{H}' \subseteq \mathcal{H}$. For a subset of vertices $V' \subseteq V(\mathcal{H})$, the *hypergraph induced by* V' is defined as $\mathcal{H}[V'] = \{e \cap V' \mid e \in \mathcal{H}\} \setminus \{\emptyset\}$. To lighten the notations, we will sometimes omit the set notation when introducing induced hypergraphs if we can easily identify the vertices. For instance, $\mathcal{H}[v_1, v_2] = \mathcal{H}[\{v_1, v_2\}]$.

For a hypergraph $\mathcal{H} = (V, E)$ and a subset of vertices $S \subseteq V$, we define the *cover number* of S with respect to $\mathcal{H}$, denoted $\mathsf{cn}_{\mathcal{H}}(S)$ to be the minimal number of edges of $\mathcal{H}$ needed to cover all vertices from S. More formally, $\mathsf{cn}_{\mathcal{H}}(S) = \mathsf{min}\{|F| \mid F \subseteq E, S \subseteq \bigcup_{f \in F} f\}$.

This can also be seen as the optimal solution of the following integer linear program:

[1] In this (very) simplified model, suppose that authors get bored easily and would never write more than one paper with exactly the same people.

$$\begin{aligned} &\min \sum_{e \in E} x_e \\ s.t. &\forall e \in E, 0 \leqslant x_e \leqslant 1, \\ &\forall s \in S, \sum_{e \mid s \in e} x_e \geqslant 1 \end{aligned}$$

When referring to this linear program, we usually call the x_e values the *weights* over the edges.

The *cover number of a hypergraph* $\mathcal{H}$ is defined as the minimal number of edges needed to cover the whole vertex set. The *fractional cover number* of S with respect to $\mathcal{H}$, denoted by $\mathsf{fcn}_{\mathcal{H}}(S)$, is a relaxation of the linear program for the cover number to rational numbers. In this case, we look for the optimal value over $\mathbb{Q}$. To avoid overcharging the notations, when the considered hypergraph is clear from the context, we omit it from the notation.

▶ **Example 1.11** (Covering the vertices with edges)
We will show a quick example of the computation of cover numbers for a simple hypergraph.

Consider the following triangle hypergraph Δ, defined on vertices $V = \{1, 2, 3\}$, such that all vertices are connected.

To find the cover number of Δ, we solve the linear program. We could assign a weight of 1 to each edge: in this case, all the vertices are covered, and we have $\mathsf{cn}(\Delta) \leqslant 3$. Another solution is to assign a weight of 1 to two distinct edges: in this case, all the vertices are still covered by a weight of at least 1, and we have $\mathsf{cn}(\Delta) \leqslant 2$. If we assign a weight of 1 to only one edge, then we will have one vertex that is not covered. The cover number of Δ is therefore 2, since you need at least two edges to cover V.

Suppose we relax the constraints on the weights to use fractional numbers. Since each vertex has degree 2, we could imagine assigning a weight of $1/2$ to each edge. This keeps each vertex covered by a weight of 1 exactly, but reduces the fractional cover number to $3/2$.

To give some intuition behind these measures, let's start by considering a simple case over a graph. Suppose that you have a list of tasks to do (that will be the vertices of the graph), and a list of workers that can work on two different tasks simultaneously (as such, they will be represented by the edges of the graph). What we want to do is find the minimal number of workers we have to hire so that each task is fully completed.

Let's first consider we can only hire the workers for full-time contracts (0 or 1). For a task

graph such as Δ from Example 1.11, we could of course hire all three workers. In this case, all the tasks would be fully completed. In fact, they would even be completed at 200% each, since two full-time workers would be doing each task. By only selecting two of the workers, we reduce the total employment load, but still complete all of the tasks. Some tasks might still be completed at more than 100%, but in this case, we cannot do better. This number is the cover number of our task graph.

Now suppose that we can offer part-time contracts to our workers (between 0 and 100%). In this case, we will have some tasks that are completed partly by one worker and partly by another. While we could technically still employ two workers for full-time contracts, this is no longer minimal, and by readapting the work load, we could be more efficient. In the last case of Example 1.11, what happens is that we offer all three of our workers a 50% time contract. All the tasks are still fully completed, however, the "total time" used is only 1.5 full-time equivalents (0.5 per worker). In this optimal solution, we are computing the fractional edge cover of Δ, which is the number of equivalent full-time contracts needed to fully complete all the tasks.

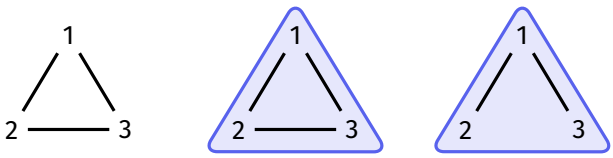
When dealing with hypergraphs, the intuition stays the same, but this time, we that a worker is no longer limited to 2 tasks, but could work on any number of them simultaneously.

1.2.5 Hypergraphs: Acyclicity and Measures

In Section 1.2.3, we introduced a notion of width for graphs as a way of characterising their structure. A natural next step is to extend these notions to hypergraphs. In this section, we will focus on presenting first a generalisation of the notion of acyclicity over hypergraphs and then a decomposition notion: the hypertree width.

The notion of acyclicity is not as trivial to explain in the context of hypergraphs than for graphs, since the notion of cycle is not necessarily as intuitive. Multiple *degrees* of acyclicity have been defined, notably by Fagin [Fag83]. However, a minimal requirement for any notion of hypergraph acyclicity is that when restricted to graphs, the notion should coincide with the usual definition of acyclicity for graphs. Example 1.12 shows multiple hypergraphs. If the first of those is intuitively cyclic, the rest can be seen as acyclic depending on the considered degree of acyclicity. A more complete introduction to these notions can be found in [Dur09] and a complete overview of the different notions, their properties and of known results on hypergraph acyclicity can be found in [Bra17].

▶ **Example 1.12** (Cycles? Where?)
Each of the hypergraphs presented below contain a path from vertex 1 to vertex 3 that could be considered as a cycle for at least one hypergraph acyclicity notion. For the leftmost graph, it is simply a cycle as defined in Sections 1.2.1 and 1.2.3, but it is a little trickier for the two other graphs. Notably, both the second and third hypergraphs can be considered acyclic.

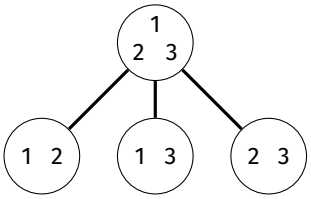


α-acyclicity. A very general notion of acyclicity is α-acyclicity, which was introduced by Yannakakis [Yan81]. This notion was later extended in [Fag83; Bee+83]. The idea of α-acyclicity is inspired by tree decompositions and uses the concept of a join tree, defined formally as follows:

▶ **Definition 1.13** (Join tree)
A *join tree* of a hypergraph $\mathcal{H}$ is a special tree decomposition (T, λ) of $\mathcal{H}$, where λ is a one-to-one mapping from the vertices of T and $\mathcal{H}$.

A hypergraph $\mathcal{H}$ is said to be *α-acyclic* if it has a join tree, which can be counterintuitive, such as shown in Example 1.14. If we consider only graphs, then the existence of a join tree is equivalent to the existence of tree decomposition of treewidth 1. This implies that when restricting the study to graphs, α-acyclicity collapses with standard graph acyclicity.

▶ **Example 1.14**
A join tree for the second hypergraph of Example 1.12 could be:



Note that this implies that this hypergraph is α-acyclic.

The notion of α-acyclicity can also be defined in terms of elimination orders, in a similar fashion as in Section 1.2.3. To do this, we can generalise the algorithm that consists in eliminating leaves from a tree. Given a hypergraph $\mathcal{H}$, we call a vertex $v \in V(\mathcal{H})$ an *α-leaf* of $\mathcal{H}$ if there exists an edge $e \in \mathcal{H}$ such that $\{v\} \cup \mathcal{N}_{\mathcal{H}}(v) \subseteq e$. That is, that v and its neighbourhood are contained in e. An *α-elimination order* for $\mathcal{H}$ is then an elimination order $(v_1, \ldots, v_n)$ of $V(\mathcal{H})$ such that, for every $1 \leqslant i \leqslant n$, v_i is an α-leaf of $\mathcal{H}[v_i, \ldots, v_n]$.

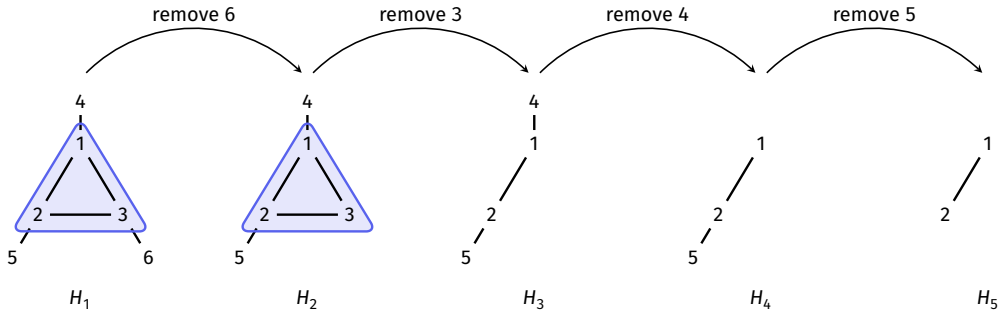
The link between the two notions can be formalised as such:

▶ **Theorem 1.15** ([Bee+83, reformulated] or [Bra17, Corollary 5 and Section 2.1])
A hypergraph $\mathcal{H}$ is α-acyclic if, and only if, there exists an α-elimination order for $\mathcal{H}$.

The notion of α-acyclicity seems simple to grasp, but can be counterintuitive, such as shown in the second hypergraph of Example 1.12 or in Example 1.16. Indeed, this notion is not hereditary: a hypergraph $\mathcal{H}$ can be α-acyclic even if there exists a subhypergraph $\mathcal{H}' \in \mathcal{H}$ that is

not. This is the case in both our examples, where, by removing one edge only, we get a triangle graph. In fact, we can make any hypergraph α-acyclic by simply adding a hyperedge covering all of the vertices. In this case, a join tree consists of a root node labelled by this hyperedge, connected to nodes labelled each with one edge of $\mathcal{H}$.

▶ **Example 1.16** (α-elimination order)
The following is an example of the α-elimination order $\pi = (6, 3, 4, 5, 1, 2)$ over a simple hypergraph:



In the original hypergraph, the vertex 6 is an α-leaf, since all of its neighbourhood is covered by one edge that also contains 6 (the edge $\{3, 6\}$). Symmetrically, this is also the case for 4 and 5. Since it is an α-leaf, we can remove it.

In the following hypergraph, 4 and 5 are still α-leaves, but now, 3 is also an α-leaf, since the edge $\{1, 2, 3\}$ covers all the neighbours and 3. We choose to remove 3 here.

From then, since we basically have a path from 4 to 5, we can remove at each step one of the extremities of the path. Here, we select 4, then 5. The removal steps for 1 and 2 are not pictured but we can remove them by the same observation.

Since, there is an α-elimination order for this hypergraph, we know from Theorem 1.15 that it is α-acyclic.

β-acyclicity. This observation that α-acyclicity is non-hereditary creates a need for another notion of acyclicity, that aims to fix this counter-intuitive aspect of α-acyclicity. In comes β-acyclicity, that imposes that any subhypergraph must be α-acyclic.

▶ **Definition 1.17** (β-acyclicity)
A hypergraph $\mathcal{H}$ is *β-acyclic* if, and only if, for every subhypergraph $\mathcal{H}' \subseteq \mathcal{H}$, $\mathcal{H}'$ is α-acyclic.

Once again, when restricted to graphs, this reduces to regular acyclicity, since any subgraph of an acyclic graph is also acyclic. In Example 1.12, the second hypergraph is α-acyclic but not β-acyclic, since it contains the triangle as a subhypergraph. However, the third hypergraph is β-acyclic.

Other equivalent characterisations of β-acyclicity have been given, one notably using elimination orders. Given a hypergraph $\mathcal{H}$, a *β-leaf* of $\mathcal{H}$ is a vertex v such that $\mathcal{H}_v = \{e \in \mathcal{H} \mid v \in e\}$ is ordered by inclusion, that is $\mathcal{H}_v = \{e_1, \dots, e_m\}$ with $e_1 \subseteq \dots \subseteq e_m$. A *$\beta$-elimination order* for $\mathcal{H}$ is then an elimination order $(v_1, \dots, v_n)$ such that, for every $1 \leqslant i \leqslant n$, v_i is a β-leaf of $\mathcal{H}[v_i, \dots, v_n]$. Once again, the link between the two notions can be formalised as such:

▶ **Theorem 1.18** ([Bee+83, reformulated] or [Bra17, Corollary 5 and Section 2.1])
A hypergraph $\mathcal{H}$ is β-acyclic if, and only if, there exists a β-elimination order for $\mathcal{H}$.

In Example 1.12, the second example does not contain any β-leaves. Indeed, the vertex 1 (and symmetrically for 2 and 3) for example belongs to the edges $\{\{1,2\},\{1,3\},\{1,2,3\}\}$ and this set cannot be ordered by inclusion. By removing one vertex such as in the third example however, 2 belongs to $\{\{1,2\},\{1,2,3\}\}$ which is ordered by inclusion, thus 2 is a β-leaf. If we remove 2 from the hypergraph, then the two remaining vertices both are β-leaves, thus the hypergraph is β-acyclic.

More different levels of hypergraph acyclicity have been defined over the years such as γ-acyclicity or Berge-acyclicity, but we will not develop these further in this manuscript. The interested reader can refer to Brault-Baron's survey [Bra17] for more details.

Width measures. An important tool to measure the complexity of a graph's structure is the notion of width. Gottlob, Leone, and Scarcello introduced the notions of hypertree width and generalised hypertree width [GLS99; GLS02]. We will present these notions here, but a more detailed overview of their applications can be found in [GLS01].

The tree width came from the decomposition of the graph into a tree. The idea will be to use the same principle for hypergraphs. In order to do this, we define the generalised hypertree decomposition.

▶ **Definition 1.19** (Generalised Hypertree Decomposition (GHD))
Let $\mathcal{H}$ be a hypergraph. A *generalised hypertree decomposition* (GHD for short) of $\mathcal{H}$ is a pair (T, λ), where T is a tree and λ a labelling function that associates to each node of T a subset of the vertices of $\mathcal{H}$. The decomposition also respects the following conditions:

1. **Completeness:** for every edge $e \in \mathcal{H}$, there exists a node $t \in T$ such that $e \subseteq \lambda(t)$; and
2. **Connectedness:** for every vertex of $\mathcal{H}$, the set $\{v \mid x \in \lambda(v)\}$ forms a connected subtree.

We can use this decomposition to define generalised hypertree width:

▶ **Definition 1.20** (Generalised Hypertree Width)
Let $\mathcal{H}$ be a hypergraph and (T, λ) a generalised hypertree decomposition of $\mathcal{H}$. If, for a given integer k and for every vertex t of T, we can find a subset of edges $S \subseteq \mathcal{H}$ such that $|S| \leqslant k$ and $\lambda(t) \subseteq \bigcup_{e \in S} e$, then T is said to be of *generalised hypertree width* at most k.

We denote by $\mathsf{ghtw}(T)$ the smallest k such that T is of generalised hypertree width k. The generalised hypertree width of $\mathcal{H}$, denoted $\mathsf{ghtw}(\mathcal{H})$, is the minimal generalised hypertree width of generalised hypertree decompositions of $\mathcal{H}$.

In the same way as for treewidth, we can use elimination orders to characterise (fractional) hypertree width.

Given a hypergraph $\mathcal{H} = (V, E)$ and a vertex $v \in V$, we will denote by $\mathcal{H}/v$ the hypergraph obtained by removing v from the set of vertices and adding a hypereedge consisting of its neighbourhood. More formally, $\mathcal{H}/v = (V \setminus \{v\}, E[V \setminus \{v\}] \cup \mathcal{N}_{\mathcal{H}}(v))$. For an order $(v_1, \dots, v_n)$ on the vertices of a hypergraph $\mathcal{H}$, we consider the family $(\mathcal{H}_i)_{1 \leqslant i \leqslant n+1}$ defined inductively as: $\mathcal{H}_1 = \mathcal{H}, \mathcal{H}_{i+1} = \mathcal{H}_i/v_i$. This allows us to define the following concepts:

▶ **Definition 1.21** (Hyperorder Width)
Given a hypergraph $\mathcal{H} = (V, E)$ and an elimination order $\prec = (v_1, \dots, v_n)$ over the vertices of $\mathcal{H}$, the *hyperorder width of* $\prec$ is defined as:

$$\mathsf{how}(\mathcal{H}, \prec) = \mathsf{max}_{i \leqslant n} \mathsf{cn}_{\mathcal{H}}(\mathcal{N}_{\mathcal{H}_i}(v_i))$$

that is, we look at the neighbourhood of v_i in $\mathcal{H}_i$, and compute the cover number of this set of vertices in the original hypergraph. The maximal value over all vertices is the hyperorder width of $\prec$.

The *hyperorder width* of $\mathcal{H}$, denoted $\mathsf{how}(\mathcal{H})$ is the minimum over every possible elimination order $\prec$ over V of $\mathsf{how}(\mathcal{H}, \prec)$.

This definition can easily be lifted to fractional values:

▶ **Definition 1.22** (Fractional Hyperorder Width)
Given a hypergraph $\mathcal{H} = (V, E)$ and an elimination order $\prec = (v_1, \dots, v_n)$ over the vertices of $\mathcal{H}$, the *fractional hyperorder width of* $\prec$ is defined as:

$$\mathsf{fhow}(\mathcal{H}, \prec) = \mathsf{max}_{i \leqslant n} \mathsf{fcn}_{\mathcal{H}}(\mathcal{N}_{\mathcal{H}_i}(v_i))$$

The *fractional hyperorder width* of $\mathcal{H}$, denoted $\mathsf{fhow}(\mathcal{H})$ is the minimum over every possible elimination order $\prec$ over V of $\mathsf{fhow}(\mathcal{H}, \prec)$.

It has already been observed multiple times (see [ANR15] or [Fic+18; Gan+22; ANR16], that, for a hypergraph $\mathcal{H}$, $\mathsf{how}(\mathcal{H})$ and $\mathsf{fhow}(\mathcal{H})$ are respectively equal to the generalised hypertree width and the fractional hypertree width of $\mathcal{H}$ and that there is a natural correspondence between a tree decomposition and an elimination order having the same width. However, to be able to express the tractability results in this thesis as a function of the order, it is more practical to define the width of orders instead of hypertree decompositions[2].

[2]Strictly speaking, the definition of $\mathsf{how}(\cdot)$ and $\mathsf{fhow}(\cdot)$ in [ANR16] differ slightly in that the elimination step of v removes every edge containing v and replace it by the neighborhood of v, where in our definition, we keep them.

1.3 Queries and Databases

This section introduces the logical and theoretical foundations underpinning queries and databases, upon which the rest of the thesis builds. We will work from the ground up, starting with the *data model* composed of tuples, relations, and databases and then move on to the query syntax.

1.3.1 Tuples, Relations and Databases

Databases and their contents can be defined by two standard representations: the *named* perspective and the *unnamed* perspective. Both have their strengths and weaknesses: the named perspective fits better to the way that databases are represented in implemented database systems and thus is clearer when presenting examples. The unnamed perspective, on the other hand, provides an easier mathematical model, which becomes useful when studying the properties of databases. These two perspectives can model the same situations, which means that we will be able to switch between them when convenient, as to always use the most appropriate presentation. In the following definitions, we will present both representations, however we will mainly use the named perspective in this work. For the interested reader, a more complete and formal presentation of these two representations can be found in [Are+22].

Tuples. We define a finite set D, that we will call the *domain*. This set represents the possible values that can appear in our data. In the *unnamed* perspective, a *tuple* τ is an element of D^k for some value $k \in \mathbb{N}$. For a tuple $\tau = \langle \tau_1, \dots, \tau_k \rangle$, we call k the *arity* of the τ.

Conversely, in the *named* perspective, a tuple is a mapping from a *variable* set X to D, that is, $\tau \colon X \to \mathsf{D}$. This can also be seen as τ being an element of D^X. Given a particular variable $x \in X$, we denote by $\tau(x)$ the value of x in τ. In this case, a tuple τ over variables $X = \{x_1, \dots, x_n\}$ where, for all $i \in [n]$, $\tau(x_i) = d_i$ with $d_i \in \mathsf{D}$ will be represented as $\tau = \langle x_1 \leftarrow d_1, \dots, x_n \leftarrow d_n \rangle$. The *empty tuple*, the only element of $\mathsf{D}^\emptyset$ is denoted by $\langle\rangle$. The main difference is that we are now *naming* the attributes of the tuple, which allows us to reference them by name and ease notations. In this case, a tuple can also be seen as a *record* of values assigned to different attributes, which is a common way for visualising databases. In the unnamed perspective, for tuples of arity k, we could see the set of variable X as the set of integers corresponding to positions $[k]$.

Sometimes however, one is not interested in the valuation of the tuple over the full set of variables (in the named perspective), but rather in a subset of those. For a tuple τ and a subset of variables $Y \subseteq X$, we denote by $\tau_{|Y}$ the *restriction* of τ over Y. Formally, $\tau_{|Y}$ is the tuple of D^Y such that $\tau_{|Y}(y) = \tau(y)$ for every $y \in Y$. Notice that this translates naturally to the unnamed perspective if we consider a subset of indexes $Y \subseteq [k]$ where k is the arity. Let $[k] \setminus Y = \{i_1, \dots, i_p\}$ so that $i_1 < i_2 < \dots < i_p$. In this case, $\tau_{|Y} = \langle \tau_{i_1}, \tau_{i_2}, \dots, \tau_{i_p} \rangle$.

When looking at several tuples, it is often important to be able to compare them. In effect, we compare tuples over their common variables. Two tuples $\tau_1 \in \mathsf{D}^{X_1}$ and $\tau_2 \in \mathsf{D}^{X_2}$ are said to be *compatible* if $\tau_{1|X_1 \cap X_2} = \tau_{2|X_1 \cap X_2}$. This is denoted by $\tau_1 \simeq \tau_2$. In this case, we denote by $\tau_1 \bowtie \tau_2$ the tuple over variables $X_1 \cup X_2$ where $(\tau_1 \bowtie \tau_2)(x)$ is $\tau_1(x)$ if $x \in X_1$ and $\tau_2(x)$ otherwise. When the two variables sets are disjoint, that is $X_1 \cap X_2 = \emptyset$, this is simply the Cartesian product of τ_1 and τ_2 and denoted $\tau_1 \times \tau_2$.

This does not change the notion of neighborhood at each step so it results in the same widths but with a slightly different definition.

Relations A *relation* R of arity k on domain D is a finite subset of all the possible tuples of arity k. In the *named* perspective, R is a set of tuples over the same variable set X, that is, $R \subseteq \mathsf{D}^X$. The size of a relation R is defined to be the number of tuples contained in R and often shorthanded $\#R$. The *active domain* of a relation is the subset of the domain composed solely of the values used in the tuples from the relation.

In a similar way as for tuples, we often work with multiple relations at the same time. When combining multiple relations, we combine the tuples of each relation on their common variables. Given two relations R_1 and R_2 over a domain D and variables X_1 and X_2 respectively, the *natural join* of R_1 and R_2, denoted by $R_1 \bowtie R_2$ is defined as $\{\tau_1 \bowtie \tau_2 \mid \tau_1 \in R_1, \tau_2 \in R_2, \tau_1 \simeq \tau_2\}$. Observe that, in a similar fashion as for tuples, if $X_1 \cap X_2 = \emptyset$, then $R_1 \bowtie R_2$ is simply the Cartesian product of R_1 and R_2 and is denoted as $R_1 \times R_2$. In Example 1.23, we show a quick example of both tuples and relations. In real-life database management systems, relations are also known as *tables*.

Given a subset $Y \subseteq X$ of variables, we denote by $R_{|Y}$ the restriction of R over Y. Formally, $R_{|Y}$ is the set of tuples $\{\tau_{|Y} \mid \tau \in R\}$. For a partial assignment τ, we denote by $R[\tau]$ the subset of tuples in R that are compatible with τ.

Throughout this thesis, we will often assume that both the domain and the variables are ordered. The order on the domain D will be denoted by $<$, and the order on the variables will be denoted by $\prec$. We will often write the domain as $\mathsf{D} = \{d_1, \dots, d_n\}$ with $d_1 < \cdots < d_n$ and the variables as $X = \{x_1, \dots, x_k\}$ with $x_1 \prec \cdots \prec x_k$. Given a value $d \in \mathsf{D}$, we denote by $\mathsf{rank}(d)$ the number of elements of D that are smaller or equal to d. The combination of both the orders on the domain and the variables induces a lexicographical order $\prec_{\mathsf{lex}}$ on D^X. This order is defined as $\tau \prec_{\mathsf{lex}} \kappa$ for two tuples τ and κ if there exists $x \in X$ such that for every $y \prec x, \tau(y) = \kappa(y)$ and $\tau(x) < \kappa(x)$.

Given a logical formula F over the variables X and domain D of a relation $R \subseteq \mathsf{D}^X$, we denote by $\sigma_F(R)$ the subset of R where F is true.

▶ **Example 1.23**

Suppose that we want to represent students. In this simplified model, a tuple representing a student will have an arity of 3. In the named perspective, we can say that a student has a name, is enrolled in a year, and has a major. If we were to represent this as a relation, we could define a variable set $X = \{\mathsf{name}, \mathsf{year}, \mathsf{major}\}$.

In this case, we consider the domain to be infinite, so it covers all the possible names, years and majors.

A tuple could then be:

in the unnamed perspective	in the named perspective
$\tau = \langle \mathtt{Alice}, 3, \mathtt{CS} \rangle$	$\tau = \langle \mathsf{name} \leftarrow \mathtt{Alice}, \mathsf{year} \leftarrow 3, \mathsf{major} \leftarrow \mathtt{CS} \rangle$

The arity of this tuple is 3. Consider a restriction of τ over a subset S:

$S = \{2, 3\}$	$S = \{\texttt{year}, \texttt{major}\}$
$\tau_{\mid S} = \langle 3, \texttt{CS} \rangle$	$\tau_{\mid S} = \langle \texttt{year} \leftarrow 3, \texttt{major} \leftarrow \texttt{CS} \rangle$.

A `Students` relation could be represented in the unnamed perspective as:

$$\begin{aligned}\texttt{Students} = \{&\langle \texttt{Alice}, 3, \texttt{CS} \rangle \\ &\langle \texttt{Bob}, 2, \texttt{CS} \rangle \\ &\langle \texttt{Corentin}, 3, \texttt{Maths} \rangle \\ &\langle \texttt{Daphne}, 1, \texttt{Physics} \rangle\}\end{aligned}$$

In the named perspective, this could be naturally translated in a similar way as above. However, it is often clearer to use a *table* representation in this context. We could therefore translate to:

Students	name	year	major
	Alice	3	CS
	Bob	2	CS
	Corentin	3	Maths
	Daphne	1	Physics

Here, while the domain for the student names was assumed to be all possible names up to 8 letters, the active domain for this variable in this relation is simply the set $\{\texttt{Alice}, \texttt{Bob}, \texttt{Corentin}, \texttt{Daphne}\}$. The size of the relation is $\#\texttt{Students} = 4$. We can also filter the relation according to the value of some variables. For instance:

$\sigma_{\texttt{year} \leqslant 2}(\texttt{Students})$	name	year	major
	Bob	2	CS
	Daphne	1	Physics

Let's add a new relation in the mix. Suppose that we have a `Rooms` relation over variables $\{\texttt{major}, \texttt{room}\}$ like this:

Rooms	major	room
	CS	104
	Maths	206
	French	704

The tuple τ we defined earlier to represent a student is compatible with the tuple $\kappa = \langle \texttt{CS}, 104 \rangle \in \texttt{Rooms}$, but no other tuple representing a room, since no other tuple has $[\texttt{major} \leftarrow \texttt{CS}]$. Notice however that κ is also compatible with another tuple from `Students`: $\langle \texttt{Bob}, 2, \texttt{CS} \rangle$. This means we can join the tuples. We then have $\tau \bowtie \kappa = \langle \texttt{Alice}, 3, \texttt{CS}, 104 \rangle$. We could also join both the relations. This would return:

Students ⋈ Rooms	name	year	major	room
	Alice	3	CS	104
	Bob	2	CS	104
	Corentin	3	Maths	206

Notice that the tuple representing the student named Daphne is not present in the join. This is because this tuple is not compatible with any tuple in the `Rooms` relation.

Remark: Note that the restriction operation is the equivalent of selecting specific fields in the `SELECT` operation, with no restriction being the equivalent of `SELECT *`. The filtering operation is the equivalent of adding a `WHERE` clause.

Databases. The elements we represent with tuples, grouped in relations, can be seen as *data*. Informally, we can define a database as a collection of data, but we will formalise this notion.

Consider a set of relation names Rel. In the unnamed perspective, a *database schema* is a partial function $S\colon \mathsf{Rel} \to \mathbb{N}$ such that the number of elements on which S is defined is finite. The function S maps (a finite number of) relation names to their arity (i.e. the arity of the tuples they contain). A *database instance* for a schema S whose domain is $\mathsf{dom}(S)$ is a function $I\colon R \in \mathsf{dom}(S) \to \mathcal{P}(\mathsf{D}^{S(R)})$, where D is a fixed, possibly infinite, set of constants. The function I maps relation names to sets of tuples of the corresponding arity. A *database* $\mathbf{D}$ is a pair composed of a schema and an instance.

In the named perspective, the database schema S is a function that maps (a finite number of) relation names to sets of attributes (the variables), that is $S\colon \mathsf{Rel} \to \mathcal{P}(X)$. A database instance I for a schema S over a set of variables X is then defined as a collection of relations $R_i \in \mathsf{Rel}$, each over a subset of variables $X_i \subseteq X$. That is, $I\colon R \in \mathsf{dom}(S) \to \mathcal{P}(\mathsf{D}^{S(R)})$. The relations in a database may share some of their attributes. Similarly to the unnamed perspective, in the named perspective, databases are pairs composed with a schema and an instance. However, in practice, as schemas can easily be inferred from instances, we will abuse notations and consider databases as the set of named relations instances we have described. Formally, the database from Example 1.23 would consist in a schema:

$$S\colon \begin{cases} \mathsf{Students} & \to \{\mathsf{name}, \mathsf{year}, \mathsf{major}\} \\ \mathsf{Rooms} & \to \{\mathsf{major}, \mathsf{room}\} \end{cases}$$

and an instance composed of the tuples present in the example. However this is heavy, and thus we prefer writing that the database $\mathbf{D}$ is simply the set $\{\texttt{Students}, \texttt{Rooms}\}$. In that example, you may note that `major` is a variable overlapping both relations.

In a similar way as for relations, for a tuple τ, we denote by $\mathbf{D}[\tau]$ the restriction of all relations in the database to the set of tuples compatible with τ. As an abuse of notation, when the database is clear from the context, we will directly use the name of the relation to refer to the set of tuples contained in the database instance. In particular, for a relation $R \in \mathbf{D}$, we will write $\tau \in R$ to signify that τ is an element of the instance of the relation in the database. We will also use $R[\tau]$ to denote the subset of tuples in R that are compatible with τ.

1.3.2 Queries

Tuples, relations and databases contain information. Traditionally, there is a separation of concerns in databases, between the querying of the data and the way it is actually represented in

memory. This has led to logical querying mechanisms, where, in place of having an algorithm that crawls through the whole database to extract the information, we describe the data we need. This brings logical queries at the heart of data retrieval. In this section, we define the standard notions behind queries.

Consider a set of variables X. An *atom* is an expression of the form $R(\mathbf{x})$, where R is a relation symbol and $\mathbf{x}$ is a subset of variables in X. A *join query* Q is a join of multiple atoms, represented as an expression $Q(\mathbf{x_h}) :\text{-}\ R_1(\mathbf{x_1}), \ldots, R_m(\mathbf{x_m})$, where $\mathbf{x_h} = \bigcup_i \mathbf{x_i}$ represents the *output* variables of Q. Note that the variables of the relations might overlap.

Queries implicitly declare part of the schema of the databases on which they can be used. As they use variable names, these databases should be defined in the named perspective. In particular, the query Q can be used on a database $\mathbf{D}$ with schema S that verifies for every $i \in [m]$, $S(R_i) = \mathbf{x_i}$. Formally, we defined a database instance $\mathbf{D}$ as an assignation to each relation symbol R of a finite relation $R^{\mathbf{D}}$. Throughout this manuscript, when the database is fixed or clear from the context, we abuse notation and simply write R for $R^{\mathbf{D}}$.

In the following definitions, we will assume databases are defined under the *named perspective*. From this point, we will also alleviate the notations a little by omitting the set notation for the variables of the atoms of the query.

In this thesis, we will mostly consider *self-join free* queries, that is, queries in which any relational symbol can appear only once. We denote the set of variables of a query Q by $\mathsf{vars}(Q)$ and the set of atoms by $\mathsf{atoms}(Q)$. The *size* of a query is defined as $\sum_{i=1}^{m} |\mathbf{x_m}|$.

We define the *output of a query* Q, also called the *answers* of Q, over a database $\mathbf{D}$, denoted by $\mathsf{ans}_{\mathbf{D}}(Q)$ to be the set of tuples τ over $\mathsf{vars}(Q)$ whose projections belong to the instances of the relations in the database. Formally, $\mathsf{ans}_{\mathbf{D}}(Q) = \{\tau \in \mathsf{D}^{\mathsf{vars}(Q)} \mid \forall R(\mathbf{x}) \in \mathsf{atoms}(Q).\, \tau_{|\mathbf{x}} \in R\}$. The output of a query can then be seen as a named relation.

In many cases, we will need to refer to tuples that only assign a subset of the variables of the query. While some of these tuples may be extended to form answers to the query, some may not. For a tuple τ defined on a subset $Y \subseteq \mathsf{vars}(Q)$ to be extendable, a necessary condition is that, for all $R(\mathbf{x}) \in \mathsf{atoms}(Q)$, $R[\tau]$ is not empty. In this case, we say that τ is *consistent* with Q and $\mathbf{D}$. When the database is clear from context, we may abuse slightly the notation and simply write that τ is consistent with Q. If a tuple does not meet this criterion, then it is *inconsistent* with Q and $\mathbf{D}$.

Conjunctive queries (often abbreviated CQs) are queries that are built from a join query and *projecting* some of the variables. They usually have the form: $Q(\mathbf{x_h}) :\text{-}\ \exists \mathbf{y}, R_1(\mathbf{x_1}), \ldots, R_m(\mathbf{x_m})$. The variables $\mathbf{x_h}$ are the free variables of the query. They form the *head* of Q. A conjunctive query is said to be *full* if all of its variables are contained in the head. A full conjunctive query is equivalent to a join query as defined above. A *Boolean query* is a conjunctive query where all the variables in the query have been projected out. Boolean queries are used when one simply wants to check for the existence of tuples compatible with the query.

In `SQL`, conjunctive queries correspond to `SELECT ... FROM ... WHERE` queries, where the `WHERE` clause only contains conditions built with the = operator, combined with `and`.

▶ **Example 1.24** (Conjunctive Query)
Consider the database $\mathbf{D} = \{\texttt{Students}, \texttt{Rooms}\}$ from Example 1.23. The following conjunctive query Q over $\mathbf{D}$:

$$Q(\texttt{name}, \texttt{year}, \texttt{major}, \texttt{room}) :\text{-}\quad \texttt{Students}(\texttt{name}, \texttt{year}, \texttt{major}),\ \texttt{Rooms}(\texttt{major}, \texttt{room})$$

is equivalent to the following SQL query:

```
SELECT s.name, s.year, s.major, r.room
FROM Students as s, Rooms as r
WHERE s.major = r.major
```

and will return the following tuples:

Q	name	year	major	room
	Alice	3	CS	104
	Bob	2	CS	104
	Corentin	3	Maths	206

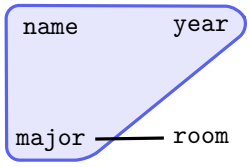
In this example, no variable is existentially quantified.

Comparing the complexity of two queries is not trivial, even over a same database. In order to achieve this, a common reference is to look at the *query hypergraph*. This is a hypergraph constructed from the query, where the vertices are the variables and the edges are the relations. An example of such a hypergraph is presented in Example 1.25. We can then use measures over these hypergraphs to compare queries. For instance, a well-studied class of conjunctive queries consists of *acyclic conjunctive queries*. A conjunctive query Q is said to be acyclic if its hypergraph $\mathcal{H}(Q)$ is α-acyclic. Using the hypergraph of a query will allow us to use some measures from hypergraphs to compare queries. For instance, the fractional cover number quantifies the minimum total weight with which atoms must be combined to ensure that every variable is covered, when atoms may contribute fractionally to the cover.

▶ **Example 1.25** (Query Hypergraph)

From the query of Example 1.24, we can build the following hypergraph.

The blue hyperedge represents the `Students` relation and the simple edge represents the `Rooms` relation.



Going further with queries. In practice, conjunctive queries often appear with additional features beyond basic logical conjunction. Two common extensions are aggregation and functional dependencies. These extensions increase the expressive power of queries and, in many cases, alter the algorithmic complexity of their evaluation. We defer the introduction and formal treatment of these notions to later chapters where necessary.

1.3.3 Complexity Analysis in Query Evaluation

In database theory, computational complexity is often used to classify the difficulty of evaluating a query on a database. However, unlike standard algorithmic settings where the input is a single string or structure, a query evaluation problem involves *two inputs*: the query and the database.

This leads to a natural distinction between two perspectives on complexity: data complexity and combined complexity.

Data complexity. The *data complexity* of a query problem considers the query as *fixed* and measures complexity only as a function of the database size. This reflects the typical setting in database systems, where a small number of queries are executed repeatedly over large and evolving datasets. For instance, evaluating a fixed Boolean conjunctive query over a relational database can be done in logarithmic space in the size of the database [Var82]. As such, data complexity is often used to identify tractable cases relevant for query optimisation.

Combined complexity. The *combined complexity* of a query problem treats both the query and the database as part of the input. This models scenarios where queries themselves can be large or dynamically generated. For example, checking whether a general (non-fixed) conjunctive query has a non-empty answer is NP-complete [CM77]. This reflects the fact that query evaluation subsumes constraint satisfaction problems (CSP), which are known to be computationally hard in general.

These two notions reflect different goals: data complexity is appropriate when studying the performance of a system with a fixed workload over varying data, while combined complexity captures the general difficulty of query evaluation in its full generality. Understanding the gap between the two is key to identifying structural restrictions that guarantee tractability.

Current chapter references

[AB16] Sanjeev **Arora** and Boaz **Barak**. *Computational Complexity: A Modern Approach*. 4th printing 2016. New York: Cambridge University Press, 2016. 579 pp.

[ANR15] Mahmoud **Abo Khamis**, Hung Q **Ngo**, and Atri **Rudra**. *FAQ: questions asked frequently*. Apr. 2015. arXiv 1504.04044.

[ANR16] Mahmoud **Abo Khamis**, Hung Q **Ngo**, and Atri **Rudra**. *FAQ: questions asked frequently*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2016, pp. 13–28. doi 10.1145/2902251.2902280.

[Are+22] Marcelo **Arenas**, Pablo **Barcelo**, Leonid **Libkin**, Wim **Martens**, and Andreas **Pieris**. *Database Theory*. Open Source. 2022. https://github.com/pdm-book/community.

[Bee+83] Catriel **Beeri**, Ronald **Fagin**, David **Maier**, and Mihalis **Yannakakis**. *On the Desirability of Acyclic Database Schemes*. In *Journal of the ACM* 30.3 (July 1983), pp. 479–513. doi 10.1145/2402.322389.

[Ber73] Claude **Berge**. *Graphes et Hypergraphes*. 2. éd. Dunod Université, 604. Série Violette; Mathématiques. Paris: Dunod, 1973. 516 pp.

[Ber87] Claude **Berge**. *Hypergraphes: combinatoire des ensembles finis*. Bordas. Paris: Gauthier-Villars [u.a.], May 1987. 240 pp.

[Bod06] Hans L. **Bodlaender**. *Treewidth: Characterizations, Applications, and Computations*. In *Graph-Theoretic Concepts in Computer Science*. Ed. by Fedor V. **Fomin**. Vol. 4271. Berlin, Heidelberg: Springer Berlin Heidelberg, June 2006, pp. 1–14. doi 10.1007/11917496_1.

[Bod93] Hans L. **Bodlaender**. *A Tourist Guide through Treewidth*. In *Acta Cybernetica* 11.1-2 (Jan. 1993), pp. 1–21. https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3417.

[Bra17] Johann **Brault-Baron**. *Hypergraph Acyclicity Revisited*. In *ACM Computing Surveys* 49.3 (Sept. 2017), pp. 1–26. doi 10.1145/2983573.

[Cap16] Florent **Capelli**. *Structural restrictions of CNF-formulas: applications to model counting and knowledge compilation*. PhD thesis. Université Paris Diderot, Sorbonne Paris Cité, June 2016. https://florent.capelli.me/publi/these_capelli.pdf.

[CM77] Ashok K. **Chandra** and Philip M. **Merlin**. *Optimal Implementation of Conjunctive Queries in Relational Data Bases*. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, May 1977, pp. 77–90. doi 10.1145/800105.803397.

[Dur09] David **Duris**. *Acyclicité des hypergraphes et liens avec la logique sur les structures relationnelles finies*. PhD thesis. Université Paris Diderot, Nov. 2009. https://www.imj-prg.fr/theses/pdf/david_duris.pdf.

[Fag83] Ronald **Fagin**. *Degrees of Acyclicity for Hypergraphs and Relational Database Schemes*. In *Journal of the ACM* 30.3 (July 1983), pp. 514–550. doi 10.1145/2402.322390.

[Fic+18] Johannes K **Fichte**, Markus **Hecher**, Neha **Lodha**, and Stefan **Szeider**. *An SMT approach to fractional hypertree width*. In *Principles and Practice of Constraint Programming: 24th International Conference, Lille, France, August 27-31, 2018, Proceedings 24*. Springer. Aug. 2018, pp. 109–127. doi 10.1007/978-3-319-98334-9_8.

[FW90] Michael L. **Fredman** and Dan E. **Willard**. *BLASTING through the information theoretic barrier with FUSION TREES*. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: Association for Computing Machinery, Apr. 1990, pp. 1–7. doi 10.1145/100216.100217.

[Gan+22] Robert **Ganian**, André **Schidler**, Manuel **Sorge**, and Stefan **Szeider**. *Threshold treewidth and hypertree width*. In *Journal of Artificial Intelligence Research* 74 (Aug. 2022), pp. 1687–1713. doi 10.1613/JAIR.1.13661.

[GLS01] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *Hypertree Decompositions: A Survey*. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science*. MFCS '01. Berlin, Heidelberg: Springer-Verlag, Jan. 2001, pp. 37–57. doi 10.1007/3-540-44683-4_5.

[GLS02] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *Hypertree Decompositions and Tractable Queries*. In *Journal of Computer and System Sciences* 64.3 (May 2002), pp. 579–627. doi 10.1006/jcss.2001.1809.

[GLS99] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *Hypertree Decompositions and Tractable Queries*. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. SIGMOD/PODS99: International Conference on Management of Data and Symposium on Principles of Database Systems. Philadelphia Pennsylvania USA: ACM, May 1999, pp. 21–32. doi 10.1145/303976.303979.

[HV21] David **Harvey** and Joris **Van Der Hoeven**. *Integer multiplication in time* $\mathcal{O}(n \log n)$. In *Annals of Mathematics* 193.2 (Mar. 2021), pp. 563–617. doi 10.4007/annals.2021.193.2.4.

[Per14] Sylvain **Perifel**. *Complexité algorithmique*. Références sciences. Paris: Ellipses, 2014.

[RS83] Neil **Robertson** and P.D. **Seymour**. *Graph Minors. I. Excluding a Forest*. In *Journal of Combinatorial Theory, Series B* 35.1 (Aug. 1983), pp. 39–61. doi 10.1016/0095-8956(83)90079-5.

[RS86] Neil **Robertson** and P.D **Seymour**. *Graph Minors. II. Algorithmic Aspects of Tree-Width*. In *Journal of Algorithms* 7.3 (Sept. 1986), pp. 309–322. doi 10.1016/0196-6774(86)90023-4.

[Var82] Moshe Y. **Vardi**. *The Complexity of Relational Query Languages (Extended Abstract)*. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing - STOC '82*. The Fourteenth Annual ACM Symposium. San Francisco, California, United States: ACM Press, May 1982, pp. 137–146. doi 10.1145/800070.802186.

[Yan81] Mihalis **Yannakakis**. *Algorithms for Acyclic Database Schemes*. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. Sept. 1981, pp. 82–94. doi 10.5555/1286831.1286840.

Chapter 2

The Landscape of Join Query Answering: A Survey

All life is problem solving.

Karl Popper

Efficient query answering over large datasets is a central concern in database theory and practice. Multiple problems have emerged in this domain, from retrieving the full set of answers to a query to more finer algorithmic tasks such as counting the answers, directly accessing a specific answer or uniformly sampling from the answer set. These tasks are particularly challenging when the answer set is large, rendering its full materialisation impractical or infeasible. Therefore, research has focused on designing representations and algorithms that enable efficient access to the answer set under structural restrictions with respect to the query or the database.

The aim of this chapter is to provide to the reader a comprehensive background on the fundamental problems we consider and the current state of the art. We will formalise these problems and review known tractability results under various assumptions about the query and the database it is being evaluated against. We start by introducing the more general problem of query evaluation, which aims at retrieving the full answer set of a query over a given database. We then introduce more specific problems that will be a central part of this manuscript, such as direct access and uniform sampling.

Outline of the current chapter

2.1 Boolean Query Evaluation

A natural way to think about queries is as questions directed to a reference source, or database, that holds the relevant information across one or more relations. The simplest of these questions are those that expect a binary answer: *yes* or *no*. At first glance, this kind of question may appear straightforward. However, verifying the answer may require checking many combinations of values across several relations. The query may be syntactically simple, but the underlying computation may be costly. In fact, this problem, known as Boolean query evaluation problem, is computationally hard in general. The hardness of these tasks, even for queries that appear structurally simple, motivates the study of which queries admit tractable evaluation and under which assumptions.

In Example 2.1, we introduce a toy example of this problem.

▶ **Example 2.1**

Consider a reference source for a university. This database consists of the two following relations:

- Enrolled(student, lecture): indicating which student is enrolled in which lecture; and
- Teaches(teacher, lecture): indicating which teacher is responsible for which lecture.

And suppose we are given the following database instance **D**:

Enrolled	student	lecture
	Alice	CS101
	Alice	CS102
	Bob	CS102
	Bob	CS103
	Corentin	CS103
	David	CS101

Teaches	teacher	lecture
	Dr. Turing	CS101
	Pr. Lovelace	CS102
	Dr. Strange	CS103

One could imagine multiple different questions to be asked over this simple database, but one of the easiest could be: "Is there a student enrolled in a class taught by Pr. Lovelace?". This can be represented by the following query:

$$Q :\text{-}\ \mathsf{Enrolled}(x, y),\ \mathsf{Teaches}(\text{Pr. Lovelace}, y)$$

Note that, in this case, the head of the query is empty, since nothing other than `true` or `false` is expected to be returned. The `true` value would be represented by the set containing only the empty tuple: $\{\langle\rangle\}$, and the `false` value would be an empty set. In our case, the answer would be `true`, since Alice is a student enrolled in CS102 taught by Pr. Lovelace.

We can formalise the problem definition as follows:

▶ **Definition 2.2** (Boolean Query Evaluation)

Let Q be a join query over a relational schema, and let $\mathbf{D}$ be a database instance over the same schema. The *Boolean query evaluation problem* asks whether $\mathsf{ans}_{\mathbf{D}}(Q)$ is empty or not.

Solving this problem is not trivial. In fact, the complexity of the model checking problem has been shown by Chandra and Merlin to be NP-complete in the combined complexity setting for conjunctive queries [CM77].

▶ **Theorem 2.3** (Complexity of model checking [CM77])
Given a conjunctive query Q and a database instance $\mathbf{D}$, the problem of checking if there exists a tuple $\tau \in \mathbf{D}$ such that τ is an answer to Q is NP-complete.

Proof. We can solve the problem by using a non-deterministic algorithm to guess a tuple τ. We can then check if this tuple satisfies all the atoms of Q, that is, for each atom $\mathsf{R}(x_1, \ldots, x_n) \in Q$, whether $\mathsf{R}(\tau(x_1), \ldots, \tau(x_n)) \in \mathbf{D}$. The number of atoms is linear in the size of Q and checking the existence of the tuple can be done in polynomial time (or even linear or constant if the database is indexed). Since the size of the tuple certificate is polynomial in the size of the query and the verification is possible in polynomial time, the model checking problem is in NP.

To show the NP-hardness of model checking, we can reduce from the 3-colourability problem, which we recall is the problem of, given a graph $G = (V, E)$, decide if we can colour the nodes of G with 3 colours such that no two adjacent nodes have the same colour. The reduction is as follows. Start by constructing a database with a single binary relation defined as:

$$\begin{aligned}\mathsf{C}(c_1, c_2) = \{&\langle c_1 \leftarrow 1, c_2 \leftarrow 2\rangle, \langle c_1 \leftarrow 1, c_2 \leftarrow 3\rangle, \langle c_1 \leftarrow 2, c_2 \leftarrow 1\rangle,\\ &\langle c_1 \leftarrow 2, c_2 \leftarrow 3\rangle, \langle c_1 \leftarrow 3, c_2 \leftarrow 1\rangle, \langle c_1 \leftarrow 3, c_2 \leftarrow 2\rangle\}\end{aligned}$$

That is, every valid combination of colours between two nodes. We use a variable x_v for every node $v \in V$. The query is then the join, over all the edges $\{u, v\}$ from the graph, of relation C applied to the extremities of the edges $\mathsf{C}(x_u, x_v)$.

This construction can be done in polynomial time as the database is constant and since we use one atom per edge of G, we can build the query in linear time. If there is an answer to Q, then there exists a 3-colouring of G, since no fact in $\mathbf{D}$ allows for two adjacent nodes to be of the same colour. It can also be shown that, if a graph G defined in this way is 3-colourable, then the related query Q is satisfiable. The 3-colourability problem is known to be NP-hard, and since we have shown that model checking is in NP, it is NP-complete. □

The problem of Boolean query evaluation is a good starting point, but more general problems exist, as we will detail in the rest of this chapter.

2.2 Query Evaluation

A natural extension of Boolean query evaluation is to move from asking if there is an answer to retrieving all of the answers to a query.

We will present two main directions aimed at evaluating the full answer set, both based on different metrics of efficiency. The first is centered around the *total time* taken to return all the answers, the second is based around a notion of *delay* between the output of two consecutive

answers. Both have their importance, in the literature and in the contributions presented in this thesis.

We start by introducing the former direction, usually referred to simply as "query evaluation".

▶ **Example 2.4**
Consider once again the database instance from Example 2.1, another question we could ask is: "Who are the students enrolled in a class taught by Dr. Strange?". In this case, it would be modelled by a query such as:

$$Q(x, y) \text{ :- } \mathsf{Enrolled}(x, y),\ \mathsf{Teaches}(\text{Dr. Strange}, y)$$

Here, we are no longer expecting a set containing the empty tuple, but a set of tuples over names and classes. In our case, the answer would be:

$$\mathsf{ans}_{\mathbf{D}}(Q) = \{\langle \mathsf{student} \leftarrow \mathsf{Bob}, \mathsf{lecture} \leftarrow \mathsf{CS103}\rangle, \langle \mathsf{student} \leftarrow \mathsf{Corentin}, \mathsf{lecture} \leftarrow \mathsf{CS103}\rangle\}$$

since both Bob and Corentin are enrolled in CS103, taught by Dr. Strange.

Formally, we can define the query evaluation problem as follows:

▶ **Definition 2.5** (Query Evaluation)
Given a join query $Q(\mathbf{x})$ over a relational schema and a database instance $\mathbf{D}$ over the same schema, the *query evaluation problem* consists in computing and returning the full answer set $\mathsf{ans}_{\mathbf{D}}(Q)$.

The complexity of the query evaluation problem follows from the complexity of the Boolean case: finding one of the answer tuples would result in solving the Boolean version.

However, in the data complexity setting, where we consider the query to be fixed and only the database to be variable, query evaluation becomes much more tractable. As shown by Vardi, the problem lies in LogSpace, that is the class of problems that can be decided using $\mathcal{O}(\log(n))$ space, where n is the size of the input [Var82]. This discrepancy between the combined and data complexities has motivated the search for restricted fragments of queries that are tractable under combined complexity. It has also led to efforts to characterise the dependency to the size of the data. Indeed, especially in real-world applications where it quickly becomes very large, the dominant complexity factor is the size of the data. While we could have an exponential coefficient, we would be happy to have a complexity measure that is linear in the size of the data as opposed to an expression of the form $|\mathbf{D}|^k$.

An important part of the complexity of a query evaluation algorithm is the sizes of the input (essentially the database) and the output (the answer set). The complexity will always be at least linear in the output size, since we have to return all of the answers. However, this has led to two major lines of research. The first has focused on building evaluation algorithms that are guaranteed to run in time linear in the worst possible output size, while the second aims to understand the dependency to structural properties of the queries and databases.

2.2.1 Worst-Case Optimal Joins

Algorithms devised to solve query evaluation problems are commonly called *join algorithms*. Classically, join algorithms rely on building a query plan and computing intermediary joins in order to construct the full answer set. This can be expensive however due to the cost of computing these intermediary joins. It is moreover crucial to be able to correctly estimate the size of a join result, as it is used to choose the best query plan. Atserias, Grohe, and Marx have shown that there exists a tight bound on the number of possible answers for a full conjunctive query [AGM13; GM14]. This bound is based on the size of the input relations, and is extensively used in the literature to characterise the complexity of join algorithms.

▶ **Example 2.6** (Bounds on the output size)
To explain the idea behind the bound, let's consider a simple query:

$$Q_\Delta :\text{-}\ \mathsf{R}(x,y),\ \mathsf{S}(y,z),\ \mathsf{T}(x,z)$$

(see Example 1.11 for a representation of the query graph).

We will also consider that the relation sizes are the same, that is: $|\mathsf{R}| = |\mathsf{S}| = |\mathsf{T}| = N$, with $N \in \mathbb{N}$.
Our objective here is to find a bound on $|\mathsf{ans}_{\mathbf{D}}(Q)|$ for a database instance $\mathbf{D}$. More simply, we want to know if we can estimate the maximum number of answers that we would get for this query over a given database.

A trivial bound on the number of answers of this query would be N^3, as it would be the Cartesian product of the considered relations. However, by joining simply two of the relations, such as $\mathsf{R} \bowtie \mathsf{S}$ (there is no loss of generality here, we could also consider $\mathsf{S} \bowtie \mathsf{T}$ or $\mathsf{R} \bowtie \mathsf{T}$), we notice that we obtain a superset of $\mathsf{ans}_{\mathbf{D}}(Q)$, since we already cover all of the variables. Indeed, $\mathsf{R} \bowtie \mathsf{S}$ already contains all the tuples τ over variables (x, y, z) where $\tau_{|\{x,y\}} \in \mathsf{R}$ and $\tau_{|\{y,z\}} \in \mathsf{S}$, and any tuple in $\mathsf{ans}_{\mathbf{D}}(Q)$ must appear in this intermediate result, therefore it is a superset of the answer set of the full join. Since any such pairwise join contains at most N^2 elements, then a better bound of $|\mathsf{ans}_{\mathbf{D}}(Q)|$ is N^2.

However, by generalising this notion of *covering* the variables to a fractional cover of the query hypergraph, Atserias, Grohe, and Marx showed that, in this case, we can more tightly bound the size of $\mathsf{ans}_{\mathbf{D}}(Q)$ by $N^{1.5}$.

More formally, this bound, that is known as the AGM bound, can be defined as:

▶ **Definition 2.7** (AGM *bound* ([AGM13; GM14]))
Given a query Q whose hypergraph is $\mathcal{H}$, if we consider the set of weigths $\Lambda = \{\lambda_e \mid e \in \mathcal{H}\}$ computed from a fractional edge cover of $\mathcal{H}$, then we have:

$$|\bowtie_{e\in\mathcal{H}} \mathsf{R}_e| \leqslant \prod_{e\in\mathcal{H}} N_e^{\lambda_e}$$

where the R_e are the relations of the query Q and where, for each R_e, N_e is a bound on the

size of the relation.

This bound has also been shown to be both *tight* and *optimal* [AGM13].

We know from Chandra and Merlin [CM77] that we cannot have a join algorithm that is linear in the size of the output in the general case. These results on bounding the size of the answer set of a query have therefore led to a different approach in the design of join algorithms, that takes into account not just the shape of the query, but the relationship between the input and output sizes. This recent line of research is focused on worst-case optimal join, and are linked to the worst-case, that is the size of the largest possible output of a query over any of the considered databases.

▶ **Definition 2.8** (Worst-Case)
Let Q be a query and $\mathcal{C}$ be a class of databases for Q that share the same relations and properties. We define the *worst-case* of $\mathcal{C}$, denoted $\mathsf{wc}(Q, \mathcal{C})$ as the size of the largest answer set possible for Q with any $\mathbf{D} \in \mathcal{C}$.

More formally,

$$\mathsf{wc}(Q, \mathcal{C}) = \mathsf{sup}_{\mathbf{D} \in \mathcal{C}} |\mathsf{ans}_{\mathbf{D}}(Q)| \ .$$

A few examples of worst-case values are presented in Example 2.9. An algorithm is said to be a *worst-case optimal join* (WCOJ for short) for $(Q, \mathcal{C})$ if it outputs the answer set $\mathsf{ans}_{\mathbf{D}}(Q)$ in time $\tilde{\mathcal{O}}(\mathsf{wc}(Q, \mathcal{C}) \cdot f(Q))$, where $f(Q)$ is a polynomial, independent of $\mathcal{C}$. This polynomial depends only on parameters from the structure of the query (the variables and atoms) and not from the database (such as the size of the relations). Of course, for this definition to make sense, the worst-case of the considered class must be finite.

This line of work was initiated by Ngo, Porat, Ré, and Rudra [Ngo+12] with an algorithm known as the NPRR join. In their work, they prove that the NPRR join has a running time matching the AGM bound, which is the shown to be optimal when the databases are defined with restrictions on the sizes of the relations. Chapter 3 goes into more details around these notions. Their work was later expanded upon and refined by algorithms such as Leapfrog Triejoin [Vel14] and GenericJoin [Ngo18].

The latter also exploits the key idea that, rather than evaluating joins pairwise in a fixed query plan, they explore the space of valid assignments one attribute at a time, following a fixed variable ordering. At each level, they efficiently intersect the sets of possible values across all relevant relations, pruning infeasible combinations early. The search is naturally guided by the query structure, not by fixed join orderings.

▶ **Example 2.9** (Some examples of worst-case)
Here, we will consider the class of databases where all relations have a size bounded by some integer N. To avoid overcharging the notation at this point, we will denote the worst-case values as $\mathsf{wc}(\cdot, N)$ and will not show the query variables.

Query	Expression	Worst-Case
Cartesian product	Q_2 :- $\mathsf{R}(x)$, $\mathsf{S}(y)$	$\mathsf{wc}(Q_2, N) = N^2$
	Q_k :- $\mathsf{R}_1(x_1)$, ..., $\mathsf{R}_k(x_k)$	$\mathsf{wc}(Q_k, N) = N^k$
Square Query	$Q_\square$:- $\mathsf{R}(x,y)$, $\mathsf{S}(y,z)$, $\mathsf{T}(z,t)$, $\mathsf{U}(t,x)$	$\mathsf{wc}(Q_\square, N) = N^2$
Triangle Query	Q_Δ :- $\mathsf{R}(x,y)$, $\mathsf{S}(y,z)$, $\mathsf{T}(z,x)$	$\mathsf{wc}(Q_\Delta, N) = N^{1.5}$
k-cycle	Q_{c_k} :- $\mathsf{R}_1(x_1,x_2)$, $\mathsf{R}_2(x_2,x_3)$, ..., $\mathsf{R}_k(x_k,x_1)$	$\mathsf{wc}(Q_{c_k}, N) = N^{k/2}$

Some of these worst-case values may seem counter-intuitive, such as the value for the square or triangle queries. The idea is, that in the case where the relations all have a size bounded by N, the worst-case can be shown to be the AGM bound, and therefore the power is equal to the fractional edge cover of the query hypergraph.

WCOJ algorithms have multiple benefits over other approaches. For instance, they handle *cyclic* queries efficiently and guarantee a runtime no worse than the worst possible output size. Moreover, they are *decomposition free*, no join tree or query plan is required, simply an order with which to evaluate the variables. They have also been used in modern database join engines, such as LogicBlox, that uses Leapfrog Triejoin.

Chapter 3 goes more into detail around worst-case optimal joins and presents one of the main contributions of this thesis: a novel branching algorithm, similar in spirit to both Leapfrog Triejoin and Generic Join, but with a simpler complexity analysis.

2.2.2 Answer Enumeration

In practical applications, it is often useful to iterate over the answers of a query. Imagine for instance we are still in the context of Examples 2.1 and 2.4. Suppose we need to retrieve all the student records from the university database. For each student, we then have to send a notification based on their courses or their performance. In this case, materialising the full answer set in memory before iterating can be both prohibitive, especially when the number of answers is very large, and unnecessary. This is common in data-intensive applications, where the number of possible output tuples can far exceed the size of the input data. Modern systems often mitigate this using streaming techniques or cursor-based evaluation, where answers are produced and consumed incrementally. This is the aim of the second approach we mentioned earlier. In this case, we are no longer considering algorithms that run in time linear to the worst possible case, but we aim to build algorithms that are able to incrementally output tuples a specific way. Here, the focus is essentially on setting guarantees on the *delay* between two consecutive answers. The goal of *enumeration algorithms* is to support an incremental behaviour efficiently: after a preprocessing phase, answers should be produced one by one, without repetition, and with a low delay between successive outputs. This avoids materialising the entire answer set and allows each answer to be processed as necessary. These problems are usually split into two parts: the first phase involves designing a datastructure from the query and the database that allows for a more efficient enumeration. This structure could for instance take the form of a different data representation or be an index of information helping the eunmeration. The second phase is devoted to the actual enumeration of the answers, ideally with strong guarantees on the delay between the output of consecutive answers.

Because one has to at least read the input data, the preprocessing cannot be less than **linear** in the size of the data. This leads to an ideal complexity target of **linear-time preprocessing** in the size of the database $|\mathbf{D}|$ and a **constant delay** between each output tuple, independant from

the size of the database. In some settings, it can also be interesting to enumerate the answers in a given order (such as the lexicographical order), which may however increase the cost of either phase.

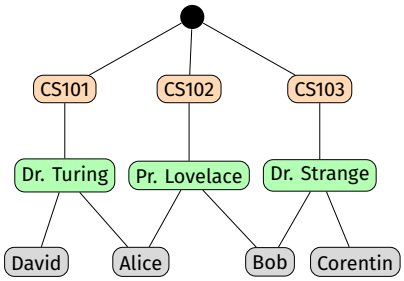
▶ **Example 2.10**
Let us consider the following query over the database instance from Example 2.1.

$$Q(x, y, z) :\text{-}\ \mathsf{Enrolled}(x, y),\ \mathsf{Teaches}(z, y)$$

This query retrieves all tuples (x, y, z) such that student x is enrolled in lecture y, taught by teacher z.

Now consider two preprocessing possibilities:

$\mathsf{ans}_{\mathbf{D}}(Q)$	student	lecture	teacher
	Alice	CS101	Dr. Turing
	Alice	CS102	Pr. Lovelace
	Bob	CS102	Pr. Lovelace
	Bob	CS103	Dr. Strange
	Corentin	CS103	Dr. Strange
	David	CS101	Dr. Turing



On the left, we have a naive table preprocessing. Building this table takes time linear in the size of the answer set, but guarantees constant delay output on the answers. By reorganising this by shared prefixes, we can build the structure on the right in time $\mathcal{O}(\#\mathsf{Enrolled} + \#\mathsf{Teaches})$. Here, outputting an answer consists simply in following a path from the root of the tree to a leaf. This can be done with a constant delay, which will be the time needed to go down the tree.

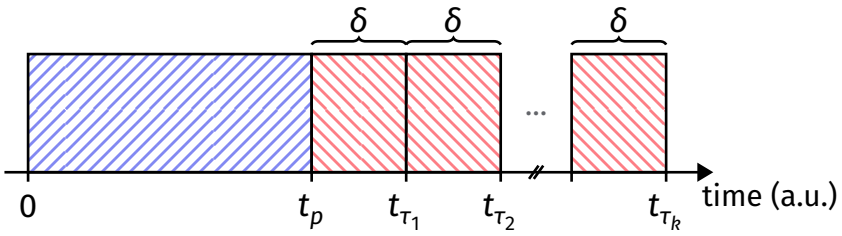
We can define the enumeration problem more formally as follows:

▶ **Definition 2.11** (Enumeration Problem)
Given a query Q and a database instance $\mathbf{D}$, the *enumeration problem* consists in designing an algorithm that, after a preprocessing phase, outputs one by one, without repetition, each tuple in the answer set $\mathsf{ans}_{\mathbf{D}}(Q)$.

The complexity of the enumeration problem is usually defined in terms of both the complexity of the preprocessing phase and of a bound on the delay between the output of two answers.

▶ **Example 2.12** (Complexity of Enumeration)
Consider the following time log of an execution of enumeration with the table preprocessing

from Example 2.10:



In this illustration, the first rectangle (filled with blue coloured diagonal lines) represents the preprocessing phase. The rectangles filled with red coloured (inverted) diagonal lines represent the delays between outputs.
At time t_p, the preprocessing has finished and the enumeration can start. At time t_{τ_i}, the i^{th} tuple is outputted.

In this example, the delay between the output of two tuples is a constant δ. We would therefore describe this algorithm as having a preprocessing time of t_p and a constant delay of δ.

Enumeration algorithm often focus on *structural restrictions* on the shape of the query. These approaches are often based on associating to each conjunctive query a hypergraph representing the variable co-occurences in its atoms, and then using measures over this hypergraph to restrict the possible queries.

The seminal result in this area is due to Yannakakis, who introduced the notion of acyclic queries[1] and showed that Boolean query evaluation becomes tractable for this fragment of queries [Yan81]. In this case, evaluation can be performed in linear time with respect to the size of the database and polynomial in the number of variables and relations, using a bottom-up dynamic programming approach on a join tree of the query.

In his seminal paper, Yannakakis presented an algorithm for full acyclic conjunctive queries that takes advantage of the structure of the query rather than simply the size of the data [Yan81]. The main idea is that, if the query is acyclic, then we can arrange the variables in a join tree. Each node then corresponds to an atom and shared variables as connected. Then, the algorithm works in two phases. In the first phase, the idea is to work *bottom-up*, that is, from the leaves to the root, and operate a *semijoin reduction*. Each relation is filtered by only keeping the tuples that are needed by its parent in the tree. This avoids keeping track of unused tuples and allows removing dead branches in the join tree early if needed. Finally, the second phase consists in constructing the join, this time *top-down*, from the root of the join tree to the leaves. Since the data has already been filtered, we can now combine the tuples in a consistent way efficiently and without generating redundant tuples.

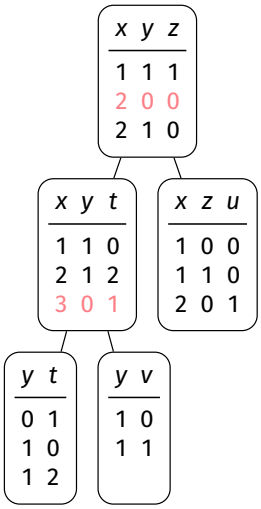
▶ **Example 2.13** (Yannakakis' algorithm)

[1] formally, Yannakakis introduces *acyclic database schemes*, but all further notions of *acyclicity* for queries directly derive from this concept.

Consider the following query:

$$Q(x, y, z, t, u, v) \text{ :- } \mathsf{R}(x, y, z),\ \mathsf{S}(x, y, t),\ \mathsf{T}(x, z, u),\ \mathsf{U}(y, t),\ \mathsf{V}(u, v)$$

We can build a join tree and load the data from the database inside, such as in the figure on the right.
The first phase consists in filtering out the tuples. We work from the leaves up, the point being to remove all the tuples that cannot be extended to a solution below. In our case, the coloured tuples cannot be extended and are therefore not useful in the full join. Note that the leaves are not filtered, and redundant tuples can still exist in these relations. However, the tuples in the root can now all be extended to full solutions.
The second phase then consists in constructing the answer set by extending the tuples from the root down to the leaves. For instance, we could build the following tuple:



$$\langle x \leftarrow 1, y \leftarrow 1, z \leftarrow 1, t \leftarrow 0, u \leftarrow 0, v \leftarrow 0 \rangle$$

which is in the answer set.

Given a query Q and a database $\mathbf{D}$, this algorithm runs in time $\mathcal{O}(|Q| \cdot (|\mathbf{D}| + |\mathsf{ans}_{\mathbf{D}}(Q)|))$. In data complexity, where we consider the query to be a constant, it is therefore asymptotically the same as reading the input (the database) and writing the output (the answers).

This has been a starting point for the study of the complexity of answering conjunctive queries as the subclass of acyclic queries was shown to be tractable for query evaluation. Since the notion of acyclic queries is very restrictive, many efforts have been subsequently made in order to find lighter restrictions on the queries that still lead to tractable conjunctive evaluation. Several different query classes have been introduced based on width measures over the query hypergraph, notably treewidth, hypertree width, fractional hypertree width or query-width [CR00; FFG01; GLS02; GSS01; KV00; Mar10]. Classes of queries built around these width restrictions have allowed different tractability characterisations of query evaluation. A more comprehensive comparison of the links between these measures can be found in [GR04].

These classes have been the focus of extensive research due to their algorithmic and descriptive robustness: many natural queries fall within them and they also support efficient query evaluation algorithms.

The Yannakakis algorithm has been widely used and twisted to answer multiple different problems (see Sections 2.3 and 2.4 for example). However, it relies on the fact that a join tree can be built, which restricts its use to (α-)acyclic queries.

Yannakakis's algorithm might be optimal for acyclic queries, but it does not extend efficiently to the general case. As soon as the query contains cycles, the intermediate joins can grow in size, even when the final result set is small. A detailed example of this scenario is presented in Chapter 3 (see Example 3.1).

When we consider conjunctive queries, the problem is more complicated. This is due to the fact that, by *forgetting* some of the variables (that is, projecting them), some answer tuples become equal. Because we cannot return the same tuple more than once, additional guards have to be placed in the algorithms. In this case, some subclasses have been shown to support efficent enumeration. Consider for instance the fragment of free-connex acyclic conjunctive queries, that is, acyclic conjunctive queries for which there exists in the query hypergraph a connex subset

of edges whose union is equal to the set of the free variables. This fragment has been shown to accept algorithms with linear preprocessing time and constant delay enumeration [BDG07; Bag09; Bra13]. Bagan, Durand, and Grandjean have also shown that self-join free conjunctive queries that are not free-connex cannot be enumerated with these guarantees [BDG07].

If we relax the constraints on the preprocessing time, that is, we no longer require linear time preprocessing, then more query classes have been shown to be tractable with constant delay enumeration. For instance, it has been shown that it is possible for conjunctive queries of bounded *free-connex* fractional hypertree width [GLS00; OZ15].

While we will not go into more details about enumeration algorithms as they are not at the centre of this thesis, the interested reader may find a more comprehensive state of the art of the progress in model enumeration in a survey by Berkholz, Gerhardt, and Schweikardt [BGS20].

2.3 Counting the Answers of a Query

Once we know how to check whether a query admits at least one solution over a database and how to retrieve the answer set, a natural extension is to be able to efficiently know how many such solutions exist. This is the goal of the counting problem. A practical example over the sample database provided in Example 2.1 could be answering a question such as: "How many students are enrolled in a lecture given by Dr. Turing?". This boils down to counting the number of tuples in the database that satisfy the query. While this may appear to merely be a modest step beyond evaluation, the complexity of this problem is significantly higher, as it requires identifying and counting all satisfying valuations, not just verifying the existence of one.

Formally, we can define the problem as such:

▶ **Definition 2.14** (Counting problem)
Given a query Q over a relational schema and a database instance $\mathbf{D}$ over the same schema, the *counting problem* consists of returning $|\mathsf{ans}_{\mathbf{D}}(Q)|$, that is, the size of the answer set of Q over $\mathbf{D}$.

The counting problem for conjunctive queries is the task of computing the number of answers of the query. This problem can therefore be seen as the counting analogue of the decision problem of query evaluation, and as such, is a typical #P problem. In fact, this problem has been shown to be #P-complete [Val79; Bau+05], and this even in the case of full conjunctive queries. As such, it is generally intractable in combined complexity, and this remains true even for queries where the evaluation is tractable. In particular, many acyclic conjunctive queries admit efficient evalution, but not efficient counting.

Therefore, finer structural decompositions of the query have once again played a central role in characterising tractable instances of model counting. Unlike evaluation, where acyclicity already suffices for linear-time algorithms, model counting however requires finer restrictions. Dalmau and Jonsson proved that join queries with bounded treewidth have tractable counting [DJ04]. These results were later extended by Pichler and Skritek to acyclic conjunctive queries with bounded hypertree width [PS13].

2.4 Direct Access

Enumeration is important for practical cases where one needs to perform some task on each element of the answer set. In some cases however, we are only interested in performing the task over a small number of specific answers. For instance, suppose that from the same university database mentioned in Example 2.1, we may wish to retrieve the student ranked 500^{th} based on their grades. We have many different approaches possible for this. A naive solution could be to build an array-like structure containing the ordered answers to the query, but this method reaches its limits fast when the number of answers is large. We could also use an enumeration algorithm to go through the first 499 students in that order, but this would result in 499 extra answers being generated needlessly.

This motivates yet another approach that aims at using a data structure to represent the set of answers, allowing for efficient access to a given answer, but without constructing the whole answer set in memory. Once again, this task, known as the *direct access problem*, is typically divided into two separate steps: first, a preprocessing phase where we build a structure that allows efficient access to individual answers, followed by the access phase, where we query a given index and retrieve the correct answer.

Moreover, one is rarely interested in accessing exactly one tuple of the answer set but is usually interested in having a data structure allowing to solve efficiently direct access tasks for any input k. The objective here is to obtain an algorithm for direct access tasks with polynomial precomputation time and access time that is polylogarithmic in the size of $\mathbf{D}$. Being able to fetch a given answer solely by its index is crucial in practice: this is especially important when answers are expensive to compute or store, or when different results need to be retrieved independently.

▶ **Remark 2.15**
Note that the direct access problem is at least as hard as the enumeration problem, since, for an equal preprocessing time, we cannot hope for an access time smaller that the cumulated delay. This can also be seen from the other side: if we have efficient direct access for a query over a database, then we immediatly have enumeration by retrieving successively all answers, starting from index 1 up to the last possible index. This also works for counting: we could use a binary search algorithm to find the number of answers by retrieving answers ranked between 1 and 2^n.

Another important factor to consider is that the answer set of a query has no inherent order: it is defined as a set, not a sequence. However, for the direct access problem to be well defined, specifically, for it to make sense to retrieve the k^{th} answer, we must impose a total order on the answers. Most results in the literature consider the lexicographic order, which is defined by fixing an order on the variables of the query and an order on the domain values. This choice is not arbitrary: lexicographic ordering aligns well with structural decompositions of the query, such as variable elimination orders or tree decompositions, and enables recursive algorithms to reason about the prefix of an answer before its suffix. Moreover, many of the tractable cases for direct access rely on exploiting this order to navigate the answer space efficiently without materialising it.

With these notions in mind, we can now introduce a formal definition of the direct access problem, which captures this notion of retrieving the k^{th} answer in the lexicographical order after a preprocessing phase.

▶ **Definition 2.16** (Direct Access Problem)
Given a query Q, a database instance $\mathbf{D}$ over an ordered domain $\mathsf{dom}(\mathbf{D})$, and a total order $\prec$ over the variables of Q, the *direct access problem* consists in constructing a data structure such that, after a preprocessing phase executed in time $\mathsf{T}_{\mathsf{preprocessing}}$, the k^{th} tuple of the answer set $\mathsf{ans}_{\mathbf{D}}(Q)$ (ordered lexicographically according to $\prec$) can be retrieved in time $\mathsf{T}_{\mathsf{access}}$, for any valid index $k \leqslant |\mathsf{ans}_{\mathbf{D}}(Q)|$ and failing otherwise.

Example 2.17 is a quick example of direct access tasks.

▶ **Example 2.17**
Consider the following query over the database instance defined in Example 2.1.

$$Q(\mathsf{student}, \mathsf{lecture}, \mathsf{teacher}) \text{ :- } \mathsf{Enrolled}(\mathsf{student}, \mathsf{lecture}),\ \mathsf{Teaches}(\mathsf{teacher}, \mathsf{lecture})$$

Suppose that we wish to have a direct access to the answer set of this query. Let us start by looking at a naive preprocessing strategy consisting in building the table of all the answers, as shown below.

$\mathsf{ans}_{\mathbf{D}}(Q)$	student	lecture	teacher
	Alice	CS101	Dr. Turing
	Alice	CS102	Pr. Lovelace
	Bob	CS102	Pr. Lovelace
	Bob	CS103	Dr. Strange
	Corentin	CS103	Dr. Strange
	David	CS101	Dr. Turing

In this case, we can directly access any given answer by simply looking up the related index. If we have stored the table as an array $\mathcal{T}$, then the 4^{th} answer is accessible with $\mathcal{T}[4]$. Here, the preprocessing phase takes the time needed to build all of the answers and uses $\mathcal{O}(|\mathsf{ans}_{\mathbf{D}}(Q)|)$ space. The access time is constant in this case, because of the properties of the table structure we use.

One important factor however is that we have to materialise the complete answer set and therefore spend a lot of time on preprocessing.

Trying to avoid having to explicitly materialise the full answer set has led to multiple research directions around direct access. As with enumeration, the objective is to find query classes that support direct access with suitable preprocessing and access times guarantees.

This problem was first been introduced by Bagan, Durand, Grandjean, and Olive [Bag+08], and is very natural in the context of databases since it can be seen as a building block for different problems such as counting, (randomized) enumeration [Bag+08] or uniformly sampling [Car+20; Kep20] from the answers of a query. While this problem is known to be #P-hard with respect to combined complexity, some query classes have an underlying structure that allows for direct access tasks to be performed more efficiently.

Previous work on this subject has focused on devising methods with better preprocessing time while still offering reasonable access performance. To do this, the problem is often reduced to the study of specific subclasses of databases or queries. Early work on direct access focused on adding restrictions to the database. For instance, in their seminal work, Bagan, Durand, Grandjean, and Olive present an algorithm for direct access with linear precomputation time and constant access time, for the class of first-order logic formulas and bounded-degree databases, that is, databases in which any domain element can only occur in a constant number of tuples [Bag+08]. Bagan later expanded these results for queries of bounded treewidth [Bag09]. While this yields interesting algorithms and complexity bounds, it severely limits practical applicability, since bounding the degree of the database restricts the data, and not the query. Real-world applications, on the other hand, often lack such uniform constraints. More recent research directions therefore aim at *query-centric* restrictions such as acyclicity, treewidth bounds or compatibility measures over the query. These structural restrictions on the query allow for preprocessing techniques to be applied uniformly over any database instance, thus expanding practical applicability. Carmeli, Zeevi, Berkholz, Kimelfeld, and Schweikardt proved that it can be done on acyclic conjunctive queries with linear preprocessing time and polylogarithmic access time for well-chosen lexicographical orders [Car+20]. This result also generalises results from Yannakakis' seminal paper [Yan81] establishing the tractability of model checking on acyclic conjunctive queries to the tractability of counting the number of answers of such queries. Later, for queries whose answers are assumed to be ordered by some lexicographical order, more precise complexity measures of the direct access task have been established in the case of acyclic queries [Car+23] and the general case [BCM22a; BCM25]. Eldar, Carmeli, and Kimelfeld also recently studied the complexity of direct access for conjunctive queries with aggregation [ECK23].

2.5 Uniform Sampling

As we have seen in the previous sections, relational database joins are known to be computationally expensive, with a cost that can grow rapidly with the size of the input, which in our case is mostly comprised of the considered database instance. In many practical cases however, the need for the whole join to be outputted is less important than the need for a less expensive computation. Moreover, many real-world applications, such as machine learning or statistical analysis, can be just as effective with random samples from the answer set.

A naive approach to sampling randomly from the answer set could consist of executing the full join, storing it in memory and then sampling uniformly by choosing a random index and returning the associated answer. However, this approach becomes clearly expensive with the number of answers, since we have to materialise the whole answer set. This has led to a focus on devising novel techniques capable of producing samples from the answer set much faster than by executing the full join. As in the enumeration or direct access problems, this often implies a preprocessing phase, where an intermediary structure is constructed, and an *access* phase, where a random sample is produced and returned. In this case, the naive sampling approach described is made harder by the fact that the answers are not explicitly stored. Moreover, we generally want strong guarantees over the uniformity of the sampling: each answer must have an equal chance to be sampled, even if some are harder to reach computationally. In the same vein as for the direct access problem, the same structure can be used multiple times to sample multiple answers from the answer set.

We can now formally define this problem as follows:

▶ **Definition 2.18** (Uniform Sampling)
Let Q be a conjunctive query over a relational schema, and let $\mathbf{D}$ be a database instance over the same schema. The *uniform sampling problem* consists of designing an algorithm that outputs a tuple $\tau \in \mathsf{ans}_{\mathbf{D}}(Q)$ in time $\mathsf{T}_{\mathsf{sample}}$, such that each tuple in $\mathsf{ans}_{\mathbf{D}}(Q)$ has equal probability of being returned:

$$\forall \tau \in \mathsf{ans}_{\mathbf{D}}(Q), \quad \Pr[\mathsf{out} = \tau] = \frac{1}{|\mathsf{ans}_{\mathbf{D}}(Q)|} \ .$$

▶ **Example 2.19**
As we mentioned earlier, if we consider the same preprocessing as in Example 2.17, then uniformly sampling from the answers of the query Q consists of two steps:

- produce a uniformly random index i, with $0 \leqslant i \leqslant |\mathsf{ans}_{\mathbf{D}}(Q)|]$; and
- return $\mathcal{T}[i]$.

However, this implies explicitly materialising the whole answer set in the form of a table. While this gives good complexity bounds on the *access* time to a random tuple and we have a strong guarantee of uniformity, the preprocessing phase will be very expensive.

As mentioned, uniform sampling from query answers has been studied as an alternative to full materialisation, especially in applications where representative instances are enough. While early solutions relied on generating the full answer set and then sampling from it, this clearly becomes intractable as the size of the answer set grows. More recent work has shown that uniform sampling can be achieved more efficiently, under structural restrictions on the query, by producing a intermediary structure that allows for efficient sampling without needing a full materialisation of the answer set.

Chaudhuri, Motwani, and Narasayya initiated research around uniformly sampling from join queries [CMN99]. They restricted the study to joins of two relations and described a structure of size $\mathcal{O}(|\mathbf{D}|)$ that allows for uniform sampling in constant time. These results were first extended by Acharya, Gibbons, Poosala, and Ramaswamy [Ach+99] to the class of queries following a *star-schema*, that is, queries that have a relation that contains a link to all other relations. The tractability of efficient uniform sampling for join queries with a $\mathcal{O}(|\mathbf{D}|)$ sized structure has later been showed to be true for acyclic join queries, with constant-time sampling [Zha+18].

More recently, one of the research directions was to look at the worst case of the considered queries and develop ways to uniformly sample with guarantees based on worst case values. Chen and Yi have shown that, even for cyclic join queries, it is possible to uniformly sample in time $\tilde{\mathcal{O}}(\frac{\mathsf{AGM}}{\max(1,|\mathsf{ans}_{\mathbf{D}}(Q)|)})$ after a linear time preprocessing step [CY20a]. This was later extended and confirmed in [Kim+23; DLT23], where lower bounds on the complexity of sampling uniformly were proven.

In [WT24], Wang and Tao extended sampling to queries with acyclic degree constraints, with the same guarantees of an expected runtime of $\tilde{\mathcal{O}}(\frac{\mathsf{upb}}{\max(1,|\mathsf{ans}_{\mathbf{D}}(Q)|)})$. Because of the extension to acyclic degree constraints, [WT24] uses a different bound, known as the *polymatroid* bound. This bound is however always *at most* AGM. We will discuss these bounds in more extensive detail in Chapter 4.

The problem of sampling uniformly from the answer set of a query is tightly connected to the problem of correctly estimating the size of said answer set. To avoid computing the exact number of answers to a query, we often rely on an approximation. This line of research was pioneered by Jerrum, Valiant, and Vazirani [JVV86]. One way to compute such an approximation is by using randomised algorithms, to sample tuples and check whether they are answers. Recent work in this area characterises the tractable query classes, as in [Are+21; Foc+25]. There are also links to the direct access problem. At first, we can present this as the fact that, if we have an efficient way of doing direct access, then we also have an efficient way of sampling random answers. This is notably the case in recent works such as [Car+23; Car+20].

2.6 Contributions

In this manuscript, we will focus on two of the fundamental query answering tasks we have presented in this chapter: direct access and uniform sampling. Although these tasks differ in nature, they are linked. The results presented in this thesis show that they share an underlying structure that can be leveraged to answer both tasks.

▶ **Uniform Sampling.** We will first address the problem of uniformly sampling from the answers of a query. To this extent, we will present a sampling algorithm based on a tree-like structure. This structure is the trace result of a simple worst-case optimal join algorithm, which will allow us to recover, with a much simpler analysis, results from the literature for uniform sampling. The results from this part are built upon from our recent paper "A Simple Algorithm for Worst Case Optimal Join and Sampling" [CIS25].

▶ **Direct Access.** We later work on the direct access problem. The results we introduce are based on enriching the structure we used for uniform sampling with new properties. We then present a direct access algorithm that recovers the previous literature result. Another core contribution of this thesis is the extension of direct access for join queries that may contain negative atoms, that is queries where we add the possibility of asking for R and *not* S for instance. This builds on another paper, [CI24; Cap+25].

Current chapter references

[Ach+99] Swarup **Acharya**, Phillip B. **Gibbons**, Viswanath **Poosala**, and Sridhar **Ramaswamy**. *Join Synopses for Approximate Query Answering*. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS99: International Conference on Management of Data and Symposium on Principles of Database Systems. Philadelphia Pennsylvania USA: ACM, June 1999, pp. 275–286. doi 10.1145/304182.304207.

[AGM13] Albert **Atserias**, Martin **Grohe**, and Dániel **Marx**. *Size bounds and query plans for relational joins*. In *SIAM Journal on Computing* 42.4 (Jan. 2013), pp. 1737–1767. doi 10.1137/110859440.

[Are+21] Marcelo **Arenas**, Luis Alberto **Croquevielle**, Rajesh **Jayaram**, and Cristian **Riveros**. *When Is Approximate Counting for Conjunctive Queries Tractable?* In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing. Virtual, Italy: ACM, June 2021, pp. 1015–1027. doi 10.1145/3406325.3451014.

[Bag+08] Guillaume **Bagan**, Arnaud **Durand**, Etienne **Grandjean**, and Frédéric **Olive**. *Computing the jth solution of a first-order query*. In *RAIRO-Theoretical Informatics and Applications* 42.1 (Jan. 2008), pp. 147–164. doi 10.1051/ita:2007046.

[Bag09] Guillaume **Bagan**. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques. (Algorithms and complexity of enumeration problems for the evaluation of logical queries)*. PhD thesis. University of Caen Normandy, France, Mar. 2009.

[Bau+05] Michael **Bauland**, Philippe **Chapdelaine**, Nadia **Creignou**, Miki **Hermann**, and Heribert **Vollmer**. *An Algebraic Approach to the Complexity of Generalized Conjunctive Queries*. In *Theory and Applications of Satisfiability Testing*. Ed. by Holger H. **Hoos** and David G. **Mitchell**. Vol. 3542. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 30–45. doi 10.1007/11527695_3.

[BCM22a] Karl **Bringmann**, Nofar **Carmeli**, and Stefan **Mengel**. *Tight Fine-Grained Bounds for Direct Access on Join Queries*. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid **Libkin** and Pablo **Barceló**. PODS '22. Association for Computing Machinery, June 2022, pp. 427–436. doi 10.1145/3517804.3526234.

[BCM25] Karl **Bringmann**, Nofar **Carmeli**, and Stefan **Mengel**. *Tight Fine-Grained Bounds for Direct Access on Join Queries*. In *ACM Trans. Database Syst.* 50.1 (Jan. 2025). doi 10.1145/3707448.

[BDG07] Guillaume **Bagan**, Arnaud **Durand**, and Étienne **Grandjean**. *On Acyclic Conjunctive Queries and Constant Delay Enumeration*. In *Proceedings of the 21st International Conference, and Proceedings of the 16th Annual Conference on Computer Science Logic*. CSL'07/EACSL'07. Lausanne, Switzerland: Springer-Verlag, Sept. 2007, pp. 208–222. doi 10.1007/978-3-540-74915-8_18.

[BGS20] Christoph **Berkholz**, Fabian **Gerhardt**, and Nicole **Schweikardt**. *Constant Delay Enumeration for Conjunctive Queries: A Tutorial*. In *ACM SIGLOG News* 7.1 (Feb. 2020), pp. 4–33. doi 10.1145/3385634.3385636.

[Bra13] Johann **Brault-Baron**. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis. Université de Caen, Apr. 2013.

[Cap+25] Florent **Capelli**, Nofar **Carmeli**, Oliver **Irwin**, and Sylvain **Salvati**. *Direct Access for Conjunctive Queries with Negations*. Oct. 2025. arXiv 2310.15800.

[Car+20] Nofar **Carmeli**, Shai **Zeevi**, Christoph **Berkholz**, Benny **Kimelfeld**, and Nicole **Schweikardt**. *Answering (unions of) conjunctive queries using random access and random-order enumeration*. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2020, pp. 393–409. doi 10.1145/3375395.3387662.

[Car+23] Nofar **Carmeli**, Nikolaos **Tziavelis**, Wolfgang **Gatterbauer**, Benny **Kimelfeld**, and Mirek **Riedewald**. *Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries*. In *ACM Transactions on Database Systems* (Jan. 2023). doi 10.1145/3578517.

[CI24] Florent **Capelli** and Oliver **Irwin**. *Direct Access for Conjunctive Queries with Negations*. In *27th International Conference on Database Theory, ICDT 2024, March 24 to March 28, 2024, Paestum, Italy*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2024, 13:1–13:20. doi 10.4230/LIPICS.ICDT.2024.13.

[CIS25] Florent **Capelli**, Oliver **Irwin**, and Sylvain **Salvati**. *A Simple Algorithm for Worst Case Optimal Join and Sampling*. In *28th International Conference on Database Theory (ICDT 2025), March 25 to March 28, 2025, Barcelona, Spain* (ICDT '25). Ed. by Sudeepa **Roy** and Ahmet **Kara**. Vol. 328. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2025, 23:1–23:19. doi 10.4230/LIPIcs.ICDT.2025.23.

[CM77] Ashok K. **Chandra** and Philip M. **Merlin**. *Optimal Implementation of Conjunctive Queries in Relational Data Bases*. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, May 1977, pp. 77–90. doi 10.1145/800105.803397.

[CMN99] Surajit **Chaudhuri**, Rajeev **Motwani**, and Vivek **Narasayya**. *On Random Sampling over Joins*. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS99: International Conference on Management of Data and Symposium on Principles of Database Systems. Philadelphia Pennsylvania USA: ACM, June 1999, pp. 263–274. doi 10.1145/304182.304206.

[CR00] Chandra **Chekuri** and Anand **Rajaraman**. *Conjunctive Query Containment Revisited*. In *Theoretical Computer Science* 239.2 (May 2000), pp. 211–229. doi 10.1016/S0304-3975(99)00220-0.

[CY20a] Yu **Chen** and Ke **Yi**. *Random Sampling and Size Estimation Over Cyclic Joins*. In *LIPIcs, Volume 155, ICDT 2020* 155 (Mar. 2020). Ed. by Carsten **Lutz** and Jean Christoph **Jung**, 7:1–7:18. doi 10.4230/LIPICS.ICDT.2020.7.

[DJ04] Víctor **Dalmau** and Peter **Jonsson**. *The Complexity of Counting Homomorphisms Seen from the Other Side*. In *Theoretical Computer Science* 329.1-3 (Dec. 2004), pp. 315–323. doi 10.1016/j.tcs.2004.08.008.

[DLT23] Shiyuan **Deng**, Shangqi **Lu**, and Yufei **Tao**. *On Join Sampling and the Hardness of Combinatorial Output-Sensitive Join Algorithms*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '23: International Conference on Management of Data. Seattle WA USA: ACM, June 2023, pp. 99–111. doi 10.1145/3584372.3588666.

[ECK23] Idan **Eldar**, Nofar **Carmeli**, and Benny **Kimelfeld**. *Direct Access for Answers to Conjunctive Queries with Aggregation*. Mar. 2023. arXiv 2303.05327. preprint.

[FFG01] Jörg **Flum**, Markus **Frick**, and Martin **Grohe**. *Query Evaluation via Tree-Decompositions: Extended Abstract*. In *Database Theory — ICDT 2001*. Ed. by Jan **Van Den Bussche** and Victor **Vianu**. Red. by Gerhard **Goos**, Juris **Hartmanis**, and Jan **Van Leeuwen**. Vol. 1973. Berlin, Heidelberg: Springer Berlin Heidelberg, Oct. 2001, pp. 22–38. doi 10.1007/3-540-44503-X_2.

[Foc+25] Jacob **Focke**, Leslie Ann **Goldberg**, Marc **Roth**, and Stanislav **Živný**. *Approximately Counting Answers to Conjunctive Queries with Disequalities and Negations*. In *ACM Transactions on Algorithms* 21.1 (Jan. 2025), pp. 1–29. doi 10.1145/3689634.

[GLS00] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *A comparison of structural CSP decomposition methods*. In *Artificial Intelligence* 124.2 (Dec. 2000), pp. 243–282. doi 10.1016/S0004-3702(00)00078-3.

[GLS02] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *Hypertree Decompositions and Tractable Queries*. In *Journal of Computer and System Sciences* 64.3 (May 2002), pp. 579–627. doi 10.1006/jcss.2001.1809.

[GM14] Martin **Grohe** and Dániel **Marx**. *Constraint solving via fractional edge covers*. In *ACM Transactions on Algorithms (TALG)* 11.1 (Aug. 2014), p. 4. doi 10.1145/2636918.

[GR04] **Georg Gottlob** and **Reinhard Pichler**. *Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width*. In *SIAM Journal on Computing* 33.2 (Jan. 2004), pp. 351–378. doi 10.1137/S0097539701396807.

[GSS01] Martin **Grohe**, Thomas **Schwentick**, and Luc **Segoufin**. *When Is the Evaluation of Conjunctive Queries Tractable?* In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*. STOC01: 33rd ACM Symposium on Theory of Computing. Hersonissos Greece: ACM, July 2001, pp. 657–666. doi 10.1145/380752.380867.

[JVV86] Mark R. **Jerrum**, Leslie G. **Valiant**, and Vijay V. **Vazirani**. *Random generation of combinatorial structures from a uniform distribution*. In *Theoretical Computer Science* 43 (Nov. 1986), pp. 169–188. doi https://doi.org/10.1016/0304-3975(86)90174-X.

[Kep20] Jens **Keppeler**. *Answering Conjunctive Queries and FO+MOD Queries under Updates*. PhD thesis. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät, June 2020. doi 10.18452/21483.

[Kim+23] Kyoungmin **Kim**, Jaehyun **Ha**, George **Fletcher**, and Wook-Shin **Han**. *Guaranteeing the Õ(AGM/OUT) Runtime for Uniform Sampling and Size Estimation over Joins*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '23: International Conference on Management of Data. Seattle WA USA: ACM, June 2023, pp. 113–125. doi 10.1145/3584372.3588676.

[KV00] Phokion G. **Kolaitis** and Moshe Y. **Vardi**. *Conjunctive-Query Containment and Constraint Satisfaction*. In *Journal of Computer and System Sciences* 61.2 (Oct. 2000), pp. 302–332. doi 10.1006/jcss.2000.1713.

[Mar10] Dániel **Marx**. *Approximating fractional hypertree width*. In *ACM Trans. Algorithms* 6.2 (Apr. 2010). doi 10.1145/1721837.1721845.

[Ngo+12] Hung Q. **Ngo**, Ely **Porat**, Christopher **Ré**, and Atri **Rudra**. *Worst-Case Optimal Join Algorithms: [Extended Abstract]*. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '12: International Conference on Management of Data. Scottsdale Arizona USA: ACM, May 2012, pp. 37–48. doi 10.1145/2213556.2213565.

[Ngo18] Hung Q. **Ngo**. *Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems*. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '18: International Conference on Management of Data. Houston TX USA: ACM, May 2018, pp. 111–124. doi 10.1145/3196959.3196990.

[OZ15] Dan **Olteanu** and Jakub **Závodný**. *Size Bounds for Factorised Representations of Query Results*. en. In *ACM Transactions on Database Systems* 40.1 (Mar. 2015), pp. 1–44. doi 10.1145/2656335.

[PS13] Reinhard **Pichler** and Sebastian **Skritek**. *Tractable Counting of the Answers to Conjunctive Queries*. In *Journal of Computer and System Sciences* 79.6 (Sept. 2013), pp. 984–1001. doi 10.1016/j.jcss.2013.01.012.

[Val79] Leslie G. **Valiant**. *The Complexity of Enumeration and Reliability Problems*. In *SIAM Journal on Computing* 8.3 (Aug. 1979), pp. 410–421. doi 10.1137/0208032.

[Var82] Moshe Y. **Vardi**. *The Complexity of Relational Query Languages (Extended Abstract)*. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing - STOC '82*. The Fourteenth Annual ACM Symposium. San Francisco, California, United States: ACM Press, May 1982, pp. 137–146. doi 10.1145/800070.802186.

[Vel14] Todd L. **Veldhuizen**. *Triejoin: A Simple, Worst-Case Optimal Join Algorithm*. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. Ed. by Nicole **Schweikardt**, Vassilis **Christophides**, and Vincent **Leroy**. OpenProceedings.org, Mar. 2014, pp. 96–106. doi 10.5441/002/icdt.2014.13.

[WT24] Ru **Wang** and Yufei **Tao**. *Join Sampling Under Acyclic Degree Constraints and (Cyclic) Subgraph Sampling*. In *27th International Conference on Database Theory (ICDT 2024)*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Vol. 290. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2024, 23:1–23:20. doi 10.4230/LIPIcs.ICDT.2024.23.

[Yan81] Mihalis **Yannakakis**. *Algorithms for Acyclic Database Schemes*. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. Sept. 1981, pp. 82–94. doi 10.5555/1286831.1286840.

[Zha+18] Zhuoyue **Zhao**, Robert **Christensen**, Feifei **Li**, Xiao **Hu**, and Ke **Yi**. *Random Sampling over Joins Revisited*. In *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD/PODS '18: International Conference on Management of Data. Houston TX USA: ACM, May 2018, pp. 1525–1539. doi 10.1145/3183713.3183739.

Chapter 3

Join Evaluation via Branching Algorithms

If we represent knowledge as a tree, we know that things that are divided are yet connected.

Wendell Berry

In this chapter, we study a branching approach to join query evaluation. Our objective is to show that a conceptually elementary algorithm suffices to achieve worst-case optimality for join queries, provided that the input databases belong to classes defined by cardinality constraints or by acyclic degree constraints.

The motivation for this study comes from the limitations of traditional plan-based strategies, which may materialise large intermediate joins and therefore fail to run in time proportional to the worst-case output size (see Section 3.1). Existing worst-case optimal join algorithms, such as *Generic Join* [Ngo+12] and *Leapfrog TrieJoin* [Vel14], have addressed this issue in the past. However, their analyses rely on non-trivial bounds such as the AGM bound, making them somewhat opaque. Our approach instead isolates a simple property of query classes, that we call prefix-closedness, which immediately entails worst-case optimality for the algorithm without having to rely on heavy machinery.

The algorithm we present in this chapter can be viewed as a simplified form of Generic Join, where the technical burden of sophisticated data structures is replaced by a light domain size reduction trick. Despite this simplicity, it matches the guarantees of previous methods, while offering a cleaner analysis and a more transparent proof of optimality. Beyond joins, the same framework can be extended to more advanced tasks, such as uniform sampling of query answers, which we will later develop in Chapter 4.

Outline of the current chapter

3.1 Motivation: the Limitations of Classical Join Plans

In traditional query engines, a join query is executed by devising a binary join plan. This essentially reduces to choosing an order to join the relations pairwise. While this plan-based approach works well in many practical scenarios, it can also be suboptimal in the worst case. The core issue is that a binary join plan may produce extremely large intermediate results that are much bigger than the final output, incurring unnecessary costs. Indeed, in the worst case, the runtime of a classical plan can far exceed the theoretical minimum required to produce the answers, because the plan enumerates many combinations of tuples that do not end up in any final answer.

One of the most emblematic examples of this problem is the triangle query, such as defined earlier in Example 2.9.

▶ **Example 3.1** (The worst query plan possible)
Consider three sets $\mathsf{D}_1, \mathsf{D}_2$ and D_3 of same size $|\mathsf{D}_1| = |\mathsf{D}_2| = |\mathsf{D}_3| = N$. Let us define a domain D consisting of the disjoint union of these sets, that is $\mathsf{D} = \mathsf{D}_1 \uplus \mathsf{D}_2 \uplus \mathsf{D}_3$, and assume we also have $0 \notin \mathsf{D}$. Now let's consider the triangle query $Q_\Delta(x, y, z) :\text{-}\, \mathsf{R}(x, y), \mathsf{S}(x, z), \mathsf{T}(y, z)$, over the following database:

R	x	y
	0	D_2
	D_1	0

S	x	z
	0	D_3
	D_1	0

T	y	z
	0	D_3
	D_2	0

Whatever the intermediate join we choose, it will have a large size: $|\mathsf{R}(x, y),\ \mathsf{S}(x, z)| = |\mathsf{R}(x, y),\ \mathsf{T}(y, z)| = |\mathsf{S}(x, z),\ \mathsf{T}(y, z)| \geqslant N^2$. Without loss of generality, consider joining R and S. When x (the common variable in this case) is 0, it causes a Cartesian product between all possible values of D_2 and D_3, which already has size N^2. To these tuples, we have to add the tuples formed as $\langle x \leftarrow d, y \leftarrow 0, z \leftarrow 0\rangle$ where $d \in \mathsf{D}_1$, leading to an intermediate join size greater than N^2. However, in the end, because none of the D_i contain 0, there are no tuples in the database compatible with the query, thus $|\mathsf{ans}_\mathbf{D}(Q_\Delta)|$.

Even in the more general setting of considering any database for Q_Δ, the intermediate joins for the triangle query will all produce a set of tuples larger than the possible number of solutions. Atserias, Grohe, and Marx established an upper bound on the maximum number of output tuples queries like the triangle query, often called the AGM bound [AGM13]. This bound is based on the fractional cover number of the query hypergraph. In particular, for the triangle query, the AGM bound states that we can have at most $N^{1.5}$ answers in any database where the size of the relations is bounded by N. An ideal algorithm might run in time proportional to this worst-case output size, however, a conventional query plan will fail to meet this bound and produce intermediate joins asymptotically larger than the worst-case, as seen in Example 3.1.

This is a fundamental limitation of a fixed binary join order: it materialises large combinations of tuples too early. A traditional query optimiser has no robust way to avoid this pathological case using binary joins alone, since any join order will have a worst-case instance yielding a quadratic blow-up. Thus, classical plans are worst-case suboptimal: their runtime can grow super-linear in the size of the largest possible output. This shortcoming motivates the search for worst-case optimal join algorithms that avoid unnecessary work by never materialising more tuples than

needed to enumerate the actual results.

3.1.1 A shift in perspective

Branching algorithms fundamentally change the way we evaluate joins. Instead of following a predetermined binary join order, a branching algorithm explores the output space of the query directly. This is a different approach than the classical DBMS approach of building a simple query plan and executing it. Conceptually, a join problem can be viewed as a constraint satisfaction problem. As such, some research directions have focused on using tools and algorithms used in constraint satisfaction problems (such as in SAT solvers) to solve joins. A branching algorithm works by incrementally constructing answer tuples by assigning values to query variables, one variable at a time, while ensuring that at each step the partial assignment is consistent with all relations in the query. In other words, each intermediate step corresponds to a partial tuple (a binding for some subset of the query's variables). This way, combinations of tuples that would have been formed in a binary join but do not lead to any final answer are never fully materialised.

Formally, consider a join query $Q(X) :- \mathsf{R}_1(\mathbf{x_1}), \mathsf{R}_2(\mathbf{x_2}), \ldots, \mathsf{R}_m(\mathbf{x_m})$ over a set of variables $X = \{x_1, \ldots, x_n\}$. A branching strategy does not choose an arbitrary join order of the relations, but instead it typically chooses an ordering $\prec$ of the variables $x_1 \prec x_2 \prec \cdots \prec x_n$. The algorithm will assign values to the variables in sequence. When setting a particular variable $x_j \leftarrow d$, the algorithm checks all relations that constrain x_j to ensure there is at least one matching tuple consistent with the current partial assignment. If at any point an assignment for x_j cannot be extended to some relation (because it would violate a join condition), that branch is backtracked and abandoned immediately. By systematically exploring only the join-consistent combinations, a branching algorithm strongly limits the risk that it enumerates a tuple that would fail to be in the final output. Of course, in the case where a variable assignment is consistent right until the last variable, then the algorithm still branches up to the last variable. Consequently, this allows for a finer refinement on the bound over the total number of materialised tuples.

This has also found a place in the search for worst-case optimal join algorithms. If Ngo, Porat, Ré, and Rudra's work [Ngo+12] is not a branching algorithm, both Leapfrog Triejoin and Generic Join are. Leapfrog Triejoin [Vel14] uses a trie-based index structure to perform a multi-way join evaluation efficiently. Their approach was later generalised as the Generic Join algorithm, which uses a recursive variable-assignment procedure guided by a variable ordering and data structures that quickly find relevant domain values for the next variable [Ngo18]. Both these algorithms were shown to be worst-case optimal. however, one downside of these methods such as they are presented is that they both rely on proofs from [Ngo+12], making the optimality proof rather complicated.

These approaches laid the groundwork for worst-case optimal join evaluation. In the next section, we introduce a conceptually simpler algorithm which achieves the same guarantees under standard constraints.

3.1.2 Our contribution: A Simple Worst-Case Optimal Join Algorithm

Despite the advances described above, implementing a worst-case optimal join algorithm can be complex. Both GenericJoin and Triejoin rely on non-trivial data structures such as tries or intersection algorithms to avoid iterating over irrelevant values. In this dissertation, we introduce a new algorithm that achieves worst-case optimality with a conceptually simple branch-and-bound strategy. The algorithm explicitly searches through variable assignments as described, and can be viewed as an extremely simplified variant of GenericJoin or Triejoin. In its most naive form, this strategy would assign query variables one by one, trying every possible domain value and

backtracking when a conflict with some relation is detected. Naively branching on every value in a large domain D would incur an extra factor of $|\mathsf{D}|$ in the running time, thus preventing worst-case optimality. Our contribution is to eliminate this overhead by a technique of branching on values bit-by-bit rather than value-by-value. That is, the algorithm performs a binary decomposition of the domain and assigns bits of the value at each step. By testing partial bit assignments against the relations, the search space is dramatically pruned: large batches of impossible values are skipped without enumerating each value explicitly. This trick ensures that the algorithm runs in time proportional to the output bound (up to polylogarithmic factors) even though it does not use complex indexing structures. In essence, we retain the simplicity of a brute-force search while guaranteeing worst-case optimal performance on join queries under the usual constraints (bounded relation sizes or even more general degree constraints, as we later show).

The proposed algorithm, first presented in "A Simple Algorithm for Worst Case Optimal Join and Sampling" [CIS25], not only matches the performance of prior WCOJ algorithms in theory, but does so with a clean and modular analysis. Our approach thus demonstrates that worst-case optimal join processing can be achieved with a very elementary procedure, bridging the gap between theoretical optimality and practical simplicity. The remainder of this chapter delves into the details of this algorithm, explain its analysis, and compare it with the classical approaches and prior work on worst-case optimality.

3.2 A Branching Algorithm for Join Queries

In this section, we describe a simple branching algorithm to evaluate join queries and provide an easy upper bound on its complexity. The algorithm can be seen as an instance of Generic Join from [Ngo18], but it is given in an extremely simple form and its analysis is elementary. However, in this way, the algorithm is not yet worst-case optimal. We will delay the proof that this algorithm is also worst-case optimal to Section 3.3, concentrating this section on the core of the algorithm and the basis of the complexity analysis.

3.2.1 The algorithm

The algorithm, whose pseudocode is given in Algorithm 3.2, is a simple recursive search: assume a fixed order $\prec$ with $x_1 \prec x_2 \prec \cdots \prec x_n$ is given on variables X. We find the answers of Q over a database instance $\mathbf{D}$ by setting variables sequentially according to this order, trying each possible value in the domain. Whenever the current partial assignment is inconsistent with Q, it is not further expanded. If every variable is assigned and the assignment is consistent with Q, then it is output.

We can visualise the trace of a run of this algorithm as a tree where the nodes are labelled by the variables and the edges by the domain values. A path from the root of the tree, that is the first variable in the considered order, to a leaf gives a tuple. The leaves are labelled in a binary fashion, indicating whether the considered tuple is a solution to the query over the database or not.

▶ **Example 3.3** (Joining the triangle query)
Consider the (now classical) triangle query Q_Δ :- $\mathsf{R}(x, y)$, $\mathsf{S}(x, z)$, $\mathsf{T}(y, z)$ over the following database instance:

Algorithm 3.2 An algorithm to compute join queries.

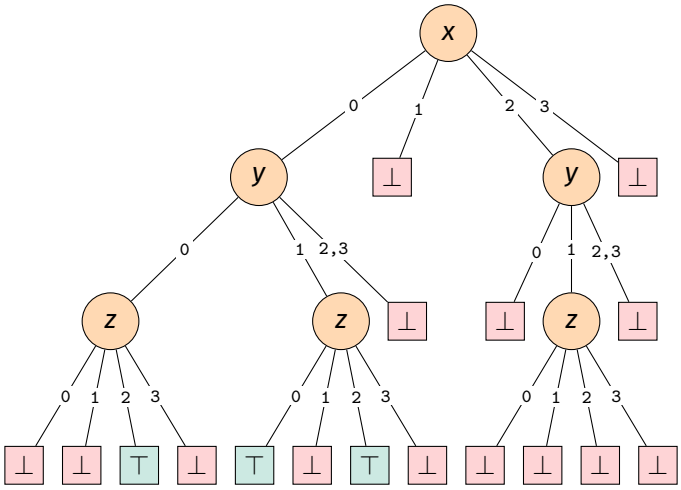
procedure WCJ$(Q, \mathbf{D}, \tau, \prec)$
  input: · Q a join query,
    · $\mathbf{D}$ a database instance for Q,
    · τ a partial tuple assignment,
    · $\prec$ an order on the variables
  output: the answer set of Q over $\mathbf{D}$
  if $\mathbf{D}[\tau]$ contains an empty relation **then exit** ▷ *τ is inconsistent with the database*
  $i \leftarrow$ last variable assigned by τ
  if $i = |\mathsf{vars}(Q)|$ **then output** τ ▷ *all variables assigned with no inconsistency*
  for all $d \in \mathsf{dom}(\mathbf{D})$ **do**
    WCJ$(Q, \mathbf{D}, \tau \times \langle x_{i+1} \leftarrow d \rangle)$ ▷ *continue assigning variables*

R	x	y
	0	0
	0	1
	2	1

S	x	z
	0	0
	0	2
	2	3

T	y	z
	0	2
	1	0
	1	2

A run of Algorithm 3.2 has a trace that can be visualised as the following tree:



Here, the considered variable order is (x, y, z). In this example, the tree solutions of Q_Δ over the database are represented by the three paths from the root to a leaf labelled $\top$.

Correctness of the algorithm. We introduce two notations to describe sets of tuples in this context. First, for a subset of variables $Y \subseteq X$, we denote by $\mathsf{ans}^Y_{\mathbf{D}}(Q)$ the set of tuples over variables Y that are still consistent with Q. That is, $\mathsf{ans}^Y_{\mathbf{D}}(Q) = \{\tau \mid \tau \in \mathsf{D}^Y, \tau \text{ is consistent with } Q\}$. Second, for a partial assignment τ, we denote by $\mathsf{ans}_{\mathbf{D}}(Q, \tau)$ the set of tuples from the answer set of the query that are compatible with τ, $\mathsf{ans}_{\mathbf{D}}(Q, \tau) = \{\sigma \mid \sigma \in \mathsf{ans}_{\mathbf{D}}(Q), \sigma \simeq \tau\}$.

The proof of the correctness of Algorithm 3.2 is made easier by these previous definitions. We will now show that Algorithm 3.2 is correct, in the sense that it outputs all the answers of the query over the database, and no extra tuple. Starting with a call $\mathsf{WCJ}(Q, \mathbf{D}, \langle\rangle)$, every recursive call is of the form $\mathsf{WCJ}(Q, \mathbf{D}, \tau)$ where τ is a tuple in D^{X_i} where $X_i := \{x_1, \ldots, x_i\}$. We claim that for every τ which assigns variables X_i, then $\mathsf{WCJ}(Q, \mathbf{D}, \tau)$ outputs $\tau \times \sigma$ for every answer σ of $\mathsf{ans}_{\mathbf{D}}(Q, \tau)$. The proof is done by induction over i.

If $i = n$, then τ is output if and only if $\mathsf{ans}_{\mathbf{D}}(Q, \tau)$ does not contain the empty relation, which, by definition, means that τ is an answer of Q. Now assume $i < n$. If τ is inconsistent with Q then nothing is output, this is coherent with our induction hypothesis since $\mathbf{D}[\tau]$ contains an empty relation, meaning that any tuple σ such that $\sigma_{|X_i} = \tau$ is not in $\mathsf{ans}_{\mathbf{D}}(Q)$. Otherwise, by induction, for every $d \in \mathsf{D}$, $\mathsf{WCJ}(Q, \mathbf{D}, \tau \times \langle x_{i+1} \leftarrow d\rangle)$ outputs $\tau \times \langle x_{i+1} \leftarrow d\rangle \times \sigma$ for every $\sigma \in \mathsf{ans}_{\mathbf{D}}(Q, \tau \times \langle x_{i+1} \leftarrow d\rangle)$, that is, for every $\sigma \in \mathsf{ans}_{\mathbf{D}}(Q, \tau)$. It completes the induction, and it directly follows that $\mathsf{WCJ}(Q, \mathbf{D}, \langle\rangle)$ outputs $\mathsf{ans}_{\mathbf{D}}(Q)$.

Now that we have shown this algorithm to be correct, we can analyse its complexity.

3.2.2 A simple complexity analysis

In this section, we compute the complexity of Algorithm 3.2 in a simple way. Since this is a recursive algorithm, we can proceed by first estimating the number of recursive calls and then studying the complexity of each call.

Number of recursive calls. We claim that Algorithm 3.2 does at most $(1+|\mathsf{D}|)\cdot\sum_{i\leqslant n}|\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)|$ recursive calls, where X_i is the set of variables $\{x_1, \ldots, x_i\}$. Indeed, as stated before, every recursive call is of the form $\mathsf{WCJ}(Q, \mathbf{D}, \tau)$ where τ is a tuple of D^{X_i}.

We will split the analysis in two distinct cases. In the first case, assume that τ is consistent with Q, which means in particular that τ is in $\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)$ by definition. Hence, there are at most $\sum_{i\leqslant n}|\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)|$ recursive calls of this type.

In the second case, assume that τ is inconsistent with Q. Since the algorithm backtracks when finding an inconsistency, this implies that the tuple from the call that issued τ was consistent. More formally, the recursive call with parameters $(Q, \mathbf{D}, \tau)$ has been issued from a call of the form $(Q, \mathbf{D}, \tau')$ where $\tau = \tau' \times \langle x_i \leftarrow d\rangle$ for some $d \in \mathsf{D}$. In particular, τ' is consistent with Q, otherwise, such a recursive call would not have happened. Hence, $\tau' \in \mathsf{ans}_{\mathbf{D}}^{X_{i-1}}(Q)$ and there are at most $|\mathsf{D}|$ possible τ for a given $\tau' \in \mathsf{ans}_{\mathbf{D}}^{X_{i-1}}(Q)]$.

Therefore, there are at most $|\mathsf{D}| \cdot \sum_{i\leqslant n}|\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)]|$ recursive calls of this form, this in total, $(|\mathsf{D}|+1)\sum_{i\leqslant n}|\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)|$ recursive calls.

Efficient implementation. Now we explain how, using a very simple data structure, one can assume that each recursive call is executed in time $\tilde{\mathcal{O}}(m)$ where m is the number of atoms in Q. By using this data structure, the explicit dependence on the size of the database will be absorbed into the $\tilde{\mathcal{O}}(\cdot)$ notation. The only non-trivial thing is to check whether $\mathbf{D}[\tau]$ contains an empty relation. To do that, we may simply assume that every relation is given sorted in lexicographical order, for the variable order $x_1 \prec x_2 \prec \cdots \prec x_n$. Note that this could be obtained with a preprocessing that is quasi-linear in the data (or even linear in the RAM model, but since we ignore polylogarithmic factors, it does not matter much).

In our situation, observe that, if R is a relation of $\mathbf{D}$ and τ a tuple in D^{X_i}, then all tuples from $\mathsf{R}[\tau]$ are consecutively stored in the table. Hence, we can represent $\mathsf{R}[\tau]$ by keeping two pointers p_1 and p_2 on the tuples of R: one towards the first tuple and one towards the last tuple in $\mathsf{R}[\tau]$. To go from the representation of $\mathsf{R}[\tau]$ to the representation of $\mathsf{R}[\tau \times \langle x_{i+1} \leftarrow d\rangle]$, we simply need to find the first and last tuple between p_1 and p_2 where $x_{i+1} = d$. If no such tuple exist, p_1 and p_2 can be undefined. This can be done via a binary search in time $\mathcal{O}(\log|\mathsf{R}|)$. To check whether $\mathsf{R}[\tau]$ is consistent, it is enough to check that both pointers are defined, and, as a sanity check, that $p_1 \leqslant p_2$. Hence, each recursive join can be executed in time $\mathcal{O}(m\log|\mathbf{D}|)$, that is, $\tilde{\mathcal{O}}(m)$.

▶ **Example 3.4**

R	x	y
→	0	0
	1	0
	1	1
	2	1 ←

(a)

R	x	y
	0	0
→	1	0
	1	1 ←
	2	1

(b)

R	x	y
	0	0
	1	0
	1	1
	2	1

(c)

In this simple example, we start in Table **(a)** with R and an empty tuple $\tau = \langle\rangle$. The pointers p_1 (in green, pointing right) and p_2 (in red, pointing left), are towards the first and the last tuples of $\mathsf{R}[\tau]$.

We then suppose that the algorithm moves to $\tau' = \tau \times \langle x \leftarrow 1\rangle$. What happens is then illustrated in Table **(b)**, where the pointers have move towards the centre of the table. Both pointers are still defined and define a non-empty range of tuples, so $\mathsf{R}[\tau']$ is not empty.

If we now consider $\tau'' = \tau' \times \langle y \leftarrow 2\rangle$ for example, we get the situation of Table **(c)**, where the pointers are no longer defined. This is because there are no tuples in R that are consistent with τ'' and thus, that this assignment cannot yield any answer.

Remark that a slightly more involved data structure would allow us to compute in time $\mathcal{O}(m)$, that is, suppressing completely the polylogarithmic factors, by representing R as a trie as in [Vel14].

In short, we have just proved:

▶ **Theorem 3.5** (Complexity of Algorithm 3.2)
Given a join query Q with m atoms, $x_1 \prec x_2 \prec \cdots \prec x_n$) an order on the variables of Q and $\mathbf{D}$ a database instance for Q on domain D, $\mathsf{WCJ}(Q, \mathbf{D}, \langle\rangle, \prec)$ computes $\mathsf{ans}_{\mathbf{D}}(Q)$ in time

$\tilde{\mathcal{O}}(m|\mathsf{D}| \cdot \sum_{i \leqslant n} |\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)|)$, where $X_i = \{x_1, \ldots, x_i\}$.

3.3 Worst-Case Optimality

The complexity of Algorithm 3.2 presented in Theorem 3.5 does not meet the criteria for worst-case optimality as we defined in Section 2.2.1, since we do not reference the worst-case (see Definition 2.8) and we depend on the size of the domain $|\mathsf{D}|$, which is not a parameter from the query. In this section, we will expand on some techniques that allow us to show that this algorithm is indeed worst-case optimal.

To do this, we first present two different database classes that we will reference in the rest of this work, then show an important property of these classes: prefix-closedness. We then present an algorithmic trick to reduce the domain size while preserving the necessary properties and move on to prove the worst-case optimality of Algorithm 3.2.

3.3.1 Database constraints

Cardinality Constraints. In practical database management systems, cardinality constraints play a central role: query optimisers rely on estimates of relation sizes to choose efficient execution plans. Motivated by this practical importance, the theoretical study of worst-case optimal joins has also considered classes of queries defined by such constraints.

Let Q be a join query, $\mathbf{D}$ be a database for Q with m relations and $\mathbf{N} \in \mathbb{N}^m$.

We say that a database $\mathbf{D}$ satisfies a *cardinality constraint* $\mathbf{N} \in \mathbb{N}^m$ if, for every relation $\mathsf{R}_i \in \mathbf{D}$, we have $|\mathsf{R}_i| \leqslant \mathbf{N}(i)$. Let $\mathcal{C}(\leqslant \mathbf{N})$ denote the class of all such databases. The worst-case output size for this class is finite, since

$$\mathsf{wc}(Q, \mathcal{C}(\leqslant \mathbf{N})) \leqslant \prod_{i \leq m} \mathbf{N}(i).$$

Where $\mathsf{wc}(Q, \mathcal{C})$ is the expression of the worst case defined in Definition 2.8, that is, the size of the largest possible answer set possible for Q with a database $\mathbf{D} \in \mathcal{C}$.

A sharper and asymptotically optimal bound is given by the AGM bound [GM14; AGM13], which characterises exactly the worst-case output size over $\mathcal{C}(\leqslant \mathbf{N})$.

Degree constraints. Another class of join queries which received attention in the literature on worst-case optimal joins is the class of queries defined with degree constraints. Degree constraints were introduced by Abo Khamis, Ngo, and Suciu in order to model practical cases of using input statistics and key constraints [ANS16; ANS17b].

Given two (possibly empty) sets of variables $A \subseteq B$, a *degree constraint* is a triplet $(A, B, N_{B|A})$ with $N_{B|A} > 0$. A relation R satisfies such a constraint if, for every assignment $\tau \in \mathsf{D}^A$, the number of extensions to B is bounded by $N_{B|A}$, that is:

$$\mathsf{max}_{\tau \in \mathsf{D}^A} |\mathsf{R}[\tau]_{|B}| \leqslant N_{B|A}$$

Degree constraints generalise cardinality constraints (take $A = \emptyset$, $B = \mathsf{var}(\mathsf{R})$) and functional dependencies (a functional dependency $X \to y$ can be seen as $(X, X \cup \{y\}, 1)$). We denote by $\mathcal{C}(\mathsf{DC})$ the class of databases satisfying a given set of degree constraints DC. An example of how having a functional dependency can directly influence the bound on the output size of a query is presented in Example 3.6.

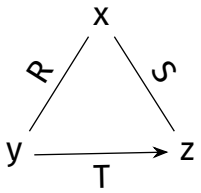
Given a set of degree constraints DC and a query Q, we define the *constraint dependency graph* G_{DC} as the directed graph with vertex set $\mathsf{var}(Q)$. This graph has an edge from a to b for every constraint $(A, B, N_{B|A}) \in \mathsf{DC}$ and every pair $(a, b) \in A \times (B \setminus A)$. If G_{DC} is acyclic, then we say that DC is a set of *acyclic degree constraints*.

Observe that it may happen that $\mathsf{wc}(Q, \mathcal{C}(\mathsf{DC})) = +\infty$. In this thesis however, we are only interested in classes where this does not happen. This is often enforced by assuming that for every relation R in the databases, at least one constraint in DC is a cardinality constraint of the form $(\emptyset, \mathsf{vars}(\mathsf{R}), N_{\mathsf{R}})$ satisfied by R. In this case, as before, it is clear that $\mathsf{wc}(Q, \mathcal{C}(\mathsf{DC})) \leqslant \prod_{\mathsf{R}} N_{\mathsf{R}} < +\infty$. Here again, more precise upper bounds are known on $\mathsf{wc}(Q, \mathcal{C}(\mathsf{DC}))$, but they will not be necessary for the analysis of our worst case optimal join algorithm presented in Algorithm 3.2 and we therefore delay this discussion to Chapter 4 where we will need them.

▶ **Example 3.6** (Adding a functional dependency)
Consider the triangle query $Q(x, y, z) :- \mathsf{R}(x, y), \mathsf{S}(x, z), \mathsf{T}(y, z)$, but this time, suppose that we have a functional dependency such that $y \to z$, that is, fixing the value of y fixes the value of z.

This query can be visualised as the following graph, where the arrowed edge represents the functional dependency.

The functional dependency here implies that each value for y can be associated with *at most one* value of z.
We can freely join R and T without asymptotic blow-up, since T can only yield one value of z per value of y. This intermediate join will therefore result in at most $|\mathsf{R}|$ tuples.
Finally, we can do the join with S without blow-up once again, since it simply consists in filtering the values of $\mathsf{R} \bowtie \mathsf{T}$ to those consistent with S.



As a result of adding this functional dependency, the join output cannot exceed $|\mathsf{R}|$ tuples and is therefore of size $\mathcal{O}(N)$. While this is still consistent with the AGM bound of $\mathcal{O}(N^{1.5})$, this bound is no longer optimal. This shows how incorporating functional dependencies can yield strictly better bounds than the AGM bound applied naively.

3.3.2 Prefix-closed Classes

To show that Algorithm 3.2 is worst-case optimal queries evaluated on a class $\mathcal{C}$ of instances, we need to bound the complexity from Theorem 3.5 by $\tilde{\mathcal{O}}(\mathsf{wc}(\cdot, \mathcal{C}))$. Of course, this will not be true for any class of instances, but it turns out that we can easily do so on classes defined by cardinality constraints or by acyclic degree constraints. Theorem 3.5 motivates the following definition:

▶ **Definition 3.7** (Prefix-closed Classes)
Let $\mathcal{C}$ be a class of database instances for join queries over variables X. Let $\prec$ be an order on the variables X with $x_1 \prec x_2 \prec \cdots \prec x_n$.

We say that $\mathcal{C}$ is *prefix-closed* for the variable order $\prec$ if, and only if, for every $i \leqslant n$ and any join query Q over variables X that can be evaluated over $\mathbf{D} \in \mathcal{C}$:

$$|\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)| \leqslant \mathsf{wc}(Q, \mathcal{C}) \ .$$

We call these classes "prefix-closed" since, while the full answer size is already bounded by the worst-case, any prefix projectino along the order $\prec$ is also bounded by the worst-case. At no point does a prefix create a combinatorial explosion, we are always inside the same worst-case bound.

This definition has a strong link to Theorem 3.5, since, if $\mathcal{C}$ is prefix-closed for the order $\prec$, then computing $\mathsf{ans}_{\mathbf{D}}(Q)$ for Q and $\mathbf{D} \in \mathcal{C}$ using Algorithm 3.2 with order $\prec$ will take $\tilde{\mathcal{O}}(mn \cdot |\mathsf{D}| \cdot \mathsf{wc}(Q, \mathcal{C}))$, where D is the domain of Q.

This is formalised in the following theorem:

▶ **Theorem 3.8**
For every class $\mathcal{C}$ that is prefix-closed for variables $X = \{x_1, \ldots, x_n\}$ ordered by $\prec$, join query Q over variables X and with m relations and for every database $\mathbf{D} \in \mathcal{C}$, $\mathsf{WCJ}(Q, \mathbf{D}, \langle\rangle, \prec)$ returns $\mathsf{ans}_{\mathbf{D}}(Q)$ in time $\tilde{\mathcal{O}}(nm \cdot |\mathsf{D}| \cdot \mathsf{wc}(Q, \mathcal{C}))$.

Proof. This is a restatement of the result from Theorem 3.5. Since we are in a prefix-closed setting, we can bound the $|\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)|$ term by $\mathsf{wc}(Q, \mathcal{C})$ for all $i \leqslant n$. Therefore, we have that $\sum_{i \leqslant n} |\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)| \leqslant n \cdot \mathsf{wc}(Q, \mathcal{C})$ and the proof is complete. □

While m and n are considered constant in our setting, we cannot assume so for $|\mathsf{D}|$. Hence, Theorem 3.5 and prefix-closedness will not be enough to establish worst-case optimality of Algorithm 3.2. With that said, in Section 3.3.3, we present a simple trick which allows circumventing this issue easily.

The classes from Section 3.3.1 for which worst-case optimal algorithms are known are prefix-closed, at least for one order on the variables. Even if cardinality constraints are less general than degree constraints, we start by showing this property for the former as a warm-up, even if the proof is essentially the same for the latter.

▶ **Theorem 3.9**
Let $\mathcal{C}(\leqslant \mathbf{N})$ be the class of databases for a join query Q and satisfying a cardinality constraint vector $\mathbf{N}$. Then $\mathcal{C}(\leqslant \mathbf{N})$ is prefix-closed for **every order**.

Proof. Let Q be a join query, $\mathbf{D} \in \mathcal{C}(\leqslant \mathbf{N})$, $x_1 \prec x_2 \prec \cdots \prec x_n$ be an order on X and $i \leqslant n$. We want to show that $|\mathsf{ans}_{\mathbf{D}}(Q_{|X_i})| \leqslant \mathsf{wc}(Q, \mathcal{C}(\leqslant \mathbf{N}))$, that is, projecting the query to the first i variables does not increase the worst-case output size. To do so, we construct a database $\mathbf{D}^* \in \mathcal{C}(\leqslant \mathbf{N})$ such that $|\mathsf{ans}_{\mathbf{D}_{|X_i}}(Q)| = |\mathsf{ans}_{\mathbf{D}^*}(Q)|$. Since $\mathbf{D}^* \in \mathcal{C}(\leqslant \mathbf{N})$, we have by definition that $|\mathsf{ans}_{\mathbf{D}^*}(Q)| \leqslant \mathsf{wc}(Q, \mathcal{C}(\leqslant \mathbf{N}))$, hence $|\mathsf{ans}_{\mathbf{D}_{|X_i}}(Q)| \leqslant \mathsf{wc}(Q, \mathcal{C}(\leqslant \mathbf{N}))$.

We now show how to build such a database. Assume that the database is built on domain $\mathsf{D} \neq \emptyset$ and let $d \in \mathsf{D}$ be some fixed element of D. We denote by $d^Y \in \mathsf{D}^Y$ the tuple defined as

$d^Y(y) = d$ for every $y \in Y$. Let $R \in \mathbf{D}_{|X_i}$. By definition, $\mathsf{R} = \mathsf{R}_{k_{|X_i}}$ for some $\mathsf{R}_k \in \mathbf{D}$. Hence, $|\mathsf{R}| \leqslant |\mathsf{R}_k| \leqslant \mathbf{N}(k)$. We then define $\mathsf{R}^*_k \subseteq \mathsf{D}^{\mathsf{var}(\mathsf{R}_k)}$ as $\mathsf{R} \times \{d^{\mathsf{var}(\mathsf{R}_k)\setminus X_i}\}$, that is, we extend every tuple from R to the original variables $\mathsf{var}(\mathsf{R}_k)$ by setting every missing variable to d. This padding ensures each projected tuple extends uniquely, so the number of answers is preserved.

Clearly, $|\mathsf{R}^*_k| = |\mathsf{R}| \leqslant |\mathsf{R}_k| \leqslant \mathbf{N}(k)$. Hence the database $\mathbf{D}^*$ defined as $\{\mathsf{R}^*_k \mid \mathsf{R} \in \mathbf{D}_{|X_i}\}$ is in $\mathcal{C}(\leqslant \mathbf{N})$. Moreover, we clearly have $\mathsf{ans}_{\mathbf{D}^*}(Q) = \mathsf{ans}_{\mathbf{D}_{|X_i}}(Q) \times \{d^{X\setminus X_i}\}$, therefore $|\mathsf{ans}_{\mathbf{D}^*}(Q)| = |\mathsf{ans}_{\mathbf{D}_{|X_i}}(Q)|$ as needed to complete the proof. □

We now generalise the previous result to classes defined via degree constraints. Observe however that such classes may not always be prefix-closed, or sometimes only for some particular order. An example of such a case is presented in Example 3.10.

▶ **Example 3.10** (For whom the order matters)
Consider the following query:

$$Q(x_1, x_2, x_3) \text{ :- } \mathsf{R}(x_1, x_3),\ \mathsf{S}(x_2, x_3)$$

and consider that the class of database instances $\mathcal{C}$ we consider when evaluating this query satisfy the functional dependencies $x_3 \to x_1$ and $x_3 \to x_2$. Equivalently, this implies that x_3 acts as a sort of key in these databases: once it is fixed, the values of x_1 and x_2 are uniquely determined. Assume that this class $\mathcal{C}$ also satisfies the following cardinality constraints: $|\mathsf{R}| \leqslant N$ and $|\mathsf{S}| \leqslant N$.

Then clearly, the worst case here is $\mathsf{wc}(Q, \mathcal{C}) \leqslant N$ since each value of x_3 can yield at most one answer tuple.

Let us however examine how our branching algorithm from Algorithm 3.2 behaves with different variable orders.

▶ **Order** (x_1, x_2, x_3). Suppose that we branch first on x_1 and x_2. Each of these variables can take N possible values, so Algorithm 3.2 will explore roughly N^2 partial assignments before realising that only at most N of these are consistent with the functional dependencies. The intermediate search tree thus grows quadratically larger than the final answer set. This leads to this conclusion: $|\mathsf{ans}_{\mathbf{D}}(Q_{|\{x_1,x_2\}})| > \mathsf{wc}(Q, \mathcal{C})$.

Remark that this is completely symmetrical for the order (x_2, x_1, x_3).

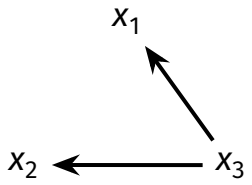
▶ **Order** (x_3, x_1, x_2). This time, we start by branching on x_3. In this case, the functional dependencies immediately fix the values of x_1 and x_2. Algorithm 3.2 therefore only explores at most N branches, matching worst-case complexity.

Again, remark that this is completely symmetrical for the order (x_3, x_2, x_1).

▶ **Order** (x_1, x_3, x_2). Finally, suppose we start by branching on x_1, but follow with x_3. In a similar way as in the first case we considered, x_1 can take N possible values. After this, x_3 can also take N possible values, leading to an intermediate search tree of size roughly N^2, which is larger than the worst-case. This is because we are not taking advantage in this case of the functional dependency. We explore all the possible values for x_1 before x_3, which might lead to many "dead-ends" immediately following this.

If we build the constraint dependency graph for $\mathcal{C}$, we obtain the graph represented on the right.

The main difference between the variable orders (x_1, x_2, x_3) and (x_3, x_1, x_2) is that the latter is a topological sort of this graph.



This motivates the following definition: if we consider a set DC of acyclic degree constraints, an order $(x_1, \dots, x_n)$ is said to be *compatible* with DC if it is a topological sort of the constraint dependency graph G_{DC}. The class of databases defined with acyclic degree constraints is denoted $\mathcal{C}(\mathsf{DC})$. Unsurprisingly, this allows to prove the following generalisation of Theorem 3.9:

▶ **Theorem 3.11**
Let $\mathcal{C}(\mathsf{DC})$ be the class of databases for a join query Q satisfying a set of acyclic degree constraints DC. Then $\mathcal{C}(\mathsf{DC})$ is prefix-closed for every order compatible with DC.

Proof. The proof is very similar to the proof of Theorem 3.9. Let Q be a join query, $\mathbf{D} \in \mathcal{C}(\mathsf{DC})$ and $i \leqslant n$. We construct $\mathbf{D}^*$ as in Theorem 3.9. We still have $|\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)| = |\mathsf{ans}_{\mathbf{D}^*}(Q)|$. We only have to check that the degree constraints still hold for $\mathbf{D}^*$.

Let $(A, B, N_{B|A}) \in \mathsf{DC}$ be a degree constraint. By definition, it is satisfied by a relation R of $\mathbf{D}$ on variables $\mathsf{var}(\mathsf{R}) \supseteq B$. We claim that $\mathsf{R}^* \in \mathbf{D}^*$, defined in the same way as in Theorem 3.9 also satisfies δ. Indeed, if $X_i \cap \mathsf{var}(\mathsf{R}) \subseteq A$, then for every $\tau \in \mathsf{D}^A$, there is at most one tuple in $\mathsf{R}^*[\tau]$ which is $\tau \times d^{\mathsf{var}(\mathsf{R}) \setminus X_i}$, hence $|\mathsf{R}^*[\tau]_{|B}| \leqslant 1 \leqslant N_{B|A}$. Otherwise, since the order is compatible with DC, $A \subseteq X_i$. Therefore, $\mathsf{R}^*[\tau] = \mathsf{R}_{|X_i}[\tau] \times d^{\mathsf{var}(\mathsf{R}) \setminus X_i}$. In particular $|\mathsf{R}^*[\tau]| = |\mathsf{R}_{|X_i}[\tau]| \leqslant |\mathsf{R}[\tau]|$. In this case, projecting to B, $|\mathsf{R}_{|X_i}[\tau]_{|B}| \leqslant |\mathsf{R}[\tau]_{|B}| \leqslant N_{B|A}$ since R satisfies the degree constraint $(A, B, N_{B|A})$. Hence, R^* also satisfies this degree constraint. Since this reasoning works for every relation $\mathsf{R}^* \in \mathbf{D}^*$, we conclude that the degree constraints still hold for $\mathbf{D}^*$.

We now see that $|\mathsf{ans}_{\mathbf{D}}^{X_i}(Q)| = |\mathsf{ans}_{\mathbf{D}^*}(Q)| \leqslant \mathsf{wc}(Q, \mathcal{C}(\mathsf{DC}))$, which is what we needed to prove. □

A direct corollary of Theorems 3.8 and 3.11 is that Algorithm 3.2 is almost worst-case optimal on classes defined by acyclic degree constraints.

▶ **Corollary 3.12**
Let Q be a join query. Let $\mathcal{C}(\mathsf{DC})$ be a class of databases for Q with m relations over n variables, D be the domain of the databases, and DC a set of acyclic degree constraints.

Assume $(x_1, \dots, x_n)$ is a variable order compatible with DC. Then for every $\mathbf{D} \in \mathcal{C}(\mathsf{DC})$, $\mathsf{WCJ}(Q, \langle\rangle)$ returns $\mathsf{ans}_{\mathbf{D}}(Q)$ in time $\tilde{\mathcal{O}}(mn \cdot |\mathsf{D}| \cdot \mathsf{wc}(Q, \mathcal{C}(\mathsf{DC})))$.

Observe that in order to prove the almost worst-case optimality of Algorithm 3.2 in Corollary 3.12, we have not used any knowledge of the actual value of the worst-case $\mathsf{wc}(Q, \mathcal{C}(\mathsf{DC}))$. This makes our approach simpler than the existing analysis of worst-case optimal join algorithms. While such knowledge is also not necessary in the analysis of Leapfrog TrieJoin [Vel14], the complexity analysis given in that paper is more complicated.

3.3.3 Binarisation

We have seen that Algorithm 3.2 achieves time $\tilde{\mathcal{O}}(mn \cdot |\mathsf{D}| \cdot \mathsf{wc}(Q, \mathcal{C}))$ complexity for a query Q when $\mathcal{C}$ is prefix-closed (see Theorem 3.8). However, this does not qualify this algorithm as a worst-case optimal join yet. Indeed, we have an extra $|\mathsf{D}|$ factor that stems from the fact that we are testing every possible value of $d \in \mathsf{D}$ for each variable, even if many of them will directly lead to inconsistencies. We could overcome this issue by exploring only relevant values, using for example the TrieJoin algorithm from [Vel14] which allows to enumerate values present in the intersection of every relation in time $\mathcal{O}(\mathsf{log}|\mathsf{D}|)$ or hash indices as in [Ngo18]. While these techniques are interesting for practical implementations, our goal in this work is to achieve the same result by using as little technical machinery as possible. Therefore, we present here a new simple technique to remove this extra $|\mathsf{D}|$ factor.

The main idea is that instead of testing every value in the domain D for each variable, we fix its value bit by bit. This could be implemented directly by modifying Algorithm 3.2 or, as we choose to present it, by transforming any database $\mathbf{D}$ on domain D with n variables into a database $\widetilde{\mathbf{D}}^b$. We also update the query Q to a query $\widetilde{Q}^b$, where we replace all the atoms $\mathsf{R}(\mathbf{x})$ by $\widetilde{\mathsf{R}}^b(\widetilde{\mathbf{x}}^b)$. This database now has $n \cdot b$ variables where $b = \lceil \mathsf{log}|\mathsf{D}| \rceil$ variables on domain $\{0, 1\}$, and is built such that the answers of $\widetilde{Q}^b$ over $\widetilde{\mathbf{D}}^b$ are in one-to-one correspondence with the answers of Q over $\mathbf{D}$. We do this by re-encoding each element of the domain D in binary.

▶ **Example 3.13** (Re-encoding the database)
Consider the following database instance $\mathbf{D}$ for the query $Q_\Delta(x, y, z) :\text{-}\ \mathsf{R}(x, y),\ \mathsf{S}(x, z),\ \mathsf{T}(y, z)$:

R	x	y
	0	0
	1	0
	1	1
	2	1

S	x	z
	0	3
	1	0
	1	2
	2	3

T	y	z
	0	2
	0	3
	1	0
	1	2

The idea is then to encode each value of the domain $\{0, 1, 2, 3\}$ by a binary word of length 2 on domain $\{0, 1\}$. This will give us the following database instance $\widetilde{\mathbf{D}}^2$:

$\widetilde{\mathsf{R}}^2$	x^2	x^1	y^2	y^1
	0	0	0	0
	0	1	0	0
	0	1	0	1
	1	0	0	1

$\widetilde{\mathsf{S}}^2$	x^2	x^1	z^2	z^1
	0	0	1	1
	0	1	0	0
	0	1	1	0
	1	0	1	1

$\widetilde{\mathsf{T}}^2$	y^2	y^1	z^2	z^1
	0	0	1	0
	0	0	1	1
	0	1	0	0
	0	1	1	0

The reencoded query would then be:

$$\widetilde{Q_\Delta}^2(x^2, x^1, y^2, y^1, z^2, z^1) \text{ :- } \widetilde{\mathsf{R}}^2(x^2, x^1, y^2, y^1),\ \widetilde{\mathsf{S}}^2(x^2, x^1, z^2, z^1),\ \widetilde{\mathsf{T}}^2(y^2, y^1, z^2, z^1)\ .$$

The encoding can be defined formally as follows. Let $\mathbf{D}$ be a database on variables X and domain D. Without loss of generality, we assume that $\mathsf{D} = \{1, \ldots, d\}$ for some d and we let $b = \lceil \log|\mathsf{D}| \rceil$ to be the number of bits needed to encode in binary every element of D. We represent each element k in D by the binary number $\widetilde{k}^b$ representing k written with b bits. For $1 \leqslant i \leqslant b$, let $\widetilde{k}^b[i]$ be the i^{th} bit of the binary representation of $k \in \mathsf{D}$ over b bits. The function $\widetilde{\cdot}^b$ is a bijection between D and its image. Given a binary number k in the image of $\widetilde{\cdot}^b$, we write $\bar{k}^b$ for the corresponding element of D, that is, the function $\bar{\cdot}^b$ is the inverse function of $\widetilde{\cdot}^b$.

We now lift the functions $\widetilde{\cdot}^b$ to pairs of bijections over tuples, relations and then over databases. For a set of variables Y, we denote by $\widetilde{Y}^b$ the set $\{y^i \mid y \in Y, 1 \leqslant i \leqslant b\}$, that is, the set containing b distinct copies of each variable of Y. For $\tau \in \mathsf{D}^Y$, we define $\widetilde{\tau}^b$ as follows: for every $y \in Y$ and $i \in [b]$, $\widetilde{\tau}^b(y^i) = \widetilde{\tau(y)}^b[i]$. Similarly, given a relation $\mathsf{R} \subseteq \mathsf{D}^Y$ we let $\widetilde{\mathsf{R}}^b = \{\widetilde{\tau}^b \mid \tau \in \mathsf{R}\}$. And finally, given a query Q over variables X and domain D and a database instance $\mathbf{D}$ for Q, we let $\widetilde{\mathbf{D}}^b = \{\widetilde{\mathsf{R}}^b \mid \mathsf{R} \in \mathbf{D}\}$.

Obviously, the answers of $\widetilde{Q}^b$ over $\widetilde{\mathbf{D}}^b$ are in one-to-one correspondence with the answers of Q over $\mathbf{D}$. Moreover, we have that the cardinalities of the relations are invariant under this transformation, that is $|\widetilde{\mathsf{R}}^b| = |\mathsf{R}|$ for any relation R. An example of the domain size reduction we operate is shown in Example 3.17.

Applying Theorem 3.5 on $\widetilde{\mathbf{D}}^b$ directly yields the following:

▶ **Theorem 3.14**
Given a join query Q on domain $\mathsf{D} \subseteq [2^b]$ with m atoms, $(x_1, \ldots, x_n)$ an order on the variables of Q, and $\mathbf{D}$ a database instance for Q, $\mathsf{WCJ}(Q, \widetilde{\mathbf{D}}^b, \langle\rangle)$ with order $(x_1^1, \ldots, x_1^b, \ldots, x_n^1, \ldots, x_n^b)$ computes $\mathsf{ans}_{\widetilde{\mathbf{D}}^b}(Q)$ in time $\tilde{\mathcal{O}}(m \cdot \sum_{i \leqslant n} \sum_{j \leqslant b} |\mathsf{ans}_{\widetilde{\mathbf{D}}^b}(Q_{|X_i^j})|)$ where $X_i^j = \{x_1^1, \ldots, x_1^b, \ldots, x_i^1, \ldots, x_i^j\}$.

This is the first step in showing the worst-case optimality of Algorithm 3.2. However, we will still have to bound $\max_{i,j} |\mathsf{ans}_{\widetilde{\mathbf{D}}^b}^{X_i^j}(Q)|$ by $\mathsf{wc}(Q, \mathcal{C})$. We do this by showing that in the case of acyclic degree constraints, the binarised database $\widetilde{\mathbf{D}}^b$ also belongs to a class $\widetilde{\mathcal{C}}^b$ defined by acyclic degree constraints where $\mathsf{wc}(Q, \widetilde{\mathcal{C}}^b) \leqslant \mathsf{wc}(Q, \mathcal{C})$, that is, that reducing the domain size does not increase the worst-case value for the class. Moreover, we show that, if $\mathcal{C}$ is prefix-closed for the variable order $(x_1, \ldots, x_n)$, then $\widetilde{\mathcal{C}}^b$ is prefix-closed for the binarised variable order $(x_1^1, \ldots, x_1^b, \ldots, x_n^1, \ldots, x_n^b)$.

The idea is to binarise the degree constraints as follows: for $b \in \mathbb{N}$ and degree constraint $\delta = (A, B, N)$, we denote by $\widetilde{\delta}^b$ the degree constraint $(\widetilde{A}^b, \widetilde{B}^b, N)$ and for a set DC of degree constraints, let $\widetilde{\mathsf{DC}}^b :- \{\widetilde{\delta}^b \mid \delta \in \mathsf{DC}\}$. We can then show:

▶ **Lemma 3.15**
Let DC be a set of degree constraints and Q be a join query. For every database $\mathbf{D} \in \mathcal{C}(\mathsf{DC})$ for Q on domain $\mathsf{D} \subseteq [2^b]$ where $b \in \mathbb{N}$, we have that $\widetilde{\mathbf{D}}^b \in \mathcal{C}(\widetilde{\mathsf{DC}}^b)$. Moreover, $\mathsf{wc}(Q, \mathcal{C}(\widetilde{\mathsf{DC}}^b)) \leqslant \mathsf{wc}(Q, \mathcal{C}(\mathsf{DC}))$.

Finally, if the set of constraints DC is acyclic and $(x_1, \ldots, x_n)$ is an order compatible with DC, then $\widetilde{\mathsf{DC}}^b$ is acyclic and $(x_1^1, \ldots, x_1^b, \ldots, x_n^1, \ldots, x_n^b)$ is an order compatible with $\widetilde{\mathsf{DC}}^b$.

Proof. The first part of the statement follows from the following observation: let $\delta = (A, B, N)$ be a degree constraint from DC and R a relation on variables $\mathsf{var}(\mathsf{R}) \supseteq B$ which satisfies δ, then $\widetilde{\mathsf{R}}^b$ respects $\widetilde{\delta}^b$. Indeed, let τ be an assignment of $\widetilde{A}^b$ and let τ' be the corresponding assignment of A on domain 2^b defined as $\tau'(x) = \sum_{i=1}^{b} 2^{i-1} \tau(x^i)$, that is, τ encodes τ' via the binary operation we defined. Then it is easy to see that $\mathsf{R}[\tau']_{|Y}$ is in one-to-one correspondence with $\widetilde{\mathsf{R}}^b[\tau]_{|\widetilde{Y}^b}$ by using the same encoding. In particular, $|\widetilde{\mathsf{R}}^b[\tau]_{|\widetilde{Y}^b}| = |\mathsf{R}[\tau']_{|Y}| \leqslant N$, and therefore, $\widetilde{\mathbf{D}}^b \in \mathcal{C}(\widetilde{\mathsf{DC}}^b)$.

Now, let $\mathbf{D}' \in \mathcal{C}(\widetilde{\mathsf{DC}}^b)$ be a database on domain D. We build $\mathbf{D}$ as the database on domain D^b where each relation R' of $\mathbf{D}'$ over attributes $\widetilde{Y}^b$ is transformed into a relation R over attributes Y as follows: for a tuple $\tau' \in \mathsf{R}'$, we build the tuple $\tau \in \mathsf{R}$ by taking for each $y \in Y$, $\tau(y) = \bigtimes_{j \leqslant b} \tau'(y^j)$. It is easy to see that if R' satisfies the degree constraint $\widetilde{\delta}^b$, then R satisfies δ and that $\mathsf{ans}_{\mathbf{D}}(Q)$ and $\mathsf{ans}_{\mathbf{D}'}(Q)$ are in one-to-one correspondence. Hence $\mathbf{D} \in \mathcal{C}(\mathsf{DC})$ and then $\mathsf{wc}(Q, \mathcal{C}(\widetilde{\mathsf{DC}}^b)) \leqslant \mathsf{wc}(Q, \mathcal{C}(\mathsf{DC}))$.

Finally, by definition, it is clear that there is an edge in G_{DC} between x and y if, and only if, there is an edge between x^i and y^j for every $i, j \leqslant b$ in $G_{\widetilde{\mathsf{DC}}^b}$. Assume towards a contradiction that there is a path from x_i^j to x_k^ℓ for some $i \geqslant k$ in $G_{\widetilde{\mathsf{DC}}^b}$. Then there is necessarily a path from

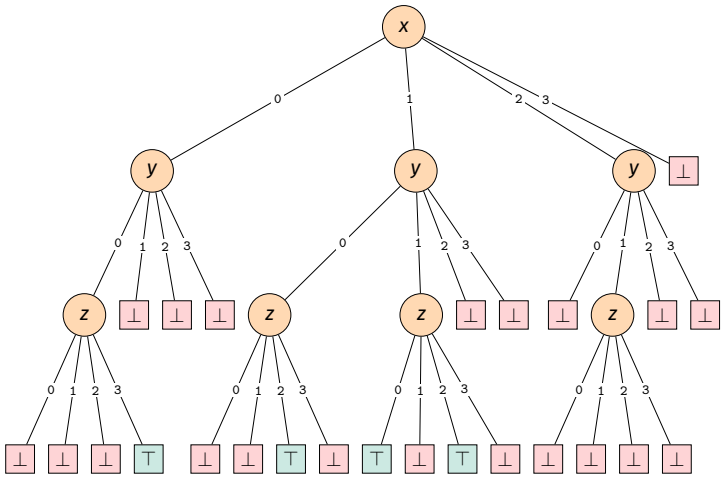
x_i to x_k in G_{DC} as argued above, which contradicts the fact that $x_1, \ldots, x_n$ is a topological sort of G_{DC}. □

A direct consequence of Lemma 3.15 is that $\mathsf{ans}_{\mathbf{D}}^{X_i^j}(Q) \leqslant \mathsf{wc}(Q, \mathcal{C}(\mathsf{DC}))$ for $X_i^j = \{x_1^1, \ldots, x_i^j\}$ whenever $j = b$. This also holds for all other values of j different from b. Indeed, fixing fewer bits corresponds to projecting the answer relation on a stricter subset of attributes, which can only merge tuples and therefore cannot increase its size. Lemma 3.15 therefore implies that $\mathcal{C}(\widetilde{\mathsf{DC}}^b)$ is prefix-closed when DC is acyclic and its worst-case is not greater than the worst-case of $\mathcal{C}(\mathsf{DC})$. Since for any $\mathbf{D} \in \mathcal{C}(\mathsf{DC})$, the domain of $\widetilde{\mathbf{D}}^b$ is two and has $b \cdot n = \tilde{\mathcal{O}}(n)$ variables, we therefore have a worst-case optimal join (up to polylogarithmic factors) algorithm for $\mathcal{C}(\mathsf{DC})$:

▶ **Corollary 3.16**
Let Q be a join query, DC be a set of acyclic degree constraints and $\mathcal{C}(\mathsf{DC})$ be a class of databases for Q satisfying DC. Let m be the number of relations in $\mathbf{D} \in \mathcal{C}(\mathsf{DC})$ and n be the number of variables. Assume $(x_1, \ldots, x_n)$ is an order compatible with DC. Then for every $\mathbf{D} \in \mathcal{C}(\mathsf{DC})$ on domain $\mathsf{D} \subseteq [2^b]$, $\mathsf{WCJ}(Q, \widetilde{\mathbf{D}}^b, \langle\rangle)$ on order $(x_1^1, \ldots, x_1^b, \ldots, x_n^1, \ldots, x_n^b)$ returns $\mathsf{ans}_{\mathbf{D}}(Q)$ in time $\tilde{\mathcal{O}}(mn \cdot \mathsf{wc}(Q, \mathcal{C}(\mathsf{DC})))$.

Notice that there is a slight abuse in the statement of Corollary 3.16 since the algorithm does not directly return $\mathsf{ans}_{\mathbf{D}}(Q)$ but a binary representation of each tuple in $\mathsf{ans}_{\mathbf{D}}(Q)$. However, it is straightforward to turn each answer of Q over $\widetilde{\mathbf{D}}^b$ back to the corresponding answer of Q in $\mathbf{D}$ in $\tilde{\mathcal{O}}(1)$.

▶ **Example 3.17** (Working with a reduced domain)
Consider the query and database instance introduced in Example 3.13. By keeping the trace of the execution of Algorithm 3.2 over this instance, we obtain the following tree:

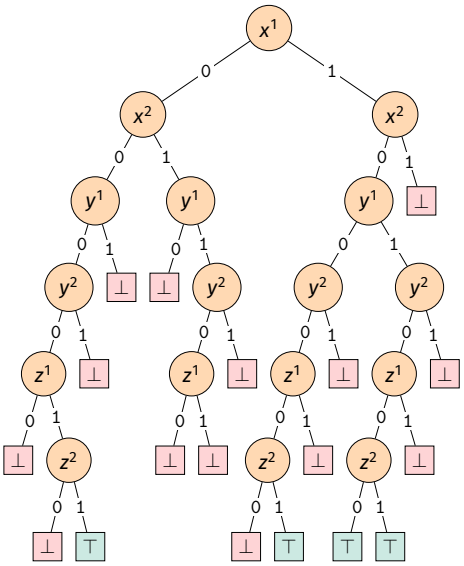


If we reencode the database instance and the query as in Example 3.13, we can then branch on the first bit x^1 of x, then the second bit x^2 and so on. A trace of a run of the same algorithm over this reencoded instance would produce the following tree:

Note that the bits are evaluated starting from the least significant bit.

In this example, we can directly see that when x is set to 0, that is, when $x^1 \leftarrow 0, x^2 \leftarrow 0$, then we do not explore the values 1 or 3 for y as we directly detect an inconsistency when setting the first bit of y to 1.

This simple bitwise branching suffices to guarantee worst-case optimality for an already simple algorithm such as Algorithm 3.2.



3.4 Comparison and Conclusion

Our algorithm is closely related to both *Generic Join* [Ngo+12] and *Leapfrog TrieJoin* [Vel14], the latter already being a particular case of Generic Join. Similarly, Algorithm 3.2 can be regarded as a simplified form of Generic Join. The main difference in the approach is that both Generic Join and TrieJoin use a specific algorithm (trie join and m-way sort merge respectively) to ensure that a variable x is branched only on values that would not introduce any inconsistency. We circumvent this need of a specific algorithm by using binarisation instead. This can be seen, from a higher perspective, as simply branching on the bits of each value. In turn, this approach could be emulated directly on the query (without binarising the query explicitly), yielding an algorithm very close to Generic Join, that is, a branching algorithm which efficiently branches only on values of x that do not contradict the query right away. However, we prefer the explicit binarisation since it keeps the analysis transparent: the complexity of Algorithm 3.2 depends only on the prefix-closed property of the class and the size of the domain, which can then be reduced via binarisation.

A key contribution of this work lies in its complexity analysis. The analysis of Generic Join from [Ngo+12; NRR14] relies on the knowledge of the worst-case bound, known as the AGM bound for cardinality constraints and a polymatroid bound for acyclic degree constraints. While proofs of these bounds are well established in the literature (see [Suc23] for a survey), they add a layer of complexity in the understanding of why such simple branch-and-bound strategies achieve worst-case optimality. Our analysis instead relies on a easily verifiable structural property of the considered classes, namely prefix-closedness: forgetting variables in intermediate relations cannot produce an instance whose number of answers exceeds the worst case. We have shown in this

chapter that classes defined by cardinality constraints or acyclic degree constraints satisfy this property, making the link both elementary and natural.

Our approach is also close in spirit to the one of [Vel14], where the notion of *renumbering* is introduced and the runtime is bounded by analysing the behaviour of the algorithm on a normalised instance in which values have been modified. While the underlying idea is similar, we argue that prefix-closedness provides a simpler and more direct notion to reason about worst-case optimality.

To summarise, this chapter presents a simple branch-and-bound algorithm for evaluating join queries, and studies the conditions under which it achieves worst-case optimality. Despite its conceptual simplicity, the algorithm matches the guarantees of more elaborate strategies found in the literature, such as Generic Join or Leapfrog TrieJoin, provided the class of databases considered for the query satisfies the natural prefix-closedness condition.

To eliminate the overhead incurred by naive value enumeration, we introduced a binarisation technique reducing arbitrary domains to the Boolean case. This allows the algorithm to branch on bits rather than values, thereby pruning the search space efficiently while preserving optimality. While the search on bits facilitates the understanding of the algorithm and its underlying complexity, a practical implementation may nevertheless find it more convenient to use larger domains such as bytes to take advantage of bit-vector operations.

Beyond its theoretical significance, the resulting framework is easy to describe, straightforward to implement, and flexible enough to support more advanced query tasks. In Chapter 4, we build on this structure to show how it can be extended to support uniform sampling of query answers.

Current chapter references

[AGM13] Albert **Atserias**, Martin **Grohe**, and Dániel **Marx**. *Size bounds and query plans for relational joins*. In *SIAM Journal on Computing* 42.4 (Jan. 2013), pp. 1737–1767. doi 10.1137/110859440.

[ANS16] Mahmoud **Abo Khamis**, Hung Q. **Ngo**, and Dan **Suciu**. *Computing Join Queries with Functional Dependencies*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS'16: International Conference on Management of Data. San Francisco California USA: ACM, June 2016, pp. 327–342. doi 10.1145/2902251.2902289.

[ANS17b] Mahmoud **Abo Khamis**, Hung Q. **Ngo**, and Dan **Suciu**. *What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?* In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS'17: International Conference on Management of Data. Chicago Illinois USA: ACM, May 2017, pp. 429–444. doi 10.1145/3034786.3056105.

[CIS25] Florent **Capelli**, Oliver **Irwin**, and Sylvain **Salvati**. *A Simple Algorithm for Worst Case Optimal Join and Sampling*. In *28th International Conference on Database Theory (ICDT 2025), March 25 to March 28, 2025, Barcelona, Spain* (ICDT '25). Ed. by Sudeepa **Roy** and Ahmet **Kara**. Vol. 328. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2025, 23:1–23:19. doi 10.4230/LIPIcs.ICDT.2025.23.

[GM14] Martin **Grohe** and Dániel **Marx**. *Constraint solving via fractional edge covers*. In *ACM Transactions on Algorithms (TALG)* 11.1 (Aug. 2014), p. 4. doi 10.1145/2636918.

[Ngo+12] Hung Q. **Ngo**, Ely **Porat**, Christopher **Ré**, and Atri **Rudra**. *Worst-Case Optimal Join Algorithms: [Extended Abstract]*. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '12: International Conference on Management of Data. Scottsdale Arizona USA: ACM, May 2012, pp. 37–48. doi 10.1145/2213556.2213565.

[Ngo18] Hung Q. **Ngo**. *Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems*. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '18: International Conference on Management of Data. Houston TX USA: ACM, May 2018, pp. 111–124. doi 10.1145/3196959.3196990.

[NRR14] Hung Q. **Ngo**, Christopher **Ré**, and Atri **Rudra**. *Skew strikes back: new developments in the theory of join algorithms*. In *SIGMOD Rec.* 42.4 (Feb. 2014), pp. 5–16. doi 10.1145/2590989.2590991.

[Suc23] Dan **Suciu**. *Applications of Information Inequalities to Database Theory Problems*. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023, Boston, MA, USA, June 26-29, 2023*. IEEE, June 2023, pp. 1–30. doi 10.1109/LICS56636.2023.10175769.

[Vel14] Todd L. **Veldhuizen**. *Triejoin: A Simple, Worst-Case Optimal Join Algorithm*. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. Ed. by Nicole **Schweikardt**, Vassilis **Christophides**, and Vincent **Leroy**. OpenProceedings.org, Mar. 2014, pp. 96–106. doi 10.5441/002/icdt.2014.13.

Chapter 4

Uniformly Sampling Query Answers

Probability is the very guide of life.

Cicero

In this chapter, we extend the ideas developed for WCOJ evaluation to the task of uniform sampling from the answers of a join query. Uniform sampling has received growing attention in recent years as an alternative to full materialisation of the answer set, especially when representative answers are sufficient for downstream applications. It is known that, for queries Q over databases belonging to classes $\mathcal{C}$ defined by cardinality constraints [DLT23; Kim+23] or by acyclic degree constraints [WT24], one can sample answers uniformly in expected time $\tilde{\mathcal{O}}(\frac{\mathsf{wc}(Q,\mathcal{C})}{\max(1,|\mathsf{ans}_{\mathbf{D}}(Q)|)})$.

Our goal here is to recover these guarantees through an elementary algorithmic approach. The central idea is to reuse the structure of the trace tree generated by the branching procedure of Chapter 3 (Algorithm 3.2), and to adapt a classical method of Rosenbaum [Ros93] to sample leaves from a tree without exploring it exhaustively. This perspective provides a clean and modular bridge from worst-case optimal join evaluation to uniform sampling. Our approach is similar to the one used by Kim, Ha, Fletcher, and Han [Kim+23] and by Chen and Yi [CY20b], but is more modular, making it easier to adapt to the more general case of acyclic degree constraints. Our analysis relies solely on structural properties of the database class, and the probability calculations are considerably simpler than those in existing analyses such as [DLT23]. The work presented in this chapter is an extension of the work presented in Chapter 3, first described in [CIS25].

This chapter starts by presenting a simple algorithm to sample from the leaves of a tree, that we adapt to use over the trace tree generated by Algorithm 3.2. We then show an example on how we can use the worst-case as an estimator to guide the sampling. Finally, we formally prove that the worst-case bounds from Chapter 3 have the required properties to allow for an efficient uniform sampling of the query answers.

Outline of the current chapter

4.1 Efficiently Sampling the Leaves of a Tree

Our core technique relies on the problem of sampling uniformly a leaf from a rooted tree. We aim at designing an algorithm solving this problem while avoiding an exhaustive tree exploration. This question has already been addressed by Rosenbaum in [Ros93] where he proposes an algorithm that explores the tree in a top-down manner, and whenever it encounters a leaf, either returns it or fails. To remove, or at least reduce, bias towards subtrees having more leaves than others in this algorithm, he guides the search with upper bounds on the number of leaves of each subtree, obtained from the depth and the branching size of the tree.

4.1.1 Rosenbaum's Algorithm

Rosenbaum's algorithm samples the leaves of tree by using a random movement procedure and repeating the movement until a leaf is reached. We choose to present this algorithm in a slightly different way to that used in the original paper, by presenting it as a Las Vegas algorithm. Consider a rooted tree T with root r. The movement choice is based on an upper bound of the number of leaves in T. This upper bound is computed with the help of a few measures: (1) the number of children of the root r, denoted m, (2) an upper bound on the maximum degree of any node that is not r, denoted w, and (3) the maximum depth of the tree, denoted d. We then have an upper bound defined as $\mathsf{r\text{-}upb} = m \cdot w^{d-1}$. Note that this measure is truly an upper bound since if we consider a tree of maximal degree w and of depth d, there the most leaves the tree can have is w^d.

These measures allow Rosenbaum to define a *random movement* procedure as follows:

- from the root r: if a child t of r is a leaf, move to t with probability $\frac{1}{\mathsf{r\text{-}upb}}$, if it is not a leaf, move to t with probability $\frac{1}{m}$.
- from any descendant t' of r at depth a: if a child t of t' is not a leaf, move to t with probability $\frac{1}{w}$, if it is a leaf, move to t with probability $\frac{1}{w^{d-a}}$.

In both these cases, there is a non-zero probability that no child or leaf t will be selected since the sum of the probabilities will always be less than 1. In these cases, the random movement procedure is defined to fail[1].

Example 4.3 showcases an example of this random movement procedure over a simple tree.

The sampling algorithm is then as simple as the following:

Algorithm 4.1 Rosenbaum's algorithm

```
1: procedure LEAF(t)
2:     input: · t a tree
3:     if t is a leaf then return t
4:     else, return random_movement(t)
```

Rosenbaum shows that Algorithm 4.1 has the following guarantees:

[1] in the original paper, the random movement simply returns to the root and continues.

▶ **Theorem 4.2** ([Ros93, Proposition 1])
Given a rooted tree T, and out the output of a run of the LEAF procedure, we have:

$$\Pr(\text{out} = \ell) = \frac{1}{\text{r-upb}(T)} \qquad \text{and} \qquad \Pr(\text{out} = \text{fail}) = 1 - \frac{N}{\text{r-upb}(T)}$$

for any given leaf ℓ, where N is the total number of leaves in T.

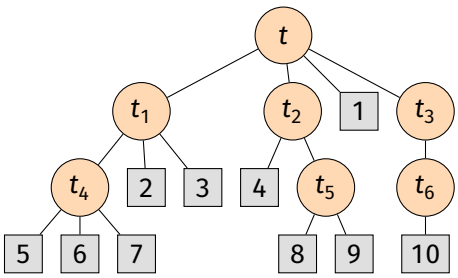
Repeating this algorithm until it succeeds will produce a uniformly sampled leaf from the tree, but its main caveat is that the number of returns to the root is high. The probability that the procedure will fail at some point before reaching a leaf is equal to $1 - \frac{N}{\text{r-upb}}$. The total number of fails before outputting a leaf follows a geometric distribution and its expected value is $\frac{\text{r-upb}}{N}$. This implies that if the tree is *sparse*, that is, if the number of leaves is small compared to the number of nodes in the tree, then we will face a high volume of fails, and thus will have to repeat the algorithm multiple times.

To compensate for this fact and therefore reduce the volume of fails, Rosenbaum proposes to reorganise the tree to reduce its depth. This is done by searching for leaves and subtrees up to a certain depth and adding them as direct children of the root. Note that, in this setting, if this does reduce the depth of the tree, it does so without changing its width (since we do not consider the degree of the root). Therefore, this manipulation simply updates the exponent in the upper bound computation.

This reorganisation of the tree allows for easier sampling. Indeed, since the depth has changed, then the expected number of fails is often less (more importantly, it is never more) than for the original tree.

Example 4.3 also shows an example of a reorganisation of the tree.

▶ **Example 4.3** (Rosenbaum tree sampling [Ros93])
Consider the simple following tree:



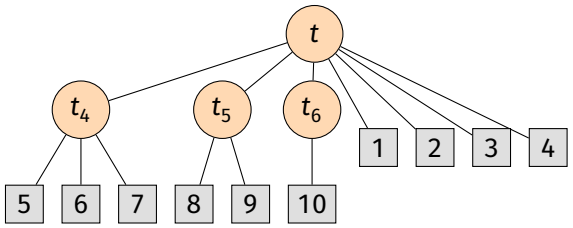
In this tree, the root has $m = 4$ children, a child width of $w = 3$ and a maximum depth $d = 3$. Rosenbaum's bound is thus $\text{r-upb} = m \cdot w^{d-1} = 4 \cdot 3^{3-1} = 36$, even though there are only 10 leaves.

From the root t, the random movement will move to t_1, t_2 or t_3 with probability $\frac{1}{4}$, to the leaf 1 with probability $\frac{1}{36}$ or fail with probability $\frac{2}{9}$.

From t_1, we would move to t_4 with probability $\frac{1}{3}$, to leaves 2 or 3 with probability $\frac{1}{9}$. More importantly, at this step, we would fail with probability $\frac{4}{9}$.

Note that the high probability of failing in this case is because of the high difference between the actual number of leaves of the tree and the upper bound we have computed.

Reorganising the tree up to depth 2 would yield the following tree:



which indeed results in a tree that, if it has more children directly from the root, has a depth of 2 and no longer 3. This in turn implies that the upper bound on the number of leaves is now $\mathsf{r\text{-}upb} = 7 \cdot 3^{2-1} = 21$. This reduces the expected value of the number of fails while sampling from the tree.

In summary, we could use Rosenbaum's algorithm as grounds for our query answer sampling problem. However, as we will see in our case, we are not interested in sampling from all the leaves, but simply a subset of them.

4.1.2 Adapting the algorithm

In this section, we adapt Rosenbaum's algorithm in a slightly more general setting. Indeed, in our approach, the leaves of the tree will correspond to cases where the recursion of Algorithm 3.2 stops. In this case, either a solution is found and we are interested in the leaf, or an inconsistency is found and we want to reject the leaf.

The main difference with Algorithm 4.1 is therefore that we now want this algorithm to work on a tree where we wish to sample from a subset of the leaves.

Moreover, the upper bounds on the leaves used in Section 4.1.1 [Ros93] are too coarse for our purposes. Therefore, we describe our version of the algorithm by using oracle calls to a function (over-)estimating this number of leaves. We delay the detailed presentation of this function to Section 4.2 in order to keep the presentation of this algorithm modular.

These conditions motivate the following definition:

▶ **Definition 4.4** (Leaf estimators)
Let T be a rooted tree. A *leaf estimator* upb for T is a function mapping nodes of T to positive values such that:

(i) for every node t with children $(t_1, \ldots, t_n)$, $\mathsf{upb}(t) \geqslant \sum_{i=1}^{n} \mathsf{upb}(t_i)$; we call functions with this property *tree-superadditive*; and

(ii) if t is a leaf, then $\mathsf{upb}(t) \in \{0, 1\}$, indicating whether we are interested in sampling the

leaf.

For a tree T rooted in r, we denote by $\mathsf{upb}(T)$ the value of $\mathsf{upb}(r)$. Now, given a tree T and a leaf estimator upb for T, we say that a leaf ℓ of T is a *1-leaf of T* if, and only if, $\mathsf{upb}(\ell) = 1$ and denote by $\mathsf{leaves}_1(T)$ the set of all 1-leaves of T. Our goal is now to uniformly sample a leaf $\ell \in \mathsf{leaves}_1(T)$.

Observe that since upb is a tree-superadditive function, for any node t, we have that $\mathsf{upb}(t)$ is an upper bound on the number of 1-leaves below t.

The upper bound used in Rosenbaum's algorithm is tree-superadditive, since, given a node at depth a, we have at most w children at depth $(a+1)$ with an upper bound of $w^{d-(a+1)}$ each, thus $w \cdot w^{d-(a+1)} \leqslant w^{d-a}$, which is the upper bound for the node at depth a.

We define $\mathsf{children}(t)$ as the function that, given a node t, returns the list of its direct children.

This allows us to define our algorithm recursively as follows:

Algorithm 4.5 A variation of the Rosenbaum algorithm [Ros93]

Sample a leaf in the subtree rooted in t as follows:

- if t is a leaf belonging to $\mathsf{leaves}_1(T)$, output the leaf with probability 1;
- if t is any other leaf, fail with probability 1; and
- if t has children $t_1, \ldots, t_n$, recursively sample a 1-leaf in t_i with probability $\frac{\mathsf{upb}(t_i)}{\mathsf{upb}(t)}$ and return it if the recursive call in t_i succeeds and fail otherwise. Note that we may directly fail without recursively sampling with probability $1 - \sum_i \frac{\mathsf{upb}(t_i)}{\mathsf{upb}(t)}$. This probability is positive, since $\sum_i \mathsf{upb}(t_i) \leqslant \mathsf{upb}(t)$ by the tree-superadditivity of upb.

In the rest of this section, we prove that Algorithm 4.5 is a Las Vegas uniform sampling algorithm for the set of $\mathsf{leaves}_1(T)$ of T, with the following guarantees:

▶ **Theorem 4.6**
Let T be a tree rooted in r and upb a leaf estimator for T. Let out be the output of Algorithm 4.5 on input r. Then, for any leaf $\ell \in \mathsf{leaves}_1(T)$, we have that Algorithm 4.5 is a uniform Las Vegas sampler with guarantees:

$$\Pr(\mathsf{out} = \ell) = \frac{1}{\mathsf{upb}(T)} \qquad \text{and} \qquad \Pr(\mathsf{out} = \mathsf{fail}) = 1 - \frac{|\mathsf{leaves}_1(T)|}{\mathsf{upb}(T)}$$

Algorithm 4.5 consists of $\mathcal{O}(B \cdot \mathsf{depth}(T))$ calls to upb, where B is the branching size of the tree, in $\mathcal{O}(\mathsf{depth}(T))$ calls to the $\mathsf{children}$ function.

Proof. We proceed by induction on the depth of the tree T. If the tree T is of depth 1, then it can be one of two cases: (i) either it belongs to $\mathsf{leaves}_1(T)$ and then $|\mathsf{leaves}_1(T)| = 1$ and therefore it is trivial to see that the algorithm samples this leaf with probability $\frac{1}{\mathsf{upb}(T)} = 1$, or (ii) it does not belong to $\mathsf{leaves}_1(T)$ and then there is nothing to sample, so the algorithm fails inevitably.

Supposing that the property holds for a tree T' of depth at most k, if we now have a tree T

of depth $k+1$, then it has children $(t_1, \ldots, t_n)$ each of depth at most k. Then, by induction, Algorithm 4.5 samples from a given t_i with probability $\frac{\mathsf{upb}(t_i)}{\mathsf{upb}(t)}$. If the recursive call has succeeded, then the algorithm has sampled a leaf from t_i with probability $\frac{1}{\mathsf{upb}(t_i)}$. Since the random choices are independent, the probability of outputting this leaf from t is $\frac{\mathsf{upb}(t_i)}{\mathsf{upb}(t)} \times \frac{1}{\mathsf{upb}(t_i)} = \frac{1}{\mathsf{upb}(t)}$.

The complexity statement is straightforward. We need to evaluate the number of selected leaves in each subtree along a path from the root to a leaf, leading to $\mathcal{O}(B \cdot \mathsf{depth}(T))$ calls to upb and for each node we visit, we need to find the list of children, leading to $\mathcal{O}(\mathsf{depth}(T))$ calls to $\mathsf{children}$. □

Since Algorithm 4.5 is a Las Vegas algorithm, we can also extend Theorem 4.6 with the following corollary:

▶ **Corollary 4.7**
Given a tree T with branching size B and oracle access to a leaf estimator function $\mathsf{upb}(\cdot)$, we can sample in $\mathsf{leaves}_1(T)$ with uniform probability $\frac{1}{|\mathsf{leaves}_1(T)|}$, when $|\mathsf{leaves}_1(T)| > 0$ or answer that $|\mathsf{leaves}_1(T)| = 0$.

This is done by repeating Algorithm 4.5 an expected $\mathcal{O}(\frac{\mathsf{upb}(T)}{\max(1, |\mathsf{leaves}_1(T)|)})$ number of times and therefore, we have:

$$\begin{cases} \mathcal{O}(\dfrac{\mathsf{upb}(T)}{\max(1, |\mathsf{leaves}_1(T)|)} \cdot B \cdot \mathsf{depth}(T)) & \text{expected calls to } \mathsf{upb}(\cdot), \text{ and} \\ \mathcal{O}(\dfrac{\mathsf{upb}(T)}{\max(1, |\mathsf{leaves}_1(T)|)} \cdot \mathsf{depth}(T)) & \text{expected calls to } \mathsf{children}(\cdot) \end{cases}$$

The proof of Corollary 4.7 is done by splitting the analysis into two cases: one where we consider the size of $\mathsf{leaves}_1(T)$ to be strictly positive and one where $|\mathsf{leaves}_1(T)| = 0$. We show that the expected number of repetitions of Algorithm 4.5 is $\frac{\mathsf{upb}(T)}{\max(1, |\mathsf{leaves}_1(T)|)}$. In parallel to the sampling, we run an exploration of T in order to cover the eventuality that $|\mathsf{leaves}_1(T)| = 0$.

Proof. We first treat the case where $|\mathsf{leaves}_1(T)| = n > 0$. Algorithm 4.5 can either fail or produce a leaf that has been sampled with uniform probability. It is thus a *Las Vegas* algorithm. It samples a leaf in $\mathsf{leaves}_1(T)$ with uniform probability $\frac{1}{\mathsf{upb}(T)}$. When $|\mathsf{leaves}_1(T)| = n > 0$, it thus outputs some data with probability $\frac{n}{\mathsf{upb}(T)}$ or fails with probability $1 - \frac{n}{\mathsf{upb}(T)}$. Now if we repeat the algorithm until it succeeds, as each 1-leaf has the same probability to be outputted in one repetition, they all have the same probability to be outputted at the end of this process. In a nutshell, this procedure uniformly chooses amongst the 1-leaves of T which all have probability $\frac{1}{n}$ to be outputted. Moreover, since the repetitions of Algorithm 4.5 are independent, the number of such repetitions until it succeeds follows a geometric distribution and its expected value is thus $\frac{\mathsf{upb}(T)}{n}$.

The case where $|\mathsf{leaves}_1(T)| = 0$ is a little trickier. Since Algorithm 4.5 can only fail, repeating it would result in an infinite loop. We can circumvent this in one of two ways.

Either we start a full exploration of T in parallel and if it does not find any 1-leaf in T, we stop running Algorithm 4.5 (if it returns a 1-leaf, we stop the exploration). A second method would be to improve Algorithm 4.5 by maintaining the parts of T that have already been explored and by updating the values of $\mathsf{upb}(t)$ for each subtree that is explored by using the

information from its children. Eventually, the algorithm explores T entirely. Indeed parts of T that have been explored are known not to contain data and have thus probability 0 to be explored again. In the end, updating the $\mathsf{upb}(t)$ value of each node will result in having $\mathsf{upb}(T) = 0$, meaning that $\mathsf{leaves}_1(T)$ is empty. We then stop the search. Both methods would cost a time of $\mathcal{O}(\mathsf{upb}(T))$. □

▶ **Remark 4.8**
Notice that the second method presented in the proof may also be useful when $|\mathsf{leaves}_1(T)|$ is small compared to $\mathsf{upb}(T)$. Indeed, updating $\mathsf{upb}(\cdot)$ at each failure of the algorithm increases the probability of success of the next iteration of the algorithm resulting in an overall improvement in the speed of convergence.

4.2 Applying Algorithm 4.5 to Join Queries

There is a strong link between the tree structure used in the sampling method from Section 4.1 and the trace of a run of the worst-case optimal join algorithm presented in Algorithm 3.2. Assume that we want to sample uniformly the results of a join query Q over a database $\mathbf{D}$ with relations over variables $X = \{x_1, \ldots, x_n\}$ and domain D. For this, we can follow the structure of the execution of Algorithm 3.2 when it uses the order $(x_1, \ldots, x_n)$ on variables. As we have seen, the trace of the execution of Algorithm 3.2 naturally constructs a tree structure whose nodes are some assignments of D^{X_i} for some $i \in [0, n]$, corresponding to the input of the recursive calls.

We call this tree the *trace tree of* $(Q, \mathbf{D})$ and denote it by $T_Q^{\mathbf{D}}$. In $T_Q^{\mathbf{D}}$, an assignment $\tau \in \mathsf{D}^{X_i}$ is the *parent* of another assignment τ' when $\tau' = \tau \times \langle x_{i+1} \leftarrow d \rangle$. Amongst all the possible assignments, the ones that are nodes in $T_Q^{\mathbf{D}}$ are those that are consistent with Q or those that are inconsistent with Q but have a parent that is consistent with Q. In Example 3.17, we depict such a tree: assignments that are inconsistent with Q are labelled $\bot$, those that are elements of the answer set are labelled $\top$. Moreover, the assignment corresponding to a node in the tree from Example 3.17 can simply be read off the path from the root to that particular node.

The leaves of $T_Q^{\mathbf{D}}$ that we want to sample in this tree are simply the elements of D^{X_n} that are consistent with Q, namely the solutions of Q, or more visually, the leaves labelled with $\top$. Algorithm 4.5 can then be applied to $T_Q^{\mathbf{D}}$.

This motivates the definition of a Q-estimator:

▶ **Definition 4.9** (Q-estimator)
Given a join query Q over a database $\mathbf{D}$ and an order $(x_1, \ldots, x_n)$ on the variables, a *Q-estimator* is a function $\mathsf{q\text{-}upb}$ on the nodes of $T_Q^{\mathbf{D}}$ that satisfies the following properties:

- when τ is a node of $T_Q^{\mathbf{D}}$ in D^{X_i} that is consistent with Q, $\mathsf{q\text{-}upb}(\tau) \geqslant \sum_{d \in \mathsf{D}} \mathsf{q\text{-}upb}(\tau \times \langle x_{i+1} \leftarrow d \rangle)$ (that is, $\mathsf{q\text{-}upb}$ is tree-superadditive);
- $\mathsf{q\text{-}upb}(\tau) = 1$ when $\tau \in \mathsf{ans}_{\mathbf{D}}(Q)$; and
- $\mathsf{q\text{-}upb}(\tau) = 0$ when τ is inconsistent with Q.

Defining Q-estimators then allows us to extend Corollary 4.7 with the following result:

▶ **Theorem 4.10**
Let Q be a join query over a database $\mathbf{D}$ with variable set X. Given a Q-estimator $\mathsf{q\text{-}upb}(\tau)$ that can be evaluated in time t for every τ, it is possible to uniformly sample $\mathsf{ans}_{\mathbf{D}}(Q)$ in expected time:

$$\mathcal{O}\left(\frac{\mathsf{q\text{-}upb}(\langle\rangle)}{\max(1, |\mathsf{ans}_{\mathbf{D}}(Q)|)} \cdot |X| \cdot |\mathsf{D}| \cdot t\right)$$

Proof. This theorem is a direct consequence of Corollary 4.7. The depth of $T_Q^{\mathbf{D}}$ is $|X|$ and its branching size is $|\mathsf{D}|$.

Each call to $\mathsf{q\text{-}upb}(\cdot)$ takes time t, and we have $\mathcal{O}\left(\frac{\mathsf{upb}(T)}{\max(1,|\mathsf{leaves}_1(T)|)} \cdot |\mathsf{D}| \cdot |X|\right)$ such calls.

Each call to $\mathsf{children}(\cdot)$ can be done in constant time with the appropriate data structure, and we have $\mathcal{O}\left(\frac{\mathsf{upb}(T)}{\max(1,|\mathsf{leaves}_1(T)|)} \cdot |X|\right)$ such calls. □

In the next section, we focus on giving a simple example of how to use a well-known bound for databases in our sampling algorithm. This will allow us to later show that the classes of databases we have been considering at this point, those defined by cardinality constraints and acyclic degree constraints allow for efficient sampling since the worst-case can be used as a Q-estimator.

4.3 Using the AGM Bound to Sample Query Answers

This section will serve as an example of an application of Algorithm 4.5 to sample uniformly from the answers of a join query.

Throughout this example, we will consider the triangle query Q_Δ :- $R(x, y)$, $S(x, z)$, $T(y, z)$ over a given database instance $\mathbf{D}$ and domain D.

Setting the context. In Section 4.2, we defined a few concepts to help link our sampling algorithm to join queries.

As such, it could be interesting to use query upper bounds as Q-estimators for Algorithm 4.5. Suppose that we have a partial assignment τ such that the next variable we are assigning is x. We consider the query estimator $\mathsf{q\text{-}upb}(\tau) = (|R[\tau]| \cdot |S[\tau]| \cdot |T[\tau]|)^{0.5}$. This is an upper bound on the number of answers of Q over $\mathbf{D}$, by AGM. In order to use this as a Q-estimator, we have to show that this bound is tree-superadditive here, that is, that:

$$\mathsf{q\text{-}upb}(\tau) = \sqrt{|R[\tau]| \cdot |S[\tau]| \cdot |T[\tau]|} \geqslant \sum_{d \in \mathsf{D}} \mathsf{q\text{-}upb}(\tau \times \langle x \leftarrow d \rangle)$$

To do this, we will use the Cauchy-Schwarz Inequality that states:

▶ **Proposition 4.11** (Cauchy-Schwarz Inequality)
For all vectors $\mathbf{u}$ and $\mathbf{v}$ of an inner product space, we have:

$$|\langle \mathbf{u}, \mathbf{v} \rangle| \leqslant \sqrt{\langle \mathbf{u}, \mathbf{u} \rangle} \cdot \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle}$$

We denote by R_d the slice $R[\tau \times \langle x \leftarrow d \rangle]$ to ease notations. We have $|R[\tau]| = \sum_{d \in \mathsf{D}} |R_d|$. This is also completely symmetrical for S. In this context, $T[\tau]$ stays constant, since x does not intervene in T.

We want to show:

$$\sqrt{|R[\tau]| \cdot |S[\tau]| \cdot |T[\tau]|} \geqslant \sum_{d \in \mathsf{D}} \sqrt{|R_d| \cdot |S_d| \cdot |T[\tau]|} = \sqrt{|T[\tau]|} \sum_{d \in \mathsf{D}} \sqrt{|R_d| \cdot |S_d|}$$

Which in this case is equivalent to showing:

$$\sqrt{|R[\tau]| \cdot |S[\tau]|} \geqslant \sum_{d \in \mathsf{D}} \sqrt{|R_d| \cdot |S_d|}$$

We define two vectors $\mathbf{R} = \left(\sqrt{|R_{d_1}|}, \ldots, \sqrt{|R_{d_n}|}\right)$ and $\mathbf{S} = \left(\sqrt{|S_{d_1}|}, \ldots, \sqrt{|S_{d_n}|}\right)$.

By Cauchy-Schwarz, we have:

$$\begin{aligned}
|\langle \mathbf{R}, \mathbf{S} \rangle| &\leqslant \sqrt{\langle \mathbf{R}, \mathbf{R} \rangle} \cdot \sqrt{\langle \mathbf{S}, \mathbf{S} \rangle} \\
\sum_{d \in \mathsf{D}} \sqrt{|R_d|} \sqrt{|S_d|} &\leqslant \sqrt{\sum_{d \in \mathsf{D}} \sqrt{|R_d|}^2} \cdot \sqrt{\sum_{d \in \mathsf{D}} \sqrt{|S_d|}^2} \\
\sum_{d \in \mathsf{D}} \sqrt{|R_d| \cdot |S_d|} &\leqslant \sqrt{\sum_{d \in \mathsf{D}} |R_d|} \cdot \sqrt{\sum_{d \in \mathsf{D}} |S_d|} \\
\sum_{d \in \mathsf{D}} \sqrt{|R_d| \cdot |S_d|} &\leqslant \sqrt{|R[\tau]|} \cdot \sqrt{|S[\tau]|} = \sqrt{|R[\tau]| \cdot |S[\tau]|}
\end{aligned}$$

Which is what we wanted to prove. Thus, the function q-upb defined via the AGM bound is tree-superadditive.

Moreover, since the restriction of the database to a tuple that is an answer to the query will only yield one tuple, our bound will be 1 when dealing with $\top$-leaves. When dealing with $\bot$-leaves, the restriction will leave at least one empty relation, leading to a bound of 0. This means that the function q-upb is a Q-estimator as it follows the properties from Definition 4.9.

Running Algorithm 4.5. We now describe how a run of Algorithm 4.5 would sample one of the answers (or fail to do so) using the AGM bound as an estimator. A visual representation of three different runs of the algorithm can be found in Figure 4.13.

In this example, the database instance $\mathbf{D}$ that we will consider is the following (Table 4.12):

R	x	y
	0	0
	1	0
	1	1
	2	1

S	x	z
	0	3
	1	0
	1	2
	2	3

T	y	z
	0	2
	0	3
	1	0
	1	2

Table 4.12: A database instance $\mathbf{D}$ for the triangle query Q_Δ

We will assume that $\mathbf{D}$ follows a cardinality constraint that bounds the number of tuples in each relation by 4. As we have seen in Example 3.17 we can apply our worst-case optimal join algorithm Algorithm 3.2 over Q and $\mathbf{D}$, and by keeping the trace of the execution, we obtain the tree T_{Q_Δ} represented in Example 3.17.

The run that we describe in detail next is a successful run of the algorithm, that does not fail.

First, suppose that we are at the root. Since the root here is not a leaf, we recursively sample from the children. To do so, we have to compute the upper bounds. The upper bound for the number of answers to the query over this database is $(4 \cdot 4 \cdot 4)^{0.5} = 8$. The upper bounds for the children of the root can be obtained by looking at how the database constraints change when setting the value of x. If we set $x \leftarrow 0$ or $x \leftarrow 2$, then both R and S now only contain 1 tuple. This implies that the upper bound for these subtrees will be $(1 \cdot 1 \cdot 4)^{0.5} = 2$. If we set $x \leftarrow 1$, then R and S contain 2 tuples each, hence the upper bound becomes $(2 \cdot 2 \cdot 4)^{0.5} = 4$. We therefore sample from the left and right subtrees with probability $\frac{2}{8} = 0.25$ and from the central subtree with probability $\frac{4}{8} = 0.5$. Note that setting $x \leftarrow 3$ leads to an inconsistency, thus the bound becomes 0 and implies that it cannot be sampled.

Suppose we sample from the central subtree (that is, we have set $x \leftarrow 1$). Since this is still not a leaf, we once again look at the subtrees. Here, if we set $y \leftarrow 0$ or $y \leftarrow 1$, we get an upper bound of $(1 \cdot 2 \cdot 2)^{0.5} = 2$ in both cases. Therefore, we can sample from each of the subtrees with probability $\frac{2}{4} = 0.5$.

Suppose we sample from the right subtree, that is, we set $y \leftarrow 1$. This is not a leaf, so we look at the subtrees. Here, we have 2 leaves that are in $\mathsf{leaves}_1(T_{Q_\Delta})$ and 2 $\bot$-leaves. Setting $z \leftarrow 0$ or $z \leftarrow 2$ will lead to an upper bound of $(1 \cdot 1 \cdot 1)^{0.5} = 1$, so we sample from one of the two $\top$-leaves with probability $\frac{1}{2} = 0.5$ each.

Finally, suppose that we set $z \leftarrow 0$. We arrive at a leaf $\ell \in \mathsf{leaves}_1(T_{Q_\Delta})$, so we output this leaf with probability 1.

At the end of the run, we have sampled the tuple $(1, 1, 0) \in \mathsf{ans}_{\mathbf{D}}(Q_\Delta)$ with probability:

$$\frac{4}{8} \cdot \frac{2}{4} \cdot \frac{1}{2} \cdot 1 = \frac{1}{8} = \frac{1}{\mathsf{upb}(T_{Q_\Delta})}$$

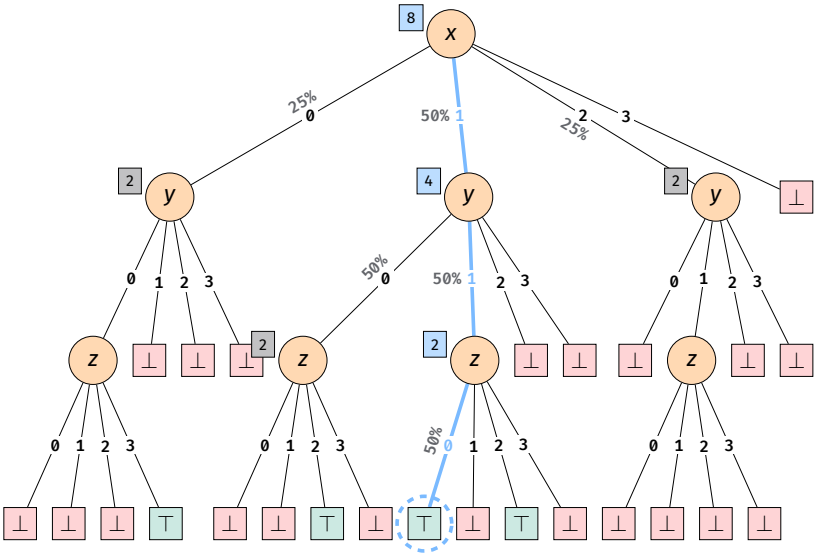
which is consistent with the guarantees from Theorem 4.6.

Figure 4.13 gives a visual representation of this run in Subfigure (a), as well as a representation of a failed run (Subfigure (b)) and of another successful run (Subfigure (c)).

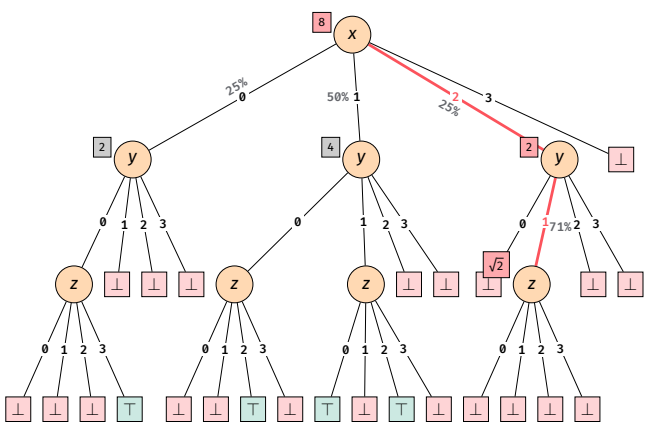
This leads us to believe that we can use the AGM bound to sample effectively from queries over databases defined with cardinality constraints. In fact, we will see in the next section that, when we have a query Q over a database $\mathbf{D}$ that belongs to a class $\mathcal{C}$ defined either by cardinality constraints or acyclic degree constraints, we can choose our Q-estimator to be equal to the worst-case $\mathsf{wc}(Q, \mathcal{C})$. This entails that Theorem 4.10 gives us a time complexity of $\mathcal{O}(\frac{\mathsf{wc}(Q,\mathcal{C})}{\max(1,|\mathsf{ans}_{\mathbf{D}}(Q)|)} \cdot |X| \cdot |\mathsf{D}|)$, where X is the set of variables and D the domain. This however depends linearly on $|\mathsf{D}|$ and is therefore not as good what is known from the literature, which is why we will also show that we can apply Algorithm 4.5 to $\widetilde{Q}^b$ (see Section 3.3.3) to improve on the complexity.

4.4 Tree-Superadditive Worst-Case Bounds

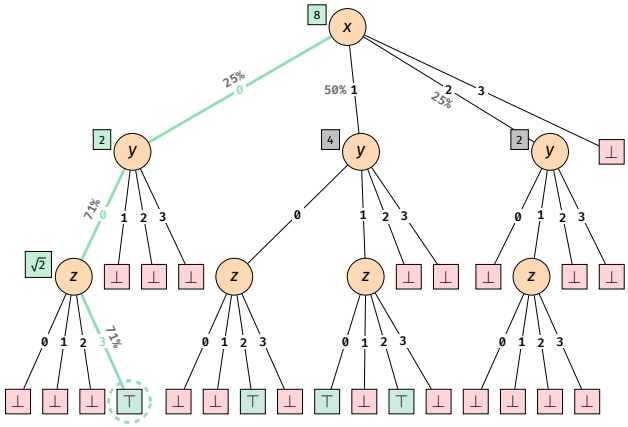
We now wish to generalise these results to whole classes of databases, such as those defined by cardinality constraints or by acyclic degree constraints. This will allow us to recover the sampling results from the literature, and notably from [DLT23; Kim+23; WT24] by using Theorem 4.10 and



(a) A successful sampling run, sampling the tuple $(1, 1, 0)$.



(b) A run that ends in a fail. This run will not return a tuple.



(c) A second successful run, this time sampling the tuple $(0, 0, 3)$.

Figure 4.13: A visual representation of three different runs of sampling with Algorithm 4.5. The chosen path in the tree is materialised by coloured, boldened edges. The coloured square boxes to the left of the nodes visualise the upper bounds for these nodes, the greyed boxes represent the upper bound for the siblings of the chosen nodes. In each run, the probabilities of choosing a given branch are noted alongside it.

the binarisation technique presented in Section 3.3.3. Since the class $\mathcal{C}(\mathsf{DC})$ generalises the class $\mathcal{C}(\leqslant \mathbf{N})$, we could have only treated the first case. We believe however that the $\mathcal{C}(\leqslant \mathbf{N})$ being simpler, it conveys more intuition and is easier to show. Our main tool is a simple consequence of an inequality by Friedgut [Fri04].

▶ **Lemma 4.14** (Generalised Weighted Entropy Lemma, [Fri04])
Given the following objects:

- A hypergraph $\mathcal{H} = (X, E)$;
- A finite set L;
- A family of subsets of X $(F_\ell)_{\ell \in L}$. Let $e_\ell = e \cap F_\ell$ for every $e \in E$. And let $E_\ell = \{e_\ell \mid e \in E\}$.
- A family of weights $W = (w_\ell)_{\ell \in L}$: w_ℓ associates a positive real number to the elements of E_ℓ.
- A family of positive real numbers $A = (\alpha_\ell)_{\ell \in L}$ so that for every $x \in X$, $\sum_{\ell | x \in F_\ell} \alpha_\ell \geqslant 1$.

We have that:

$$\sum_{e \in E} \prod_{\ell \in L} w_\ell(e_\ell) \leqslant \prod_{\ell \in L} \left(\sum_{e_\ell \in E_\ell} w_\ell(e_\ell)^{1/\alpha_\ell} \right)^{\alpha_\ell} .$$

We slightly adapt Lemma 4.14 into the following corollary:

▶ **Corollary 4.15**
For every finite sets I and J, every family of positive real numbers $(\omega_j)_{j \in J}$ so that $\sum_{j \in J} \omega_j \geqslant 1$, and every family of positive real numbers $(a_{i,j})_{i \in I, j \in J}$, we have:

$$\sum_{i \in I} \prod_{j \in J} a_{i,j}^{\omega_j} \leqslant \prod_{j \in J} \left(\sum_{i \in I} a_{i,j} \right)^{\omega_j} .$$

Proof. This is a direct consequence of Lemma 4.14.
By setting:

- $X = I$, $E = \{\{i\} \mid i \in I\}$,
- $L = J$ and for every $j \in J$, $F_j = I$, as a consequence for every $e \in E$ and $j \in J$, $e_j = e$, and thus $E_j = E$.
- For each $j \in J$, we let $w_j(\{i\}) = a_{i,j}^{\omega_j}$.
- Finally, we let $A = (\omega_j)_{j \in J}$. As for every $i \in I$ and $j \in J$, we have that $i \in F_j$, the hypothesis that A must satisfy is a consequence of the hypothesis $\sum_{j \in J} \omega_j \geqslant 1$.

Lemma 4.14 gives us:

$$\sum_{i \in I} \prod_{j \in J} a_{i,j}^{\omega_j} \leqslant \prod_{j \in J} \left(\sum_{i \in I} (a_{i,j}^{\omega_j})^{1/\omega_j} \right)^{\omega_j} = \prod_{j \in J} \left(\sum_{i \in I} a_{i,j} \right)^{\omega_j} .$$

Which is the expected inequality, thus proving our claim. □

The proof of the superadditivity of AGM in Section 4.3 using the Cauchy-Schwarz inequality is a special case of Corollary 4.15. By using Corollary 4.15, we can therefore generalise this proof to the full AGM with arbitrary fractional weights.

4.4.1 Bounds for classes defined by cardinality constraints

Let Q be a join query over variables X and $\mathcal{C}(\leqslant \mathbf{N})$ be a class of databases for Q defined with and cardinality constraints $\mathbf{N} \subseteq \mathbb{N}^{|X|}$. In [AGM13], Atserias, Grohe, and Marx show that $\mathsf{wc}(Q, \mathcal{C}(\leqslant \mathbf{N}))$ can be computed from the solutions of the following linear program:

$$\begin{aligned} &\mathsf{min} \sum_{R \in Q}^{m} \omega_R \mathsf{log}(\mathbf{N}(R)) \\ &\mathsf{s.t.}\ \forall i = 1, \ldots, n, \quad \sum_{R : x_i \in \mathsf{var}(R)} \omega_R \geqslant 1 \end{aligned}$$

and prove $\mathsf{wc}(Q, \mathcal{C}(\leqslant \mathbf{N}))$ is $\prod_{R=1}^{m} \mathbf{N}(R)^{\omega_R}$ up to polylogarithmic factors. Note that the vector ω is a fractional cover of the query hypergraph $\mathcal{H}$.

Let us consider an optimal solution to this linear program ω for a query Q over a database instance $\mathbf{D} \in \mathcal{C}(\leqslant \mathbf{N})$. We take as Q-estimator the function $\mathsf{agm\text{-}upb}(\tau)$ as follows:

$$\mathsf{agm\text{-}upb}(\tau) = \begin{cases} 0 & \text{when } \tau \text{ is inconsistent with } Q \\ \prod_{j=1}^{m} |R_j[\tau]|^{\omega_j} & \text{otherwise} \end{cases} .$$

When τ is inconsistent with Q, then, by definition, $\mathsf{agm\text{-}upb}(\tau) = 0$. Furthermore, when τ is in $\mathsf{ans}_{\mathbf{D}}(Q)$, for every j, $|R_j[\tau]| = 1$ and therefore, $\mathsf{agm\text{-}upb}(\tau) = 1$, as we have seen in Section 4.3. To show that $\mathsf{agm\text{-}upb}$ is a Q-estimator, we finally need to show that for every τ assigning variables up to x_i and consistent with Q, we have $\mathsf{agm\text{-}upb}(\tau) \geqslant \sum_{d \in \mathsf{D}} \mathsf{agm\text{-}upb}(\tau \times \langle x_{i+1} \leftarrow d \rangle)$, that is, that AGM is tree-superadditive in this more general case.

For every j we have:

$$\begin{cases} |R_j[\tau]| = \sum_{d \in \mathsf{D}} |R_j[\tau \times \langle x_{i+1} \leftarrow d \rangle]| & \text{when } x_{i+1} \text{ is in } X_{R_j} - X_i \\ |R_j[\tau]| = |R_j[\tau \times \langle x_{i+1} \leftarrow d \rangle]|, \text{ for every } d \in \mathsf{D} & \text{otherwise.} \end{cases}$$

We let $K = \{k \in [1, m] \mid x_{i+1} \in X_{R_k} \setminus X_i\}$ and $L = [m] \setminus K$. Since $\bigcup_j \mathsf{var}(R_j) = X$, we must have $K \neq \emptyset$. We thus have:

$$\begin{aligned} \sum_{d \in \mathsf{D}} \mathsf{agm\text{-}upb}(\tau \times \langle x_{i+1} \leftarrow d \rangle) &= \sum_{d \in \mathsf{D}} \prod_{k \in K} |R_k[\tau \times \langle x_{i+1} \leftarrow d \rangle]|^{\omega_k} \times \prod_{l \in L} |R_l[\tau \times \langle x_{i+1} \leftarrow d \rangle]|^{\omega_l} \\ &= \sum_{d \in \mathsf{D}} \prod_{k \in K} |R_k[\tau \times \langle x_{i+1} \leftarrow d \rangle]|^{\omega_k} \times \prod_{l \in L} |R_l[\tau]|^{\omega_l} \\ &= \prod_{l \in L} |R_l[\tau]|^{\omega_l} \times \sum_{d \in \mathsf{D}} \prod_{k \in K} |R_k[\tau \times \langle x_{i+1} \leftarrow d \rangle]|^{\omega_k} \end{aligned}$$

Then, $\mathsf{agm\text{-}upb}(\tau) \geqslant \sum_{d \in \mathsf{D}} \mathsf{agm\text{-}upb}(\tau \times \langle x_{i+1} \leftarrow d \rangle)$ follows from:

$$\sum_{d\in\mathsf{D}}\prod_{k\in K}|R_k[\tau\times\langle x_{i+1}\leftarrow d\rangle]|^{\omega_k}\leqslant\prod_{k\in K}\left(\sum_{d\in\mathsf{D}}|R_k[\tau\times\langle x_{i+1}\leftarrow d\rangle|\right)^{\omega_k}$$
$$=$$
$$\leqslant\prod_{k\in K}|R_k[\tau]|^{\omega_k}$$

To summarise, we have the following chain:

$$\sum_{d\in\mathsf{D}}\mathsf{agm\text{-}upb}(\tau\times\langle x_{i+1}\leftarrow d\rangle)\leqslant\prod_{j\in[m]}|R_j[\tau]|^{\omega_j}\cdot\prod_{k\in K}|R_k[\tau]|^{\omega_k}=\prod_{j\in[m]}|R_j[\tau]|^{\omega_j}=\mathsf{agm\text{-}upb}(\tau)$$

By definition of K and since (ω_j) is a solution of our linear program, we have $\sum_{k\in K}\omega_k\geqslant 1$. We can therefore directly get the bound using Friedgut's inequality from Corollary 4.15.

Now that we have established that the AGM bound results in a tree-superadditive estimator, we can use it in an application of our sampling framework presented in Theorem 4.10.

Suppose that we are given a query Q, a database instance $\mathbf{D}$, a (maybe partial) assignment τ and a weight vector ω computed from the AGM bound. This leads to the following adaptation of Algorithm 4.5:

Algorithm 4.16 Sampling with AGM

```
1:  procedure SAMPLE(Q, D, τ, ω)
2:      input: · Q a join query,
               · D a database instance for Q,
               · τ a partial tuple assignment,
               · ω a AGM weight vector
3:      output: a uniformly sampled tuples of ans_D(Q) or fail
4:      if D[τ] contains an empty relation then fail
5:      i ← last variable assigned by τ
6:      if i = n then return τ
7:      for d ∈ D do
8:          b_d ← ∏_{R|x_{i+1}∈var(R)} ( |R[τ×⟨x_{i+1}←d⟩]| / |R[τ]| )^{ω_R}   ▷ selection probability for that child
9:
10:     δ ← pick d ∈ D with probability b_d
11:     if δ = nil then
12:         fail                                                  ▷ no child was selected
13:     else
14:         return SAMPLE(Q, D, τ × ⟨x_{i+1} ← δ⟩, ω)             ▷ a child is selected, continue sampling
```

Note that the algorithm in itself is very similar to the one we describe in Algorithm 3.2. The fail at Algorithm 4.16 can happen if the sum of the upper bounds of the children of the current node is smaller than the upper bound of the current node. In this case, there is a non-zero possibility that no child is selected.

Consider a query Q over a database $\mathbf{D}$ with variables X and domain D. One run of the procedure from Algorithm 4.16 over the trace tree of $(Q,\mathbf{D})$ will result in $\mathcal{O}(|X|)$ calls to the

children function (one per variable) and $\mathcal{O}(|\mathsf{D}| \cdot |X|)$ calls to the agm-upb function (one for each possible valuation of each variable), which is coherent with the results from Theorem 4.6.

These results allow us to specify the results from Theorem 4.10 in the following way:

▶ **Theorem 4.17**
Given a join query Q over variables X and $\mathcal{C}(\leqslant \mathbf{N})$ a class of databases for Q with m relations defined with cardinality constraints $\mathbf{N} \subseteq \mathbb{N}^{|X|}$, for every database $\mathbf{D}$ in $\mathcal{C}(\leqslant \mathbf{N})$, it is possible to uniformly sample $\mathsf{ans}_{\mathbf{D}}(Q)$ with expected time

$$\tilde{\mathcal{O}}\left(\frac{\mathsf{wc}(Q, \mathcal{C}(\leqslant \mathbf{N}))}{\max(1, |\mathsf{ans}_{\mathbf{D}}(Q)|)} \cdot |X| \cdot \log(|\mathsf{D}|) \cdot m\right)$$

Proof. Given $(Q, \mathbf{D})$ with active domain D, we let $b = \lceil \log(|\mathsf{D}|) \rceil$ and we are going to sample from the answers of $\widetilde{Q}^b$ over $\widetilde{\mathbf{D}}^b$. As we have seen in Section 3.3.3, $\mathsf{ans}_{\mathbf{D}}(Q)$ and $\mathsf{ans}_{\widetilde{\mathbf{D}}^b}(\widetilde{Q}^b)$ can be considered to be the same set. Moreover, $\widetilde{\mathbf{D}}^b$ belongs to a class $\widetilde{\mathcal{C}}^b$ defined by cardinality constraints where $\mathsf{wc}(\widetilde{Q}^b, \widetilde{\mathcal{C}}^b) \leqslant \mathsf{wc}(Q, \mathcal{C}(\leqslant \mathbf{N}))$. This implies that there is a $\widetilde{Q}^b$-estimator agm-upb$(\cdot)$.

Using the data structure described in Chapter 3 to represent $R[\tau]$ or a trie structure annotated with cardinalities to represent every relation of Q, we can compute $|R[\tau \times \langle x_{i+1} \leftarrow d \rangle]|$ in $\mathcal{O}(\log|R|)$. Therefore, computing agm-upb$(\cdot)$ takes time $\tilde{\mathcal{O}}(m)$.

Finally, by definition of agm-upb$(\cdot)$, we have agm-upb$(\langle\rangle) \leqslant \mathsf{wc}(\widetilde{Q}^b, \widetilde{\mathcal{C}}^b)$. This allows us to apply Theorem 4.10 and yields the claimed complexity. □

Thus, for classes of databases following a set of cardinality constraints, the AGM bound can be used as a Q-estimator and we can sample uniformly from the answer set of a query. In the next section, we show how this can also be generalised to classes of databases defined with acyclic degree constraints.

4.4.2 Bounds for classes defined by acyclic degree constraints

Let Q be a join query over variables X, and $\mathcal{C}(\mathsf{DC})$ be a class of databases following a set DC of acyclic degree constraints $\delta = (A_\delta, B_\delta, N_\delta)$. We introduce the *polymatroid bound* as a generalisation of the AGM bound that can be formulated over acyclic degree constraints with the solutions of the following linear program:

$$\begin{aligned} &\min \sum_{\delta \in \mathsf{DC}} \omega_\delta \log(N_\delta) \\ &\text{s.t. } \forall x \in X, \sum_{\delta : x \in B_\delta \setminus A_\delta} \omega_\delta \geqslant 1 \end{aligned}$$

For this program to have a solution, we need to assume that for every $x \in X$, there is at least one constraint δ such that $x \in B_\delta \setminus A_\delta$. As stated in Chapter 3, to ensure that acyclic degree constraints induce a finite worst-case, we generally assume that $\bigcup_j \mathsf{var}(R_j) = X$ and that for each relation R, we have at least one cardinality constraint $(\emptyset, \mathsf{var}(R), N_R)$. In other terms, we add the constraint that each relation also satisfies a cardinality constraint. We assume these

conditions to be met here. Now for any solution ω_δ of the previous program, it has been shown by Ngo [Ngo18] that $\mathsf{wc}(Q, \mathcal{C}(\mathsf{DC}))$ is $\prod_{\delta \in \mathsf{DC}} N_\delta^{\omega_\delta}$ up to some polylogarithmic factors.

Let us fix a solution ω of our linear program, and an order $(x_1, \ldots, x_n)$ on the variables X that is compatible with the set DC. For a database $\mathbf{D}$ in $\mathcal{C}(\mathsf{DC})$ and a degree constraint $\delta = (A_\delta, B_\delta, N_\delta)$ in DC, we let R_δ to be a relation in Q that satisfies δ. To lighten the notations, we denote by $R'_\delta = R_\delta|_{B_\delta}$, since the degree constraint is applied to this projection of R_δ and not R_δ itself. Moreover, given a tuple τ in D^{X_i}, we let:

$$N_\delta[\tau] = \begin{cases} 0 & \text{if } \tau \text{ is inconsistent with } Q \\ N_\delta & \text{if } A_\delta \setminus X_i \neq \emptyset \\ |R'_\delta[\tau]| & \text{otherwise} \end{cases}$$

We now take as our Q-estimator the function $\mathsf{pm\text{-}upb}(\tau)$, defined as follows:

$$\mathsf{pm\text{-}upb}(\tau) = \prod_{\delta \in \mathsf{DC}} N_\delta[\tau]^{\omega_\delta} \ .$$

When τ is inconsistent with Q, by definition of $N_\delta[\tau]$, $\mathsf{pm\text{-}upb}(\tau) = 0$. If τ is in $\mathsf{ans}_{\mathbf{D}}(Q)$, since for every δ, $A_\delta \setminus X_n = \emptyset$, $N_\delta[\tau] = |R'_j[\tau]| = 1$ and therefore, $\mathsf{pm\text{-}upb}(\tau) = 1$. This follows from the same argument as before. Once more, proving that $\mathsf{pm\text{-}upb}$ is a Q-estimator finally requires showing that for every τ that is consistent with Q, we have $\mathsf{pm\text{-}upb}(\tau) \geqslant \sum_{d \in \mathsf{D}} \mathsf{pm\text{-}upb}(\tau \times \langle x_{i+1} \leftarrow d \rangle)$.

We let $K = \{\delta \in \mathsf{DC} \mid x_{i+1} \in B_\delta \setminus A_\delta\}$ and $L = \mathsf{DC} \setminus K$. As stated before, for the linear program to have a solution, we assumed $K \neq \emptyset$. We make two observations:

1. For $\delta \in K$, $N_\delta[\tau] = |R'_\delta[\tau]|$ and for any $d \in \mathsf{D}$, $N_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle] = |R'_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle]|$,
2. For $d \in \mathsf{D}$, $N_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle] \leqslant N_\delta[\tau]$.

The first observation follows directly from the fact that $x_{i+1} \in B_\delta \setminus A_\delta$ by definition of K and that $x_1, \ldots, x_n$ is compatible with DC which implies that for any $\delta \in K$, $A_\delta \subseteq X_i$.

For the second inequality, first assume $A_\delta \setminus X_{i+1} \neq \emptyset$. Then both sides are equal to N_δ. Now assume $A_\delta \setminus X_{i+1} = \emptyset$. Then $N_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle] = |R'_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle]|$ and either $N_\delta[\tau] = |R'_\delta[\tau]|$, then the equality is clear, or $N_\delta[\tau] = N_\delta$ and the inequality follows from the fact that R_δ satisfies δ by definition.

From these observations, we obtain:

$$\begin{aligned} \sum_{d \in \mathsf{D}} \mathsf{pm\text{-}upb}(\tau \times \langle x_{i+1} \leftarrow d \rangle) &= \sum_{d \in \mathsf{D}} \prod_{\delta \in K} |R'_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle]|^{\omega_\delta} \times \prod_{\delta \in L} N_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle]^{\omega_\delta} \\ &\leqslant \sum_{d \in \mathsf{D}} \prod_{\delta \in K} |R'_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle]|^{\omega_\delta} \times \prod_{\delta \in L} N_\delta[\tau]^{\omega_\delta} \\ &= \prod_{\delta \in L} N_\delta[\tau]^{\omega_\delta} \times \sum_{d \in \mathsf{D}} \prod_{\delta \in K} |R'_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle]|^{\omega_\delta} \end{aligned}$$

The superadditivity of this function, that is, the fact that $\sum_{d \in \mathsf{D}} \mathsf{pm\text{-}upb}(\tau \times \langle x_{i+1} \leftarrow d \rangle) \leqslant \mathsf{pm\text{-}upb}(\tau)$ is obtained by showing the following statement:

$$\sum_{d \in \mathsf{D}} \prod_{\delta \in K} |R'_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle]|^{\omega_\delta} \leqslant \prod_{\delta \in K} \left(\sum_{d \in \mathsf{D}} |R'_\delta[\tau \times \langle x_{i+1} \leftarrow d \rangle| \right)^{\omega_\delta} = \prod_{\delta \in K} R'_\delta[\tau]^{\omega_\delta} = \prod_{\delta \in K} N_\delta[\tau]^{\omega_\delta} \ .$$

The inequality above is a direct consequence of Friedgut's lemma (Corollary 4.15). Indeed, $\sum_{\delta \in K} \omega_\delta \geqslant 1$ by definition of K and because of the constraints ω_δ verifies in the linear program.

We can then build a very similar algorithm as Algorithm 4.16 using the polymatroid bound. This allows us to conclude with a last result, obtained exactly in the same way as Theorem 4.17.

▶ **Theorem 4.18**
Given a join query Q over variables X, and given $\mathcal{C}(\mathsf{DC})$ a class of databases for Q with m relations satisfying a set of acyclic degree constraints DC, we have that, for every database $\mathbf{D}$ in $\mathcal{C}(\mathsf{DC})$, it is possible to uniformly sample $\mathsf{ans}_{\mathbf{D}}(Q)$ with expected time

$$\mathcal{O}\left(\frac{\mathsf{wc}(Q, \mathcal{C}(\mathsf{DC}))}{\max(1, |\mathsf{ans}_{\mathbf{D}}(Q)|)} \cdot |X| \cdot \log(|\mathsf{D}|) \cdot m\right)$$

4.5 Conclusion

In this chapter, we studied the problem of uniformly sampling answers to join queries. We began by revisiting Rosenbaum's classical algorithm for sampling leaves of a tree without full exploration, and adapted it to a more general setting where only a subset of the leaves are of interest. This led us to introduce the notion of a leaf estimator, a tree-superadditive function that allows us to bound the number of answer leaves in any subtree.

We then applied this perspective to join queries, by showing how the execution trace of our WCOJ algorithm introduced in Chapter 3 naturally unfolds into a tree. Leaves of this trace tree correspond to either inconsistencies or valid answers; and leaf estimators can be constructed from worst-case bounds. This provides a clean reduction from uniform answer sampling for a query over a database to leaf sampling in a tree.

We have also shown how we can use known worst-case bounds over query answers as estimators for this sampling algorithm. Through the AGM bound for cardinality constraints, we showed informally how each partial assignment can be given a natural estimator, and illustrated the procedure on a concrete example. The essential property here is the superadditivity of the estimator, which guarantees the correctness of the sampling procedure.

Finally, we established more general results by proving that both the AGM bound for cardinality constraints and the polymatroid bound for acyclic degree constraints are tree-superadditive, thereby yielding uniform samplers in expected time $\mathcal{O}(\frac{\mathsf{wc}(Q,\mathcal{C})}{\max(1,|\mathsf{ans}_{\mathbf{D}}(Q)|)} \cdot |X| \cdot \log(|\mathsf{D}|) \cdot m)$. Combined with the binarisation technique presented in Chapter 3, this allows us to match recent results in the literature, while relying on a simpler and more modular analysis.

Current chapter references

[AGM13] Albert **Atserias**, Martin **Grohe**, and Dániel **Marx**. *Size bounds and query plans for relational joins*. In *SIAM Journal on Computing* 42.4 (Jan. 2013), pp. 1737–1767. doi 10.1137/110859440.

[CIS25] Florent **Capelli**, Oliver **Irwin**, and Sylvain **Salvati**. *A Simple Algorithm for Worst Case Optimal Join and Sampling*. In *28th International Conference on Database Theory (ICDT 2025), March 25 to March 28, 2025, Barcelona, Spain* (ICDT '25). Ed. by Sudeepa **Roy** and Ahmet **Kara**. Vol. 328. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2025, 23:1–23:19. doi 10.4230/LIPIcs.ICDT.2025.23.

[CY20b] Yu **Chen** and Ke **Yi**. *Random sampling and size estimation over cyclic joins*. In *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*. Ed. by Carsten **Lutz** and Jean Christoph **Jung**. Vol. 155. LIPIcs. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2020, 7:1–7:18. doi 10.4230/LIPIcs.ICDT.2020.7.

[DLT23] Shiyuan **Deng**, Shangqi **Lu**, and Yufei **Tao**. *On Join Sampling and the Hardness of Combinatorial Output-Sensitive Join Algorithms*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '23: International Conference on Management of Data. Seattle WA USA: ACM, June 2023, pp. 99–111. doi 10.1145/3584372.3588666.

[Fri04] Ehud **Friedgut**. *Hypergraphs, Entropy, and Inequalities*. In *The American Mathematical Monthly* 111.9 (June 2004), pp. 749–760. doi 10.2307/4145187.

[Kim+23] Kyoungmin **Kim**, Jaehyun **Ha**, George **Fletcher**, and Wook-Shin **Han**. *Guaranteeing the Õ(AGM/OUT) Runtime for Uniform Sampling and Size Estimation over Joins*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '23: International Conference on Management of Data. Seattle WA USA: ACM, June 2023, pp. 113–125. doi 10.1145/3584372.3588676.

[Ngo18] Hung Q. **Ngo**. *Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems*. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '18: International Conference on Management of Data. Houston TX USA: ACM, May 2018, pp. 111–124. doi 10.1145/3196959.3196990.

[Ros93] Paul R. **Rosenbaum**. *Sampling the Leaves of a Tree with Equal Probabilities*. In *Journal of the American Statistical Association* 88.424 (Dec. 1993), pp. 1455–1457. doi 10.1080/01621459.1993.10476433.

[WT24] Ru **Wang** and Yufei **Tao**. *Join Sampling Under Acyclic Degree Constraints and (Cyclic) Subgraph Sampling*. In *27th International Conference on Database Theory (ICDT 2024)*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Vol. 290. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2024, 23:1–23:20. doi 10.4230/LIPIcs.ICDT.2024.23.

Chapter 5

Using Circuits to Answer Join Queries

Let us change our traditional attitude to the construction of programs

Donald Knuth

In Chapters 3 and 4, we presented a worst-case optimal join algorithm whose trace can be used to sample efficiently from the answers of a query. Notably, the trace of Algorithm 3.2 builds a tree-like structure where the internal nodes are *decision* nodes labelled by variables and the leaves are labelled $\top$ or $\bot$ according to whether the path ending there from the root leads to an assignment of the variables along the path that satisfies the query. In a way, we can see the trace of Algorithm 3.2 as a *trie* representation of the answer set of a query. This data structure representing the answers of a query has interesting algorithmic properties, as we have notably seen in Chapter 4. We can however work on optimising this data structure by allowing for new operations, such as by factorising similar nodes or by simplifying the computation of the subtrees based on the context we evaluate them in.

In this chapter, we therefore present a slightly more complex data structure to represent the answers to a query Q over a database $\mathbf{D}$ as an extension of Algorithm 3.2 with a few extra optimisations. This data structure, that we call $\{\times, \mathsf{dec}\}$-circuit, will also represent $\mathsf{ans}_{\mathbf{D}}(Q)$ in a factorised way, but will also integrate gate factorisation and subquery separation. We also describe a simple, top-down algorithm that builds a $\{\times, \mathsf{dec}\}-$*circuit* from Q and $\mathbf{D}$ (see Algorithm 5.5). The idea behind our algorithm is very similar to the idea of Exhaustive DPLL, an algorithm that was introduced by Sang, Bacchus, Beame, Kautz, and Pitassi [San+04] as a way to solve instances of the #SAT problem. While Exhaustive DPLL does not construct a data structure as such, Huang and Darwiche later observed that the *trace* of the execution of Exhaustive DPLL implicitly builds a Boolean circuit [HD05]. This matches our definition of a $\{\times, \mathsf{dec}\}$-circuit on domain $\{0, 1\}$ and enjoys interesting tractability properties. The algorithm and the analysis presented here were first introduced in [CI24; Cap+25].

The structure of this chapter differs from the presentation of [CI24] and [Cap+25].

We start by introducing relational circuits, the data structure we will be using. Then, we introduce the algorithm that compiles a $\{\times, \mathsf{dec}\}$-circuit from a join query and a database. Later, we introduce signed join queries as a natural extension of join queries and show how we can adapt our algorithm to this new framework. Finally, we establish the complexity of the algorithm, and use similar methods as in Chapter 3 to shave factors from this complexity.

Outline of the current chapter

5.1 Decision Diagrams and Relational Circuits

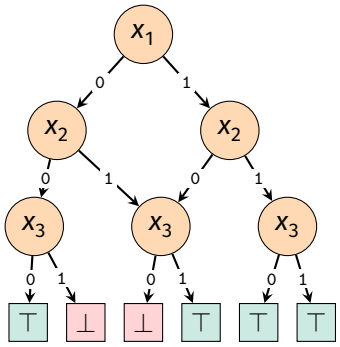
In this section, we introduce the main structures that we will be using in the rest of this chapter and later in Chapter 6.

5.1.1 Decision Diagrams

A *binary decision diagram* (often abbreviated BDD) is a data structure that represents a Boolean function. This is done by representing the function as a rooted, directed, acyclic graph consisting of *decision nodes* labelled by a variable and leaves labelled by $\bot$ or $\top$. The out edges of a decision node are labelled by 0 or 1. If we impose an order $\prec$ over the variables we are considering, then we obtain an *ordered binary decision diagram* (OBDD). A common extension to these structures is to allow them to have a *multivalued* domain, in which case, a decision node can have up to d children, where d is the size of the domain. The interest in representations such as (O)BDDs lies in their ease of use in various applications. For instance, in Example 5.1, we show how we can represent a binary valued relation as an OBDD.

▶ **Example 5.1**
Consider the following relation and its representation as an OBDD:

R	x_1	x_2	x_3
	0	0	0
	0	1	1
	1	0	1
	1	1	0
	1	1	1



Note that a tuple in the relation R corresponds to a path from the root to a leaf labelled by $\top$. The order over the variables we consider here is $x_1 \prec x_2 \prec x_3$.

This OBDD uses factorisation, since the middle gate labelled by x_3 has two parents. This is due to the fact that, whatever the value for x_1, if x_2 is different from x_1, then x_3 has to be set to 1 if we want a tuple from R. An equivalent "tree-like" OBDD could be constructed by duplicating this gate and linking each copy to one parent. This, however, would result in duplication of the data.

▶ **Remark:** the OBDD presented above looks very similar to the data structure that we built with Algorithm 3.2 (see Example 3.17 for example). The only difference between the two structures is that the OBDDs now allow node sharing.

While this representation is compressed due to factorisation, and can already be used in various applications, we want to go even further in the optimisations. Notably, when considering

a join between multiple relations, we could use the structure of the relations to simplify the analysis and find inconsistencies faster. This observation motivates the following definitions.

5.1.2 Relational Circuits

To improve on the previous data structure, we introduce relational circuits. A *multi-directed* graph is a directed graph in which we allow, for two nodes u and v, multiple edges to exist from u to v. A *$\{\bowtie, \mathsf{dec}\}$-circuit* C on variables $X = \{x_1, \ldots, x_n\}$ and domain D is a kind of multi-directed acyclic graph, where the nodes of the graph can either be decision nodes (as in OBDDs) or special nodes called $\times$-nodes. To stay coherent with the logical circuit context, we will often refer to the vertices of the graph as *gates*. A $\{\bowtie, \mathsf{dec}\}$-circuit has one distinguished gate $\mathsf{out}(C)$ called the *output* of C. Moreover, the circuit is labelled as follows:

- any gate of C with no ingoing edge will be called an *input gate* of C and will be labelled by either $\bot$ or $\top$;
- a gate v labelled by a variable $x \in X$ is called a *decision gate*. Each ingoing edge e of v is labelled by a value $d \in \mathsf{D}$ and for each $d \in \mathsf{D}$, there is at most one ingoing edge of v labelled by d. This implies that a decision gate has at most $|\mathsf{D}|$ ingoing edges; and
- any other gate is labelled by $\bowtie$.

The set of all the decision gates in a circuit C is denoted by $\mathsf{decision}(C)$. Given a gate v of C, we denote by C_v the *subcircuit of C rooted in v* to be the circuit whose gates are the gates from which v is reachable by following a directed path in C. We define the *variable set of v*, denoted by $\mathsf{var}(v) \subseteq X$, to be the set of variables x labelling a decision gate in C_v. The variable labelling a specific decision gate v is denoted by $\mathsf{decvar}(v)$. The *size* $|C|$ of a $\{\bowtie, \mathsf{dec}\}$-circuit is defined to be the number of edges of its underlying directed acyclic graph.

We define the *relation $\mathsf{rel}(v) \subseteq \mathsf{D}^{\mathsf{var}(v)}$ computed at gate v* inductively as follows: if v is an input labelled by $\bot$, then $\mathsf{rel}(v) = \emptyset$. If v is an input labelled by $\top$, then $\mathsf{rel}(v) = \{\langle\rangle\}$, that is, $\mathsf{rel}(v)$ is the relation containing only the empty tuple. Otherwise, let $v_1, \ldots, v_k$ be the inputs of v, that is, there exists in the circuit an edge between v_i and v. If v is a $\bowtie$-gate, then $\mathsf{rel}(v)$ is defined to be $\mathsf{rel}(v_1) \bowtie \ldots \bowtie \mathsf{rel}(v_k)$. If v is a decision gate labelled by a variable x, $\mathsf{rel}(v) = R_1 \cup \cdots \cup R_k$ where $R_i = \langle x \leftarrow d_i \rangle \bowtie \mathsf{rel}(v_i) \bowtie \mathsf{D}^{\mathsf{var}(v) \setminus (\mathsf{var}(v_i) \cup \{x\})}$ and d_i is the label of the incoming edge (v_i, v). It is readily verified that $\mathsf{rel}(v)$ is a relation on domain D and variables $\mathsf{var}(v)$. The *relation computed by C over a set of variables X* (assuming $\mathsf{var}(C) \subseteq X$), denoted by $\mathsf{rel}_X(C)$, is defined to be $\mathsf{rel}(\mathsf{out}(C)) \times \mathsf{D}^{X \setminus \mathsf{var}(\mathsf{out}(C))}$.

To ease notation, we use the following shortcuts: first, when the variable set X is not stated explicitly, then we assume that $X = \mathsf{var}(C)$ and second, if v is a decision gate and we have a domain value $d \in \mathsf{D}$, we denote by v_d the gate of C that is connected to v by an edge (v_d, v) labelled by d.

Deciding whether the relation computed by a $\{\bowtie, \mathsf{dec}\}$-circuit is non-empty is NP-complete by a straightforward reduction to the problem of model checking of conjunctive queries [CM77] (see Section 2.3 for more details). As they are presented, such circuits are therefore of little use to get tractability results. We are therefore more interested in a restriction of $\{\bowtie, \mathsf{dec}\}$-circuits which we will denote $\{\times, \mathsf{dec}\}$-circuits. A *$\{\times, \mathsf{dec}\}$-circuit* C is a $\{\bowtie, \mathsf{dec}\}$-circuit that satisfies two extra properties.

1. For every $\bowtie$-gate v of C with inputs $v_1, \ldots, v_k$ and $i < j \leqslant k$, it holds that $\mathsf{var}(v_i) \cap \mathsf{var}(v_j) = \emptyset$. The sets of variables on each side being disjoint, we make this property explicit by changing the notation to a Cartesian product and the $\bowtie$-gate is now denoted by a $\times$-gate.

2. For every decision gate v of C labelled by x with inputs $v_1, \ldots, v_k$ and $i \leqslant k$, it holds that

$x \notin \mathsf{var}(v_i)$. That is, x is not evaluated or assigned at any other point in the subcircuit rooted in v.

Checking whether the relation computed by a $\{\times, \mathsf{dec}\}$-circuit C is non-empty can be done in time $\mathcal{O}(|C|)$ by a dynamic programming algorithm propagating in a bottom-up fashion whether $\mathsf{rel}(v)$ is empty. Similarly, given a $\{\times, \mathsf{dec}\}$-circuit C, one can compute the size of $\mathsf{rel}(C)$ in polynomial time in $|C|$ by a dynamic programming algorithm propagating in a bottom-up fashion $|\mathsf{rel}(v)|$. The idea of the counting algorithm is that input gates provide an answer of 0 or 1, the decision gates sum the values from their inputs and the $\times$-gates multiply the values from their inputs. Example 5.2 presents a simple $\{\times, \mathsf{dec}\}$-circuit, with factorisation, over a relation.

A $\{\times, \mathsf{dec}\}$-circuit C is said to be *complete* if, for every decision gate v on variable x with inputs $v_1, \ldots, v_k$, we have that $\mathsf{var}(v_i) = \mathsf{var}(v) \setminus \{x\}$ for every $i \leqslant k$. Moreover, C is said to be *complete with respect to* X if it is complete and if $\mathsf{var}(C) = X$.

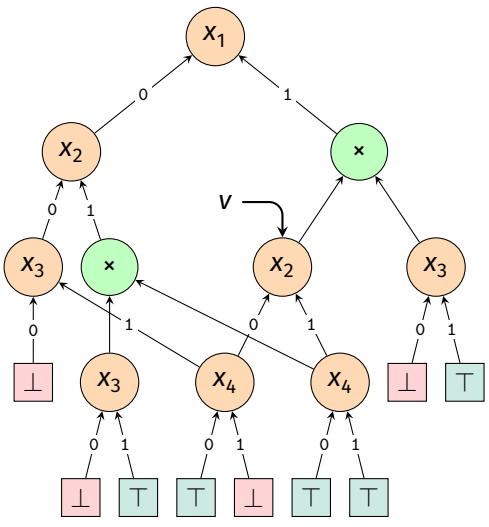
5.1.3 Ordered Relational Circuits

Let X be a set of variables and $\prec$ an order on X. We say that a $\{\times, \mathsf{dec}\}$-circuit C on domain D and variables X is a *$\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit* if for every decision gate v of C labelled with $x \in X$, it holds that for every $y \in \mathsf{var}(v) \setminus \{x\}$, $x \prec y$. We simply say that a circuit C is an ordered $\{\times, \mathsf{dec}\}$-circuit if there exists some order $\prec$ on X such that C is a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit.

▶ **Example 5.2** (A simple ordered $\{\times, \mathsf{dec}\}$-circuit)
Below is a simple example of an ordered $\{\times, \mathsf{dec}\}$-circuit. The domain used is $\{0, 1\}$ and the variable order is $x_1 \prec x_2 \prec x_3 \prec x_4$. On the left, we represent R, the relation computed by the circuit.

R	x_1	x_2	x_3	x_4
	0	0	1	0
	0	1	1	0
	0	1	1	1
	1	0	1	0
	1	1	1	0
	1	1	1	1

$\mathsf{rel}(v)$	x_2	x_4
	0	0
	1	0
	1	1



Notice how the variables on both sides of the $\times$-gates are interlaced: there is no side of the $\times$-gates where every variable is greater than the one appearing on the other side, this is because the order property is only at the decision gates level. This circuit is not complete, since for example, all the subcircuits under the left-most x_3 decision gate do not assign all the remaining

variables. Indeed, the branch where x_3 is assigned to 0 stops immediately without having a decision gate for x_4.

The size of this circuit is 20. This circuit also showcases an example of factorisation. The two decision nodes labelled by x_4 are shared by two different parts of the circuit each, leading to a reduced total number of nodes. For example, there are only 5 $\top$-gates, but 6 total solutions.

If we consider the gate that we call v on the illustration, the relation $\mathsf{rel}(v)$ computed at gate v is the relation represented on the left, the current decision variable is $\mathsf{decvar}(v) = x_2$ and the variable set of v is the set $\mathsf{rel}(v) = \{x_2, x_4\}$.

▶ **Remark 5.3**

You may notice that the edges in the circuit presented in Example 5.2 are directed in the opposite direction as in the OBDD from Example 5.1, as the decision nodes are oriented towards the variables. This is not a classical construction, but it allows us to preserve the *logical circuit* way of seeing things.

5.1.4 Related Frameworks

In the previous sections, we introduced two kinds of relational circuits. While we chose to detail these two constructions, many more types of circuits exist in the literature, each designed to efficiently answer specific problems, with different trade-offs between compactness, tractability or supported operations.

This idea of representing logical formulas in a compact form is at the basis of *knowledge compilation*, whose aim is to change (by a preprocessing phase called *compilation*) the representation of some knowledge base to make it easier to analyse. In more relaxed terms, the idea behind knowledge compilation is that, for a given (hard) problem, one might find a more efficient way of solving it by changing the way the input is represented. This often leads to a computational overhead due to the *preprocessing* of the input, but this overhead can usually be amortized by the efficiency gain. Knowledge compilation then consists in the study of different *target languages*, that is, forms into which we can compile data. Many of those target languages are circuit families. For instance, OBDDs are circuits in Negation Normal Form (or NNF) and the ordered $\{\times, \mathsf{dec}\}$-circuits we presented can be seen as a generalisation of as dec-DNNF (or Decision-Decomposable NNF) to non-binary domains [DM02]. A more complete survey of the different compilation languages and their uses can be found in [DM02] and a more detailed introduction to the concepts of knowledge compilation can be found in [Cap16; Zan25].

Unsurprisingly, there exists a strong link between knowledge compilation and another research direction, *factorised databases* [Olt16], where the objective is to find more compact representations of databases and query answers. Factorised databases were introduced by Olteanu and Závodnỳ [OZ12]. This has been a fruitful line of research, notably around finding size bounds over these factorised representations [OZ12; OZ15] or including more complex operators such as aggregation or ordering [Bak+13].

The relational circuits we will use in the following chapters, and notably ordered $\{\times, \mathsf{dec}\}$-circuits are examples of factorised representations such as d-representations [OZ15]. However, they do not need to be structured along a tree, which will allow us to handle more queries. This is especially important when dealing with queries with negative atoms, which we will define later in

Section 5.3.1: β-acyclic signed conjunctive queries for instance are a class of queries that cannot be represented by polynomial size d-representations [Cap17].

5.2 From Queries to Relational Circuits

Now that we have introduced relational circuits as a data structure, we still have to show how we can construct such circuits in such a way that they are useful when dealing with join queries. The gist behind the method we will present now is analog to that of Chapter 3 where we were joining relations in a worst-case optimal way. As we mentioned briefly in this chapter's introduction, the algorithm we present in Algorithm 5.5 can be viewed as an extension of Exhaustive DPLL, first introduced in [San+04]. While this method was originally devised to solve the #SAT problem, that is, counting the number of satisfying assignments to a logical formula in conjunctive normal form, it has been shown that the trace of this algorithm builds a relational circuit [HD05].

5.2.1 Connecting the Dots: WCOJ and DPLL

In Chapter 3, we introduced Algorithm 3.2 as an worst-case optimal join algorithm. The trace of this algorithm produces a structure, such as shown in Example 3.3 forms a rooted, directed and acyclic graph. The nodes of this graph are labelled by the variables of the considered query and the leaves are labelled by $\top$ or $\bot$. As such, the trace of Algorithm 3.2 is a *multivalued* decision diagram. Thus, when considering the binarised version from Example 3.17, we obtain an OBDD, which can also be seen as a $\{\mathsf{dec}\}$-circuit, without factorisation.

From Algorithm 3.2, a simple algorithm, we therefore transformed (or *compiled*) a query over a database into a form of relational circuit whose relation represents the answer set of the query. A natural direction stemming from this observation is to ask whether we could efficiently compile a query into different forms of relational circuits while taking advantage of their structure. The rest of this chapter is devoted to showing how we can in fact construct a ordered $\{\times, \mathsf{dec}\}$-circuit from a query and a database, by slightly adapting Algorithm 3.2 to take better advantage of both the structure of the query and the properties of the circuit.

The main changes from Algorithm 3.2 consist in *factorising* the circuit as we build it, that is, avoiding the reconstruction of gates computing the same result, and using the $\times$-gates of ordered $\{\times, \mathsf{dec}\}$-circuits to divide the query into smaller components that will be easier to handle. Note that, with little change, we could include factorisation in Algorithm 3.2 and still keep the OBDD trace.

5.2.2 Exhaustive DPLL for Join Queries

The main idea of our version of DPLL for join queries is the following: given an order $\prec$ on the variables of a join query Q and a database $\mathbf{D}$, we construct a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit (where $x \succ y$ if, and only if, $y \prec x$)[1] computing $\mathsf{ans}_{\mathbf{D}}(Q)$ by successively testing the variables of Q with decision gates, from the highest to the lowest with respect to $\prec$. At its simplest form, the algorithm picks the highest variable x of Q with respect to the order $\prec$, creates a new decision gate v on x and then, for every value d from the domain D, sets x to d and recursively computes a gate v_d computing the subset of $\mathsf{ans}_{\mathbf{D}}(Q)$ where $x = d$. We then add v_d as an input of v. The whole pseudocode for this algorithm is presented in Algorithm 5.5.

[1]While it may seem strange at this point that the order given in the input differs from the order used by the output circuit, we will later see that $\prec$ corresponds to a structural parameter, generalising existing results, but we have to follow it in reverse to build the circuit.

Until this point, our new method is the same as the one we use in Algorithm 3.2. However, the approach used in Algorithm 3.2 is an approach for worst-case optimal joining of relations. While it does provide optimality in the worst case, it will not be enough for us to obtain interesting tractability results, and we provide improvements to the algorithm to balance this. Indeed, we have to consider that multiple variable assignments might give us the same subqueries by eliminating the same relations for example. By using this naive approach, we would be recomputing the same result multiple times. A second important point lays in the fact that, by assigning certain variables, we could get a query that consists in different parts that do not contain the same variables. In this case, simply assigning any possible domain value to the decision nodes would result in a suboptimal circuit, since we would be working on relations that are not directly involved. Example 5.4 shows a simple example of these shortcomings.

▶ **Example 5.4**
Consider a query $Q(x, y, z) :- \mathsf{R}(x, y), \mathsf{S}(y, z)$.

Suppose we follow an order $x \prec y \prec z$. If the tuples $\tau_1 = \langle x \leftarrow 1, y \leftarrow 1\rangle$ and $\tau_2 = \langle x \leftarrow 2, y \leftarrow 1\rangle$ are both in R, then the recursive call for τ_1 and for τ_2 have the same answer set, because they both correspond to the answer of $\mathsf{S}[\langle y \leftarrow 1\rangle]$.

For another example, suppose we now followed an order $y \prec x \prec z$. After setting the value of y to a value d, the problem then is solving $Q' :- \mathsf{R}(x, d), \mathsf{S}(d, z)$, which, since the atoms do not share variables, boils down to the Cartesian product of $\mathsf{R}[\langle y \leftarrow d\rangle]$ with $\mathsf{S}[\langle y \leftarrow d\rangle]$

We therefore add a few optimizations. First, we keep a *cache* of already computed queries so that, if we recursively call the algorithm twice on the same input, we can directly return the previously constructed gate. Moreover, if we detect that the answers of Q are the Cartesian product of two or more subqueries $Q_1, \ldots, Q_k$, then we create a new $\times$-gate v, recursively call the algorithm on each component Q_i to construct a gate w_i and plug each w_i to v. Detecting such cases is mainly done syntactically, by checking whether the query can be partitioned into subqueries having disjoint variables.

The complexity of the previously described algorithm may however vary if one is not careful in the way the recursive calls are actually made. We therefore give a more formal presentation of the algorithm, whose pseudocode is given in Algorithm 5.5. We will prove upper bounds on the runtime of Algorithm 5.5 parameterised by the structure of the query in Section 5.4. Since we are not interested in complexity analysis yet, we deliberately let the underlying data structures for encoding relations unspecified and delay this discussion to Section 5.4. Algorithm 5.5 relies on notation that will be introduced immediately after the pseudocode.

Comparison with WCOJ. While Algorithm 5.5 is more complicated than Algorithm 3.2, both their structures are the same. Line 4 is an addition but is the consequence of factorising the circuit. Lines 5 and 6 are the same end cases as in Algorithm 3.2 and the loop starting at Line 8 serves the same function as the loop from our WCOJ (see Line 8) but is more detailed from the division of the query into disjoint subqueries.

A few novel notations are used in Algorithm 5.5. First, we recall two definitions from Chapter 3: (1) for a subset of variables $Y \subseteq X$, we denote by $\mathsf{ans}_{\mathbf{D}}^{Y}(Q)$ the set of tuples over variables Y that are still consistent with Q, that is, $\mathsf{ans}_{\mathbf{D}}^{Y}(Q) = \{\tau \mid \tau \in \mathsf{D}^{Y}, \tau \text{ is consistent with } Q\}$; (2), for a partial assignment τ, we denote by $\mathsf{ans}_{\mathbf{D}}(Q, \tau)$ the set of tuples from the answer set of the query that are compatible with τ, $\mathsf{ans}_{\mathbf{D}}(Q, \tau) = \{\sigma \mid \sigma \in \mathsf{ans}_{\mathbf{D}}(Q), \sigma \simeq \tau\}$. Recall that a tuple τ is said

Algorithm 5.5 An algorithm to compute a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit representing $\mathsf{ans}_{\mathbf{D}}(Q)$

```
1:  procedure DPLL(Q, τ, D, ≺)
2:      input: · Q a join query,
               · τ a partial tuple assignment,
               · D a database instance for Q,
               · ≺ an order on the variables
3:      output: a ≺-ordered {×, dec}-circuit for ans_D(Q)
4:      if (Q, τ) is in cache then return cache(Q, τ)
5:      if τ is inconsistent with Q then return ⊥-gate
6:      if τ assigns every variable in Q then return ⊤-gate
7:      x ← max_≺ var(Q)
8:      for d ∈ D do
9:          τ' ← τ × ⟨x ← d⟩
10:         if τ' is inconsistent with Q then  v_d ← ⊥-gate
11:         else
12:             let Q_1, ..., Q_k be the τ'-connected components of Q
13:             for i = 1 to k do
14:                 w_i ← DPLL(Q_i, τ_i, D, ≺) where τ_i = τ'_{|var(Q_i)}
15:             if k = 1 then
16:                 v_d ← w_1
17:             else
18:                 v_d ← new ×-gate with inputs w_1, ..., w_k
19:     v ← new dec-gate connected to v_d by a d-labelled edge for every d ∈ D
20:     cache(Q, τ) ← v
21:     return v
```

to be inconsistent with an atom $\mathsf{R}(\mathbf{x})$ if the restriction of R to the tuples compatible with τ is empty, that is, $\mathsf{R}[\tau] = \emptyset$. Note that if a query Q contains at least one atom R with which a tuple τ is inconsistent, then Q has no answers compatible with τ and therefore we can lift the notation and say that τ is inconsistent with Q.

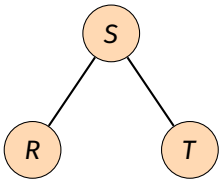
For a tuple $\tau \in \mathsf{D}^Y$ assigning a subset Y of variables of Q, the *τ-intersection graph $\mathcal{I}_\tau^Q$ of Q* is the graph whose vertices are the atoms of Q having at least one variable not in Y, and there is an edge between two atoms a and b of Q if a and b share a variable that is not in Y. Observe that $\mathcal{I}_\tau^Q$ does not depend on the values of τ but only on the variables it sets. Hence it can be computed in polynomial time in the size of Q only. A connected component $\mathcal{C}$ of $\mathcal{I}_\tau^Q$ naturally induces a subquery Q_C of Q and is called a *τ-connected component*. Q is partitioned into its τ-connected components and the set of atoms whose variables are completely set by τ. More precisely, $Q = \bigcup_{\mathcal{C} \in \mathsf{CC}} Q_C \cup Q'$ where CC are the connected components of $\mathcal{I}_\tau^Q$ and Q' contains every atom a of Q on variables set $\mathbf{x}$ such that $\mathbf{x}$ only has variables in Y. Observe that if τ is an answer of Q', then $\mathsf{partial\text{-}ans}_\tau(Q) = \bigtimes_{\mathcal{C} \in \mathsf{CC}} \mathsf{partial\text{-}ans}_{\tau_\mathcal{C}}(Q_\mathcal{C})$ where $\tau_\mathcal{C} = \tau_{|\mathsf{var}(Q_\mathcal{C})}$ since if $\mathcal{C}_1$ and $\mathcal{C}_2$ are two distinct τ-connected components of $\mathcal{I}_\tau^Q$, then $\mathsf{var}(Q_{\mathcal{C}_1}) \cap \mathsf{var}(Q_{\mathcal{C}_2}) \subseteq Y$.

▶ **Example 5.6** (Intersection Graphs & Connected Components)
To illustrate the previous notions, consider the query:

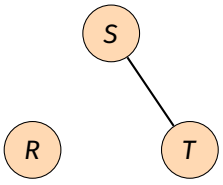
$$Q \text{ :- } \mathsf{R}(x, y),\ \mathsf{S}(y, z),\ \mathsf{T}(z)$$

and a domain value d.

We represent two intersection graphs below:



This is the $\langle\rangle$-intersection graph of Q. All three atoms are represented since no variable has been assigned. There is an edge between R and S since they share y and an edge between S and T since they share z.



This is the $\langle y \leftarrow d\rangle$-intersection graph of Q. All three atoms are still represented since none of the atoms have all there variables assigned. The edge between R and S no longer exists as they do not share an unassigned variable anymore.

From the figure on the right, the tuple $\tau = \langle y \leftarrow d\rangle$ splits the intersection graph (and thus, Q) into two τ-connected components.

If we had set z instead of y, then the intersection graph would consist of the nodes for R and S linked by an edge, with the node for T being removed due to the fact that all its variables are assigned.

5.2.3 Running DPLL

This section presents a simple example of running Algorithm 5.5 for a query over a given database.

▶ **Example 5.7** (Running Algorithm 5.5)
In this example, we will illustrate a run of Algorithm 5.5 over the simple join query:

$$Q(x_1, x_2, x_3, x_4) \text{ :- } \mathsf{R}(x_1, x_2),\ \mathsf{S}(x_1, x_3),\ \mathsf{T}(x_2, x_4)$$

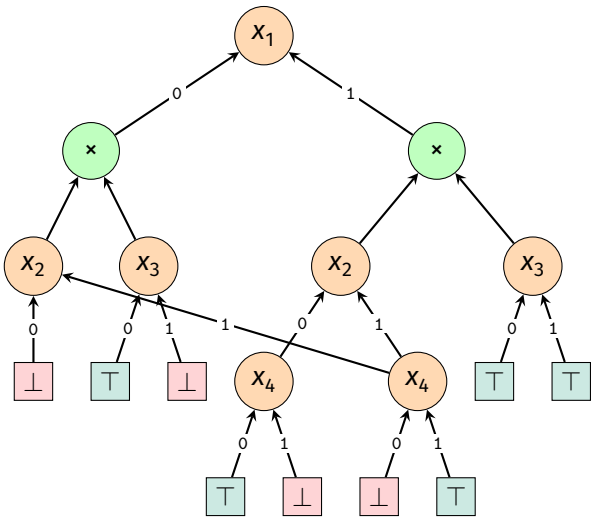
that we evaluate over the following database $\mathbf{D}$ over the binary domain $\{0, 1\}$:

R	x_1	x_2
	0	1
	1	0
	1	1

S	x_1	x_3
	0	0
	1	0
	1	1

T	x_2	x_4
	0	0
	1	1

Let's consider an order $\prec$ such that $x_4 \prec x_3 \prec x_2 \prec x_1$. The execution of Algorithm 5.5 with these parameters, that is, a call to $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$, would yield the following ordered $\{\times, \mathsf{dec}\}$-circuit:



▶ **Details about the run.**
The first recursive call happens on assignment $\tau_0 = \langle x_1 \leftarrow 0\rangle$, which is still consistent with the query. At this point, the τ_0-intersection graph of Q connects R and T through x_2 and S is disconnected from the other two relations. This implies that there exists *two* connected components and thus, a Cartesian product gate is produced and two recursive calls happen: the first has parameters $(Q' \text{ :- } \mathsf{R}(x_1, x_2), \mathsf{T}(x_2, x_4), \langle x_1 \leftarrow 0\rangle)$ and the second has $(Q' \text{ :- } \mathsf{S}(x_1, x_3), \langle x_1 \leftarrow 0\rangle)$. We focus on the first call in this example.

After setting a decision gate by x_2, a new recursive call happens on assignment $\tau_1 = \langle x_1 \leftarrow 0, x_2 \leftarrow 0\rangle$. Now, R and τ_1 are inconsistent, since $\tau_1 \notin \mathsf{R}$, and therefore a $\bot$-gate is created and the algorithm backtracks and sets x_2 to 1 to build $\tau_2 = \langle x_1 \leftarrow 0, x_2 \leftarrow 1\rangle$ on Line 9.

This assignment does not split the intersection graph into more than 1 connected components, however, the τ_2-connected component of Q does not contain R anymore since all variables in R have been assigned in τ_2. The parameters of this recursive call are thus $(Q' :\text{-}\, \mathsf{T}(x_2, x_4), \tau_2)$. Since τ_2 is still consistent with the query, then a decision gate is created for x_4 (note that x_4 is the next variable to be evaluated and not x_3 since x_3 is not present in the current connected component). Two recursive calls are then made: the first assigning x_4 to 0, leading to an inconsistency (and thus a $\bot$-gate) since the tuple $\langle x_2 \leftarrow 1, x_4 \leftarrow 0\rangle \notin \mathsf{T}$, and the second assigning x_4 to 1, leading to a consistent assignment (and thus a $\top$-gate), since all the variables for this subquery are assigned.

The algorithm has then finished with this subquery and returns to Line 12 for the call with parameters $(Q' :\text{-}\, \mathsf{S}(x_1, x_3), \langle x_1 \leftarrow 0\rangle)$.

Later, the algorithm will eventually backtrack to the first call in the stack and set $\tau'_0 = \langle x_1 \leftarrow 1\rangle$ on Line 9. This will again split the intersection graph into *two* connected components and create a Cartesian product gate. Most of the following calls adhere to the same idea as we have described above. Let us focus on one specific call, that happens after constructing the decision gate for x_2. One of the recursive calls at that point will be on $(Q' :\text{-}\, \mathsf{T}(x_2, x_4), \langle x_1 \leftarrow 1, x_2 \leftarrow 1\rangle$. Recall now that we have already computed a gate matching this call (see paragraph above). A cache hit then occurs on Line 4, leading to sharing in the circuit.

▶ **Remark:** The interested reader may find an interactive version of this example and a few others here.

5.3 Handling Negations with Relational Circuits

While conjunctive queries already capture a large fragment of practical database tasks, they only allow for *positive* information: an atom $\mathsf{R}(\mathbf{x})$ requires that a tuple of values assigned to the variables $\mathbf{x}$ belongs to the relation R. In many situations however, one is also interested in expressing *negative* conditions such as "a tuple τ must *not* belong to a relation R".

Intuitively, the presence of negative atoms disrupts the usually monotonic behaviour of joins. In a positive join query, extending a partial assignment of variables can only eliminate answers. Indeed, once a tuple ceases to satisfy one of the atoms, it can no longer satisfy the entire query and therefore cannot be an answer. When we consider queries with negation, this monotonicity no longer holds: if a tuples ceases to satisfy a negative atom, it does not imply that it is inconsistent with the query. On the other hand, a tuple becomes inconsistent with the query if it assigns all of the variables of a negative atom and is still consistent with the atom. Consequently, using signed join queries brings both new challenges and new opportunities. Negation introduces the risk of exponential blow-up in naive evaluation, yet it also allows for simplifications whenever a negated atom becomes true, as one can then decompose the query into smaller independent components. In the rest of this section, we will formally introduce our signed join query framework and show that Algorithm 5.5 can be naturally extended to take these negations into account.

5.3.1 Adding negation to join queries

We will now consider an extension of the join query model that we defined earlier in Chapter 1, in which we allow for certain atoms to be negated. Formally:

▶ **Definition 5.8** (Signed Join Queries)
A *signed join query* Q over variables X is an expression of the form:

$$Q(X) :\text{-}\ R_1(\mathbf{x_1}),\ \ldots,\ R_\ell(\mathbf{x}_\ell),\ \neg S_{\ell+1}(\mathbf{x}_{\ell+1}),\ \ldots,\ \neg S_m(\mathbf{x_m})$$

where each R_i and S_j are relation symbols and the $\mathbf{x_k}$ are tuples of variables in X.

When dealing with signed join queries, we call the elements of the form $R_i(\mathbf{x_i})$ *positive atoms* and elements of the form $S_j(\mathbf{x_j})$ *negative atoms*. Given a signed join query Q, we denote the set of positive atoms (respectively negative atoms) of Q by $\mathsf{atoms}^+(Q)$ (respectively $\mathsf{atoms}^-(Q)$).

5.3.2 Extending Algorithm 5.5 to signed queries

Extending our version of DPLL from Algorithm 5.5 to work with signed join queries is possible when taking into account the changes brought by negation. In Section 5.2, we detect syntactically whether a query can be divided into independent subqueries in order to build a Cartesian product gate. However, this syntactical detection, if kept as such, would fail to give good complexity bounds in the presence of negative atoms. To achieve the best complexity, we therefore also remove from Q every negative atom as soon as it is satisfied by the current partial assignment. This allows us to discover more cases in which the query has more than one connected component.

We also previously defined an tuple τ to be inconsistent with an atom R (and thus a query) if there are no tuples compatible with τ in R, that is, $\mathsf{R}[\tau] = \emptyset$. Now that we also consider negative atoms, there is also a second case to consider. If a query Q contains a negative atom $\neg\mathsf{R}(\mathbf{x})$ such that τ assigns every variable of $\mathbf{x}$ and $\tau(\mathbf{x}) \in \mathsf{R}$, then $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau) = \emptyset$. We now say that a tuple τ inconsistent with Q if any of these two cases arises.

Finally, before presenting the updated version of the algorithm, we introduce a new operation. Observe that if $\neg\mathsf{R}(\mathbf{x})$ is a negative atom of Q such that τ is inconsistent with $\mathsf{R}(\mathbf{x})$, then $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau) = \mathsf{partial\text{-}ans}_{\mathbf{D}}(Q', \tau) \times \mathsf{D}^W$ where $Q' = Q \setminus \{\neg\mathsf{R}(\mathbf{x})\}$ and $W = \mathsf{var}(Q) \setminus (\mathsf{var}(Q') \cup \mathsf{var}(\tau))$ (some variables of Q may only appear in the atom $\neg\mathsf{R}(\mathbf{x})$). This motivates the following definition: the *simplification of Q with respect to τ and $\mathbf{D}$*, denoted by $Q \Downarrow \langle \tau, \mathbf{D} \rangle$ or simply by $Q \Downarrow \tau$ when $\mathbf{D}$ is clear from context, is defined to be the subquery of Q obtained by removing from Q every negative atom $\neg\mathsf{R}(\mathbf{x})$ of Q such that τ is inconsistent with $\mathsf{R}(\mathbf{x})$. From what precedes, we clearly have $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau) = \mathsf{partial\text{-}ans}_{\mathbf{D}}(Q', \tau) \times \mathsf{D}^W$ where $Q' = Q \Downarrow \langle \tau, \mathbf{D} \rangle$ and $W = \mathsf{var}(Q) \setminus (\mathsf{var}(Q') \cup \mathsf{var}(\tau))$.

The extension of the algorithm then consists in removing these satisfied negative atoms before looking for connected components. In Algorithm 5.9, this is shown at Line 12.

As for the first version, we provide a quick example of a run of Algorithm 5.9. To avoid redundancy, we will only focus on the parts that are different due to the addition of negative atoms.

▶ **Example 5.10** (Running Algorithm 5.9)
We illustrate the previous definitions and a run of Algorithm 5.9 on the following signed query:

$$Q(x_1, x_2, x_3, x_4) :\text{-}\ \neg S(x_1, x_2, x_3, x_4),\ T(x_1, x_3),\ R(x_2, x_4)$$

Algorithm 5.9 Computing a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit representing the answers of an SJQ

procedure DPLL$(Q, \tau, \mathbf{D}, \prec)$
  input: · Q a join query,
    · τ a partial tuple assignment,
    · $\mathbf{D}$ a database instance for Q,
    · $\prec$ an order on the variables
  output: a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit for $\mathsf{ans}_{\mathbf{D}}(Q)$
  if (Q, τ) is in cache **then return** $\mathsf{cache}(Q, \tau)$
  if Q is inconsistent with τ **then return** $\bot$-gate
  if τ assigns every variable in Q **then return** $\top$-gate
  $x \leftarrow \mathsf{max}_{\prec}\mathsf{var}(Q)$
  for $d \in \mathsf{D}$ **do**
    $\tau' \leftarrow \tau \times \langle x \leftarrow d \rangle$
    if Q is inconsistent with τ' **then** $v_d \leftarrow \bot$-gate
    else
      let $Q_1, \ldots, Q_k$ be the τ'-connected components of $Q \Downarrow \tau'$
      for $i = 1$ *to* k **do**
        $w_i \leftarrow \mathsf{DPLL}(Q_i, \tau_i, \mathbf{D}, \prec)$ where $\tau_i = \tau'_{|\mathsf{var}(Q_i)}$
      if $k = 1$ **then**
        $v_d \leftarrow w_1$
      else
        $v_d \leftarrow$ **new** $\times$-gate with inputs $w_1, \ldots, w_k$
  $v \leftarrow$ **new** dec-gate connected to v_d by a d-labelled edge for every $d \in \mathsf{D}$
  $\mathsf{cache}(Q, \tau) \leftarrow v$
  return v

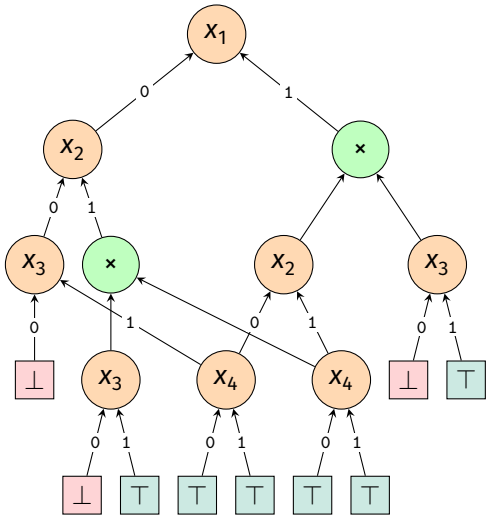
and the following relations on the binary domain $\{0, 1\}$:

S	x_1	x_2	x_3	x_4
	0	0	0	0

R	x_2	x_4
	0	0
	0	1
	1	0
	1	1

T	x_1	x_3
	0	1
	1	1

Executing Algorithm 5.9 then builds the following circuit:



▶ **Details about the run.**

While we will not go into all the details of this run, we can focus on the important calls where the addition of the negative atom modifies the behaviour of Algorithm 5.9.

After building the first few gates, the algorithm will encounter an inconsistency, build a $\bot$-gate (the left-most one) and backtract, setting x_3 to 1 to build partial assignment $\tau' = \langle x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 1\rangle$ on Line 9.

Now, observe that $Q \Downarrow \tau'$ only contains the atom $\mathsf{R}(x_2, x_4)$ because $\mathsf{S}(x_1, x_2, x_3, x_4)$ is inconsistent with τ' and hence we can remove the negative atom $\neg\mathsf{S}$. Moreover, all variables of $\mathsf{T}(x_1, x_3)$ are set by τ' and it is also removed in $Q \Downarrow \tau'$, which is something that Algorithm 5.5 would have missed.

The τ'-intersection graph of $Q \Downarrow \tau$ therefore has only one vertex $\mathsf{R}(x_2, x_4)$. A recursive call is then issued with input $(\mathsf{R}(x_2, x_4), \langle x_2 \leftarrow 0\rangle)$ since $\tau'_{|x_2,x_4} = \langle x_2 \leftarrow 0\rangle$. A decision gate is created for x_4, both values 0 and 1 give answers of Q and therefore the calls create two $\top$-gates.

Now, the algorithm backtracks to the decision gate labelled by x_2 and deals with this part of the recursive call.

Later, the algorithm will eventually backtrack to the first call in the stack and set $\tau' = \langle x_1 \leftarrow 1\rangle$ on Line 9. For the same reason as previously, albeit faster, $\neg\mathsf{S}$ will be simplified in $Q \Downarrow \tau'$. Hence the τ'-intersection graph of $Q \Downarrow \tau'$ has at that point only one vertex for T and one vertex of R.

However, here, the atoms do not share variables, which means a Cartesian product gate is created and two recursive calls happen: the first one has parameters $(\mathsf{T}(x_1, x_3), \langle x_1 \leftarrow 1\rangle)$ and the other one $(\mathsf{R}(x_2, x_4), \langle\rangle)$. The algorithm then treats these two calls, which we will not describe in detail.

Both Algorithms 5.5 and 5.9 use the observations we made previously when describing the algorithms to produce a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit. More precisely:

▶ **Theorem 5.11**
Let Q be a signed join query, $\mathbf{D}$ a database and $\prec$ an order on $\mathsf{var}(Q)$, then $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ constructs a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit C and returns a gate v of C such that $\mathsf{rel}_{\mathsf{var}(Q)}(v) = \mathsf{ans}_{\mathbf{D}}(Q)$.

Proof. The proof is by induction on the number of variables of Q that are not assigned by τ. We claim that $\mathsf{DPLL}(Q, \tau, \mathbf{D}, \prec)$ returns a gate computing $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau)$ which is stored into $\mathsf{cache}(Q, \tau)$. If every variable is assigned, then $\mathsf{DPLL}(Q, \tau, \mathbf{D}, \prec)$ returns either a $\top$-gate or a $\bot$-gate depending on whether τ is inconsistent with Q or not, which clearly is $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau)$. Otherwise, it returns and adds in the cache a decision gate v connected to a gate v_d by a d-labelled edge for each $d \in \mathsf{D}$. We claim that v_d computes $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau \times \langle x \leftarrow d\rangle)$. It is enough since in this case, by definition of the relation computed by a decision gate, v computes $\bigcup_{d\in\mathsf{D}} \mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau \times \langle x \leftarrow d\rangle) \times \langle \tau \leftarrow d\rangle = \mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau)$.

To prove that v_d computes $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau')$ where $\tau' = \tau \times \langle x \leftarrow d\rangle$, we separate two cases: if τ' is inconsistent with Q then $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau')$ is empty and v_d is a $\bot$-gate, which is what is expected. Otherwise, let $Q_1, \ldots, Q_k$ be the τ'-connected components of $Q \Downarrow \tau'$ and let $\tau_i = \tau'_{|\mathsf{var}(Q_i)}$. From what precedes, we have $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q, \tau') = \bigtimes_{i=1}^{k} \mathsf{partial\text{-}ans}_{\mathbf{D}}(Q_i, \tau_i)$. The algorithm uses a gate w_i from Line 14, obtained from a recursive call to $\mathsf{DPLL}(Q_i, \tau_i, \mathbf{D}, \prec)$ where the number of variables not assigned by τ_i in Q_i is less than the number of variables unassigned by τ in Q. Hence, by induction, w_i computes $\mathsf{partial\text{-}ans}_{\mathbf{D}}(Q_i, \tau_i)$ and since v_d is a $\times$-gate connected to each w_i, we indeed have $\mathsf{rel}(v_d) = \bigtimes_{i=1}^{k} \mathsf{partial\text{-}ans}_{\mathbf{D}}(Q_i, \tau_i)$. □

The complexity in the worst case of our DPLL may be high when no cache hit occurs. This would result in at least $|\mathsf{ans}_{\mathbf{D}}(Q)|$ recursive calls. However, when $\prec$ has good properties with respect to Q, we can prove better bounds. Section 5.4 gives upper bounds on the complexity of DPLL depending on $\prec$, using measures first defined in Section 1.2.5, that we will refine to consider signed join queries.

5.4 Complexity of Exhaustive DPLL

In this section, we focus on measuring the complexity of Algorithm 5.9 depending on the query, the database, and the order in which we consider the variables.

5.4.1 Signed Hypergraphs: Definitions and Measures

Since Algorithm 5.9 is designed to work for signed join queries, we naturally generalise some definitions from Chapter 1 to fit to this context better.

A *signed hypergraph* is a specification of hypergraphs where we consider both *positive* and *negative* edges. Formally, a signed hypergraph $\mathcal{H}$ is defined as $\mathcal{H} = (V, E_+, E_-)$, where E_+ is the set of positive edges and E_- the set of negative edges. With $\mathcal{H} = (V, E_+, E_-)$ a signed hypergraph, we say that $\mathcal{H}'$ is a *negative subhypergraph* of $\mathcal{H}$, denoted by $\mathcal{H}' \subseteq^- \mathcal{H}$ if it is a hypergraph of the form $\mathcal{H}' = (V, E_+ \cup E')$ for some subset $E' \subseteq E_-$. This definition of signed hypegraph is coherent with the way negative atoms are handled by Algorithm 5.9. During the execution of the algorithm, negative atoms may be eliminated as soon as they are satisfied or detected to be inconsistent with the current partial assignment. As a result, depending on the database instance and on the explored branch of the search, the algorithm may effectively operate on a query where only a subset of the original negative atoms remains. In other words, we consider a subset of the negative edges as if they were positive.

We can also generalise the notion of hyperorder width to signed hypergraphs. To measure the signed hyperorder width of a signed hypergraph, we find, for a fixed order $\prec$ over the vertices of the hypergraph, the worst possible hyperorder width over all possible subhypergraphs $\mathcal{H}' \subseteq^- \mathcal{H}$. We can then compare the results for all the possible orders over the vertices and take the best out of those. This is what is described in Definition 5.12 with the min over the orders of the max of the widths.

In this sense, signed hyperorder width captures the intrinsic difficulty of a signed join query under variable elimination, independently of which negative constraints are actually active on a given database instance.

More formally, we have the following definition:

▶ **Definition 5.12** (Signed (fractional) hyperorder width)
Let $\mathcal{H} = (V, E_+, E_-)$ be a signed hypergraph. Given an order $\prec$ on V, we define:

- the *signed hyperorder width of $\prec$ for $\mathcal{H}$*, denoted $\mathsf{show}(\mathcal{H}, \prec)$ and the corresponding *signed hyperorder width of $\mathcal{H}$*, denoted $\mathsf{show}(\mathcal{H})$, as:

$$\mathsf{show}(\mathcal{H}, \prec) = \mathsf{max}_{\mathcal{H}' \subseteq^- \mathcal{H}} \mathsf{how}(\mathcal{H}, \prec)$$
$$\mathsf{show}(\mathcal{H}) = \mathsf{min}_{\prec} \mathsf{show}(\mathcal{H}, \prec)$$

and its fractional counterpart:

- the *signed fractional hyperorder width of $\prec$ for $\mathcal{H}$*, denoted $\mathsf{sfhow}(\mathcal{H}, \prec)$ and the corresponding *signed fractional hyperorder width of $\mathcal{H}$*, denoted $\mathsf{sfhow}(\mathcal{H})$ as:

$$\mathsf{sfhow}(\mathcal{H}, \prec) = \mathsf{max}_{\mathcal{H}' \subseteq^- \mathcal{H}} \mathsf{fhow}(\mathcal{H}, \prec)$$
$$\mathsf{sfhow}(\mathcal{H}) = \mathsf{min}_{\prec} \mathsf{sfhow}(\mathcal{H}, \prec)$$

5.4.2 Compilation Complexity

The complexity of our version of DPLL from Algorithm 5.9 on a conjunctive query Q and considering an order $\prec$ can be bounded in terms of the hyperorder width of $\mathcal{H}(Q)$ with respect to $\prec$:

▶ **Theorem 5.13**
Let Q be a signed join query, $\mathbf{D}$ a database over domain D and $\prec$ an order on $\mathsf{var}(Q)$. Then $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ produces a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit C of size $\mathcal{O}((\mathsf{poly}_k|Q|) \cdot |\mathbf{D}|^k \cdot |\mathsf{D}|)$ such that $\mathsf{rel}(C) = \mathsf{ans}_{\mathbf{D}}(Q)$ and:

- $k = \mathsf{fhow}(\mathcal{H}(Q), \prec)$ if Q is positive,
- $k = \mathsf{show}(\mathcal{H}(Q), \prec)$ otherwise.

Moreover, the runtime of $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ is at most $\tilde{\mathcal{O}}(\mathsf{poly}_k(|Q|) \cdot |\mathbf{D}|^k \cdot |\mathsf{D}|)$.

The rest of this section is devoted to proving Theorem 5.13. For this proof, we fix a signed join query Q that has exactly one $\langle\rangle$-component. Note that this can be done without loss of generality since the case where Q has many $\langle\rangle$-components can be easily dealt with by constructing the Cartesian product of each $\langle\rangle$-component of Q. We also fix a database $\mathbf{D}$ and an order $\prec$ on $\mathsf{var}(Q) = \{x_1, \dots, x_n\}$ where $x_1 \prec \cdots \prec x_n$. We let D be the domain of $\mathbf{D}$, n be the number of variables of Q and m be the number of atoms of Q. To ease notation, we will write X instead of $\mathsf{var}(Q)$. For $i \leqslant n$, we denote $\{x_1, \dots, x_i\}$ by $X_{\preceq x_i}$. Similarly, $X_{\prec x_i} = X_{\preceq x_i} \setminus \{x_i\}$, $X_{\succ x_i} = \mathsf{var}(Q) \setminus X_{\preceq x_i}$ and $X_{\succeq x_i} = \mathsf{var}(Q) \setminus X_{\prec x_i}$.

Finally, we let $\mathbf{R}_Q^{\mathbf{D}}$ be the set of (K, σ) such that $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ makes at least one recursive call to $\mathsf{DPLL}(K, \sigma, \mathbf{D}, \prec)$ and such that K is consistent with σ.

Our first step consists in bounding the size of the circuit and the runtime in terms of the number of recursive calls, which we formalise in Lemma 5.14:

▶ **Lemma 5.14**
$\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ produces a circuit of size at most $\mathcal{O}(|\mathbf{R}_Q^{\mathbf{D}}| \cdot |\mathsf{D}| \cdot \mathsf{poly}(|Q|))$ in time $\tilde{\mathcal{O}}(|\mathbf{R}_Q^{\mathbf{D}}| \cdot |\mathsf{D}| \cdot \mathsf{poly}(|Q|))$.

Proof. Given $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$, we bound the number of edges created in the circuit during the first recursive call with these parameters. There are at most $m+1$ such edges for each $d \in \mathsf{D}$. Indeed, for a value $d \in \mathsf{D}$, there are at most m σ'-connected components for $\sigma' = \sigma \times \langle x \leftarrow d \rangle$, hence the first recursive call creates at most m edges between v_d and w_i and one edge between v and v_d.

Observe that any other recursive call with these parameters will not add any extra edges in the circuit since it will result in a cache hit. Moreover, any recursive call of the form (K, σ) with K inconsistent with σ will return a $\bot$-gate without creating any new edge. Hence, the size of the circuit produced in the end is at most $|\mathbf{R}_Q^{\mathbf{D}}| \cdot |\mathsf{D}| \cdot (m+1) = \mathcal{O}(|\mathbf{R}_Q^{\mathbf{D}}| \cdot |\mathsf{D}| \cdot \mathsf{poly}(Q))$.

Moreover, as we have seen in Chapter 3, each operation in Algorithm 5.9 can be done in time polynomial in $|Q|$ if one stores the relation using a well-chosen data structure. Indeed, if one sees a relation R on variables $x_1 \prec \cdots \prec x_n$ as a set of words on an alphabet D whose first letter is x_n and last is x_1, one can store it as a *trie* of size $\tilde{\mathcal{O}}(|R|)$. Then, it is possible to project R on $(x_1, \dots, x_{n-1})$ in $\tilde{\mathcal{O}}(1)$ (this is the encoding used in Veldhuizen's TrieJoin algorithm [Vel14]). Hence, we can test for inconsistency in time $\tilde{\mathcal{O}}(|Q|)$ after having fixed the highest variables in Q to a value $d \in \mathsf{D}$ by going over every atom of Q. During a recursive call (K, σ) where K is inconsistent with σ, we have to check for this inconsistency before returning $\bot$ and cannot really do it for free. To simplify the analysis, we assume that this check is performed before calling the function and hence we can assume every recursive call (K, σ) where K is inconsistent with

σ is done for free. In other words, we incur the cost of this call to the calling function.

Remark that computing the σ'-connected components can be done in polynomial time in $|Q|$ since it boils down to finding the connected components of a graph having at most m nodes. Such a graph can be constructed in polynomial time in $|Q|$ by testing intersections of variables in atoms. Finally, from the previous discussion, a recursive call to Algorithm 5.9 creates at most $m+1$ edges for each $d \in \mathsf{D}$.

Reading and writing values in the cache can be done in time $\tilde{\mathcal{O}}(\mathsf{poly}(|Q|))$ by using a hash table. Indeed, the cached values are subqueries of Q together with partial variable assignments, hence they can be encoded with $\mathcal{O}(|Q|\mathsf{log}|\mathsf{D}|)$ bits. If we account for the cost of reading the cache in a recursive call and the cost of checking inconsistency (as explained before) directly on Line 14, the time for each $(K,\sigma) \in \mathbf{R}_Q^{\mathbf{D}}$ is $\tilde{\mathcal{O}}(|\mathsf{D}|\cdot\mathsf{poly}(|Q|))$ and the total time is $\tilde{\mathcal{O}}(|\mathbf{R}_Q^{\mathbf{D}}|\cdot|\mathsf{D}|\cdot\mathsf{poly}(|Q|))$. □

What we still need to do is to bound the size of $\mathbf{R}_Q^{\mathbf{D}}$, which is done in two steps.t we formalise in two lemmas. First, Lemma 5.16 characterises the structure of the elements of $\mathbf{R}_Q^{\mathbf{D}}$ and second, Lemma 5.18 shows the connections with the structure of the hypergraph of Q.

However, we first need a few notations. Let $Q' \subseteq Q$ be a subquery of Q and x, y two variables of Q' such that $y \prec x$. An *x-path to y in Q'* is a list $x_0, a_0, x_1, a_1, \ldots, a_{p-1}, x_p$ where a_i is an atom of Q' on variables $\mathbf{x_i}$, x_i and x_{i+1} are variables of $\mathbf{x_i}$, $x_0 = x$, $x_p = y$ and $x_i \preceq x$ for every $i \leqslant p$. Thus, x-paths to y in the hypergraph of Q' are simply paths that start from x and ends in y and are only allowed to go through variables smaller than x.

By extension, given an atom a of Q', an *x-path to a in Q'* is an x-path to some variable y in a. The *x-component of Q'* is the set of atoms a of Q' such that there exists an x-path to a in Q'. An *x-access to y in Q'*, is an x-path to some atom a such that y is a variable of a.

Notice that there is an x-access to y in Q' if, and only if, y occurs in the x-component of Q'. Notice that when there is an x-access to y in Q', it may be the case that $x \prec y$.

It turns out that the recursive calls performed by DPLL are in correspondence with the x-components of some $x \in X$ and $Q' \subseteq Q$ where Q' is obtained from Q by removing negative atoms. Intuitively, these removed atoms are the ones that cannot be satisfied anymore by the current assignment of variables.

This observation is stated formally in Lemma 5.16, whose proof is quite technical, which is in part due to heavy notations and concepts that are necessary to truly formalise what is happening. An illustration of the proof of Lemma 5.16 can be found in Figure 5.15.

▶ **Lemma 5.16**
Let $(K,\sigma) \in \mathbf{R}_Q^{\mathbf{D}}$ and let x be the biggest variable of K not assigned by σ. There exists τ a partial assignment of $X_{\succ x}$ such that: $\sigma = \tau_{|\mathsf{var}(K)}$ and K is the x-component of $Q \Downarrow \tau$.

Proof. The proof is done by induction on the order of the recursive calls. We start with the first call, $(Q, \langle\rangle)$.

Since, by our original assumption, Q has one $\langle\rangle$-component, the x_n-component of Q is Q itself. Moreover, since $Q \Downarrow \langle\rangle = Q$, we have that Q is the x_n-component of $Q \Downarrow \langle\rangle$. Now, let $(K,\sigma) \in \mathbf{R}_Q^{\mathbf{D}}$.

By definition, this recursive call is made during the execution of another recursive call with parameters (K',σ'). Assume by induction that, for (K',σ'), the statement of Lemma 5.16 holds. In other words, let x' be the biggest variable of K'. Then K' is the x'-component of $Q \Downarrow \tau'$ for

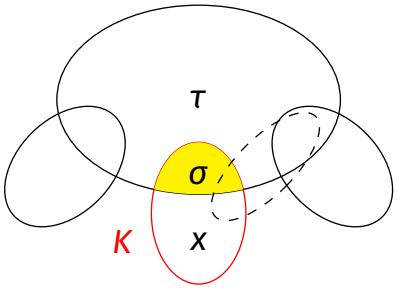


Figure 5.15: Illustration of the proof of Lemma 5.16: τ is the assignment of variables so far when the first recursive call to (K, σ) occurs and it separates Q into disjoint τ-components. Some negative atoms of Q (represented with dashed edges) have been simplified away in $Q \Downarrow \tau$. K, in red, is a τ-connected component of $Q \Downarrow \tau$: it is connected only through the atoms not simplified in $Q \Downarrow \tau$. σ is the part of τ that sets variables in K, and x is the largest variable of K not assigned by σ. Lemma 5.16 proves that K is exactly the x-component of $Q \Downarrow \tau$, that is, every atom that can be reached from x using atoms in $Q \Downarrow \tau$ and variables smaller than x.

some partial assignment τ' of $X_{\succ x'}$ such that $\tau'_{|\mathsf{var}(K')} = \sigma'$. The recursive call then made to (K, σ) has the following form: $\sigma = \sigma''_{|\mathsf{var}(K)}$ where $\sigma'' = \sigma' \times \langle x' \leftarrow d \rangle$ for some $d \in \mathsf{D}$ and K is a σ''-component of $K' \Downarrow \sigma''$.

We claim that K is the x-component of $Q \Downarrow \tau$, where $\tau = \tau' \times \langle x' \leftarrow d \rangle$. First, observe that every atom a of K is in $Q \Downarrow \tau$. This is because, if a is positive, then a is also in $Q \Downarrow \tau$ by definition. On the contrary, if $a = \neg R(\mathbf{x})$ is negative, we claim that a is not inconsistent with τ. Indeed, by induction, a is in $Q \Downarrow \tau'$, hence it is not inconsistent with τ'. Now, since a is also in K and K is a subset of $K' \Downarrow \sigma''$ and $\sigma''(x') = d$, we know that a is not inconsistent with $\tau' \times \langle x \leftarrow d \rangle$ which is τ by definition. Therefore, a is in $Q \Downarrow \tau$.

Now let a be an atom of K. K is a σ''-connected component of $K' \Downarrow \sigma''$. Hence, by definition, we have a path in K from x to some variable y in a that does not use any variable assigned by σ'', which is equivalent to saying that it does not use any variable assigned by σ since, by definition, $\sigma = \sigma''_{|\mathsf{var}(K)}$. As x is defined to be the biggest variable of K that is not assigned by σ, the path from x to y only uses variables smaller than x. In other words, there is an x-path to y in K. That is, every atom a of K is in the x-component of $Q \Downarrow \tau$.

Finally, let a be an atom that is in the x-component of $Q \Downarrow \tau$. Let y be a variable in a such that there is an x-path to y in $Q \Downarrow \tau$. We call this path p, and start by showing that p is also in K'. Since $Q \Downarrow \tau \subseteq Q \Downarrow \tau'$, p is also an x-path to y of $Q \Downarrow \tau'$. As x is occurring in K and $K \subseteq K'$, x occurs in some atom of K'. Moreover, as K' is the x'-component of $Q \Downarrow \tau'$, there is an x'-path to x, call it p', in K'. Since $K' \subseteq Q \Downarrow \tau'$, p' is also an x'-path to x in $Q \Downarrow \tau'$. The path $p'p$, obtained by concatenating the paths p' and p in $Q \Downarrow \tau'$, is an x'-path to y in $Q \Downarrow \tau'$. Since K' is the x'-component of $Q \Downarrow \tau'$, $p'p$ is in K' and hence p is in K'.

We now show that p is in K. By definition, K is the σ''-connected component of $K' \Downarrow \sigma''$ that contains x. In particular, it contains every path from x to other variables in K' that does not pass through x'. As p is an x-path to y, it cannot pass through x' (since $x \prec x'$) and is thus in K'. This finally shows that a is in K and concludes the proof. □

The following lemma establishes a connection between x-components and the structure of the underlying hypergraph. In essence, it allows us to bound the number of atoms needed to cover $X_{\succ x}$ in an x-component using the signed hyperorder width. Figure 5.17 illustrates the lemma

and give an idea of the proof.

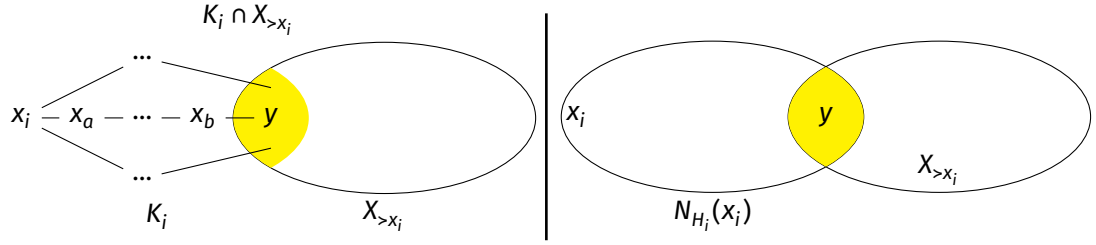


Figure 5.17: Illustration of Lemma 5.18: in the first picture, K_i is pictured on the left and the yellow area represents the variables in K_i that are greater than x_i, that is, that can be reached by following paths of variables smaller than x_i. When removing variables smaller than x_i, these paths are progressively merged into the neighbourhood of x_i, resulting in the second picture: in H_i, the neighbourhood of x_i is exactly the yellow area from the first picture, that is, the vertices of K_i greater than x_i. Every other variable from K_i has been removed. The proof of Lemma 5.18 consists in proving by induction on the paths' length that both yellow areas are the same.

▶ **Lemma 5.18**
Let Q be a signed join query on variable set $X = \{x_1, \dots, x_n\}$. Let $1 \leqslant i \leqslant n$ and K_i the x_i-component of Q. We let $\mathcal{H}$ be the hypergraph of Q, $\mathcal{H}_1 = \mathcal{H}$ and $\mathcal{H}_{j+1} = \mathcal{H}_j / x_j$. We have $\mathcal{N}_{\mathcal{H}_i}(x_i) = \mathsf{var}(K_i) \cap X_{\succeq x_i}$.

Proof. The proof is by induction on i.

We start by proving the equality for $i = 1$. Since there are no variable of K_1 smaller than x_1, it is clear that $K_1 = \{a \in \mathsf{atoms}(Q) \mid x_1 \in \mathsf{var}(a)\}$. Hence $\mathsf{var}(K_1) \cap X_{\succeq x_1} = \mathsf{var}(K_1)$. Moreover, $\mathcal{N}_{\mathcal{H}}(x_1)$ is exactly the set of variables of atoms of Q containing x_1 since there is one hyperedge in $\mathcal{H}$ per atom of Q. Hence, $\mathsf{var}(K_1) \cap X_{\succeq x_1} = \mathcal{N}_{\mathcal{H}}(x_1) = \mathcal{N}_{\mathcal{H}_1}(x_1)$, the last equality being a consequence of $\mathcal{H}_1 = \mathcal{H}$.

Now assume that the equality has been established up to x_{i-1}. We start by proving that $\mathsf{var}(K_i) \cap X_{\succeq x_i} \subseteq \mathcal{N}_{\mathcal{H}_i}(x_i)$. Let $w \in \mathsf{var}(K_i) \cap X_{\succeq x_i}$. First assume $w \in \mathcal{N}_{\mathcal{H}}(x_i)$. That is, there is an edge e in $\mathcal{H}$ such that both w and x_i are in e. Moreover, it is easy to see that $e \setminus \{x_1, \dots, x_{i-1}\}$ is an edge of $\mathcal{H}_i$. Therefore, it is clear that $w \in \mathcal{N}_{\mathcal{H}_i}(x_i)$. Otherwise, if $w \notin \mathcal{N}_{\mathcal{H}}(x_i)$, as $w \in \mathsf{var}(K_i)$, by definition of K_i, there is an x_i-access of length greater than 1 to w in K_i.

Let x_j be the biggest node on that path that is different from x_i. By definition, an x_i-access is an x_i-path and $j < i$. Moreover, splitting that of x_i-access at x_j gives an x_j-access to x_i and an x_j-access to w in K_i. Thus, x_i and w occur in the x_j-component of K_i. Since $K_i \subseteq \mathcal{H}$, they also occur in K_j, the x_j-component of $\mathcal{H}$. By induction, we thus have $w \in \mathcal{N}_{\mathcal{H}_j}(x_j)$ and $x_i \in \mathcal{N}_{\mathcal{H}_j}(x_j)$. Hence w and x_i are neighbours in $\mathcal{H}_{j+1}$ since the edge $\mathcal{N}_{\mathcal{H}_j}(x_j) \setminus \{x_j\}$ has been added in $\mathcal{H}_{j+1}$. In particular, since $i > j$, it means that $\mathcal{H}_i$ contains the edge $\mathcal{N}_{\mathcal{H}_j}(x_j) \setminus \{x_j, \dots, x_{i-1}\}$, which contains both w and x_i, hence $w \in \mathcal{N}_{\mathcal{H}_i}(x_i)$. This establishes the first part of the proof, that is, $\mathsf{var}(K_i) \cap X_{\succeq x_i} \subseteq \mathcal{N}_{\mathcal{H}_i}(x_i)$.

We now prove the converse inclusion. Let $w \in \mathcal{N}_{\mathcal{H}_i}(x_i)$. By definition, $w \in X_{\succeq x_i}$ since

$x_1, \dots, x_{i-1}$ have been removed in $\mathcal{H}_i$. It remains to prove that w is in an atom a such that there is an x_i-path to a. If x_i and w are neighbours in $\mathcal{H}$, then it means that they appear together in an atom of Q and it is clear that $w \in \mathsf{var}(K_i)$. Otherwise, let j be the smallest value for which $w \in \mathcal{N}_{\mathcal{H}_j}(x_i)$, which exists since $w \in \mathcal{N}_{\mathcal{H}_i}(x_i)$. We must have $j > 1$ since x_i and w are not neighbours in $\mathcal{H} = \mathcal{H}_1$. The minimality of j implies that x_i and w are not neighbours in $\mathcal{H}_{j-1}$. Since the only edge added in $\mathcal{H}_j$ is $\mathcal{N}_{\mathcal{H}_{j-1}}(x_{j-1}) \setminus \{x_{j-1}\}$, it means that both x_i and w are neighbours of x_{j-1} in $\mathcal{H}_{j-1}$, that is, $x_i \in \mathcal{N}_{\mathcal{H}_{j-1}}(x_{j-1})$ and $w \in \mathcal{N}_{\mathcal{H}_{j-1}}(x_{j-1})$. By induction, both x_i and w are variables of K_{j-1}. In other words, there is an x_{j-1}-access to x_i and an x_{j-1}-access to w in K_{j-1}. Since $j - 1 < i$, the paths that constitute these x_{j-1}-accesses only pass through variables y so that $y \prec x_{j-1} \prec x_i$. This shows that that there is an x_i-access to w in K_{j-1}. So w is in the x_i-component of K_{j-1} and thus in the x_i-component of $\mathcal{H}$. Hence $w \in \mathsf{var}(K_i)$. □

Finally, we are now ready to prove the upper bound on $|\mathbf{R}_Q^{\mathbf{D}}|$ depending on the width of $\prec$.

▶ **Lemma 5.19**

Let m be the number of atoms of Q and n the number of variables. We have:

- if Q is a positive join query, $|\mathbf{R}_Q^{\mathbf{D}}| \leqslant n|\mathbf{D}|^k$ where $k = \mathsf{fhow}(\mathcal{H}(Q), \prec)$.
- otherwise $|\mathbf{R}_Q^{\mathbf{D}}| \leqslant nm^{k+1}|\mathbf{D}|^k$ where $k = \mathsf{show}(\mathcal{H}(Q), \prec)$.

Proof. We start with the case where Q is a positive join query. Let $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$. In this case, we know by Lemma 5.16 that K is the x_i-component of $Q \Downarrow \tau$ for some $\tau \supseteq \sigma$. Now, since Q does not have negative atoms, $Q = Q \Downarrow \tau$ since $Q \Downarrow \tau$ is obtained from Q by removing negative atoms only. In other words, K is the x_i-component of Q and σ assigns the variables of K that are greater than x_i. We also know that σ is not inconsistent with the atoms of K by definition of $\mathbf{R}_Q^{\mathbf{D}}$. Hence, σ satisfies every atom of K when projected on $X_{\succ x_i}$. By Lemma 5.18, $\mathsf{var}(K) \cap X_{\succ x_i} = N_{x_i}(\mathcal{H}_i)$ where $\mathcal{H}_i$ is defined as in Lemma 5.18. Thus, by definition, there exits a fractional cover of $\mathcal{N}_{\mathcal{H}_i}(x_i)$ using the atoms of Q with value at most $k = \mathsf{fhow}(\mathcal{H}(Q), \prec)$. In other words, σ can be seen as the projection on $\mathsf{var}(K) \cap X_{\succ x_i}$ of an answer of the join of the atoms involved in the fractional cover. By the AGM bound (see Definition 2.7), this join query has at most $|\mathbf{D}|^k$ answers. Hence, there are at most $n|\mathbf{D}|^k$ possible elements in $\mathbf{R}_Q^{\mathbf{D}}$: there are at most n x_i-components (one for each $i \leqslant n$), and at most $|\mathbf{D}|^k$ associated assignments σ to each component.

The case of signed queries is similar but requires more attention. Again, for $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$, we know that K is the x_i-component of $Q \Downarrow \tau$ for some $\tau \supseteq \sigma$ and, as before, τ is consistent with every positive atom in $Q \Downarrow \tau$. Moreover, if $\neg R(\mathbf{x})$ is an atom of $Q \Downarrow \tau$, then τ is consistent with $R(\mathbf{x})$, since otherwise $\neg R(\mathbf{x})$ would not be in $Q \Downarrow \tau$. Now, let $\mathcal{H}'$ be the hypergraph of $Q \Downarrow \tau$. By definition, it is a subhypergraph of $\mathcal{H}(Q)$, where only negative edges have been removed. Hence, by Lemma 5.18, $\mathsf{var}(K) \cap X_{\succ x_i} = \mathcal{N}_{\mathcal{H}'_i}(x_i)$ is covered by at most $\mathsf{show}(\mathcal{H}', \prec) \leqslant \mathsf{show}(\mathcal{H}(Q), \prec) = k$ edges. Hence, σ, which corresponds to τ restricted to $\mathsf{var}(K) \cap X_{\succ x_i}$, can be seen as the projection to $\mathsf{var}(K) \cap X_{\prec x_i}$ of an answer of a positive join query having at most k atoms of Q. Indeed, even if an edge of $\mathcal{H}(Q)$ used to cover $\mathsf{var}(K) \cap X_{\succ x_i}$ corresponds to a negative atom $\neg R$ of Q, we know that τ is consistent with R, otherwise, the atom $\neg R$ would have been simplified by τ in $Q \Downarrow \tau$.

Consider $I_k(Q)$ to be the following set: $I_k(Q)$ contains every pair (L, α) such that L is a

subset of at most k atoms $P_1, \dots, P_{k_1}, \neg N_1, \dots, \neg N_{k_2}$ of Q (that is $k_1 + k_2 \leqslant k$) and α is a solution of the join query with atoms $P_1, \dots, P_{k_1}, N_1, \dots, N_{k_2}$ on $\mathbf{D}$. By definition, it is easy to see that $|I_k(Q)| \leqslant m^{k+1}|\mathbf{D}|^k$. Indeed, there are at most $\sum_{j=1}^{k} \binom{m}{j} \leqslant m^{k+1}$ choices for L and since α is an answer of a positive join query having at most k relations, there are at most $|\mathbf{D}|^k$ such answers.

From what precedes, we know that for every $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$, there exists some α such that $(K, \alpha) \in I_k(Q)$ and $\sigma = \alpha_{|\mathsf{var}(K) \cap X_{\succ x_i}}$ for some $x_i \in X$. Hence there are at most $nm^{k+1}|\mathbf{D}|^k$ elements in $\mathbf{R}_Q^{\mathbf{D}}$, the extra n-factor originating from the choice of x_i. □

At this point, Theorem 5.13 becomes a direct corollary of Lemmas 5.14 and 5.19. If Q is not $\langle\rangle$-connected, then the first recursive call of DPLL will simply break Q into at most $\mathcal{O}(m)$ connected components and recursively call itself on each now $\langle\rangle$-component of Q.

Observe that our result is based on signed hyperorder width, which is not the fractional version. This is because, with our current understanding of the algorithm, we are not able to prove a polynomial combined complexity bound on the runtime of DPLL when only the signed fractional hyperorder width is bounded. Indeed, the proof of Lemma 5.19 breaks in this case.

The proof would be analogous up to the definition of $I_k(Q)$. Indeed, to account for fractional cover, we should not consider $I_k(Q)$ anymore but $J_k(Q)$ defined as follows: $J_k(Q)$ contains every pair (L, α) such that L is a subset of atoms $P_1, \dots, P_{k_1}, \neg N_1, \dots, \neg N_{k_2}$ of Q **having a fractional cover of at most** k and α is a solution of the join query with atoms $P_1, \dots, P_{k_1}, N_1, \dots, N_{k_2}$ on $\mathbf{D}$. Now, bounding the number of α once L is fixed by $|\mathbf{D}|^k$ can still be done by Definition 2.7 but we cannot bound the number of distinct L by m^{k+1} anymore.

At this point, it is not clear to us whether this number can be bounded by a polynomial (where k is considered a constant) in m. The best that we can do is to bound this number by 2^m, that is, just saying that L is one subset of atoms of Q. By doing this, we can still bound $|\mathbf{R}_Q^{\mathbf{D}}|$ by $2^m n|\mathbf{D}|^k$ which yields the following result:

▶ **Theorem 5.20**
Let Q be a signed join query with n variables and m atoms, $\mathbf{D}$ a database over domain D and $\prec$ an order on $\mathsf{var}(Q)$.

Then $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ produces a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit C of size $\mathcal{O}((2^m \mathsf{poly}(|Q|) \cdot |\mathbf{D}|^k \cdot |\mathsf{D}|)$ such that $\mathsf{rel}(C) = \mathsf{ans}_{\mathbf{D}}(Q)$ where $k = \mathsf{sfhow}(\mathcal{H}(Q), \prec)$.

Moreover, the runtime of $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ is at most $\tilde{\mathcal{O}}(2^m \mathsf{poly}(|Q|) \cdot |\mathbf{D}|^k \cdot |\mathsf{D}|)$.

5.5 Reducing the Domain Size

In this section, we explain how one can remove the $|\mathsf{D}|$ factor in the complexity of our version of DPLL from Algorithm 5.9. This is desirable since this extra factor shows a gap between our approach and the optimal bound. We start by observing that this factor is not an over-estimation in the analysis of the algorithm but that it can effectively appear on a concrete example.

▶ **Example 5.21**

Consider the following query:

$$Q :\text{-}\ \mathsf{A}(x_1),\ \mathsf{B}(x_2),\ \neg\mathsf{R}(x_1, x_2)$$

▶ **Note:** this query has already been considered by Zhao, Fan, Ouyang, and Koutris [Zha+24] for similar reasons.

Consider the order x_1, x_2 and take $\mathsf{A}(x_1) = [d]$, $\mathsf{B}(x_2) = [d]$ and $\mathsf{R}(x_1, x_2) = \{(v, v) \mid v \in [d]\}$. In practice, this implies that $\neg\mathsf{R}(x_1, x_2)$ enforces $x_1 \neq x_2$.

The size of the database is $\mathcal{O}(d)$, and the signed hyperorder width of Q is 1, but DPLL will produce a quadratic size circuit. Indeed, $\neg\mathsf{R}(x_1, x_2)$ prevents DPLL from creating any Cartesian product gates.

Moreover, for $d' \in [d]$, $Q \Downarrow \langle x_1 \leftarrow d' \rangle$ is $\big(\mathsf{B}(x_2),\ x_1 \neq x_2\big)$. Therefore, for $d' \neq d''$, $(Q \Downarrow \langle x_1 \leftarrow d' \rangle, \langle x_1 \leftarrow d' \rangle)$ and $(Q \Downarrow \langle x_1 \leftarrow d'' \rangle, \langle x_1 \leftarrow d'' \rangle)$ are syntactically distinct: since they do not assign the same value to x_1, no caching occurs on the recursive calls for each value of $[d]$ assigned to x_1. In turn, this implies that the resulting circuit will have size $\mathcal{O}(d^2)$ since the returned circuit is essentially a tree with one path for each solution of the form $\langle x_1 \leftarrow d_1, x_2 \leftarrow d_2 \rangle$ and $d_1 \neq d_2$.

To overcome this inefficiency, we rely on a simple trick, in the same way as in Chapter 3. This trick was already used in the design of worst-case optimal join algorithms (see [CIS25] or Section 3.3.3 for example), and consists in encoding the domain over the Boolean domain and transforming the query and the database accordingly. It turns out that this transformation can be done in a way that preserves the structure of the query. Moreover, the isomorphism between the answers of the original query and the answers of the transformed one preserves the order. In Chapter 6, this will also allow us to get direct access to the answers of the original query from the answers of the transformed one.

The binarisation operation we will use in this section is the same as the one described in Chapter 3. While we will not go into the details of the construction and the links between the original query and database and their binarised version, we start by refreshing a few of the notations. We will focus on showing that applying this operation to our query and database does not change the width measure that we use to establish the complexity of Algorithm 5.9.

In this section, we consider a query Q on variables set X, a database $\mathbf{D}$ on domain $\mathsf{D} = [d]$ and let $b = \lceil \log(d) \rceil$. For $v \in [d]$, we denote by $\widetilde{v}^b$ the binary encoding of v over b bits and let $\widetilde{v}^b[i]$ be the i^{th} bit of $\widetilde{v}^b$ for every $1 \leqslant i \leqslant b$. For a variable $x \in X$, we introduce fresh variables $x^1, \ldots, x^b$ and let $\widetilde{X}^b = \{x^i \mid x \in X, 1 \leqslant i \leqslant b\}$. For a tuple $\tau \in \mathsf{D}^Y$, we let $\widetilde{\tau}^b$ be the tuple in $\mathsf{D}^{\widetilde{Y}^b}$ such that for every $y \in Y$ and $1 \leqslant i \leqslant b$, $\widetilde{\tau}^b(y^i) = \widetilde{\tau(y)}^b[i]$. For a relation $R \subseteq \mathsf{D}^Y$, we let $\widetilde{R}^b = \{\widetilde{\tau}^b \mid \tau \in R\} \subseteq \mathsf{D}^{\widetilde{Y}^b}$. For a (positive or negative) atom A on variable set Y, we let $\widetilde{A}^b$ be the corresponding atom on variables set $\widetilde{Y}^b$.

We let $\widetilde{Q}^b$ be the query on variables set $\widetilde{X}^b$ whose atoms are $\widetilde{A}^b$ for each atom A of Q. Given a database $\mathbf{D}$ for Q, we let $\widetilde{\mathbf{D}}^b$ be a database for $\widetilde{Q}^b$ where every relation symbol R appearing in Q and interpreted by $R^{\mathbf{D}}$ in $\mathbf{D}$ is now interpreted by $\widetilde{R^{\mathbf{D}}}^b$ in $\widetilde{\mathbf{D}}^b$. A visual and concrete example of the application of binarisation over a query and database can be seen in Example 3.17.

The natural isomorphism between $\mathsf{ans}_{\mathbf{D}}(Q)$ and $\mathsf{ans}_{\widetilde{\mathbf{D}}^b}(\widetilde{Q}^b)$ is interesting since the analysis of DPLL made in Theorem 5.13 has a linear dependency in the size of the domain. This domain is bounded by 2 when considering $\widetilde{\mathbf{D}}^b$ instead of D. Since the number of atoms does not change and the number of variables only increases by a logarithmic factor $\log|\mathsf{D}|$, running Algorithm 5.9 on $\widetilde{Q}^b$ and $\widetilde{\mathbf{D}}^b$ would offer an improvement on the complexity. However, a necessary condition

is that the hyperorder width of Q for $\prec$ and of $\widetilde{Q}^b$ for $\prec^b$ are the same, where $\prec^b$ is the order defined on $\widetilde{X}^b$ as $x^i \prec^b y^j$ if, and only if, $x \prec y$ or $x = y$ and $i > j$. In other words, if $x_1, \ldots, x_n$ is the order on X, then the order $\prec^b$ on $\widetilde{X}^b$ is $x_1^b, \ldots, x_1^1, \ldots, x_n^b, \ldots, x_n^1$. This is in fact true and formalised in Theorem 5.22.

▶ **Theorem 5.22**
For every join query Q and $b \in \mathbb{N}$, the following equalities hold:

$$\mathsf{show}(\mathcal{H}(\widetilde{Q}^b), \prec^b) = \mathsf{show}(\mathcal{H}(Q), \prec)$$
$$\mathsf{sfhow}(\mathcal{H}(\widetilde{Q}^b), \prec^b) = \mathsf{sfhow}(\mathcal{H}(Q), \prec)$$

The rest of this section is dedicated to the proof of Theorem 5.22. It is based on the following notion: given a signed hypergraph $\mathcal{H} = (V, E_+, E_-)$ and a vertex $u \in V$, we define the signed hypergraph $\mathsf{clone}(\mathcal{H}, u)$ to be the hypergraph obtained by *cloning* u *in* $\mathcal{H}$, that is, by adding a new vertex u' in every edge where u appears. More formally, the vertices of $\mathsf{clone}(H, u)$ are $V \cup \{u'\}$ where u' is a fresh vertex not in V and the positive (resp. negative) edges of $\mathsf{clone}(H, u)$ are $\{e \cup \{u'\} \mid e \in E_+, u \in e\} \cup \{e \in E_+ \mid u \notin e\}$ (resp. $\{e \cup \{u'\} \mid e \in E_-, u \in e\} \cup \{e \in E_- \mid u \notin e\}$). We will use $\mathsf{clone}(e, u)$ as a shorthand for $e \cup \{u'\}$ if $u \in e$ and for e otherwise.

It is easy to see that $\mathcal{H}(\widetilde{Q}^b)$ can be obtained from $\mathcal{H}(Q)$ by iteratively cloning $(b-1)$ times each variable x of Q. Proving Theorem 5.22 then boils down to proving that cloning a vertex u does not change the width of a hypergraph as long as the copy u' of u is removed right after u.

In other words, we will show that $\mathsf{show}(\mathcal{H}, \prec) = \mathsf{show}(\mathcal{H}', \prec_u)$ and $\mathsf{sfhow}(\mathcal{H}, \prec) = \mathsf{sfhow}(\mathcal{H}', \prec_u)$ where $\mathcal{H}' = \mathsf{clone}(H, u)$ and $\prec_u$ is the order obtained from $\prec$ by inserting u' after u.

The first thing needed for the proof is to observe that cloning and removing vertices commute in the following way:

▶ **Lemma 5.23**
Let $\mathcal{H} = (V, E)$ be a hypergraph and $u \in V$. For every $v \neq u$, $\mathsf{clone}(H/v, u) = \mathsf{clone}(H, u)/v$.

Proof. Since $u \neq v$, $\mathsf{clone}(\mathcal{H}/v, u)$ and $\mathsf{clone}(H, u)/v$ have the same vertex set. It is then enough to show that they have the same edges.

Let f be an edge of $\mathsf{clone}(\mathcal{H}/v, u)$. By definition, either $f = \mathsf{clone}(e \setminus \{v\}, u)$ for e an edge of $\mathcal{H}$ or $f = \mathsf{clone}(\mathcal{N}^*_{\mathcal{H}}(v), u)$. In the first case, $f = \mathsf{clone}(e, u) \setminus \{v\}$ since $u \neq v$ and so it is an edge of $\mathsf{clone}(\mathcal{H}, u)/v$. In the second case, we claim that $f = \mathcal{N}^*_{\mathsf{clone}(H,u)}(v)$, and hence f is also an edge of $\mathsf{clone}(H, u)/v$.

Indeed, we first show $f \subseteq \mathcal{N}^*_{\mathsf{clone}(H,u)}(v)$. Let $w \in f$ and u' be the vertex added in $\mathsf{clone}(H, u)$. If $w \neq u'$, it means that $w \in \mathcal{N}^*_{\mathcal{H}}(v)$. Hence there is an edge $e' \in E$ such that $\{w, v\} \subseteq e'$. In this case, w is in $\mathsf{clone}(e', u)$. In particular, $w \in \mathcal{N}^*_{\mathsf{clone}(H,u)}(v)$. Now if $w = u'$, then it can only happen if $u \in f$, that is, $u \in \mathcal{N}^*_{\mathcal{H}}(v)$. In this case, there is $e' \in E$ such that $\{u, v\} \subseteq e'$ and in particular $\{v, u'\} \subseteq \mathsf{clone}(e', u)$. It means that $u' \in \mathcal{N}^*_{\mathsf{clone}(H,u)}(v)$.

We now show $\mathcal{N}^*_{\mathsf{clone}(H,u)}(v) \subseteq f$. Let $w \in \mathcal{N}^*_{\mathsf{clone}(H,u)}(v)$ and u' be the vertex added to

$\mathsf{clone}(H,u)$. As before, if $w \neq u'$, then $w \in \mathcal{N}^*_{\mathcal{H}}(v)$ and $w \in \mathsf{clone}(\mathcal{N}^*_{\mathcal{H}}(v),u)$. If $w = u'$, then u must be in $\mathcal{N}^*_{\mathsf{clone}(H,u)}(v)$, but then it means it is in $\mathcal{N}^*_{\mathcal{H}}(v)$ too. Hence $u' \in \mathsf{clone}(\mathcal{N}^*_{\mathcal{H}}(v),u)$.

It remains to show that every edge f of $\mathsf{clone}(H,u)/v$ is also an edge of $\mathsf{clone}(\mathcal{H}/v,u)$. This is completely symmetrical to the proof above. Indeed, either $f = \mathsf{clone}(e,u) \setminus \{v\}$ for some e and since $u \neq v$, we have $f = \mathsf{clone}(e \setminus \{v\},u)$ and it proves that f is an edge of $\mathsf{clone}(\mathcal{H}/v,u)$, or we have $f = \mathcal{N}^*_{\mathsf{clone}(H,u)}(v)$ but we just proved that $f = \mathsf{clone}(\mathcal{N}^*_{\mathcal{H}}(v),u)$ which is an edge of $\mathsf{clone}(\mathcal{H}/v,u)$. □

We can now address the missing case from Lemma 5.23, where $v = u$:

▶ **Lemma 5.24**
Let $\mathcal{H} = (V,E)$ be a hypergraph and $u \in V$. Let $\mathcal{H}' = \mathsf{clone}(H,u)/u$. Then $\mathcal{H}/u = \mathcal{H}' \setminus \{u'\}$ and $\mathcal{N}^*_{\mathcal{H}'}(u') = \mathcal{N}^*_{\mathcal{H}}(u)$.

Proof. Let f be an edge of $\mathcal{H}'$. We show that $f \setminus \{u'\}$ is an edge of $\mathcal{H}/u$. By definition, f is either of the form $\mathsf{clone}(e,u)\setminus\{u\}$ for e an edge of $\mathcal{H}$ or $\mathcal{N}^*_{\mathsf{clone}(H,u)}(u)$. In the first case, $f\setminus\{u'\} = e\setminus\{u\}$ which is an edge of $\mathcal{H}/u$. In the second case, we clearly have $f = \mathcal{N}^*_{\mathsf{clone}(H,u)}(u) = \mathcal{N}^*_{\mathcal{H}}(u) \cup \{u'\}$. Hence $f \setminus \{u'\} = \mathcal{N}^*_{\mathcal{H}}(u)$ is an edge of $\mathcal{H}/u$.

Similarly, let f be an edge of $\mathcal{H}/u$. We show that either f or $f \cup \{u'\}$ is an edge of $\mathcal{H}'$. By definition, f is either equal to $e \setminus \{u\}$ for some edge e of $\mathcal{H}$ or $f = \mathcal{N}^*_{\mathcal{H}}(u)$. In the first case, there are two sub-cases depending on whether $u \notin e$ or $u \in e$.

Assume $u \notin e$ first. Then $f = e$ is also an edge of $\mathsf{clone}(H,u)$, and hence of $\mathsf{clone}(H,u)/u = \mathcal{H}'$. Now if $u \in e$, then $\mathsf{clone}(e,u) = e \cup \{u'\}$ is an edge of $\mathsf{clone}(H,u)$ and then $(e \setminus \{u\}) \cup \{u'\} = f \cup \{u'\}$ is an edge of $\mathcal{H}'$.

In the second case, we have $f = \mathcal{N}^*_{\mathcal{H}}(u)$. But then, as before $\mathcal{N}^*_{\mathsf{clone}(H,u)}(u) = f \cup \{u'\}$. Hence $f \cup \{u'\}$ is an edge of $\mathsf{clone}(H,u)/u = \mathcal{H}'$.

From this analysis, observe that u' in $\mathcal{H}'$ only appears in $\mathcal{N}^*_{\mathsf{clone}(H,u)}(u)$ or in edge f of the form $(e \setminus \{u\}) \cup \{u'\}$ where e is an edge of $\mathcal{H}$ such that $u \in e$. Therefore, $\mathcal{N}^*_{\mathcal{H}'}(u') = \mathcal{N}^*_{\mathcal{H}}(u)$. □

We are now ready to prove the following lemma, which implies Theorem 5.22:

▶ **Lemma 5.25**
Let $\mathcal{H} = (V,E)$ be a hypergraph, $\prec$ an order on V and $u \in V$. Let u' be the clone of u in $\mathcal{H}' = \mathsf{clone}(H,u)$ and $\prec_u$ be the order on $V \cup \{u'\}$ where u' is put right after u. Then $\mathsf{how}(\mathcal{H},\prec) = \mathsf{how}(\mathcal{H}',\prec_u)$ and $\mathsf{fhow}(\mathcal{H},\prec) = \mathsf{fhow}(\mathcal{H}',\prec_u)$.

Proof. We write V as $u_1,\dots,u_n$, with $u_1 \prec \cdots \prec u_n$. Assume $u = u_j$ and let $\mathcal{H}_0 = \mathcal{H}$ and $\mathcal{H}_i = \mathcal{H}/u_1/\dots/u_i$. Moreover, let $\mathcal{H}'_0 = \mathcal{H}'$ and for $i < j$, let $\mathcal{H}'_i = \mathcal{H}'/u_1/\dots/u_i$. For $i \geqslant j$, let $\mathcal{H}'_i = \mathcal{H}'/u_1/\dots/u_j/u'/u_{j+1}/\dots/u_i$. By Lemma 5.23, for $i < j$, $\mathcal{H}'_i = \mathsf{clone}(\mathcal{H}_i,u_j)$. Hence, it directly follows that $\mathcal{N}_{\mathcal{H}_i}(u_{i+1}) = \mathcal{N}_{\mathcal{H}'_i}(u_{i+1}) \setminus \{u'\}$ and that $\mathsf{cn}(\mathcal{N}_{\mathcal{H}_i}(u_{i+1}),\mathcal{H}) = \mathsf{cn}(\mathcal{N}_{\mathcal{H}'_i}(u_{i+1}),\mathcal{H}')$ and $\mathsf{fcn}(\mathcal{N}_{\mathcal{H}_i}(u_{i+1}),\mathcal{H}) = \mathsf{fcn}(\mathcal{N}_{\mathcal{H}'_i}(u_{i+1}),\mathcal{H}')$.

Up to $i = j$, removing u_i from $\mathcal{H}_i$ or $\mathcal{H}'_i$ results in the same effect on the width. Now, consider the case where u' is removed from $\mathcal{H}'_j$. By Lemma 5.24, the neighbourhood of u' in

$\mathcal{H}'_j$ is the same as the neighbourhood of u_j in $\mathcal{H}_{j-1}$. Thus it can be covered with the same (fractional) number of edges and hence removing u' from $\mathcal{H}'_j$ does not increase the width of the order.

Finally, we observe that by Lemma 5.24 once more, $\mathcal{H}'_j/u' = \mathcal{H}_j$. Indeed, since $\mathcal{N}^*_{\mathcal{H}'_j}(u') = \mathcal{N}^*_{\mathcal{H}_{j-1}}(u_j)$, $\mathcal{H}'_j/u'$ does not introduce any new edge and then $\mathcal{H}'_j/u' = \mathcal{H}'_j \setminus \{u'\}$ which is equal to $\mathcal{H}_{j-1}/u_j = \mathcal{H}_j$ by Lemma 5.24. Thus, by a direct induction, $\mathcal{H}'_i = \mathcal{H}_i$ for $i > j$.

This concludes our proof that both elimination orders have the same width. □

The last remaining task is to generalise Lemma 5.25 to signed hyperorder width:

▶ **Lemma 5.26**
Let $\mathcal{H} = (V, E_+ \cup E_-)$ be a signed hypergraph, $\prec$ an order on V and $u \in V$. Let u' be the clone of u in $\mathcal{H}' = \mathsf{clone}(H, u)$ and $\prec_u$ be the order on $V \cup \{u'\}$ where u' is put right after u. Then $\mathsf{show}(\mathcal{H}, \prec) = \mathsf{show}(\mathcal{H}', \prec_u)$ and $\mathsf{sfhow}(\mathcal{H}, \prec) = \mathsf{sfhow}(\mathcal{H}', \prec_u)$.

Proof. Let $\mathcal{H}_0 \subseteq^- \mathcal{H}$ be a negative subhypergraph of $\mathcal{H}$ and let $\mathcal{H}'_0 = \mathsf{clone}(\mathcal{H}_0, u)$. It is easy to see that $\mathcal{H}'_0$ is a negative subhypergraph of $\mathcal{H}'$. By Lemma 5.25, $\mathsf{how}(\mathcal{H}_0, \prec) = \mathsf{how}(\mathcal{H}'_0, \prec_u)$ and $\mathsf{fhow}(\mathcal{H}_0, \prec) = \mathsf{fhow}(\mathcal{H}'_0, \prec_u)$. Therefore, $\mathsf{show}(\mathcal{H}, \prec) \leqslant \mathsf{show}(\mathcal{H}', \prec_u)$ and $\mathsf{sfhow}(\mathcal{H}, \prec) \leqslant \mathsf{sfhow}(\mathcal{H}', \prec_u)$.

Similarly, for a negative subhypergraph $\mathcal{H}'_0$ of $\mathcal{H}'$, there is a negative subhypergraph $\mathcal{H}_0$ of $\mathcal{H}$ such that $\mathcal{H}'_0 = \mathsf{clone}(\mathcal{H}_0, u)$. In this case, we similarly get $\mathsf{show}(\mathcal{H}, \prec) \geqslant \mathsf{show}(\mathcal{H}', \prec_u)$ and $\mathsf{sfhow}(\mathcal{H}, \prec) \geqslant \mathsf{sfhow}(\mathcal{H}', \prec_u)$, which concludes the proof. □

Theorem 5.22 is a direct consequence of Lemma 5.26 since $\mathcal{H}(\widetilde{Q}^b)$ is obtained by cloning each vertex of $\mathcal{H}(Q)$ b times and the copies x^i of vertex x are consecutive in $\prec^b$.

▶ **Remark 5.27**
Consider once again the instance

$$Q :\text{-}\ \mathsf{A}(x_1),\ \mathsf{B}(x_2),\ \neg\mathsf{R}(x_1, x_2)$$

from Example 5.21 where we proved that caching would never occur with this instance, resulting in a circuit of size at least $\tilde{\mathcal{O}}(d^2)$.

Running DPLL on $\widetilde{Q}^b$ however, will result in cache hits now happening often. Indeed, assume that the algorithm is in a state where every copy $x_1^b, \ldots, x_1^1$ of x_1 was set already to values $v_b, \ldots, v_1$ and the algorithm is now setting variables $x_2^b, \ldots, x_2^1$. Observe that as soon as some copy x_2^i is set to the value $1 - v_i$, then $\neg\widetilde{\mathsf{R}}^b$ is satisfied since the value assigned to x_1 is necessarily different from the value assigned to x_2. Thus, the atom $\neg\widetilde{\mathsf{R}}^b$ is now simplified away and caching can occur. Each subset of $\widetilde{\mathsf{B}}^b$ where some prefix of $x_2^b, \ldots, x_2^1$ will be cached, which boils down to $d \cdot \log d$ values, which is less than the $\tilde{\mathcal{O}}(d^2)$ from Example 5.21.

We conclude this section by stating how one can use binarisation as a means of improving the construction of ordered $\{\times, \mathsf{dec}\}$-circuits representing answers of signed conjunctive queries.

▶ **Theorem 5.28**
Let Q be a signed join query, $\mathbf{D}$ a database over domain D and $\prec$ an order on $\mathsf{var}(Q)$. Then $\mathsf{DPLL}(\widetilde{Q}^b, \langle\rangle, \widetilde{\mathbf{D}}^b, \prec_b)$ produces a $\succ_b$-ordered $\{\times, \mathsf{dec}\}$-circuit C of size $\tilde{\mathcal{O}}((\mathsf{poly}_k|Q|) \cdot |\mathbf{D}|^k)$ on domain $\{0,1\}$ such that $\mathsf{rel}(C) = \mathsf{ans}_{\mathbf{D}}(\widetilde{Q}^b)$ and:

- $k = \mathsf{fhow}(\mathcal{H}(Q), \prec)$ if Q is positive,
- $k = \mathsf{show}(\mathcal{H}(Q), \prec)$ otherwise.

Moreover, the runtime of this construction is at most $\tilde{\mathcal{O}}(\mathsf{poly}_k(|Q|) \cdot |\mathbf{D}|^k)$.

Proof. This is simply Theorem 5.13 applied to $\widetilde{Q}^b$ with $|\mathsf{D}| = 2$ together with the fact that $\mathsf{sfhow}(\mathcal{H}(\widetilde{Q}^b), \prec_b) = \mathsf{sfhow}(\mathcal{H}(Q), \prec)$ and $\mathsf{fhow}(\mathcal{H}(\widetilde{Q}^b), \prec_b) = \mathsf{fhow}(\mathcal{H}(Q), \prec)$ from Theorem 5.22. □

5.6 Conclusion

In this chapter, we have seen join query evaluation as a *compilation* task. Instead of enumerating answers naively, we build on the work we introduced in Chapter 3 and build a data structure that represents the answer set of the query. This new data structure is a relational circuit that leverages factorisation of gates and independent subquery evaluation to be more compact. We then extended this construction to queries with negative atoms. This extension of queries brings more challenges, but also new possibilities for decomposition and simplification.

The main result of this chapter is that it is possible to compile a (signed) join query over a database instance into an ordered $\{\times, \mathsf{dec}\}$-circuit representing the answer set of the query. The complexity of this compilation is formalised in Theorems 5.20 and 5.28. The two most important elements in the complexity analysis are the (signed) fractional hyperorder width of the order we consider over the variables and the size of the domain. In Section 5.5, we show that we can use the same method as in Chapter 3 to reduce the size of the domain to a manageable constant (in our case, 2 by binarisation).

In Chapter 6, we take advantage of the structure of ordered $\{\times, \mathsf{dec}\}$-circuits to efficiently answer direct access tasks for signed join queries.

Current chapter references

[Bak+13] Nurzhan **Bakibayev**, Tomáš **Kočiský**, Dan **Olteanu**, and Jakub **Závodný**. *Aggregation and Ordering in Factorised Databases*. In *Proceedings of the VLDB Endowment* 6.14 (Sept. 2013), pp. 1990–2001. doi 10.14778/2556549.2556579.

[Cap+25] Florent **Capelli**, Nofar **Carmeli**, Oliver **Irwin**, and Sylvain **Salvati**. *Direct Access for Conjunctive Queries with Negations*. Oct. 2025. arXiv 2310.15800.

[Cap16] Florent **Capelli**. *Structural restrictions of CNF-formulas: applications to model counting and knowledge compilation*. PhD thesis. Université Paris Diderot, Sorbonne Paris Cité, June 2016. https://florent.capelli.me/publi/these_capelli.pdf.

[Cap17] Florent **Capelli**. *Understanding the complexity of #SAT using knowledge compilation*. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, June 2017, pp. 1–10. doi 10.1109/LICS.2017.8005121.

[CI24] Florent **Capelli** and Oliver **Irwin**. *Direct Access for Conjunctive Queries with Negations*. In *27th International Conference on Database Theory, ICDT 2024, March 24 to March 28, 2024, Paestum, Italy*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2024, 13:1–13:20. doi 10.4230/LIPICS.ICDT.2024.13.

[CIS25] Florent **Capelli**, Oliver **Irwin**, and Sylvain **Salvati**. *A Simple Algorithm for Worst Case Optimal Join and Sampling*. In *28th International Conference on Database Theory (ICDT 2025), March 25 to March 28, 2025, Barcelona, Spain* (ICDT '25). Ed. by Sudeepa **Roy** and Ahmet **Kara**. Vol. 328. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2025, 23:1–23:19. doi 10.4230/LIPIcs.ICDT.2025.23.

[CM77] Ashok K. **Chandra** and Philip M. **Merlin**. *Optimal Implementation of Conjunctive Queries in Relational Data Bases*. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, May 1977, pp. 77–90. doi 10.1145/800105.803397.

[DM02] A. **Darwiche** and P. **Marquis**. *A Knowledge Compilation Map*. In *Journal of Artificial Intelligence Research* 17 (Sept. 2002), pp. 229–264. doi 10.1613/jair.989.

[HD05] Jinbo **Huang** and Adnan **Darwiche**. *DPLL with a trace: from SAT to knowledge compilation*. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. IJCAI'05. Edinburgh, Scotland: Morgan Kaufmann Publishers Inc., July 2005, pp. 156–162. doi 10.5555/1642293.1642318.

[Olt16] Dan **Olteanu**. *Factorized Databases: A Knowledge Compilation Perspective*. In *AAAI Workshop: Beyond NP*. Feb. 2016.

[OZ12] Dan **Olteanu** and Jakub **Závodnỳ**. *Factorised Representations of Query Results: Size Bounds and Readability*. In *Proceedings of the 15th International Conference on Database Theory*. ACM. Mar. 2012, pp. 285–298. doi 10.1145/2274576.2274607.

[OZ15] Dan **Olteanu** and Jakub **Závodný**. *Size Bounds for Factorised Representations of Query Results*. en. In *ACM Transactions on Database Systems* 40.1 (Mar. 2015), pp. 1–44. doi 10.1145/2656335.

[San+04] Tian **Sang**, Fahiem **Bacchus**, Paul **Beame**, Henry A **Kautz**, and Toniann **Pitassi**. *Combining Component Caching and Clause Learning for Effective Model Counting*. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*. May 2004. http://www.satisfiability.org/SAT04/programme/21.pdf.

[Vel14] Todd L. **Veldhuizen**. *Triejoin: A Simple, Worst-Case Optimal Join Algorithm*. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. Ed. by Nicole **Schweikardt**, Vassilis **Christophides**, and Vincent **Leroy**. OpenProceedings.org, Mar. 2014, pp. 96–106. doi 10.5441/002/icdt.2014.13.

[Zan25] Bruno **Zanuttini**. *Introduction à la compilation de connaissances*. In *Informatique fondamentale et ses Mathématiques : Une photographie en 2025*. Ed. by Pascal **Vanier**. CNRS Alpha. CNRS Editions, July 2025, pp. 255–302.

[Zha+24] Hangdong **Zhao**, Austen Z. **Fan**, Xiating **Ouyang**, and Paraschos **Koutris**. *Conjunctive Queries with Negation and Aggregation: A Linear Time Characterization*. In *Proc. ACM Manag. Data* 2.2 (May 2024). doi 10.1145/3651138.

Chapter 6

Direct Access for Conjunctive Queries with Negations

To be, or not to be: that is the question...

Hamlet

Given a conjunctive query Q and a database $\mathbf{D}$, direct access to the answers of Q over $\mathbf{D}$ is the operation of returning, given an index k, the k^{th} answer for some order on its answers. While this problem is #P-hard in general with respect to combined complexity, many conjunctive queries have an underlying structure that allows for a direct access to their answers for some lexicographical ordering. This direct access takes polylogarithmic time in the size of the database after a polynomial preprocessing time. Previous work has precisely characterised the tractable classes and given fine-grained lower bounds on the preprocessing time needed depending on the structure of the query.

In this chapter, we generalise these tractability results to the case of signed conjunctive queries, that is, conjunctive queries that may contain *negative* atoms. Our technique is based on ordered $\{\times, \mathsf{dec}\}$-circuits, which we introduced in Chapter 5 as a class of circuits that can represent relational data. This chapter is based on the methods first presented in [CI24; Cap+25].

We will start by reducing the problem of direct access for signed join queries to the problem of direct access for *positive* join queries. This will allow us to use existing results from the literature, and notably existing complexity bounds. However, this algorithm, if optimal in data complexity, has an exponential complexity in the size of the query. We subsequently offer a resolution to this problem by using the properties of ordered $\{\times, \mathsf{dec}\}$-circuits. We will use the bounds on the size of the circuit needed to represent the answer set of signed conjunctive queries from Chapter 5, that depend on their structure. Both results combined together allow us to prove the tractability of direct access for a large class of conjunctive queries. We will also use these results to show that both the class of β-acyclic negative conjunctive queries and the class of bounded *nest set* width negative conjunctive queries admit tractable direct access.

Outline of the current chapter

6.1 Setting the Stage

Before we dive into the main contributions of this chapter, we recall some useful notations from Chapter 1 and introduce some new concepts that will prove important later on.

In our model of computation, it is known that *radix sort* allows us to sort m value whose total size is bounded by n in time $\mathcal{O}(m+n)$ [Cor+22]. This fact will allow us to sort the relations of a database $\mathbf{D}$ in time $\mathcal{O}(|\mathbf{D}|)$, which will prove to be very useful in the algorithms we will present here. Indeed, we will rely on the relations being sorted to efficiently count the sizes of the relations restricted to certain variable assignments.

In Chapter 1, we defined a relation R on domain D as a finite set of tuples of same arity. More formally, in the *named* perspective, given a variable set X, a relation is a set of tuples over these variables $\mathsf{R} \subseteq \mathsf{D}^X$. We also defined a *filtering* operation on relations, denoted by $\sigma_F(\mathsf{R})$, that represents the subset of R where a logical formula F is true. We assume that F can be build using a conjunction of atomic formulas of the form $x = d, x < d, x \leqslant d, x > d, x \geqslant d$, for a variable $x \in X$ and a domain value $d \in \mathsf{D}$. For example, $\sigma_{x \leqslant d}(\mathsf{R})$ contains all the tuples $\tau \in \mathsf{R}$ such that $\tau(x) \leqslant d$.

Of course, this does not make sense if we do not have an order on the answers of the query. Throughout this chapter, we will assume that both the domain D and the variables set X are totally ordered. We denote the order on the domain values by $<$ and the order on the variables by $\prec$. To ease notations, we will also often write that $\mathsf{D} = \{d_1, \dots, d_p\}$ with $d_1 < \cdots < d_p$ and $X = \{x_1, \dots, x_n\}$ with $x_1 \prec \cdots \prec x_n$. For a subset of variables $Y \subseteq X$, we denote by $\mathsf{min}_{\prec}(Y)$ (or $\mathsf{min}(Y)$ if the order is clear from the context) the minimal element of Y with respect to $\prec$. We will also often refer to the lexicographical order $\prec_{\mathsf{lex}}$ induced by $<$ and $\prec$. To ease the notations, for an integer $k \leqslant \#\mathsf{R}$, we will denote by $\mathsf{R}[k]$ the k^{th} tuple in R with respect to the order $\prec_{\mathsf{lex}}$.

6.1.1 Assigning a Value in τ

One of the core elements of our direct access method will be way in which we choose a value to assign to each variable in a tuple τ. Lemma 6.1 formalises an observation that will prove to be useful for the rest of this chapter:

▶ **Lemma 6.1**
Let $\tau = \mathsf{R}[k]$ and let $x = \mathsf{min}(\mathsf{var}(\mathsf{R}))$. Then the following holds:

$$\tau(x) = \mathsf{min}\{d \mid \#\sigma_{x \leqslant d}(\mathsf{R}) \geqslant k\} \ .$$

Moreover, $\tau = \mathsf{R}'[k']$, where $\mathsf{R}' = \sigma_{x=d}(\mathsf{R})$ is the subset of R where $x = d$ and $k' = k - \#\sigma_{x<d}(\mathsf{R})$.

Before formally proving this statement, we propose a visual representation in Figure 6.2.
We can now proceed and formally prove Lemma 6.1.

Proof. Let $A = \{d \mid \#\sigma_{x \leqslant d}(\mathsf{R}) \geqslant k\}$.

We start by showing that $\tau(x) \in A$, meaning $\#\sigma_{x \leqslant \tau(x)} \geqslant k$. Let $\alpha \in \mathsf{R}$ such that $\alpha \preceq_{\mathsf{lex}} \tau$. Since x is the smallest variable, it follows that $\alpha \in \sigma_{x \leqslant \tau(x)}(\mathsf{R})$ as $\alpha(x) \leqslant \tau(x)$. Moreover, as there exists exactly k such assignments α (by definition of τ which is the k^{th} tuple of R), we have $\#\sigma_{x \leqslant \tau(x)}(\mathsf{R}) \geqslant k$.

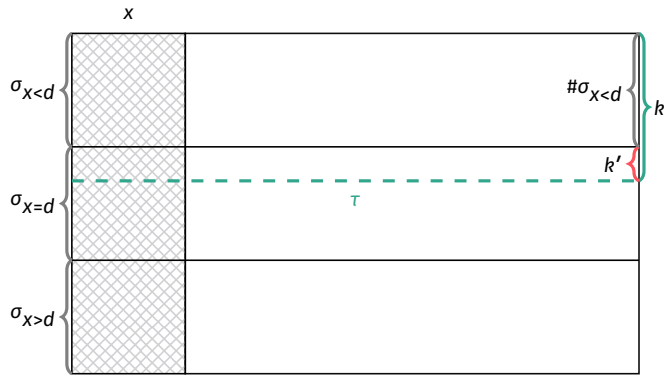


Figure 6.2: Visual representation of Lemma 6.1.

The rows represent tuples in increasing order and the columns represent variables. On the left side, we depict the subsets of the relation for different values of x. On the right side, we depict the ranks of the tuples in the relation.

The value on x of the k^{th} tuple in R, depicted by a coloured dashed line, is the smallest value $d \in \mathsf{D}$ such that R contains more than k tuples with a value smaller or equal to d. For any smaller value of d, we would be below the dashed line, as there would not be enough tuples. The value of k', which is the rank of the tuple in the subrelation $\sigma_{x=d}(\mathsf{R})$, is computed by subtracting from k the size of $\sigma_{x<d}(\mathsf{R})$.

We now show that, given a value $d' < \tau(x)$, we have that $d' \notin A$ and therefore $\tau(x)$ is indeed the smallest value in A. Let $\alpha \in \sigma_{x \leqslant d'}(\mathsf{R})$. It follows that $\alpha(x) < \tau(x)$, and therefore that $\alpha \prec_{\mathsf{lex}} \tau$. We therefore have that $\sigma_{x \leqslant d'}(\mathsf{R}) \subset \{\alpha \mid \alpha \prec_{\mathsf{lex}} \tau\}$. By definition of τ as the k^{th} tuple, the latter set has less than $k-1$ elements. Hence $d' \notin A$. This shows that $\tau(x)$ is indeed the smallest value d such that there exists at least k tuples α where $\alpha(x) \leqslant d$.

The second part of the lemma follows from the following observation: when assigning a value d to the variable x, one actually *removes* a certain number of tuples from the initial set. Specifically, the tuples that assign a different value to x.

By definition, k is the cardinal of the set $\{\tau' \mid \tau' \preceq_{\mathsf{lex}} \tau\}$. This set can also be written as the disjoint union of the set of tuples where $\tau'(x) < d$ (which are all smaller than τ) and the set of tuples smaller than τ where $\tau'(x) = d$. We therefore have $k = \#\{\tau \mid \tau(x) < d\} + \#\{\tau' \mid \tau' \preceq_{\mathsf{lex}} \tau, \tau'(x) = d\}$. By definition, the first set is $\sigma_{x<d}(\mathsf{R})$. The second part of the sum is exactly the index of the tuple in the subset of R where $\tau(x) = d$. We can rewrite the sum as $k = \#\sigma_{x<d}(\mathsf{R}) + k'$, implying $k' = k - \#\sigma_{x<d}(\mathsf{R})$. □

6.1.2 Direct Access and Ranking

We introduce the problem of ranking as a problem that can be seen as the *reverse* of direct access.

▶ **Definition 6.3** (Ranking)
Given an ordered set of tuples S and a tuple τ, *ranking* is the problem of finding the *rank* of τ in S, that is, the number of tuples of S that are smaller or equal to τ.

If τ is not in S, then we return the rank of the largest element of S that is smaller than τ.

If τ is smaller than the first element of S, then we return 0.

Definition 6.3 allows us to formulate the following property:

▶ **Lemma 6.4** (Subtraction Lemma)
Let S be a set ordered by $\prec$ and let S_1, S_2 be two subsets of S such that $S_1 \subseteq S_2$. Assume that, for any value k, we can output the k^{th} element of S_1 (respectively, of S_2) for the order $\prec$ in time t_1 (respectively, in time t_2).

Then, given any k, we can output the k^{th} element of $S_2 \setminus S_1$ for the order $\prec$ in time $C \cdot (t_2 + t_1 \cdot \log|S_1|) \cdot \log|S_2|$, for some constant C.

Proof. We observe that having direct access for a set of tuples S in time t implies having ranking on S in time $\mathcal{O}(\log|S| \cdot t)$. This holds since we can simply do a binary search on S to find the correct rank for any given tuple.

Next, we claim that, given an element τ of rank r_2 in S_2, we can find its rank in the subtraction of the sets $S_2 \setminus S_1$ in time $\mathcal{O}(\log|S_1| \cdot t_1)$.

We use ranking for τ to find its rank r_1 in S_1. Then, because we have $S_1 \subseteq S_2$, we deduce that the rank of τ in $S_2 \setminus S_1$ is $r_2 - r_1$. Finally, we can simply do a binary search over the ranks of S_2 to compute the value of r_2.

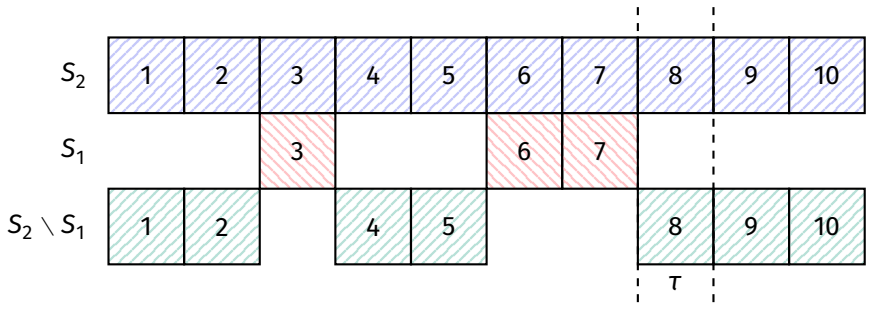
For each rank r_2, we access its element τ in time t_2, and check its ranking in $S_2 \setminus S_1$ as described above. This leads to direct access for $S_2 \setminus S_1$ in time $\mathcal{O}((t_2 + t_1 \cdot \log|S_1|) \cdot \log|S_2|)$. □

▶ **Example 6.5** (Subtraction via ranking and binary search)
We illustrate Lemma 6.4 on a small ordered universe. Let

$$S = \{1, 2, \ldots, 10\}, \qquad S_2 = S, \qquad S_1 = \{3, 6, 7\},$$

ordered by the natural order $<$ and represented in the following illustration. Clearly, $S_1 \subseteq S_2$, and

$$S_2 \setminus S_1 = \{1, 2, 4, 5, 8, 9, 10\}.$$



We assume that we have direct access to both sets, that is, for any k, we can return the k^{th}

element of S_1 (respectively S_2) in time t_1 (respectively t_2).

As observed in the proof, having direct access to the elements of a set S_i implies having the ability to compute the rank of any element $\tau \in S_i$ by binary search in time $\mathcal{O}(\log|S_i| \cdot t_i)$. In particular, given $\tau \in S_2$, we can compute its rank r_1 in S_1 in time $\mathcal{O}(\log|S_1| \cdot t_1)$.

▶ **Rank in the difference.** Let $\tau \in S_2$ have rank r_2 in S_2, and rank r_1 in S_1. Since $S_1 \subseteq S_2$, exactly r_1 elements of S_1 appear before τ in the order $<$. The rank of τ in $S_2 \setminus S_1$ is then $r_2 - r_1$.

For example, for $\tau = 8$, we have $r_2 = 8$ in S_2 and $r_1 = 3$ in S_1 (since $3, 6, 7 < 8$). Therefore, τ has rank $8 - 3 = 5$ in $S_2 \setminus S_1$.

▶ **Binary search over S_2.** To compute the k^{th} element of $S_2 \setminus S_1$, we perform a binary search over ranks $r_2 \in \{1, \ldots, |S_2|\}$. For each candidate rank r_2, we:

1. access $\tau = S_2[r_2]$ in time t_2;
2. compute its rank r_1 in S_1 in time $\mathcal{O}(\log|S_1| \cdot t_1)$; and
3. compare $r_2 - r_1$ with k.

The smallest r_2 such that $r_2 - r_1 \geqslant k$ yields the desired element.
Suppose we want the 5^{th} element of $S_2 \setminus S_1$. The search could be illustrated as follows:

Step 1	**Step 2**	**Step 3**	**Step 4**
$r_2 = 6$	$r_2 = 9$	$r_2 = 8$	$r_2 = 7$
$r_1 = 2$	$r_1 = 3$	$r_1 = 3$	$r_1 = 3$
$r_2 - r_1 = 4$	$r_2 - r_1 = 6$	$r_2 - r_1 = 5$	$r_2 - r_1 = 4$
✗	✗	✓	✗

While, at Step 2, the search has already found an index which is greater than 5, we have to check that it is the smallest such index. At the end of the search, we have that the 5^{th} element of the difference is the 8^{th} element of S_2.

▶ **Complexity.** The outer binary search performs $\mathcal{O}(\log|S_2|)$ iterations, each iteration costing $t_2 + \mathcal{O}(\log|S_1| \cdot t_1)$. This matches the bound stated in Lemma 6.4.

The following lemma will also be useful in the rest of this chapter. A similar claim can be found in [BCM22a]:

▶ **Lemma 6.6**
Assume that we are given a relation $\mathsf{R} \subseteq \mathsf{D}^X$ with $X = \{x_1, \ldots, x_n\}$ and an oracle $\mathcal{A}$ such that for every prefix assignment $\tau \in \mathsf{D}^{\{x_1,\ldots,x_p\}}$ and $N \in \mathbb{N}$, it returns the smallest value $d \in \mathsf{D}$ such that $\#\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{R})) \geqslant N$ and the value $\#\sigma_{x_{p+1} < d}(\sigma_\tau(\mathsf{R}))$.
Then, for any $k \leqslant |\mathsf{R}|$, we can compute $\mathsf{R}[k]$ using $\mathcal{O}(|X|)$ calls to oracle $\mathcal{A}$.

Proof. Let $\alpha = \mathsf{R}[k]$ be the answer we are looking for. We iteratively find the value of α on X by applying Lemma 6.1. More precisely, assume that we already know $\alpha(x_1), \ldots, \alpha(x_p)$ for some $p < n$. Moreover, we know k_p such that $\alpha = \sigma_\beta(\mathsf{R})[k_p]$ where β is the prefix assignment of $\{x_1, \ldots, x_p\}$ such that $\beta(x_i) = \alpha(x_i)$. We know by Lemma 6.1 that $\alpha(x_{p+1}) = \mathsf{min}\{d \mid \#\sigma_{x_{p+1} \leqslant d}(\sigma_\beta(\mathsf{R})) \geqslant k_p\}$ and $k_{p+1} = k_p - \#\sigma_{x_{p+1} < d}(\sigma_\beta(\mathsf{R}))$, which we can compute with *one* call to oracle $\mathcal{A}$. Therefore, by induction, we can fully reconstruct α with $\mathcal{O}(|X|)$ calls to oracle $\mathcal{A}$. □

6.2 An Optimal, yet Inefficient Algorithm

In this section, we reduce between the problem of answering direct access tasks on join queries with negative atoms and the problem of answering direct access tasks on positive join queries (*without* negative atoms). This reduction will allow us to directly use results from [BCM22a] and get, for any given Q, an algorithm to answer direct access tasks with optimal data complexity. Both the algorithm and its optimality will directly follow from the algorithm and the lower bound of [BCM22a]. However, this algorithm has exponential complexity in the size of the query.

The rest of the chapter will be devoted to the presentation of a both direct and optimal algorithm with better combined complexity.

6.2.1 Reducing from Join Queries to Positive Join Queries

We start by showing that, in terms of data complexity, and if we ignore polylogarithmic factors, direct access for a signed query has the same complexity as direct access for the *worst* positive query, obtained by considering some of the negated relations to be positive.

A signed join query Q can be seen as a join between two subqueries:

- one consisiting of all the positive atoms; and
- one consisting of all the negative atoms.

With a slight abuse of notation, we denote this by Q :- P, N. Here, P is the join of the positive atoms and N the join of the negated atoms.

Let Q be a signed join query. For a set of relations $\mathsf{N} \subseteq \mathsf{atoms}^-(Q)$, we denote by $\overline{\mathsf{N}}$ the same relations, but considered as if they were positive atoms. Given a disjoint pair $(\mathsf{N}_1, \mathsf{N}_2)$ of subsets of $\mathsf{atoms}^-(Q)$ that do not have to form a partition of $\mathsf{atoms}^-(Q)$, we denote by $Q_{\mathsf{N}_1,\mathsf{N}_2}$ the query computed by considering all the relations in N_1 as if they were positive atoms, and keeping the negated atoms of N_2 and removing all other negated atoms. That is, $Q_{\mathsf{N}_1,\mathsf{N}_2}$:- P, $\overline{\mathsf{N}_1}$, N_2.

▶ **Example 6.7**
Consider the following join query:

$$Q(a, b, x, y, z) \text{ :- } \mathsf{U}(a, b),\ \neg\mathsf{R}(x, y),\ \neg\mathsf{S}(y, z),\ \neg\mathsf{T}(z, x)$$

Suppose we define $\mathsf{N}_1 = \{\neg\mathsf{R}(x, y)\}$ and $\mathsf{N}_2 = \{\neg\mathsf{T}(z, x)\}$. Then, we have:

$$Q_{\mathsf{N}_1,\mathsf{N}_2}(a, b, x, y, z) \text{ :- } \mathsf{U}(a, b),\ \mathsf{R}(x, y),\ \neg\mathsf{T}(z, x)\ .$$

▶ **Note:** Observe that, since we did not include S in N_1 or N_2, it is simply absent from $Q_{\mathsf{N}_1,\mathsf{N}_2}$.

▶ **Remark 6.8**
This method of considering some of the negative atoms as positive matches the method we used in Section 5.4.1 to compute the width of a signed hypergraph.

Establishing our lower bound relies on the following strategy: we show that having direct access to Q is equivalent to having direct access to $Q_{\mathsf{N},\emptyset}$ for every $\mathsf{N} \subseteq \mathsf{atoms}^-(Q)$, which is a positive query, and thus for which we already have lower bounds from [BCM22a]. To do so, we actually prove that this is equivalent to having a direct access to $Q_{\mathsf{N}_1,\mathsf{N}_2}$ for every disjoint $\mathsf{N}_1, \mathsf{N}_2 \subseteq \mathsf{atoms}^-(Q)$. The proof structure we will follow is summarised in Figure 6.9.

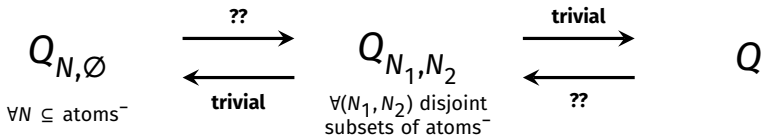


Figure 6.9: Reductions between direct access for signed queries and direct access for positive queries.

We start by showing the reduction on the left of Figure 6.9, from $Q_{\mathsf{N},\emptyset}$ to $Q_{\mathsf{N}_1,\mathsf{N}_2}$ for any disjoint pair $(\mathsf{N}_1, \mathsf{N}_2)$. This is formalised in Lemma 6.10.

This establishes the left reduction from Figure 6.9. We prove the right reduction in Lemma 6.10.

▶ **Lemma 6.10**
Let Q be a signed join query with n variables. Assume that for all $\mathsf{N} \subseteq \mathsf{atoms}^-(Q)$ we have direct access for $Q_{\mathsf{N},\emptyset}$ with preprocessing time t_p and access time t_a.

Then we have direct access to $Q_{\mathsf{N}_1,\mathsf{N}_2}$ for any disjoint pair $(\mathsf{N}_1, \mathsf{N}_2)$ of subsets of $\mathsf{atoms}^-(Q)$ with:

preprocessing time: $2^{|\mathsf{atoms}^-(Q)|} \cdot t_p$; and

access time: $(A \cdot n \cdot \log|\mathsf{D}|)^{2|\mathsf{atoms}^-(Q)|} \cdot t_a$ for some constant A.

Proof. The lemma follows by proving the following statement by induction over the size of N_2:

> For any subset $\mathsf{N}_1 \subseteq \mathsf{atoms}^-(Q)$ such that N_1 and N_2 are disjoint, we have direct access for $Q_{\mathsf{N}_1,\mathsf{N}_2}$ with preprocessing time $2^{|\mathsf{N}_2|} \cdot t_p$ and access time $(2C \cdot n \cdot \log|\mathsf{D}|)^{2|\mathsf{N}_2|} \cdot t_a$ where C is the constant from Lemma 6.4.

The base case is when $\mathsf{N}_2 = \emptyset$, and it is given by the hypothesis of the lemma statement.

Now, let us consider N_2 to be a non-empty set. Since it is not empty, we can write it as $\mathsf{N}_2 = \mathsf{N}_2' \cup \{\neg \mathsf{R}(\mathbf{x})\}$, where $\neg \mathsf{R}(\mathbf{x})$ is an arbitrarily chosen atom from N_2.

From the induction hypothesis, since $|\mathsf{N}_2'| < |\mathsf{N}_2|$, for any subset $\mathsf{N}_1' \subseteq \mathsf{atoms}^-(Q)$ such that $\mathsf{N}_1' \cap \mathsf{N}_2' = \emptyset$, we have direct access for the query $Q_{\mathsf{N}_1',\mathsf{N}_2'}$ with preprocessing time $2^{|\mathsf{N}_2|-1} \cdot t_p$ and access time $(2C \cdot n \cdot \log|\mathsf{D}|)^{2|\mathsf{N}_2|-1} \cdot t_a)$.

We can then rewrite the answer set of $Q_{\mathsf{N}_1,\mathsf{N}_2}$ as:

$$\begin{aligned}\mathsf{ans}(Q_{\mathsf{N}_1,\mathsf{N}_2}) &= \mathsf{ans}(\mathsf{P},\ \overline{\mathsf{N}_1},\ \mathsf{N}_2',\ \neg\mathsf{R}(\mathbf{x})) \\ &= \mathsf{ans}(\mathsf{P},\ \overline{\mathsf{N}_1},\ \mathsf{N}_2') \setminus \mathsf{ans}(\mathsf{P},\ \overline{\mathsf{N}_1},\ \mathsf{N}_2',\ \mathsf{R}(\mathbf{x}))) \\ &= \mathsf{ans}(Q_{\mathsf{N}_1,\mathsf{N}_2'}) \setminus \mathsf{ans}(Q_{\mathsf{N}_1\cup\{\mathsf{R}(\mathbf{x})\},\mathsf{N}_2'})\end{aligned}$$

Notice that $(\mathsf{N}_1 \cup \{\mathsf{R}(\mathbf{x})\}, \mathsf{N}_2')$ is a disjoint pair since $(\mathsf{N}_1, \mathsf{N}_2' \cup \{\mathsf{R}(\mathbf{x})\})$ is a disjoint pair, and $\mathsf{R}(\mathbf{x})$ does not appear in N_2 more than once since we consider self-join free queries. We know from the induction hypothesis that we have direct access to the answers of both $Q_{\mathsf{N}_1,\mathsf{N}_2'}$ and $Q_{\mathsf{N}_1\cup\{\mathsf{R}(\mathbf{x})\},\mathsf{N}_2'}$. Moreover, since $\mathsf{N}_1 \subset \mathsf{N}_1 \cup \{\mathsf{R}(\mathbf{x})\}$ and adding positive atoms can only restrict the answer set of a query, we have that $\mathsf{ans}_{\mathbf{D}}(Q_{\mathsf{N}_1\cup\{\mathsf{R}(\mathbf{x})\},\mathsf{N}_2'}) \subseteq \mathsf{ans}_{\mathbf{D}}(Q_{\mathsf{N}_1,\mathsf{N}_2'})$. We apply Lemma 6.4 to get direct access to the answers of $Q_{\mathsf{N}_1,\mathsf{N}_2}$. Preprocessing consists in doing the preprocessing for $Q_{\mathsf{N}_1\cup\{\mathsf{R}(\mathbf{x})\},\mathsf{N}_2'}$ and $Q_{\mathsf{N}_1,\mathsf{N}_2'}$. By induction, both of them take time $2^{|\mathsf{N}_2|-1} \cdot t_p$, hence a preprocessing time of $2^{|\mathsf{N}_2|} \cdot t_p$. For the access time, we apply Lemma 6.4 with $t_1 = t_2 = (2C \cdot n \cdot \log|\mathsf{D}|)^{2(|\mathsf{N}_2|-1)} \cdot t_a$ and bound S_1, S_2 with $|\mathsf{D}|^n$ and get:

$$2 \cdot (2C \cdot n \cdot \log|\mathsf{D}|)^{2(|\mathsf{N}_2|-1)} \cdot t_a \cdot (C \cdot n \cdot \log|\mathsf{D}|)^2 \leqslant (2C \cdot n \cdot \log|\mathsf{D}|)^{2|\mathsf{N}_2|} \cdot t_a$$

This concludes our induction step. □

We can now present the lower bound for signed join queries in Lemma 6.11.

▶ **Lemma 6.11**

Let Q be a signed join query with n variables and without self-joins. Assume we have direct access for Q with preprocessing time t_p and access time t_a.

Then, for any disjoint pair $(\mathsf{N}_1, \mathsf{N}_2)$ of $\mathsf{atoms}^-(Q)$, we have direct access to $Q_{\mathsf{N}_1,\mathsf{N}_2}$ with:

preprocessing time: $2^{|\mathsf{atoms}^-(Q)|} \cdot t_p$; and

access time: $(A \cdot n \cdot \log|\mathsf{D}|)^{2|\mathsf{atoms}^-(Q)|} \cdot t_a$ for some constant A.

Proof. We prove by induction over the size of N_1 that, for any disjoint pair $(\mathsf{N}_1, \mathsf{N}_2)$, we have direct access for $Q_{\mathsf{N}_1,\mathsf{N}_2}$ with preprocessing time $2^{|\mathsf{N}_1|} \cdot t_p$ and access time $(2Cn \cdot \log|\mathsf{D}|)^{2|\mathsf{N}_1|} \cdot t_a$ where C is the constant from Lemma 6.4.

The base case is when $\mathsf{N}_1 = \emptyset$. If $\mathsf{N}_2 = \mathsf{atoms}^-(Q)$, then since $Q_{\emptyset,\mathsf{atoms}^-(Q)} = Q$, it is given by the hypothesis of the lemma statement. If $\mathsf{N}_2 \subset \mathsf{atoms}^-(Q)$, then we can use the algorithm of $Q_{\emptyset,\mathsf{atoms}^-(Q)}$ for answering Q by setting the negated relations that are not in N_2 to be empty. Notice that we can do so freely because the query has no self-joins.

Now, let us consider N_1 to be a non-empty set. We can then denote it as $\mathsf{N}_1 = \mathsf{N}_1' \cup \{\neg\mathsf{R}\}$. By induction, we have direct access for the answers of $Q_{\mathsf{N}_1',\mathsf{N}_2}$ for every N_2 disjoint from N_1 with

preprocessing time $2^{|\mathsf{N}_1|-1} \cdot t_p$ and access time $(2C \cdot n \cdot \log|\mathsf{D}|)^{2(|\mathsf{N}_1|-1)} \cdot t_a$.

We can rewrite the answer set of $Q_{\mathsf{N}_1,\mathsf{N}_2}$ as:

$$\begin{aligned}
\mathsf{ans}(Q_{\mathsf{N}_1,\mathsf{N}_2}) &= \mathsf{ans}(\mathsf{P},\ \overline{\mathsf{N}_1},\ \mathsf{N}_2) \\
&= \mathsf{ans}(\mathsf{P},\ \overline{\mathsf{N}'_1},\ \mathsf{N}_2,\ \mathsf{R}) \\
&= \mathsf{ans}(\mathsf{P},\ \overline{\mathsf{N}'_1},\ \mathsf{N}_2) \setminus \mathsf{ans}(\mathsf{P},\ \overline{\mathsf{N}'_1},\ \mathsf{N}_2,\ \neg\mathsf{R}) \\
&= \mathsf{ans}(Q_{\mathsf{N}'_1,\mathsf{N}_2}) \setminus \mathsf{ans}(Q_{\mathsf{N}'_1,\mathsf{N}_2\cup\{\neg\mathsf{R}\}})
\end{aligned}$$

We can use the induction hypothesis on both $Q_{\mathsf{N}'_1,\mathsf{N}_2}$ and $Q_{\mathsf{N}'_1,\mathsf{N}_2\cup\{\neg\mathsf{R}\}}$. Since having more negated atoms implies that the answer set is more restricted, we have that $\mathsf{ans}_{\mathbf{D}}(Q_{\mathsf{N}'_1,\mathsf{N}_2\cup\{\neg\mathsf{R}\}}) \subseteq \mathsf{ans}_{\mathbf{D}}(Q_{\mathsf{N}'_1,\mathsf{N}_2})$. We apply Lemma 6.4 to get direct access to the answers of $Q_{\mathsf{N}_1,\mathsf{N}_2}$. Preprocessing takes $2 \cdot 2^{|\mathsf{N}_1|-1} \cdot t_p = 2^{|\mathsf{N}_1|} \cdot t_p$ time.

For the access time, we apply Lemma 6.4 by bounding S_1 and S_2 by $|\mathsf{D}|^n$ and by using $t_1 = t_2 = (2C \cdot n \cdot \log|\mathsf{D}|)^{2(|\mathsf{N}_1|-1)}$ that we have by induction. It gives a total access time bounded by $(2C \cdot n \cdot \log|\mathsf{D}|)^{2(|\mathsf{N}_1|-1)} \cdot (C \cdot n \cdot \log|\mathsf{D}|)^2$ which is itself bounded by $(2C \cdot n \cdot \log|\mathsf{D}|)^{2|\mathsf{N}_1|}$. This concludes our induction step. □

The objective at this point is to transfer the knowledge we have about positive queries to conclude on the complexity of direct access for signed queries. This is formalised in Theorem 6.12.

▶ **Theorem 6.12**

Let Q be a self-join free signed join query with n variables, and A be a constant.

- If Q has direct access with preprocessing time t_p and access time t_a, then for all $\mathsf{N} \subseteq \mathsf{atoms}^-(Q)$ we have direct access for $Q_{\mathsf{N},\emptyset}$ with preprocessing time $2^{|\mathsf{atoms}^-(Q)|} \cdot t_p$ and access time $(An \cdot \log|\mathsf{D}|)^{2|\mathsf{atoms}^-(Q)|} \cdot t_a$.
- If for all $\mathsf{N} \subseteq \mathsf{atoms}^-(Q)$ we have direct access for $Q_{\mathsf{N},\emptyset}$ with preprocessing time t_p and access time t_a, then Q has direct access with preprocessing time $2^{|\mathsf{atoms}^-(Q)|} \cdot t_p$ and access time $(An \cdot \log|\mathsf{D}|)^{2|\mathsf{atoms}^-(Q)|} \cdot t_a$.

Proof. The first part of the statement follows from applying Lemma 6.11 with $\mathsf{N}_2 = \emptyset$. The second part follows from Lemma 6.10 applied with $\mathsf{N}_1 = \emptyset$ and $\mathsf{N}_2 = \mathsf{atoms}^-(Q)$ since $Q_{\emptyset,\mathsf{atoms}^-(Q)} = Q$. □

Theorem 6.12 suggests that answering direct access tasks on a signed join query Q is as hard as the hardest positive query that can be obtained by keeping only a subset of negative atoms and turning them positive. The complexity of answering direct access tasks on positive join queries is well understood from [BCM22a]. We can use this to show upper and lower bounds for direct access over signed join queries.

6.2.2 An Algorithm for Direct Access on SJQs

In this section, we will prove that there is an algorithm to answer direct access tasks for signed join queries. This section is base on the results presented in [BCM22a]. However, these results are expressed as a function of an *incompatibility number* $\iota(\cdot,\cdot)$. The *incompatibility number* $\iota(Q,\prec)$ of a query Q for an order $\prec$ on its variables is a function that only depends on Q and the chosen

order and characterises the optimal preprocessing time needed. We will now connect ι with hypergraph decomposition techniques, and notably those presented in Sections 1.2.5 and 5.4.1.

The link between the incompatibility number of a query Q and Q's fractional hypertree width has already been observed in [BCM22a]. In this section, we revisit this connection and extend this analysis to signed hypergraphs.

The definition of the fractional hyperorder width of an elimination order $\prec$ matches the definition of the incompatibility number $\iota(Q, \succ)$ from [BCM22a] as we will use later. However, this incompatibility number is stated for the reverse of the elimination order. In this chapter, we choose to make the connection with the existing literature on elimination orders for hypergraphs more explicit and hence use the reversed order.

We first state the following:

▶ **Theorem 6.13**
For every positive join query Q on variables set X and order $\prec$ on X, the following holds:

$$\iota(Q, \succ) = \mathsf{fhow}(\mathcal{H}(Q), \prec) \ .$$

The case where $\iota(Q, \succ) = 1$ therefore matches the case $\mathsf{fhow}(\mathcal{H}(Q), \prec) = 1$, that is, $\mathcal{H}(Q)$ is α-acyclic and $\prec$ is an α-elimination order witnessing this acyclicity, as defined in [Bra17]. (Reversed) orders witnessing this have also previously been called *orders without disruptive trio* [Car+23], and exactly match the classical notion of elimination ordering witnessing α-acyclicity[1]. Here, we will now mostly use a terminology based on the hypergraph decomposition of the query rather than the incompatibility number. This is because we also aim at comparing our results with other results on negative join queries, which are stated in terms of hypergraphs.

In the case of signed join queries, we can match the notion of signed hyperorder width $\mathsf{sfhow}()$ from Chapter 5 to an extended definition of the incompatibility number of a signed join query. From Definition 5.12 and by Theorem 6.13, we immediately have:

▶ **Theorem 6.14**
For every signed join query Q on variables set X and $\prec$ an order on X, the following holds:

$$\iota(Q, \succ) = \mathsf{sfhow}(\mathcal{H}(Q), \prec) \ .$$

From this point onwards, we will work with hypergraph measures instead of the incompatibility number when dealing with join queries. We start by restating the upper bound from [BCM22a]:

▶ **Theorem 6.15** ([BCM22a, Theorem 44, restated])
For every self-join free join query Q (that we consider constant) on variables set X and order $\prec$ on X, the following property holds.

For every database $\mathbf{D}$, direct access tasks for $\mathsf{ans}_{\mathbf{D}}(Q)$ with order $\prec_{\mathsf{lex}}$ can be answered with

[1] In Brault-Baron's survey [Bra17], they are called α-elimination orders

preprocessing time $\mathcal{O}((p(|Q|) \cdot |\mathbf{D}|)^{\mathsf{fhow}(Q,\succ)})$ and access time $\mathcal{O}(p(|Q|) \cdot \mathsf{log}(|\mathbf{D}|))$.

Theorems 6.12 and 6.15 directly imply the following generalisation of Theorem 6.15 to signed join queries:

▶ **Theorem 6.16**
There exists a function $g\colon \mathbb{N} \to \mathbb{N}$ such that, given a self-join free signed join query Q on variables set X and an order $\prec$ on X, the following property holds.

For every database $\mathbf{D}$, direct access tasks for $\mathsf{ans}_{\mathbf{D}}(Q)$ and order $\prec_{\mathsf{lex}}$ can be answered with preprocessing time $g(|Q|) \cdot |\mathbf{D}|^{\mathsf{sfhow}(Q,\succ)}$ and access time $g(|Q|) \cdot \mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|)$.

Proof. Let Q be a signed join query, m be the number of negative atoms of Q and let $k = \mathsf{sfhow}(Q,\succ)$. Moreover, let p be the polynomial from Theorem 6.15.

By definition, we have that for every $\mathsf{N} \subseteq \mathsf{atoms}^-(Q)$, $\mathsf{fhow}(Q_{\mathsf{N},\emptyset},\succ)) \leqslant k$. Hence, by Theorem 6.15, we can solve the direct access task for $\mathsf{ans}_{\mathbf{D}}(Q_{\mathsf{N},\emptyset})$ using the order $\prec_{\mathsf{lex}}$ with preprocessing time $\mathcal{O}((p(|Q|) \cdot |\mathbf{D}|)^k)$ and access time $\mathcal{O}(p(|Q|) \cdot \mathsf{log}(|\mathbf{D}|))$.

By Lemma 6.10, we then have direct access for Q on $\mathbf{D}$ with preprocessing time

$$\mathcal{O}(2^{|\mathsf{atoms}^-(Q)|}(p(|Q|)|\mathbf{D}|)^k)$$

and access time

$$\mathcal{O}(p(|Q|) \cdot |X| \cdot (A|X|)^{2|\mathsf{atoms}^-(Q)|} \cdot \mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|)) \ .$$

Observe that A is a constant and $|X| \leqslant |Q|$, hence we can define $g(m) = K(p(m) \cdot m \cdot A|X|^{2m})$ for some large enough constant K and we have the desired bound: the preprocessing time can be done in $g(|Q|)|\mathbf{D}|^k$ and the access time in $g(|Q|) \cdot \mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|)$. □

This shows that there exists an algorithm for direct access for signed join queries. We can now prove its optimality.

6.2.3 Optimality of Direct Access for SJQs

The optimality of the algorithm from Section 6.2.2 has also been shown in [BCM22a]. Their result, restated with hypergraph measures, is:

▶ **Theorem 6.17** ([BCM22a, Theorem 44, restated])
For every self-join free join query Q (that we consider constant) on variables set X and order $\prec$ on X, the following property holds.

If there exists a function $f\colon \mathbb{N} \to \mathbb{N}$ and $\varepsilon > 0$ such that for every database $\mathbf{D}$ and constant $\delta > 0$, direct access tasks for $\mathsf{ans}_{\mathbf{D}}(Q)$ with order $\prec_{\mathsf{lex}}$ can be solved with preprocessing time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\mathsf{fhow}(Q,\succ)-\epsilon})$ and access time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\delta})$, then the Zero-Clique Conjecture is

false.

The *Zero-Clique Conjecture* is a widely believed conjecture in the domain of fine-grained complexity, and expresses that, for every k and $\varepsilon > 0$, there is no randomised algorithm with complexity $\mathcal{O}(n^{k-\varepsilon})$ able to decide whether there exists a k-clique whose edge weights sum to 0 in an edge-weighted graph.

In other words, unless such an algorithm exists, the best preprocessing one can hope for to solve direct access tasks for positive join queries is $\mathcal{O}(|\mathbf{D}|^{\mathsf{fhow}(Q,\succ)})$. The complexity measures presented in Theorems 6.15 and 6.17 (restated from [BCM22a]) are given in terms of data complexity. However, it is not hard to see that the dependency on $|Q|$ is polynomial for the constant $\mathsf{fhow}(Q,\succ)$.

We can now generalise Theorem 6.17 to signed join queries.

▶ **Theorem 6.18**
There exists a function $g\colon \mathbb{N} \to \mathbb{N}$ such that, given a self-join free signed join query Q on variables set X and an order $\prec$ on X, the following property holds.

If there exists $f\colon \mathbb{N} \to \mathbb{N}$ and $\varepsilon > 0$ such that for every database $\mathbf{D}$ and constant $\delta > 0$, direct access tasks for $\mathsf{ans}_{\mathbf{D}}(Q)$ with order $\prec_{\mathsf{lex}}$ can be solved with preprocessing time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\mathsf{sfhow}(\mathcal{H}(Q),\succ)-\varepsilon})$ and access time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\delta})$, then the Zero-Clique Conjecture is false.

Proof. Let Q be a signed join query, m be the number of negative atoms of Q and let $k = \mathsf{sfhow}(\mathcal{H}(Q),\succ)$. Moreover, let p be the polynomial from Theorem 6.17.

Assume that there exists a function $f\colon \mathbb{N} \to \mathbb{N}$ and a value $\varepsilon > 0$ such that, for every $\delta' > 0$, we have a direct access scheme for Q with preprocessing time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{k-\varepsilon})$ and access time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\delta'})$.

Let $\mathsf{N} \subseteq \mathsf{atoms}^-(Q)$ be such that $\mathsf{sfhow}(Q_{\mathsf{N},\emptyset},\succ) = k$. Let $\delta > 0$ and set $\delta' = \delta/2$. By Lemma 6.11, we have direct access to $\mathsf{ans}_{\mathbf{D}}(Q_{\mathsf{N},\emptyset})$ with preprocessing time $\mathcal{O}(2^{|\mathsf{atoms}^-(Q)|} f(|Q|) \cdot |\mathbf{D}|^{k-\varepsilon}) = \mathcal{O}(h(|Q_{\mathsf{N},\emptyset}|) \cdot |\mathbf{D}|^{k-\varepsilon})$ and access time $\mathcal{O}(f(|Q|) \cdot \mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|)|\mathbf{D}|^{\frac{\delta}{2}}) = \mathcal{O}(h(|Q_{\mathsf{N},\emptyset}|) \cdot |\mathbf{D}|^{\delta})$ for some function $h\colon \mathbb{N} \to \mathbb{N}$.

In this case however, by Theorem 6.17, we have an algorithm for $Q_{\mathsf{N},\emptyset}$ whose complexity implies that the Zero-Clique Conjecture is false. □

Observe that the complexity obtained for the preprocessing of a signed query has an exponential dependency on the size of the query. This is due to the fact that it is obtained using Lemma 6.10, which intuitively does a direct access preprocessing for every subquery of Q. This exponential dependency in $|Q|$ is not present for positive queries in [BCM22a]. Section 6.3 is dedicated to the design of a better algorithm, using tools from Chapter 5, thus achieving the same data complexity as in Theorems 6.16 and 6.18, but with a better dependency on $|Q|$ and a more direct algorithm.

We conclude this section with a final observation regarding Theorems 6.16 and 6.18. It is proved in [BCM22a] that both Theorems 6.15 and 6.17 also hold when Q contains self-joins. This is however not true for signed join queries which is why we explicitly assumed Q to be self-join free.

Indeed, for any join query Q containing an atom $\mathsf{R}(\mathbf{x})$ and order $\prec$, the signed query $Q' :\!-\ Q\ \neg\mathsf{R}(\mathbf{x})$ is such that $\mathsf{sfhow}(Q,\succ) = \mathsf{sfhow}(Q',\succ)$ can easily be answered with preprocessing and access time independent on $|\mathbf{D}|$, which shows that the lower bound from Theorem 6.18

does not hold for signed join queries with self-joins. The upper bound, on the other hand, holds, as we can always consider each occurrence of a relation as an unique relation. Understanding the complexity of signed join queries with self-joins is an interesting research direction that we will leave open for future work.

To ease future comparisons, let us unify Theorems 6.15 to 6.18 into one main result:

▶ **Theorem 6.19**
There exists a polynomial p such that given a self-join free signed join query Q, on variables set X and an order $\prec$ on X, we have:

- for every database $\mathbf{D}$, direct access tasks for $\mathsf{ans}_{\mathbf{D}}(Q)$ and order $\prec_{\mathsf{lex}}$ can be answered with:

  preprocessing time: $\mathcal{O}(2^{|\mathsf{atoms}^-(Q)|}(p(|Q|)\cdot|\mathbf{D}|)^k)$; and

  access time: $\mathcal{O}(p(|Q|)\cdot|X|\cdot\mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}|\mathbf{D}|)$.
- if there exists $f\colon\mathbb{N}\to\mathbb{N}$ and $\varepsilon>0$ such that, for every database $\mathbf{D}$ and constant $\delta>0$, direct access tasks for $\mathsf{ans}_{\mathbf{D}}(Q)$ with order $\prec_{\mathsf{lex}}$ can be solved with:

  preprocessing time: $\mathcal{O}(f(|Q|)\cdot|\mathbf{D}|^{k-\varepsilon})$; and

  access time: $\mathcal{O}(f(|Q|)\cdot|\mathbf{D}|^{\delta})$

  then the Zero-Clique Conjecture is false.

Here, k is defined as : $k=\mathsf{sfhow}(\mathcal{H}(Q),\succ)$.

▶ **Remark 6.20**
Notice that, in the case where the query is positive (that is, that there are no negative atoms), $k=\mathsf{sfhow}(\mathcal{H}(Q),\succ)=\mathsf{fhow}(\mathcal{H}(Q),\succ)$.

In the rest of this chapter, we propose another, more direct, approach for solving direct access on signed conjunctive queries. This new approach also reduces the combined complexity to a polynomial in both $|Q|$ and $|\mathbf{D}|$ when parameterising by the non-fractional version of hyperorder width of Q (that is $\mathsf{show}(\mathcal{H}(Q),\prec)$). Moreover, the algorithm presented in the next section has a better access time than the one we obtained in Theorem 6.19.

6.3 Direct Access for Ordered Relational Circuits

In this section, we will take advantage of the structure of ordered $\{\times,\mathsf{dec}\}$-circuits as we defined them in Chapter 5 to design an algorithm for direct access.

More precisely, we define the notion of direct access tasks for circuits as follows:

▶ **Definition 6.21** (Direct Access for Circuits)
Let C be a relational circuit on variables X and let $\prec$ be an order on X. We say that an algorithm $\mathcal{D}$ solves direct access for C and order $\prec_{\mathsf{lex}}$ with preprocessing time t_p and access time t_a if:

- there exists an algorithm $\mathcal{P}$ that runs in time t_p and builds a data structure C'; and
- there exists an algorithm $\mathcal{A}$ that runs in time t_a such that, using C', it returns the k^{th} tuple of $\mathsf{rel}(C)$ for the order $\prec_{\mathsf{lex}}$ on input k.

The main result of this chapter is an algorithm that allows for direct access for an ordered $\{\times, \mathsf{dec}\}$-circuit on domain D and variables set X. More precisely, we will prove the following:

▶ **Theorem 6.22**
Let $\prec$ be an order on X and C be a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit on domain D and variables set X, then we can solve direct access tasks on $\mathsf{rel}(C)$ for order $\prec_{\mathsf{lex}}$ with:

precomputation time: $\mathcal{O}(|X| \cdot \mathsf{log}|X| \cdot |C|)$; and

access time: $\mathcal{O}(|X|^3 \cdot \mathsf{log}|X| + |X|^2 \cdot \mathsf{log}|\mathsf{D}|)$

6.3.1 Preprocessing Phase

In this section, we assume that C is a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit with respect to X. Moreover, for every decision gate v, we assume that its ingoing edges are stored in a list sorted by increasing value of their label. More precisely, we assume that the ingoing edges of v are a list $[e_1, \ldots, e_k]$ such that $d_1 < \cdots < d_k$ where d_i is the label of e_i. If the list of ingoing edges of v was not sorted we could always sort it. This would cost a time linear in the size of the list, which is the size of the domain, since we could use *radix sort* to sort as mentioned in Section 6.1.

The $\mathsf{nrel}_C(\cdot,\cdot)$ values

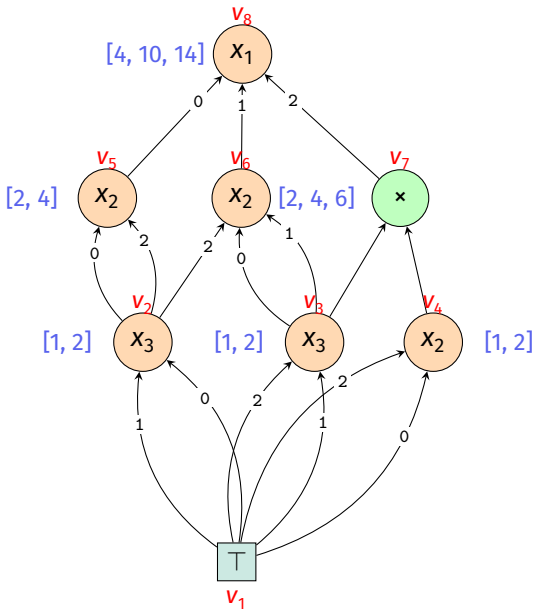
We are interested in the following values: for every decision gate v of C labelled with variable x and ingoing edges $[e_1, \ldots, e_k]$ with respective labels $[d_1, \ldots, d_k]$, we define for every $i \leqslant k$, $\mathsf{nrel}_C(v, d_i)$ as $\#\sigma_{x \leqslant d_i}(\mathsf{rel}(v))$. That is, $\mathsf{nrel}_C(v, d_i)$ is the number of tuples from $\mathsf{rel}(v)$ that assign a value on x smaller or equal than d_i. The precomputation step aims to compute nrel_C so that we can access $\mathsf{nrel}_C(v, d_i)$ quickly for every v and d_i.

▶ **Example 6.23** (An annotated relational circuit)
In this example, we show a $\{\times, \mathsf{dec}\}$-circuit annotated with nrel_C values (on the left), and the relation computed by the circuit (on the right). The domain used is $\{0, 1, 2\}$ for variables x_1, x_2 and x_3.

On the circuit, the lists shown to the left of the decision gates represent the values of nrel_C for those gates.

▶ **Remark:** For direct access, the circuit has to be annotated in a more complete way. This full annotation of the circuit is a bit more complicated and will be described later in this section.

In this example, each decision-gate is labelled with a list. For a decision-gate v in the circuit with incoming edges $e_1, \ldots, e_k$ labelled by $d_1, \ldots, d_k$ with $d_1 < \cdots < d_k$, the i^{th} entry of the list is $\mathsf{nrel}_C(v, d_i)$.



x_1	x_2	x_3
0	0	0
0	0	1
0	2	0
0	2	1
1	0	1
1	0	2
1	1	1
1	1	2
1	2	0
1	2	1
2	0	1
2	2	2
2	0	1
2	2	2

For instance, gate v_2 is labelled by $[1, 2]$. This is because the relation computed by v_2 is on variables $\{x_3\}$ and contains two tuples: $\langle x_3 \leftarrow 0\rangle$ and $\langle x_3 \leftarrow 1\rangle$. This implies that there is *one* tuple in this relation which sets x_3 to a value smaller or equal to 0, and *two* tuples which set x_3 to a value smaller or equal to 1. Since there is no incoming edge labelled by 2, we do not need to compute $\mathsf{nrel}_C(v_2, 2)$. That is, $\mathsf{nrel}_C(v_2, 0) = 1$ and $\mathsf{nrel}_C(v_2, 1) = 2$.

Incidently, observe that gate v_3 is also labelled by $[1, 2]$. However, in this case, this corresponds to $\mathsf{nrel}_C(v_3, 1) = 1$ and $\mathsf{nrel}_C(v_3, 2) = 2$ since there is no incoming edge labelled by 0.

The gate v_6 is labelled by $[2, 4, 6]$. This is because there are *two* tuples with $x_2 = 0$ in $\mathsf{rel}(v_6)$. Both of these tuples come from following the 0-labelled incoming edge of v_6 from v_3. The size of the relation computed by v_3 is 2, which can be read from $\mathsf{nrel}_C(v_3, 2)$. This implies that $\mathsf{nrel}_C(v_6, 0) = 2$. Similarly, by following the 1-labelled incoming edge from v_3, we know that there are 2 tuples with $x_2 = 1$ in $\mathsf{rel}(v_6)$, and thus 4 tuples with $x_2 \leqslant 1$, which means $\mathsf{nrel}_C(v_6, 1) = 2 + \mathsf{nrel}_C(v_6, 0) = 4$. Finally, we get 2 tuples with $x_2 = 2$ from v_2, giving $\mathsf{nrel}_C(v_6, 2) = 2 + \mathsf{nrel}_C(v_6, 1) = 6$.

Computing $\mathsf{nrel}_C(\cdot)$ with dynamic programming

As hinted in Example 6.23, our algorithm performs a *bottom-up* computation to compute $\mathsf{nrel}_C(\cdot, \cdot)$ values. However, some other values need to be computed along them to ease later

computation.

The first extra value we need represents the number of tuples in $\mathsf{rel}(v)$ for every gate v of C. We compute this value inductively.

1. If v is an input gate, then we obviously have $|\mathsf{rel}(v)| = 0$ if it is labelled by $\bot$, and $|\mathsf{rel}(v)| = 1$ if it is labelled by $\top$.

2. Now, suppose v is a decision gate on variable x with (sorted) ingoing edges $e_1 = (v_1, v), \ldots, e_k = (v_k, v)$ respectively labelled by $d_1 = \ell(e_1) < \cdots < d_k = \ell(e_k)$. For a gate w that is an input of v, we denote by $\Delta(v, w)$ the set of the variables of v that are not variables of w, that is $\Delta(v, w) = \mathsf{var}(v) \setminus (\{x\} \cup \mathsf{var}(w))$. We can then compute $|\mathsf{rel}(v)|$ inductively as:

$$|\mathsf{rel}(v)| = \sum_{i=1}^{k} |\mathsf{rel}(v_i)| \times |\mathsf{D}|^{|\Delta(v,v_i)|} \qquad \text{(Eq. 6.24)}$$

   Similarly, $\mathsf{nrel}_C(v, d_i)$ can be computed by restricting the previous relation on the inputs of v:

$$\mathsf{nrel}_C(v, d_i) = \sum_{j=1}^{i} |\mathsf{rel}(v_j)| \times |\mathsf{D}|^{|\Delta(v,v_j)|} \qquad \text{(Eq. 6.25)}$$

   In particular, observe that

$$\mathsf{nrel}_C(v, d_1) = |\mathsf{rel}(v_1)| \times |\mathsf{D}|^{|\Delta(v,v_1)|}, \text{ and}$$
$$\mathsf{nrel}_C(v, d_{i+1}) = \mathsf{nrel}_C(v, d_i) + |\mathsf{rel}(v_{i+1})| \times |\mathsf{D}|^{|\Delta(v,v_{i+1})|}$$

3. Finally, v can also be a Cartesian product gate. In that case, we have that $|\mathsf{rel}(v)| = \prod_{w \in \mathsf{input}(v)} |\mathsf{rel}(w)|$.

Therefore, one can compute nrel_C using a dynamic programming algorithm that inductively computes $\mathsf{nrel}_C(v, d)$ for each decision-gate v of the circuit with an ingoing edge labelled by d. Observe however, that in order to efficiently compute these values, we also need to have easy access to $|\mathsf{rel}(v)|$ and $\mathsf{var}(v)$ for every gate v. Fortunately, as we hinted above, these values are also easy to compute inductively.

More precisely, the dynamic programing algorithm works as follows: we start by performing a topological ordering $(v_1, \ldots, v_N)$ of the gates of C that is compatible with the underlying DAG of the circuit. In particular, it means that for a gate v and an input w of v, the topological ordering has to place w before v. This can be computed in time $\mathcal{O}(|C|)$.

We dynamically compute, for every gate v, the values $|\mathsf{rel}(v)|$, the sets $\mathsf{var}(v)$, and, if v is a decision gate with an ingoing edge labelled by $d \in D$, we also compute $\mathsf{nrel}_C(v, d)$.

For this, we proceed in the following way: first, we allocate a table T_{rel} of size N such that for $i \leqslant N$, $T_{\mathsf{rel}}[i]$ is initialised to 0. At the end of the algorithm, $T_{\mathsf{rel}}[i]$ will contain $|\mathsf{rel}(v_i)|$. We also allocate a table T_{var} such that for $i \leqslant N$, $T_{\mathsf{var}}[i]$ is initialised with a $|X|$-bitvector containing only 0. At the end of the algorithm, $T_{\mathsf{var}}[v_i][k]$ is 1 if, and only if, $x_k \in \mathsf{var}(v_i)$. This initialisation step takes $\mathcal{O}(N \cdot |X|) = \mathcal{O}(|C| \cdot |X|)$.

We now initialise T_{nrel_C} of size N where, for $i \leqslant N$, the entry $T_{\mathsf{nrel}_C}[i]$ is intialised with an array of size $|\mathsf{input}(v_i)|$ containing only -1 values. Clearly, T_{nrel_C} has size $\mathcal{O}(|C|)$ since it contains one entry per edge of C. At the end of the algorithm, for $i \leqslant N$, if v_i is a decision gate, then

$T_{\mathsf{nrel}_C}[i][p]$ contains $\mathsf{nrel}_C(v_i, d_p)$ where d_p is the label of the p^{th} input of v_i. If v_i is a Cartesian product gate or an input gate, $T_{\mathsf{nrel}_C}[i]$ is not relevant and is not modified after the initialisation.

We then populate each entry of these tables following the previously constructed topological ordering and using the relations written above (see Equation (6.25) and Equation (6.25)) and the fact that $\mathsf{var}(v) = \bigcup_{w \in \mathsf{input}(v)} \mathsf{var}(w)$. To compute nrel_C for a decision gate v, we let $(w_1, v), \ldots, (w_k, v)$ be the incoming edges of v ordered by increasing labels $d_1 < \cdots < d_k$. We set $T_{\mathsf{nrel}_C}[v, d_1]$ to $\mathsf{nrel}_C(v, d_1)$ which is equal to $|\mathsf{rel}(w_1)| \cdot |D|^{|\Delta(v,w_1)|}$. We then compute $T_{\mathsf{nrel}_C}[v, d_{i+1}]$ as $T_{\mathsf{nrel}_C}[v, d_i] + |\mathsf{rel}(w_{i+1})| \cdot |\mathsf{D}|^{|\Delta(v,w_{i+1})|}$ using Equation (6.25).

It is clear from Equation (6.25) that, at the end of this precomputation, we have that $T_{\mathsf{nrel}_C}[v, d]$ contains $\mathsf{nrel}_C(v, d)$ if d labels an incoming edge of v. Pseudocode for this preprecessing algorithm can be found in Algorithm 6.26.

Algorithm 6.26 An algorithm to annotate a circuit C on variables X and domain D

```
1:  procedure ANNOTATE(C, X, D)
2:      input: · C an ordered {×, dec}-circuit,
                · X a set of variables,
                · D a domain for the variables
3:      output: an annotated ≺-ordered {×, dec}-circuit for ans_D(Q)
4:      v_1, ..., v_N ← a topological ordering of the gates of C
5:      T_rel ← table of size N, initialised to 0
6:      T_var ← table of size N, initialised with 0-bitvectors of size |X|
7:      T_nrel_C ← table of size N;
8:      for i = 1 to N do
9:          if v_i is ⊤-input then T_rel[i] ← 1; continue
10:         if v_i is ⊥-input then T_rel[i] ← 0; continue
11:         if v_i is a ×-gate then
12:             T_rel[i] ← ∏_{v_j ∈ input(v_i)} T_rel[j]
13:             T_var[i] ← ⋃_{v_j ∈ input(v_i)} T_var[j]
14:         else if v_i is a decision gate on x_k then
15:             T_var[i] ← {x_k} ∪ ⋃_{v_j ∈ input(v_i)} T_var[j]
16:             T_rel[i] ← Σ_{v_j ∈ input(v_i)} T_rel[j] × |D|^{|T_var[j] \ T_var[i]| − 1}
17:             Order incoming edges (v_{a_1}, v_i), ..., (v_{a_k}, v_i) of v_i by increasing label
18:             T_nrel_C[i] ← new array of size k initialised to 0
19:             T_nrel_C[i][1] ← T_rel[a_1]
20:             for j = 2 to k do
21:                 T_nrel_C[i][k] ← T_nrel_C[i][j − 1] + T_rel[a_j]
22:     return T_rel, T_var, T_nrel_C
```

▶ **Example 6.27**

From the circuit in Example 6.23, assume we have populated T_{rel}, T_{var} and T_{nrel_C} up to $i = 6$. We now need to compute the seventh entry of each table.

Since v_7 is a $\times$-gate, we do not need to do anything for $T_{\mathsf{nrel}_C}[7]$.

Now, $T_{\mathsf{rel}}[7]$ must contain the size of the relation computed by v_7. Since v_7 is a $\times$-gate, this is

$|\mathsf{rel}(v_3)| \times |\mathsf{rel}(v_4)|$ which is equal to $T_{\mathsf{rel}}[3] \times T_{\mathsf{rel}}[4]$ by induction. So we set $T_{\mathsf{rel}}[7] = T_{\mathsf{rel}}[3] \times T_{\mathsf{rel}}[4]$. Similarly, $T_{\mathsf{var}}[7] = T_{\mathsf{var}}[3] \cup T_{\mathsf{var}}[4]$. Since $T_{\mathsf{var}}[7]$ is encoded as a bitvector, the union boils down to making a bitwise OR operation.

Finally, we need to compute $T_{\mathsf{rel}}[8], T_{\mathsf{var}}[8], T_{\mathsf{nrel}_C}[8]$. Since v_8 is a decision gate, $T_{\mathsf{var}}[8] = T_{\mathsf{var}}[5] \cup T_{\mathsf{var}}[6] \cup T_{\mathsf{var}}[7] \cup \{x_1\}$. Now, $T_{\mathsf{rel}}[8] = T_{\mathsf{rel}}[5] + T_{\mathsf{rel}}[6] + T_{\mathsf{rel}}[7]$. Observe that, in this case, we do not need to add a corrective factor $\Delta(\cdot)$ since v_5, v_6 and v_7 define relations on the same variables set. If the variables sets were different however, we would have to adjust the value following Equation (6.27). It then remains to compute $T_{\mathsf{nrel}_C}[8]$. Since v_8 has three incoming edges, we will have to compute $T_{\mathsf{nrel}_C}[8][1], T_{\mathsf{nrel}_C}[8][2]$, and $T_{\mathsf{nrel}_C}[8][3]$. We order the edges by increasing label and find that:

$$
\begin{aligned}
T_{\mathsf{nrel}_C}[8][1] &= T_{\mathsf{rel}}[5] = 4 \\
T_{\mathsf{nrel}_C}[8][2] &= T_{\mathsf{nrel}_C}[8][1] + T_{\mathsf{rel}}[6] = 4 + 6 = 10 \\
T_{\mathsf{nrel}_C}[8][3] &= T_{\mathsf{nrel}_C}[8][2] + T_{\mathsf{rel}}[7] = 10 + 4 = 14
\end{aligned}
$$

The following lemma follows directly from the previously described algorithm:

▶ **Lemma 6.28** (Precomputation complexity)
Given a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit C, we can compute a data structure in time $\mathcal{O}(|X|\mathsf{log}(|X|) \cdot |C|)$ that allows us to access in constant time to the following:

- $\mathsf{var}(v)$, $|\mathsf{rel}(v)|$ for every gate v of C; and
- $\mathsf{nrel}_C(v, d)$ for every decision gate v having an ingoing edge labelled by $d \in \mathsf{D}$.

Proof. The data structure simply consists of the three tables $T_{\mathsf{nrel}_C}, T_{\mathsf{rel}}$ and T_{var}.

It is easy to see that each entry of T_{var} can be computed in time $\mathcal{O}(|X|)$ since we only have to compute the union of sets of elements in X and we can use a bitmask of size $|X|$ to do it efficiently. Indeed, we can represent $\mathsf{rel}(v)$ as a bitvector $T_{\mathsf{var}}[v]$ with $|X|$ entries for each v such that $T_{\mathsf{var}}[v][i] = 1$ if, and only if, $x_i \in \mathsf{rel}(v)$. This implies that computing $T_{\mathsf{var}}[v][i]$ can be done by looking at every input w of v and check whether $T_{\mathsf{var}}[w][i] = 1$. For each gate w, we will have one such look-up for each incoming edge and each variable, hence one can compute T_{var} in time $\mathcal{O}(|C| \cdot |X|)$.

Now, observe that to compute T_{nrel_C}, one has to perform at most *two* arithmetic operations for each edge of C. Indeed, to compute $T_{\mathsf{rel}}[v]$, one has to perform at most one addition and one multiplication for each $w \in \mathsf{input}(v)$, whose cost can be associated to the edge (w, v). Similarly, to compute $T_{\mathsf{nrel}_C}[v, d]$ as described in the previous paragraph, we do one addition and one multiplication for each edge (w, v) in the circuit. Hence, we perform at most $\mathcal{O}(|C|)$ arithmetic operations. The cost of these arithmetic operations is at most $\mathcal{O}(|X|\mathsf{log}(|X|))$ in the unit-cost RAM model (see Chapter 1 for a discussion). Indeed, all operations are performed on integers representing sizes of relations on a domain D and variables set $Y \subseteq X$. Their value can therefore not exceed $|\mathsf{D}|^{|X|}$.

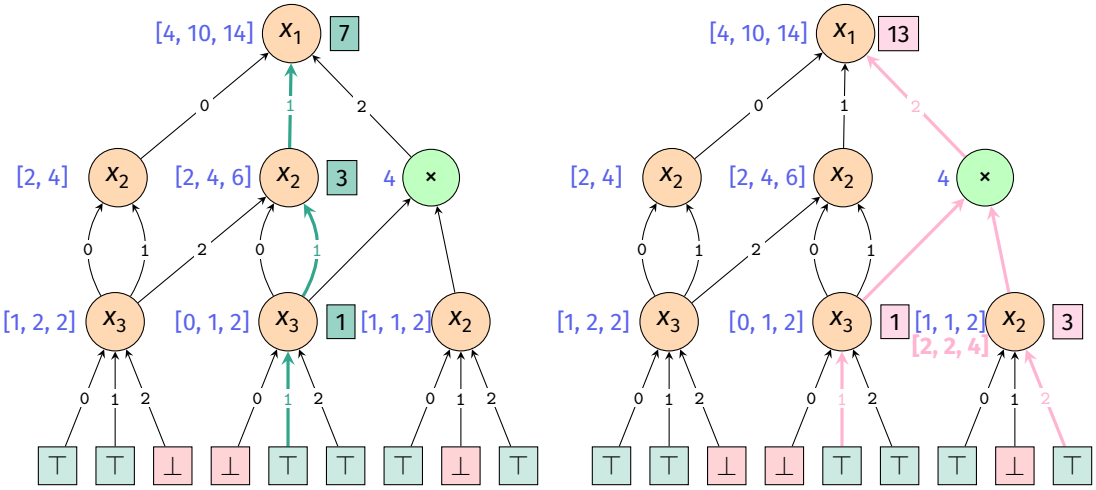
Thus, we have a total complexity of $\mathcal{O}(|X|\mathsf{log}(|X|) \cdot |C|)$. □

6.3.2 Direct Access

We now show how the precomputation from Lemma 6.28 allows us to get direct access for ordered $\{\times, \mathsf{dec}\}$-circuits. We first show how one can solve a direct access task for any relation as long as

one has access to very simple counting oracles. We then show that one can quickly simulate these oracle calls in ordered $\{\times, \mathsf{dec}\}$-circuits using precomputed values.

▶ **Example 6.29** (Direct access over a relational circuit)
In this example, we show how our direct access algorithm works.



(a) $k = 7$ **(b)** $k = 13$

Illustration (**a**) shows the paths taken in the circuit to evaluate the 7^{th} tuple in the circuit. The key idea here resides in the fact that we can connect direct access with counting tasks. If we want the 7^{th} tuple in the circuit, we cannot assign $x_1 \leftarrow 0$, since we know that there are only 4 possible extensions in that case. We should however assign $x_1 \leftarrow 1$, since there are 10 possible assignments with $x_1 \leqslant 1$. We then move down the circuit following the edge labelled by 1, and search in the new gate, but we have to adapt the index since 4 of the possible extensions have been discarded (because they involve assigning $x_1 \leftarrow 0$). We are now therefore looking for the $7^{\text{th}} - 4 = 3^{\text{rd}}$ tuple in this subcircuit. Again, assigning $x_2 \leftarrow 0$ will only give 2 possible extensions, which is not enough to find the third solution. Assigning $x_2 \leftarrow 1$ however is enough since there are 4 assignments with $x_2 \leqslant 1$. We hence set $x_2 \leftarrow 1$ and proceed with the last decision gate. By assigning $x_2 \leftarrow 1$, we discarded 2 tuples, so we are now looking for the first answer in this last decision gate, which is obtained by setting $x_3 \leftarrow 1$. We continue the search in this manner until reaching a terminal gate.

A perhaps more involved example can be found in Illustration (**b**). In this case, we are looking for the 13^{th} tuple of the circuit. However, after assigning (with the same method as before) $x_1 \leftarrow 2$, we reach a $\times$-gate, and therefore do not assign a value to a given variable. What happens is we are now looking for the $13 - 10 = 3^{\text{rd}}$ solution of a set of circuits comprised of the bottom-centre gate and the bottom-right gate. We choose which of the variables from the roots of the set of circuits to assign first based on the order, so in this case, we start with x_2.

We have to remember that for each possible solution of the circuit rooted in x_2, there are

two possible extensions from the circuit rooted in x_3 (see Lemma 6.31). This is the same thing as imagining the gate being virtually relabelled by $[2, 2, 4]$. In other words, $x_2 \leftarrow 0$ provides two tuples for the Cartesian product, which is not enough to get the third answer. The same happens for $x_2 \leftarrow 1$ but $x_2 \leftarrow 2$ has enough extension since there are 4 assignments for the Cartesian product that have $x_2 \leqslant 2$. We hence assign $x_2 \leftarrow 2$. We discarded 2 assignments with this choice, we are hence looking for the first assigmnent for x_3 which is found by setting x_3 to 1.

In order to evaluate the true complexity of answering direct access tasks, we now also have to evaluate the complexity of a single oracle call. To do that, we need to understand how an assignment of the form $\mathsf{D}^{\{x_1,\dots,x_p\}}$ for some p behaves with respect to the circuit. We hence define a *prefix assignment* of size p to be an assignment $\tau \in \mathsf{D}^{\{x_1,\dots,x_p\}}$ with $p \leqslant n$. We also define the *sinking* operation:

▶ **Definition 6.30** (Sinking through $\times$-gates)
For a gate v in C, we define $\mathsf{sink}(v)$ as:

$$\mathsf{sink}(v) = \begin{cases} \bigcup_{w \in \mathsf{inputs}(v)} \mathsf{sink}(w) & \text{if } v \text{ is a } \times\text{-gate} \\ \{v\} & \text{otherwise (that is, } v \text{ is an input or a decision gate)} \end{cases}$$

We can then state the following property:

▶ **Lemma 6.31**
For any gate v, we have $\mathsf{rel}(v) = \bigtimes_{w \in \mathsf{sink}(v)} \mathsf{rel}(w)$.

Proof. We prove this by induction on the circuit.

If v is a decision gate or an input, then, as $\mathsf{sink}(v) = \{v\}$, the property is trivial.

If v is a $\times$-gate, then by definition, $\mathsf{rel}(v) = \bigtimes_{w \in \mathsf{inputs}(v)} \mathsf{rel}(w)$. By our induction hypothesis, $\mathsf{rel}(w) = \bigtimes_{g \in \mathsf{sink}(w)} \mathsf{rel}(g)$ for all inputs w of v. Therefore, by associativity and commutativity of the Cartesian product,

$$\begin{aligned} \mathsf{rel}(v) &= \bigtimes_{w \in \mathsf{inputs}(v)} \bigtimes_{g \in \mathsf{sink}(w)} \mathsf{rel}(g) \\ &= \bigtimes_{g \in \bigcup_{w \in \mathsf{inputs}(v)} \mathsf{sink}(w)} \mathsf{rel}(g) \\ &= \bigtimes_{g \in \mathsf{sink}(v)} \mathsf{rel}(g) \end{aligned}$$

□

Given a prefix assignment τ, we want to characterise the set of tuples in C that are compatible with τ. To do so, we can follow every decision gate whose variables are set by τ and follow every

input of $\times$-gates. This gives a set of gates in the circuit that we call the *frontier of* τ, such that any assignment of $\mathsf{rel}(C)$ extending τ must be made from assignments of each gate in the frontier. More formally:

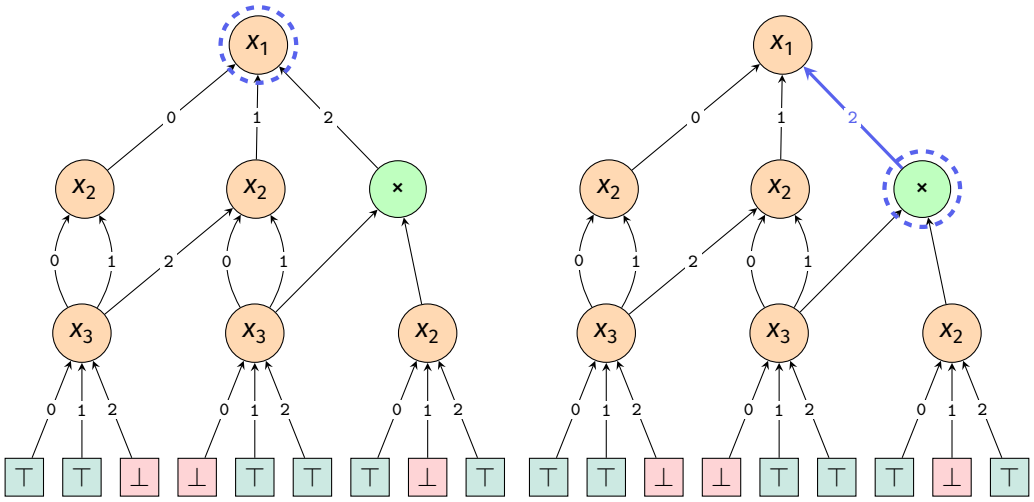
▶ **Definition 6.32** (Frontier of a tuple)
We introduce the *frontier* f_τ *of* τ in C as follows:

1. instantiate a set $F = \{\mathsf{out}(C)\}$, containing only the root of the circuit;
2. as long as it is possible, transform F as follows:
   - if $v \in F$ is a $\times$-gate, remove v from F and add $\mathsf{sink}(v)$ to F;
   - if $v \in F$ is a decision gate and the variable x labelling v is assigned in the prefix, *i.e.* $x \in \{x_1, \ldots, x_p\}$, remove v from F and add v_d to F, where v_d is the input of v such that edge (v_d, v) is labelled by $\tau(x)$.
3. if F contains a $\bot$-gate, then $f_\tau = \emptyset$, otherwise $f_\tau = F$.

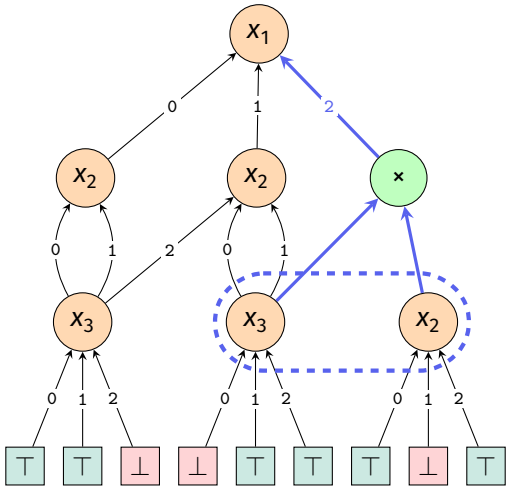
If, for a given gate v, the set $\mathsf{sink}(v)$ contains a $\bot$-gate, then the circuit is no longer satisfiable, which is why we return $\emptyset$ in this case. Note that this should not happen while building the k^{th} solution for C.

Frontiers are particularly useful since they can represent the following relation: the *frontier relation* denoted by $\mathsf{rel}(f_\tau)$ is defined as the relation $\bigtimes_{v \in f_\tau} \mathsf{rel}(v)$. This relation is defined on variables set $\mathsf{var}(f_\tau) := \bigcup_{v \in f_\tau} \mathsf{var}(v)$. An example of a frontier computation is given in Example 6.33.

▶ **Example 6.33** (Computing a frontier)
Consider the same circuit as in Example 6.29, represented in (**a**).



(a) **(b)**



(c)

And suppose we are computing the frontier for the prefix assignment $\tau = \langle x_1 \leftarrow 2 \rangle$. We start with the decision gate labelled with x_1, as it is the root of the circuit. At this point, the frontier f_τ consists of the root node, circled in a dashed, coloured line. Since this is a decision gate and x_1 is assigned in τ, we remove it from the frontier and we add the input labelled by $\tau(x_1) = 2$. This is illustrated in **(b)**, where the edge taken is shown coloured and the current frontier is again circled.

Now we have a frontier consisting of a $\times$-gate. We therefore remove this gate from f_τ and apply the sink operation. This implies that we return the union of the inputs of our $\times$-gate.

Our new frontier f_τ is therefore circled in (**c**). Since the frontier now consists of decision gates whose variables have not been assigned in τ, the computation stops.

For a prefix τ on variables set $\{x_1, \ldots, x_p\}$, we denote by $\sigma_\tau(\mathsf{R})$ the relation $\sigma_{x_1=\tau(x_1),\ldots,x_p=\tau(x_p)}(\mathsf{R})$. We have the following connection between the relation of a frontier and prefix assignments:

▶ **Lemma 6.34**
Let C be an ordered $\{\times, \mathsf{dec}\}$-circuit on variables set $X = \{x_1, \ldots, x_n\}$ and τ be a prefix assignment on variables set $\{x_1, \ldots, x_p\}$. Then,

$$\sigma_\tau(\mathsf{rel}_X(C)) = \{\tau\} \times \mathsf{rel}(f_\tau) \times \mathsf{D}^{\{x_{p+1},\ldots,x_n\} \setminus \mathsf{var}(f_\tau)}$$

Proof. We prove the lemma by induction on the size of the prefix.

For an empty prefix $\tau = \langle\rangle$, we have $f_\tau = \mathsf{sink}(\mathsf{out}(C))$. Indeed, if $\mathsf{out}(C)$ is a decision gate or an input, then it is trivial since $\mathsf{sink}(\mathsf{out}(C)) = \{\mathsf{out}(C)\}$. Otherwise we simply *sink* through the $\times$-gate since no variable is assigned by τ. We have that $\sigma_{\langle\rangle}(\mathsf{rel}_X(C)) = \mathsf{rel}_X(C)$, which is itself by definition equal to $\mathsf{rel}(\mathsf{out}(C)) \times \mathsf{D}^{X \setminus \mathsf{var}(\mathsf{out}(C))}$. From Lemma 6.31, we know that $\mathsf{rel}(\mathsf{out}(C)) = \bigtimes_{w \in \mathsf{sink}(\mathsf{out}(C))} \mathsf{rel}(w)$. We know that $\mathsf{var}(\mathsf{out}(C)) = \mathsf{var}(f_{\langle\rangle})$. Thus, we have that $\sigma_{\langle\rangle}(\mathsf{rel}_X(C)) = \bigtimes_{w \in \mathsf{sink}(\mathsf{out}(C))} \mathsf{rel}(w) \times \mathsf{D}^{\{x_1,\ldots,x_n\} \setminus \mathsf{var}(f_{\langle\rangle})}$.

Now, suppose the property holds for any prefix τ of size p. We show that, in this case, it also holds for a prefix $\tau' = \tau \times [x_{p+1} \leftarrow d]$, for $d \in \mathsf{D}$.

We can rewrite $\sigma_{\tau'}(\mathsf{rel}_X(C))$ as $\sigma_{x_{p+1}=d}(\sigma_\tau(\mathsf{rel}_X(C)))$. Applying the induction hypothesis to this equality gives:

$$\sigma_{\tau'}(\mathsf{rel}_X(C)) = \sigma_{x_{p+1}=d}\left(\{\tau\} \times \mathsf{rel}(f_\tau) \times \mathsf{D}^{\{x_{p+1},\ldots,x_n\} \setminus \mathsf{var}(f_\tau)}\right)$$

From here on, we have two possibilities: either there exists a decision gate $v \in f_\tau$ such that $\mathsf{decvar}(v) = x_{p+1}$ or not.

▶ **First case: there exists $v \in f_\tau$ such that $\mathsf{decvar}(v) = x_{p+1}$.**
In this case, we have by definition: $f_{\tau'} = f_\tau \setminus \{v\} \cup \mathsf{sink}(v_d)$.

We start by pointing out that, for a decision gate v with $x_{p+1} = \mathsf{decvar}(v)$ and $d \in \mathsf{D}$, we have $\sigma_{x_{p+1}=d}(\mathsf{rel}(v)) = \{[x_{p+1} \leftarrow d]\} \times \mathsf{rel}(v_d) \times \mathsf{D}^{\mathsf{var}(v) \setminus (\{x_{p+1}\} \cup \mathsf{var}(v_d))}$. In other words, every tuple in the relation computed by v where $x_{p+1} = d$ is of the form $[x_{p+1} \leftarrow d] \times \tau' \times \sigma'$ where $\tau' \in \mathsf{rel}(v_d)$ and σ' is any tuple on domain D and variables set $\mathsf{var}(v) \setminus (\{x_{p+1}\} \cup \mathsf{var}(v_d))$. We can therefore write:

$$\sigma_{\tau'}(\mathsf{rel}_X(C)) = \sigma_{x_{p+1}=d}\left(\{\tau\} \times \mathsf{rel}(f_\tau) \times \mathsf{D}^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)}\right)$$

since x_{p+1} only appears in the frontier :

$$= \{\tau\} \times \mathsf{D}^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)} \times \sigma_{x_{p+1}=d}(\mathsf{rel}(v)) \times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w)$$

from the previous relation:

$$= \{\tau\} \times D^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)} \times \{[x_{p+1} \leftarrow d]\} \times \mathsf{rel}(v_d) \times \mathsf{D}^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup\mathsf{var}(v_d))} \times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w)$$

from Lemma 6.31:

$$= \{\tau\} \times D^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)} \times \{[x_{p+1} \leftarrow d]\} \times \bigtimes_{w\in\mathsf{sink}(v_d)} \mathsf{rel}(w) \times D^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup\mathsf{var}(v_d))} \times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w)$$

$$= \{\tau\} \times \{[x_{p+1} \leftarrow d]\} \times \mathsf{D}^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)} \times \mathsf{D}^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup\mathsf{var}(v_d))} \times \bigtimes_{w\in\mathsf{sink}(v_d)} \mathsf{rel}(w) \times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w)$$

$$= \{\tau'\} \times \bigtimes_{w\in f_{\tau'}} \mathsf{rel}(w) \times \mathsf{D}^{\{x_{p+2},\dots,x_n\}\setminus\mathsf{var}(f'_\tau)}$$

▶ **Second case: there are no $v \in f_\tau$ such that $\mathsf{decvar}(v) = x_{p+1}$.**

In this case, since the circuit is ordered, it means that $x_{p+1} \notin \mathsf{var}(f_\tau)$. Moreover $f_\tau = f_{\tau'}$. We can therefore write:

$$\sigma_{\tau'}(\mathsf{rel}_X(C)) = \{\tau\} \times \bigtimes_{w\in f_\tau} \mathsf{rel}(w) \times \sigma_{x_{p+1}=d}(\mathsf{D}^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)})$$

since x_{p+1} does not appear in the frontier or τ:

$$= \{\tau\} \times \bigtimes_{w\in f_\tau} \mathsf{rel}(w) \times \{[x_{p+1} \leftarrow d]\} \times \mathsf{D}^{\{x_{p+2},\dots,x_n\}\setminus\mathsf{var}(f_\tau)}$$

$$= \{\tau'\} \times \bigtimes_{w\in f_{\tau'}} \mathsf{rel}(w) \times \mathsf{D}^{\{x_{p+2},\dots,x_n\}\setminus\mathsf{var}(f_{\tau'})}$$

since $f_\tau = f_{\tau'}$.

Since the property is true for the empty prefix and inductively true, we conclude that it is true for any prefix τ. □

In order to be useful in practice, building and using the frontier of a prefix assignment τ cannot be too expensive. We formulate the following complexity statement:

▶ **Lemma 6.35**

Let τ be a prefix assignment over the set of variables set $X = \{x_1, \ldots, x_p\}$. We can compute f_τ in time $\mathcal{O}(|X|)$.

Proof. Let τ be a prefix assignment of size p. The frontier f_τ is built in a top-down fashion, by following the edges corresponding to the variable assignments in τ. For each variable x assigned by τ, we follow at most one edge from a decision gate v such that $\mathsf{decvar}(v) = x$ and the edge is labelled $\tau(x)$. This means p edges are followed for the assignments. Moreover, to get to v, we might have to follow edges from $\times$-gates. Since the variable sets underneath $\times$-gates are disjoint from one another, we have that the number of such edges is bounded by $|X|$. This implies the total cost of building the frontier f_τ is $\mathcal{O}(|X| + p) = \mathcal{O}(|X|)$. □

We are now ready to prove the main connection between frontiers and Lemma 6.6:

▶ **Lemma 6.36** (Oracle Complexity)
Let C be an ordered $\{\times, \mathsf{dec}\}$-circuit on variables set $X = \{x_1, \ldots, x_n\}$ such that $\mathsf{nrel}_C(v, d)$ and $\mathsf{var}(v)$ have been precomputed as defined in Lemma 6.28. Let τ be a prefix assignment of $\mathsf{D}^{\{x_1,\ldots,x_p\}}$.

Then, for every N, one can compute the smallest value $d \in \mathsf{D}$ such that $\#\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) \geqslant N$ and the value $\#\sigma_{x_{p+1} < d}(\sigma_\tau(\mathsf{rel}(C)))$ in time $\mathcal{O}(|X|^2 \mathsf{log}|X| + |X|\mathsf{log}(|\mathsf{D}|))$.

Proof. We start by building the frontier f_τ associated with the prefix assignment τ. From Lemma 6.35, we know this can be done in time $\mathcal{O}(|X|)$. By Lemma 6.34, we can rewrite:

$$\begin{aligned}\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) &= \sigma_{x_{p+1} \leqslant d}\left(\{\tau\} \times \mathsf{rel}(f_\tau) \times \mathsf{D}^{\{x_{p+1},\ldots,x_n\} \setminus \mathsf{var}(f_\tau)}\right) \\ &= \{\tau\} \times \sigma_{x_{p+1} \leqslant d}\left(\mathsf{rel}(f_\tau) \times \mathsf{D}^{\{x_{p+1},\ldots,x_n\} \setminus \mathsf{var}(f_\tau)}\right)\end{aligned}$$

There are now two possible outcomes: either there is a decision gate v labelled by variable x_{p+1} in f_τ or there is not. In the first case, since x_{p+1} only appears in the frontier, we have for every $d \in \mathsf{D}$:

$$\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) = \{\tau\} \times \sigma_{x_{p+1} \leqslant d}(\mathsf{rel}(v)) \times \bigtimes_{w \in f_\tau \setminus \{v\}} \mathsf{rel}(w) \times \mathsf{D}^{\{x_{p+2},\ldots,x_n\} \setminus \mathsf{var}(f_\tau)}$$

Therefore, we have:

$$\begin{aligned}\#\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) &= \#\sigma_{x_{p+1} \leqslant d}(\mathsf{rel}(v)) \times \prod_{w \in f_\tau \setminus \{v\}} \#\mathsf{rel}(w) \times |\mathsf{D}|^{|\{x_{p+2},\ldots,x_n\} \setminus \mathsf{var}(f_\tau)|} \\ &= \mathsf{nrel}_C(v, d) \times \prod_{w \in f_\tau \setminus \{v\}} \#\mathsf{rel}(w) \times |\mathsf{D}|^{|\{x_{p+2},\ldots,x_n\} \setminus \mathsf{var}(f_\tau)|} \\ &= \mathsf{nrel}_C(v, d) P\end{aligned}$$

where P is a product containing at most $|X|$ precomputed integers, all of them of size at most $|\mathsf{D}|^{|X|}$ and hence, it can be evaluated in time $\mathcal{O}(|X|^2\mathsf{log}|X|)$. Indeed, every value $\#\mathsf{rel}(w)$ for $w \in f_\tau \setminus \{v\}$ have been precomputed and can be accessed in constant time. Moreover, $\mathsf{var}(f_\tau) = \bigcup_{v \in f_\tau} \mathsf{var}(v)$. Since $\mathsf{var}(v)$ has been precomputed too, we can compute $e := |\{x_{p+2}, \dots, x_n\} \setminus \mathsf{var}(f_\tau)|$ in $\mathcal{O}(|X|)$ and then $|\mathsf{D}|^e$ in $\mathcal{O}(|X|^2\mathsf{log}|X|)$.

Now, observe that the smallest value d such that $\#\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) \geqslant N$ has to be the label of one ingoing edge of v. Indeed, if d does not label any ingoing edge of v, then $\mathsf{nrel}_C(v, d) = \mathsf{nrel}_C(v, d_0)$ where $d_0 < d$ is the greatest value smaller than d, which labels one ingoing edge of v (or 0 if no such d_0 exists). Hence, the smallest value d we are looking for has to be the label of one ingoing edge of v and is the smallest one among them such that $\mathsf{nrel}_C(v, d) \geqslant \frac{N}{P}$. Since P has been evaluated already, we can evaluate $\frac{N}{P}$ in time $\mathcal{O}(|X|^2\mathsf{log}|X|)$.

To find the smallest value d where $\mathsf{nrel}_C(v, d)$ is greater than $\frac{N}{P}$, it remains to do a binary search among the ingoing edges of v, which have already been sorted by increasing label. Comparing $\mathsf{nrel}_C(v, d)$ to $\frac{N}{P}$ can be done in $\mathcal{O}(|X|)$ for any d since the values do not exceed $|\mathsf{D}|^{|X|}$. There are at most $|\mathsf{D}|$ ingoing edge, meaning this binary search can be performed in time $\mathcal{O}(|X|\mathsf{log}|\mathsf{D}|)$. Therefore, finding the value d we are interested in has a total complexity of $\mathcal{O}(|X|^2\mathsf{log}|X| + |X|\mathsf{log}|\mathsf{D}|)$.

Now that we know the value $d \in \mathsf{D}$ we are looking for, it remains to compute $\#\sigma_{x_{p+1} < d}(\sigma_\tau(\mathsf{rel}(C)))$. This value is equal to $\#\sigma_{x_{p+1} \leqslant d'}(\sigma_\tau(\mathsf{rel}(C)))$ where d' is the value labelling the ingoing edge of v just before d. If no such value exists, then $\#\sigma_{x_{p+1} < d}(\sigma_\tau(\mathsf{rel}(C)))$ is 0. Otherwise, we can compute it as before in time $\mathcal{O}(|X|^2\mathsf{log}|X|)$.

In the second case, where there is no decision gate labelled by x_{p+1} in f_τ, we have that $x_{p+1} \notin \mathsf{var}(f_\tau)$ since the circuit is ordered. Hence, we can apply a similar reasoning to obtain:

$$\begin{aligned}\#\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) &= \#\sigma_{x_{p+1} \leqslant d}(\mathsf{D}^{\{x_{p+1}\}}) \cdot |\mathsf{D}|^{|\{x_{p+2}, \dots, x_n\} \setminus \mathsf{var}(f_\tau)|} \cdot \prod_{w \in f_\tau} \#\mathsf{rel}(w) \\ &= \mathsf{rank}(d) \times P\end{aligned}$$

where $\mathsf{rank}(d)$ is the number of elements smaller than d in D and P a product whose elements have been precomputed and which can thus be evaluated in time $\mathcal{O}(|X|^2\mathsf{log}|X|)$. Therefore, the smallest value $d \in \mathsf{D}$ such that $\#\sigma_{x_{p+1} \leqslant d}(\mathsf{D}^{\{x_{p+1}\}}) \geqslant N$ is the one whose rank is $\lceil \frac{N}{P} \rceil$. Moreover, $\#\sigma_{x_{p+1} < d}(\sigma_\tau(\mathsf{rel}(C))) = \mathsf{max}(0, (\lceil \frac{N}{P} \rceil - 1)P)$. □

In short, we can follow the edges in the circuit by choosing the correct edge from the precomputed values in nrel_C. A short visual example of the followed paths for different direct access tasks over the same annotated circuit is presented in Example 6.29. Notice how in the case where $k = 13$ (case **(b)** of the example), the fact that we meet a $\times$-gate implies that we follow both paths at once. At one point of the algorithm, a frontier containing both the gates for x_2 and x_3 exists. The values shown at the right of the reached decision gates show how the index of the searched tuples evolves during a run of the search algorithm.

The proof of Theorem 6.22 is now an easy corollary of Lemmas 6.6 and 6.36. Indeed, after having precomputed nrel_C and var using Lemma 6.28, we can answer direct access tasks using the oracle based algorithm from Lemma 6.6. Lemma 6.36 shows that these oracle accesses are, in fact, tractable in ordered circuits.

We conclude this section with a high level description of the direct access algorithm using pseudocode. The first procedure we need is the FIND-FRONTIER procedure from Algorithm 6.37. Given a circuit C and a prefix assignment τ of $\{x_1, \dots, x_p\}$, this procedure returns the frontier of τ by following every decision gate whose variable is assigned by τ and sinking through every

$\times$-gate, implementing the procedure described in Lemma 6.35. Procedure Count-Extensions then computes the number of tuples in $\mathsf{rel}(C)$ compatible with τ and such that $x_{p+1} \leqslant d$ for some domain value $d \in \mathsf{D}$. It uses the frontier of τ and the equality described in Lemma 6.36. Observe that one needs access to $\mathsf{nrel}_C(v, d)$, $\mathsf{var}(v)$ and $|\mathsf{rel}(v)|$ for this, which can be accessed from the annotations of the circuit described in the previous section.

The last function is the actual direct access procedure. The Next-Value (Algorithm 6.39) function takes a circuit, a prefix assignment τ of $x_1, \ldots, x_{p-1}$, an index k and finds the value on x_p of the k^{th} tuple of $\sigma_\tau(\mathsf{rel}(C))$. It does so by using calls to the Count-Extensions function and does a binary search over the domain values. We then implement the approach described in Lemma 6.6 in the Direct-Access function by iteratively constructing τ using Next-Value and also updating the index to take into account discarded tuples from $\mathsf{rel}(C)$.

Algorithm 6.37 Finding the frontier

```
1:  procedure Find-Frontier(C, τ)
2:      input: · C an ordered {×, dec}-circuit,
                · τ a partial assignment,
3:      output: the frontier in C for τ
4:      tmp ← {out(C)}
5:      F ← ∅
6:      while tmp ≠ ∅ do
7:          for v ∈ tmp do
8:              if v is a ⊥-gate then return ∅
9:              if v is a ×-gate then tmp ← (tmp \ {v}) ∪ input(v);
10:             if v is a decision-gate on x and τ(x) is defined then
11:                 if v has no incoming edge labelled by τ(x) then return ∅
12:                 (w, v) ← the incoming edge of v labelled by τ(x)
13:                 tmp ← (tmp \ {v}) ∪ {w}
14:             else
15:                 F ← F ∪ {v}
16:                 tmp ← tmp \ {v}
17:     return F
```

Algorithm 6.38 Counting the extensions of a prefix assignment

```
procedure Count-Extensions(C, τ)
    input: · C an ordered {×, dec}-circuit,
           · τ a partial assignment,
    output: the number of valid tuple extensions from τ in C
    F ← Find-Frontier(C,τ)
    Y ← {x_{p+2}, ..., x_n} \ ⋃_{v∈F} var(v)
    if F contains a decision-gate w on variable x_{p+1} then
        return |D|^{|Y|} × ∏_{w∈F\{v}} |rel(w)| × nrel_C(v, d)
    else
        return |D|^{|Y|} × ∏_{w∈F} |rel(w)| × rank(d)
```

Algorithm 6.39 Finding the next value to assign

procedure NEXT-VALUE(C, τ, k, m, M)
  input: · C an ordered $\{\times, \mathsf{dec}\}$-circuit,
    · τ a partial assignment,
    · the index to search for,
    · m the minimal value,
    · M the maximal value.
  output: the value for the next variable to assign in τ for index k
  if $m = M$ **then**
    if COUNT-EXTENSIONS(C,τ,d_m)$\geqslant k$ **then return** d_m **Else return** $\bot$
  else
    $mi = \lfloor \frac{m+M}{2} \rfloor$
    if COUNT-EXTENSIONS(C, τ, d_{mi}) $\geqslant k$ **then return** NEXT-VALUE(C, τ, k, m, mi)
    else return NEXT-VALUE(C, τ, k, mi, M)

Algorithm 6.40 Direct Access

procedure DIRECT-ACCESS(C, k)
  input: · C an ordered $\{\times, \mathsf{dec}\}$-circuit,
    · the index to search for.
  output: the k^{th} tuple in C
  $p \leftarrow 1$
  $\tau \leftarrow \langle\rangle$
  while $p \leqslant n$ **do**
    $d \leftarrow$ NEXT-VALUE($C, \tau, k, 1, |\mathsf{D}|$)
    if $d = \bot$ **then return** $\bot$
    $\tau \leftarrow \tau \times [x_p \leftarrow d]$
    $p \leftarrow p + 1$
    $i \leftarrow \mathsf{rank}(d)$ (that is, $d = d_i$)
    if $i > 1$ **then** $k \leftarrow k -$COUNT-EXTENSIONS(C, τ, d_{i-1})

▶ **Remark 6.41**
Observe that in NEXT-VALUE and Lemma 6.36, we implemented the oracle from Lemma 6.6 using a binary search on the ingoing edges of the decision gates, which is the origin of the $\mathcal{O}(\mathsf{log}|\mathsf{D}|)$ factor in the access time. As a matter of fact, the oracle call that we use boils down to solving a problem known as FindNext in *priority queues*: given values $v_1 < \cdots < v_p$ and N, find the smallest j such that $v_j \geqslant N$. Some data structures such as *van Emde Boas trees* [van75] support this operation in time $\mathsf{log}\ \mathsf{log}\ p$, which would allow us to improve the direct access to $\mathcal{O}(\mathsf{poly}(|Q|)\mathsf{log}\ \mathsf{log}\ |\mathsf{D}|)$ time. This would be traded for a small sacrifice in the preprocessing time (since inserting in van Emde Boas trees takes time $\mathcal{O}(\mathsf{log}\ \mathsf{log}\ |\mathsf{D}|)$) if we store the $\mathsf{nrel}_C(v, d)$ values in such data structures instead of ordered lists.

▶ **Remark 6.42**
You may also notice that our use of the FIND-FRONTIER function is not optimal. Indeed, each call to FIND-FRONTIER is dealt with independently while they are actually only updating a given frontier by iteratively adding new values in τ. Thus, in practice, we could get better performances by updating the frontier at each iteration of the loop in DIRECT-ACCESS instead of recomputing it from scratch.

6.4 Tractability of Queries for Direct Access

In this section, we connect the tractability results for direct access on ordered circuits from Section 6.3 with the algorithm presented in Chapter 5 to obtain tractability results concerning the complexity of direct access on signed join queries. Moreover, we show how our approach can be used to get direct access for signed conjunctive queries, that is, signed join queries with projected variables.

One of the key elements of Chapter 5 that we will use is the *binarisation* presented in Section 5.5. The order $\prec^b$, that was derived from the order $\prec$ over the variables X of the query, naturally induces an order $\prec^b_{\mathsf{lex}}$ on $\{0,1\}^{\widetilde{X}^b}$. For $\tau, \tau' \in [d]^X$, we have that $\tau \prec_{\mathsf{lex}} \tau'$ if, and only if, $\widetilde{\tau}^b \prec^b_{\mathsf{lex}} \widetilde{\tau'}^b$. This is because the natural ordering on $[b]$ can be seen as the lexicographical order on its bit-representation, starting from the most significant bit of $\tau(x)$ (which is $\widetilde{\tau}^b(x^b)$), to its least significant bit (which is $\widetilde{\tau}^b(x^1)$). We can formalise this is the following proposition:

▶ **Proposition 6.43**
Let Q be a join query on variables X, $\prec$ be an order on X and $\mathbf{D}$ a database for Q on domain $[d]$ for some $d \in \mathbb{N}$. Let $b = \lceil \mathsf{log}(d) \rceil$.

The function $\widetilde{\cdot}^b$ is an isomorphism between $\mathsf{ans}_{\widetilde{\mathbf{D}}^b}(\widetilde{Q}^b)$ and $\mathsf{ans}_{\mathbf{D}}(Q)$. It can be computed in time $\mathcal{O}(nb)$ such that, if τ is the k^{th} solution of $\widetilde{Q}^b$ over $\widetilde{\mathbf{D}}^b$ for the order $\prec^b_{\mathsf{lex}}$, then $\bar{\tau}^b$ is the k^{th} solution of Q over $\mathbf{D}$ for the order $\prec_{\mathsf{lex}}$, with $\bar{\cdot}^b$ being the inverse function of $\widetilde{\cdot}^b$.

A consequence of Proposition 6.43 is that, if we have direct access for the answers $\widetilde{Q}^b$ over a database $\widetilde{\mathbf{D}}^b$ for the order $\prec^b_{\mathsf{lex}}$ with preprocessing time $\mathcal{P}$ and access time $\mathcal{A}$, then we directly get direct access for Q over the database $\mathbf{D}$ with the same preprocessing time $\mathcal{P}$ and access time $\mathcal{A} + nb$. This is because we can perform the same preprocessing as for the binarised query and database, but to get the k^{th} answer, we have to compute the k^{th} answer τ of $\widetilde{Q}^b$ and return $\bar{\tau}^b$.

We will also use Theorem 5.22 as it shows that the binarisation operation does not change the width of the involved elimination orders.

6.4.1 Direct Access for Signed Join Queries

We first focus on signed join queries. We can show that the tractability results from Section 6.3 and Algorithm 5.9 can be connected in the following theorem:

▶ **Theorem 6.44**
Given a signed join query Q, an order $\prec$ on $\mathsf{var}(Q)$ and a database $\mathbf{D}$ on domain D, we can solve the direct access problem for $\prec_{\mathsf{lex}}$ with:

preprocessing time: $\tilde{\mathcal{O}}(|\mathbf{D}|^k \mathsf{poly}_k(|Q|))$; and

access time: $\mathcal{O}(\mathsf{poly}(|Q|) \cdot (\mathsf{log}|\mathsf{D}|)^3(\mathsf{loglog}|\mathsf{D}|))$

With k defined as follows:

- if Q does not contain any negative atom, then $k = \mathsf{fhtw}(\mathcal{H}(Q), \succ)$;
- otherwise $k = \mathsf{show}(\mathcal{H}(Q), \succ)$.

Proof. We start by constructing a $\prec_b$ ordered circuit computing $\mathsf{ans}_{\mathbf{D}}(\widetilde{Q}^b)$ using Theorem 5.28 for $b = \lceil \mathsf{log}|\mathsf{D}| \rceil$ on a domain of size 2 and with variable set $\widetilde{X}^b = \mathsf{var}(\widetilde{Q}^b)$.

We preprocess this circuit as in Theorem 6.22. Building and preprocessing the circuit constitutes the *preprocessing phase* of our direct access algorithm and can be executed in $\tilde{\mathcal{O}}(|\mathbf{D}|^k \mathsf{poly}_k(|Q|))$ by Theorem 5.28.

Now, to find the i^{th} tuple in $\mathsf{ans}_{\mathbf{D}}(Q)$, we simply find the i^{th} solution of $\mathsf{rel}(C)$ using the algorithm from Section 6.3, which is the i^{th} solution of $\mathsf{ans}_{\widetilde{\mathbf{D}}^b}(\widetilde{Q}^b)$. Moreover, by Proposition 6.43, we can reconstruct from it the i^{th} solution of $\mathsf{ans}_{\mathbf{D}}(Q)$ in time $\mathcal{O}(\mathsf{log}|\mathsf{D}|)$.

By Theorem 6.22, the access time is therefore $\mathcal{O}(\mathsf{poly}(|Q|) \cdot (\mathsf{log}|\mathsf{D}|)^3(\mathsf{loglog}|\mathsf{D}|))$, because the number of variables of C is now $\mathcal{O}(|\mathsf{var}(Q)|\mathsf{log}|\mathsf{D}|)$. □

The main difference between Theorem 6.44 and Theorem 6.19 is that the complexity given in Theorem 6.44 is polynomial in $|Q|$ for queries with bounded signed hyperorder width or bounded fractional hypertree width, which matches Theorems 6.15 and 6.17 in this case.

Suppose however that we are interested in signed *fractional* hyperorder width. In that case, we are not able to prove that our version of DPLL from Chapter 5 produces a circuit that is polynomial in $|Q|$. The key comes from Theorem 5.20, which tells us that a run of DPLL on the binarisation of Q produces a circuit of size $\tilde{\mathcal{O}}(2^m n|\mathbf{D}|^k)$. This then matches the complexity given in Theorems 6.15 and 6.17 for the preprocessing phase but has a better access time since the degree of $\mathsf{log}|\mathsf{D}|$ does not depend on $|Q|$ anymore. Formally, we get an improved version of the upper bound of Theorem 6.19 in Theorem 6.45. In particular, observe that even if the preprocessing time is exponential in the query size, this is not the case for the access time:

▶ **Theorem 6.45**
Given a signed join query Q, an order $\prec$ on $\mathsf{var}(Q)$ and a database $\mathbf{D}$ on domain D, we can solve the direct access problem for $\prec_{\mathsf{lex}}$ with:

preprocessing time: $\tilde{\mathcal{O}}(|\mathbf{D}|^{\mathsf{sfhow}(\mathcal{H}(Q),\succ)}2^{|\mathsf{atoms}(Q)(Q)|}\mathsf{poly}(|Q|))$; and

access time: $\mathcal{O}(\mathsf{poly}(|Q|)\cdot(\log|\mathsf{D}|)^3(\log\log|\mathsf{D}|))$.

6.4.2 Direct Access for (Signed) Conjunctive Queries

As we mentioned before, Theorem 6.44 allows us to recover the tractability of direct access for positive join queries with bounded fractional hypertree width. These tractability results were first proved in [Car+23; BCM22a] and restated here in Theorems 6.15 and 6.17.

However, [Car+23] (see also [BCM22b] which is the arXiv version of [BCM22a]) also generalises their algorithm to conjunctive queries, that is, join queries with projections.

The aim of this section is therefore to demonstrate the versatility of the circuit-based approach by showing how one can also handle quantifiers directly on the circuit:

▶ **Theorem 6.46**
Let C be a $\prec$-ordered $\{\times,\mathsf{dec}\}$-circuit on domain D, variable set $X=\{x_1,\dots,x_n\}$ such that $x_1\prec\cdots\prec x_n$ and $j\leqslant n$.

One can compute in time $\mathcal{O}(|C|\cdot\mathsf{poly}(n)\cdot\mathsf{polylog}|\mathsf{D}|)$ a $\prec$-ordered $\{\times,\mathsf{dec}\}$-circuit C' of size at most $|C|$ such that $\mathsf{rel}(C')=\mathsf{rel}(C)_{|\{x_1,\dots,x_j\}}$.

Proof. Let v be a decision gate on variable x_k with $k>j$. By definition, every decision gate in the circuit rooted at v tests a variable $y\in\{x_{k+1},\dots,x_n\}$. Therefore, we know that $\mathsf{rel}(v)\subseteq\mathsf{D}^Y$ with $Y\subseteq\{x_k,\dots,x_n\}$.

Moreover, one can compute $|\mathsf{rel}(v)|$ for every gate v of C in time $\mathcal{O}(|C|\cdot\mathsf{poly}(n)\cdot\mathsf{polylog}|\mathsf{D}|)$ as explained in Lemma 6.28. Hence, after this preprocessing, we can decide whether $\mathsf{rel}(v)$ is the empty relation in time $\mathcal{O}(1)$ by simply checking whether $|\mathsf{rel}(v)|=\mathsf{nrel}_C(v,d_0)\neq 0$ where d_0 is the largest element of D.

We construct C' by replacing every decision gate v on a variable x_k with $k>j$ by a constant gate $\top$ if $\mathsf{rel}(v)\neq\emptyset$ and $\bot$ otherwise. We clearly have that $|C'|\leqslant|C|$ and from what precedes, we can compute C' in $\mathcal{O}(|C|\cdot\mathsf{poly}(n)\cdot\mathsf{polylog}|\mathsf{D}|)$. Finally, it is straightforward to show by induction that every gate v' of C', which corresponds to a gate v of C, computes $\mathsf{rel}(C)_{|\{x_1,\dots,x_j\}}$, which concludes the proof. □

We can now use Theorem 6.46 to handle conjunctive queries by first using Theorem 5.13 on the underlying join query to obtain a $\prec$-circuit, and by then projecting the variables directly in the circuit.

This approach works only when the quantified variables are the largest variables in the circuit. This motivates the following definition:

▶ **Definition 6.47** (S-connexity of elimination orders)
Let $\mathcal{H} = (V, E)$ be a hypergraph.

An elimination order $(v_1, \ldots, v_n)$ of V is *S-connex* if, and only if, there exists a value i such that $\{v_i, \ldots, v_n\} = S$.

In other words, the elimination order *starts* by eliminating $V \setminus S$ and then proceeds to S.

Given a conjunctive query Q and an elimination order $\prec$ on $\mathsf{var}(Q)$, we say that the elimination is *free-connex* if it is a $\mathsf{free}(Q)$-connex elimination order of $\mathcal{H}(Q)$ where $\mathsf{free}(Q)$ are the free variables of Q[a].

[a] The notion of S-connexity already exists for tree decompositions. We use the same name here as the existence of an S-connex tree decomposition of (fractional) hypertree width k is equivalent to the existence of an S-connex elimination order of (fractional) hyperorder width k.

From this definition, we can directly show the following:

▶ **Theorem 6.48**
Given a signed conjunctive query $Q(Y)$, a free-connex order $\succ$ on $\mathsf{var}(Q)$ and a database $\mathbf{D}$ on domain D, we can solve the direct access problem for $\prec_{\mathsf{lex}}$ with:

preprocessing time: $\tilde{\mathcal{O}}(|\mathbf{D}|^k \mathsf{poly}_k(|Q|))$; and

access time: $\mathcal{O}(\mathsf{poly}(|Q|) \cdot (\mathsf{log}|\mathsf{D}|)^3 \mathsf{loglog}|\mathsf{D}|)$.

With k being computed as follows:

- if Q does not contain any negative atom, then $k = \mathsf{fhtw}(\mathcal{H}(Q), \succ)$;
- otherwise $k = \mathsf{show}(\mathcal{H}(Q), \succ)$.

Proof. By running $\mathsf{DPLL}(\widetilde{Q}^b, \langle\rangle, \widetilde{\mathbf{D}}^b, \succ^b)$ for $b = \lceil \mathsf{log}|\mathsf{D}| \rceil$, one obtains a $\prec^b$-ordered circuit computing $\mathsf{ans}_{\widetilde{\mathbf{D}}^b}(\widetilde{Q}^b)$. The size of this circuit is $\tilde{\mathcal{O}}(|\mathbf{D}|^k \mathsf{poly}_k(|Q|))$ by Theorem 5.13.

We now use the fact that $\succ^b$ is free-connex, which implies that $\succ^b$ is of the form $z_1^b \succ \cdots \succ z_1^1 \succ \cdots \succ z_n^b \succ \cdots \succ z_n^1$, and that there exists j such that $\{z_j, \ldots, z_n\} = \mathsf{free}(Q)$. Therefore, by Theorem 6.46, we can construct a $\prec^b$-ordered $\{\times, \mathsf{dec}\}$-circuit of size at most $\mathcal{O}(|\mathbf{D}|^k \mathsf{poly}_k(|Q|))$ computing $\mathsf{ans}(\widetilde{Q}^b)[\widetilde{\mathbf{D}}^b]_{|\widetilde{\mathsf{free}(Q)}^b}$, which concludes the proof using Theorem 6.22. ◻

We observe that our notion of free-connex elimination orders for Q is akin that of [BCM22b], with one key difference: in [BCM22b], it is allowed to only specify a *preorder* on $\mathsf{free}(Q)$. The complexity of the algorithm is then stated with the best possible compatible ordering.

This would also be possible in our framework. In our case, Theorem 6.48 builds the structure for direct access for $\prec_{\mathsf{lex}}$ when $\prec$ is free-connex, so Theorem 6.48 proves the same tractability result as [BCM22b], with the same complexity.

Finally, one may also observe that there is another way of recovering the result of Theorem 6.48. We could modify DPLL so that, when only projected variables remain, instead of compiling, it calls an efficient join algorithm. This algorithm could then decide whether the answer set of the query is empty or not and return either $\top$ or $\bot$ in the circuit depending on the answer. This

approach could be more efficient because it is able to exploit more complex measures such as the *submodular width* of the resulting hypergraph, using a join algorithm such as PANDA [ANS17a].

6.5 Negative Join Queries and SAT

One particularly interesting application of this result is its application to *negative* join queries, that is, join queries where *every* atom is negated. Tractability results for negative join queries have previously been established in the literature. One of the first results in this direction was the study of β-acyclic negative join queries. It was shown in [Bra12] that the class of negative conjunctive queries where evaluation can be done in time linear in the size of the database is exactly the class of β-acyclic conjunctive queries. A slight generalisation of this result to acyclic signed join queries can be found in Brault-Baron's thesis [Bra13]. This result was first generalised for counting in [BCM15] and, more recently, to queries with more complex aggregation in [Zha+24]. A generalisation of β-acyclic queries, namely negative join queries with bounded *nest set* width, was proposed by Lanzinger [Lan23], where evaluation of such queries was shown to be tractable.

Another interesting application of our result consists in offering new tractability results for aggregation problems in SAT solving. Indeed, the SAT problem inputs are CNF (Conjunctive Normal Form) formulas, which can be seen as a particular case of negative join queries. Indeed, a CNF formula F with m clauses can be directly transformed into a negative join query Q_F with m atoms having the same hypergraph, and a database $\mathbf{D}_F$ on domain $\{0,1\}$ of size at most m such that $\mathsf{ans}_{\mathbf{D}_F}(Q_F)$ is the set of satisfying assignments of F. In other words, a clause of the CNF can be seen as the negation of a relation containing exactly one tuple. For example, the formula $F = x \vee y \vee \neg z$ can be seen as the query $Q_F :\text{-}\ \neg\mathsf{R}(x,y,z)$, where R only contains the tuple $\langle x \leftarrow 0, y \leftarrow 0, z \leftarrow 1\rangle$. Therefore, any polynomial-time algorithm in *combined complexity* on negative join queries directly transfers to CNF formulas, where many tractability results for SAT and #SAT are known when the hypergraph of the input CNF is restricted [OPS13; Bov+15; PSS16; SS13; SS10; STV14] (see [Cap16] for a complete survey).

We now show that Theorem 6.44 generalises many of these results because most of the hypergraph families known to give tractability results for negative join queries can be shown to have bounded signed hyperorder width. We first study the notion of signed hyperorder width restricted to negative join queries and compare it to existing measures from literature.

6.5.1 Hyperorder Width for Negative Join Queries

If a join query is negative, then we can consider its signed hypergraph to be simply the hypergraph induced by its negative atoms. Thus, the notion of signed hyperorder width can be simplified without the separation between the positive and negative edges. We use the following definitions:

▶ **Definition 6.49** (β-hyperorder width)
Let $\mathcal{H} = (V, E)$ be a hypergraph and $\prec$ be an order on V.

The *β-hyperorder width β-$\mathsf{how}(\mathcal{H}, \prec)$ of $\prec$ for $\mathcal{H}$* is defined as $\mathsf{max}_{\mathcal{H}' \subseteq \mathcal{H}} \mathsf{how}(\mathcal{H}', \prec)$.

The *β-hyperorder width β-$\mathsf{how}(\mathcal{H})$ of $\mathcal{H}$* is defined as the width of the *best* possible elimination order, that is, $\beta\text{-}\mathsf{how}(\mathcal{H}) = \mathsf{min}_{\prec} \beta\text{-}\mathsf{how}(\mathcal{H}, \prec)$.

Similarly, the *β-fractional hyperorder width* of an order $\prec$ and of a hypergraph (denoted

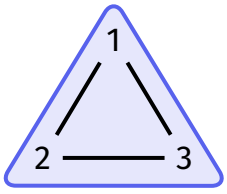
β-fhow$(\mathcal{H}, \prec)$ and β-fhow$(\mathcal{H})$) can be defined by simply replacing how$(\cdot)$ by fhow$(\cdot)$ in the previous definitions.

We now turn to comparing the notion of β-hyperorder width with other measures known from the literature. The definition of the β-hyperorder width can be seen as a hereditary closure of generalised hypertree width. Indeed, it is a well known fact that hypertree width is not a hereditary measure. That is, a subhypergraph may have a greater hypertree width than the hypergraph itself, as shown in Example 6.50.

▶ **Example 6.50**

Consider the following hypergraph (represented on the right), consisting of a triangle on vertices $\{1, 2, 3\}$ together with a hyperedge $\{1, 2, 3\}$.

This hypergraph is α-acyclic, and thus has hypertree width 1. However, if we consider the subhypergraph consisting of just the triangle, this subhypergraph has a fractional hypertree width of $3/2$ and a generalised hypertree width of 2.



The traditional way of working around the fact that fractional hypertree width is not a hereditary measure, has been to take the largest width over every subhypergraph. This led to the definition of the *β-fractional hypertree width* of $\mathcal{H}$, denoted β-fhtw$(\mathcal{H})$. This measure is defined as β-fhtw$(\mathcal{H}) = \max_{\mathcal{H}' \subseteq \mathcal{H}}$fhtw$(\mathcal{H}')$ [GR04]. The *β-hypertree width* β-htw$(\mathcal{H})$ is defined in a similar way, by replacing fhtw$(\cdot)$ by htw$(\cdot)$.

If we use the ordered characterisation of fhtw$(\mathcal{H}')$ in this definition, we then obtain the following equivalent way of defining fractional β-hypertree width: β-fhtw$(\mathcal{H}) = \max_{\mathcal{H}' \subseteq \mathcal{H}} \min_{\prec}$fhow$(\mathcal{H}', \prec)$. Now recall that β-fhow$(\mathcal{H}) = \min_{\prec} \max_{\mathcal{H}' \subseteq \mathcal{H}}$fhow$(\mathcal{H}', \prec)$. Hence, the difference between β-fhtw$(\mathcal{H})$ and β-fhow$(\mathcal{H})$ boils down to inverting the min and the max in the definition. It is easy to then notice that β-fhtw$(\mathcal{H}) \leqslant \beta$-fhow$(\mathcal{H})$ and β-htw$(\mathcal{H}) \leqslant \beta$-how$(\mathcal{H})$ for every $\mathcal{H}$ (see Theorem 6.51 for details). The main advantage of β-fractional hyperorder width is that it comes with a natural notion of decomposition — the best elimination order $\prec$ — which can be used algorithmically. For β-fhtw$(\cdot)$, no such decomposition that can be used algorithmically has yet been found.

The case where β-fhtw$(\mathcal{H}) = 1$, also known as β-acyclicity, is the only one for which tractability results are known. For example, SAT [OPS13], #SAT or #CQ for β-acyclic instances [Cap17; BCM15]. This is due to the fact that, in this case, an order-based characterisation is known. The elimination order is based on the notion of *nest points*. In a hypergraph $\mathcal{H} = (V, E)$, a *nest point* is a vertex $v \in V$ such that $E(v)$ can be ordered by inclusion, that is, $E(v) = \{e_1, \dots, e_p\}$ with $e_1 \subseteq \cdots \subseteq e_p$. A *$\beta$-elimination order $(v_1, \dots, v_n)$ for $\mathcal{H}$* is an ordering of V such that for every $i \leqslant n$, v_i is a nest point of $\mathcal{H} \setminus \{v_1, \dots, v_{i-1}\}$. We show in Theorem 6.51 that β-elimination orders correspond exactly to elimination orders having β-hyperorder width 1, hence proving that our width notion generalises β-acyclicity.

We actually prove a more general result: the notion of β-acyclicity has been recently generalised

by Lanzinger [Lan23], by using a notion called *nest sets*. A set of vertices $S \subseteq V$ is a *nest set of* $\mathcal{H}$ if $\{e \setminus S \mid e \in E, e \cap S \neq \emptyset\}$ can be ordered by inclusion. A *nest set elimination order* is therefore a list $\Pi = (S_1, \dots, S_p)$ such that:

- $\bigcup_{i=1}^{p} S_i = V$;
- $S_i \cap S_j = \emptyset$; and
- S_i is a nest set of $\mathcal{H} \setminus \bigcup_{j<i} S_j$.

The width of a nest set elimination order is defined as $\mathsf{nsw}(\mathcal{H}, \Pi) = \mathsf{max}_i |S_i|$ and the *nest set width* $\mathsf{nsw}(\mathcal{H})$ *of* $\mathcal{H}$ is defined to be the smallest possible width of a nest set elimination order of $\mathcal{H}$. As it turns out, our notion of width generalises the notion of nest set width; that is, we have $\beta\text{-}\mathsf{how}(\mathcal{H}) \leqslant \mathsf{nsw}(\mathcal{H})$. More particularly, any order $\prec$ obtained from a nest set elimination order $\Pi = (S_1, \dots, S_p)$ by ordering each S_i arbitrarily verifies $\mathsf{nsw}(\mathcal{H}, \Pi) \geqslant \beta\text{-}\mathsf{how}(\mathcal{H}, \prec)$.

The above discussion can be formalised in the following theorem:

▶ **Theorem 6.51**
For every hypergraph $\mathcal{H} = (V, E)$, we have:

$$\beta\text{-}\mathsf{htw}(\mathcal{H}) \leqslant \beta\text{-}\mathsf{how}(\mathcal{H}) \leqslant \mathsf{nsw}(\mathcal{H}) \ .$$

In particular, $\mathcal{H}$ is β-acyclic if and only if $\beta\text{-}\mathsf{how}(\mathcal{H}) = 1$.

The proof mainly follows from Lemma 6.52. Intuitively, this lemma says that, if S is a nest set of $\mathcal{H}$ of size k, then iteratively removing the vertices of S in $\mathcal{H}$ implies that the introduced edges can always be covered by k edges from the original hypergraph. This is because they are made of some of the vertices from S (at most k of them) and some other vertices that are all covered by the maximal edge covering $\{e \setminus S \mid e \cap S \neq \emptyset\}$.

▶ **Lemma 6.52**
Let $\mathcal{H} = (V, E)$ be a hypergraph and S be a nest set of $\mathcal{H}$ of size k. We let f be the maximal element (for inclusion) of $\{e \setminus S \mid e \in E, e \cap S \neq \emptyset\}$, which exists by definition and $(s_1, \dots, s_k)$ an ordering of S.

For every $i \leqslant k$ and edge e of $\mathcal{H}/s_1/\dots/s_i$, either:

- $e \cap S \neq \emptyset$ and $e \subseteq f \cup S$; or
- $e \cap S = \emptyset$ and e is an edge of $\mathcal{H} \setminus \{s_1, \dots, s_i\}$.

Proof. We prove this lemma by induction on i.

For $i = 0$, it is clear since if $e \cap S \neq \emptyset$, then $e \setminus S \subseteq f$ by definition of f. Hence $e \subseteq f \cup S$.

Now, assuming the hypothesis holds for some i, let $\mathcal{H}_i = \mathcal{H}/s_1/\dots/s_i$ and $\mathcal{H}_{i+1} = \mathcal{H}_i/s_{i+1}$. By definition, the edges of $\mathcal{H}_{i+1}$ are: (1) the edges of $\mathcal{H}_i$ without the vertex s_{i+1}; or (2) the additional edge $\mathcal{N}^*_{\mathcal{H}_i}(s_{i+1})$. Let e be an edge of $\mathcal{H}_{i+1}$ that is not $\mathcal{N}^*_{\mathcal{H}_i}(s_{i+1})$. We have two cases: either e is already in $\mathcal{H}_i$, in which case the induction hypothesis still holds, or e is not in $\mathcal{H}_i$, which means that $e = e' \setminus \{s_{i+1}\}$ for some edge e' of $\mathcal{H}_i$ containing s_{i+1}. In particular, $s_{i+1} \in e' \cap S$ and then $e' \cap S \neq \emptyset$. By induction, we have $e' \subseteq f \cup S$ and then $e = e' \setminus \{s_{i+1}\} \subseteq f \cup S$ and the

induction hypothesis follows.

Finally, assume $e = \mathcal{N}^*_{\mathcal{H}_i}(s_{i+1})$, that is, e is obtained by taking the union of every e' in $\mathcal{H}_i$ where s_{i+1} is in e' and removing s_{i+1}. Then $e' \cap s_{i+1} \neq \emptyset$. By induction again, $e' \subseteq f \cup S$ hence $\mathcal{N}^*_{\mathcal{H}_i}(s_{i+1}) \subseteq f \cup S$ and the induction hypothesis follows. □

Lemma 6.52 then allows us to formally prove Theorem 6.51.

Proof of Theorem 6.51. The inequality $\beta\text{-}\mathsf{htw}(\mathcal{H}) \leqslant \beta\text{-}\mathsf{how}(\mathcal{H})$ is straightforward using:

- $\beta\text{-}\mathsf{htw}(\mathcal{H}) = \mathsf{max}_{\mathcal{H}' \subseteq \mathcal{H}} \mathsf{min}_{\prec} \mathsf{how}(\mathcal{H}', \prec)$
- $\beta\text{-}\mathsf{how}(\mathcal{H}) = \mathsf{min}_{\prec} \mathsf{max}_{\mathcal{H}' \subseteq \mathcal{H}} \mathsf{how}(\mathcal{H}', \prec)$

Indeed, let $\prec_0$ be an elimination order that is minimal for $\beta\text{-}\mathsf{how}(\mathcal{H}, \prec)$. By definition, for $\mathcal{H}' \subseteq \mathcal{H}$, $\mathsf{how}(\mathcal{H}', \prec_0) \geqslant \mathsf{min}_{\prec} \mathsf{how}(\mathcal{H}', \prec)$. Therefore,

$$\beta\text{-}\mathsf{how}(\mathcal{H}) = \mathsf{max}_{\mathcal{H}' \subseteq \mathcal{H}} \mathsf{how}(\mathcal{H}', \prec_0) \geqslant \mathsf{max}_{\mathcal{H}' \subseteq \mathcal{H}} \mathsf{min}_{\prec} \mathsf{how}(\mathcal{H}', \prec) = \beta\text{-}\mathsf{htw}(\mathcal{H}) \ .$$

We now prove $\beta\text{-}\mathsf{how}(\mathcal{H}) \leqslant \mathsf{nsw}(\mathcal{H})$. Let $k = \mathsf{nsw}(\mathcal{H})$ and $\Pi = (S_1, \ldots, S_p)$ a nest set elimination order of $\mathcal{H}$ of width k, that is, for every i, $|S_i| \leqslant k$. Let $\prec$ be an order on $V = (v_1, \ldots, v_n)$ with $v_1 \prec \cdots \prec v_n$, obtained from Π by ordering each S_i arbitrarily, that is, if $x \in S_i$ and $y \in S_j$ with $i < j$, we require that $x \prec y$.

We claim that $\beta\text{-}\mathsf{how}(\mathcal{H}, \prec) \leqslant \mathsf{nsw}(\mathcal{H}, \Pi)$. First of all, we observe that, if $(S_1, \ldots, S_p)$ is a nest set elimination order for $\mathcal{H}$, then it is also a nest set elimination order for every $\mathcal{H}' \subseteq \mathcal{H}$, which is formally proven in [Lan23][a]. Consequently, it is enough to prove that $\mathsf{how}(\mathcal{H}, \prec) \leqslant k$. Indeed, if we prove this, then we know that, for every hypergraph $\mathcal{H}' \subseteq \mathcal{H}$, $\prec$ is also a nest set elimination order for $\mathcal{H}'$, and thus, $\mathsf{how}(\mathcal{H}', \prec) \leqslant k$ too.

This follows from Lemma 6.52. Indeed, let $(v_1, \ldots, v_t)$ be the prefix of $(v_1, \ldots, v_n)$ such that $S_1 = \{v_1, \ldots, v_t\}$. By Lemma 6.52, when v_{i+1} is removed from $\mathcal{H}_i^{\prec} = \mathcal{H}/v_1/\ldots/v_i$, then $N_{i+1} = \mathcal{N}_{\mathcal{H}_i}(v_{i+1})$ is included in $f \cup S_1$ since $N_{i+1} \cap S_1 \neq \emptyset$ (both sets contain v_{i+1}). Therefore, N_{i+1} is covered by at most t edges: f – which contains at least one element of S_1 – plus at most one edge for each remaining element of S_1. Hence, up to the removing of v_t, the hyperorder width of $\prec$ is at most $t \leqslant k$. Now, when removing $(v_1, \ldots, v_t)$ from $\mathcal{H}$, by Lemma 6.52 again, $\mathcal{H}_t^{\prec} = \mathcal{H} \setminus \{v_1, \ldots, v_t\}$ since no edge of $\mathcal{H}_t^{\prec}$ has a non-empty intersection with S_1. It follows that S_2 is a nest set of $\mathcal{H}_t^{\prec}$ and we can remove it in a similar way to S_1 and so on. Hence $\beta\text{-}\mathsf{how}(\mathcal{H}, \prec) \leqslant k = \mathsf{nsw}(\mathcal{H}, \Pi)$ which settles the inequality stated in the theorem.

This directly implies that $\mathcal{H}$ is β-acyclic if and only if $\beta\text{-}\mathsf{how}(\mathcal{H}) = 1$. Indeed, if $\mathcal{H}$ is β-acyclic, then $\mathsf{nsw}(\mathcal{H}) = 1$ (the definition of nest set width elimination order of width 1 directly corresponds to the definition of β-elimination orders) and $\beta\text{-}\mathsf{how}(\mathcal{H}) \leqslant \mathsf{nsw}(\mathcal{H}) = 1$ by the previously established bound. □

[a] Lemma 4 of [Lan23] establishes the result for a *connected subhypergraph* of $\mathcal{H}$ but the same proof works for non-connected subhypergraphs.

The goal of this section (or of [Cap+25]) is not to give a thorough analysis of β-fractional hyperorder width so this leaves for future research several related open questions.

We observe that we do not know the exact complexity of computing or approximating the β-fractional hyperorder width of an input hypergraph $\mathcal{H}$. It is very likely hard to compute exactly since it is not too difficult to observe that when $\mathcal{H}$ is a graph, $\beta\text{-}\mathsf{fhow}(\mathcal{H})$ is "sandwiched" between the half of the treewidth of $\mathcal{H}$ and the treewidth of $\mathcal{H}$ itself and it is known that treewidth is NP-hard to compute [Bod93]. This does not rule out the possibility that deciding whether $\beta\text{-}\mathsf{how}(\mathcal{H}) \leqslant k$ is tractable for every k as it is the case for treewidth [Bod93], but we observe that deciding whether $\mathsf{fhow}(\mathcal{H}) \leqslant 2$ is known to be NP-hard [FGP18]. We also leave open many questions concerning how β-fractional hyperorder width compares with other widths such as

(incidence) treewidth, (incidence) cliquewidth or MIM-width. For these width measures, #SAT, a problem close to computing the number of answers in signed join queries, is known to be tractable (see [Cap16] for a survey). We leave open the most fundamental question of comparing the respective powers of β-fhtw$(\cdot)$ and β-fhow$(\cdot)$:

▶ **Open question 6.53**
Does there exist a family $(\mathcal{H}_n)_{n\in\mathbb{N}}$ of hypergraphs such that there exists $k \in \mathbb{N}$ such that for every n, β-fhtw$(\mathcal{H}_n) \leqslant k$ while β-fhow$(\mathcal{H}_n)$ is unbounded?

One may wonder why the definition of β-hyperorder width has not appeared earlier in the literature, as it just boils down to swapping a min and a max in the definition of β-hypertree width while enabling an easier algorithmic treatment. We argue that the expression of hypertree width in terms of elimination orders – which is not the widespread way of working with this width in previous literature – is necessary to make this definition interesting. Indeed, if one swaps the min and max in the traditional definition of β-hypertree width, we get the following definition: β-htw$'(\mathcal{H}, T) = \min_T \max_{\mathcal{H}'\subseteq\mathcal{H}} \text{htw}(\mathcal{H}', T)$ where T runs over every tree decomposition of $\mathcal{H}$ and hence is valid for every $\mathcal{H}' \subseteq \mathcal{H}$ since, as every edge of $\mathcal{H}$ is covered by T, so are the edges of $\mathcal{H}'$. This definition, while being obtained in the same way as β-how$(\cdot)$, is not really interesting however because it does not generalise the notion of β-acyclicity:

▶ **Proposition 6.54**
There exists a family of β-acyclic hypergraphs $(\mathcal{H}_n)$ such that for every $n \in \mathbb{N}$, β-htw$'(\mathcal{H}_n) = n$.

Proof. Consider the hypergraph $\mathcal{H}_n$ whose vertex set is $[n]$ and edges are $\{0, i\}$ for $i > 0$ and $[n]$. That is $\mathcal{H}_n$ is a star centered in 0 and additionnally has an edge containing every vertex. $\mathcal{H}_n$ is clearly β-acyclic (any elimination order that ends with 0 is a β-elimination order) but we claim that β-htw$'(\mathcal{H}_n) = n$. Indeed, let T be a tree decomposition for $\mathcal{H}_n$. By definition, it contains a bag that covers $[n]$. Now consider the subhypergraph $\mathcal{H}'_n$ of $\mathcal{H}_n$ obtained by removing the edge $[n]$. The hypertree width of T with respect to $\mathcal{H}'_n$ is n since one needs the edge $\{i, 0\}$ for every i to cover vertex i in the bag $[n]$ since i appears only in this edge. □

6.5.2 Applications

In the previous section, we have shown that β-hyperorder width generalises β-acyclicity and nest set width. Therefore, Theorem 6.44 can be used to show that direct access is tractable for the class of queries with bounded nest set width. In particular, counting the number of answers is tractable for this class, a question left open in [Lan23]:

▶ **Theorem 6.55**
Let Q be a negative join query of bounded nest set width and $\mathbf{D}$ be a database. Then we can

compute $|\mathsf{ans}_{\mathbf{D}}(Q)|$ in polynomial time.

Proof. Let k be the nest set width of Q. It was proved in [Lan23] that a nest set elimination order for Q can be found in time $2^{\mathcal{O}(k^2)}\mathsf{poly}(|\mathcal{H}(Q)|)$, which induces an elimination order $\prec$ for $\mathcal{H}(Q)$ of β-hyperoder width k.

Now, using our version of DPLL on this order and on the binarised form of Q, we can construct a circuit computing $\mathsf{ans}_{\mathbf{D}}(Q)$ in time $\tilde{\mathcal{O}}((\mathsf{poly}(\mathcal{H}(Q))|\mathbf{D}|)^k)$ and extract $|\mathsf{ans}_{\mathbf{D}}(Q)|$ from it using Lemma 6.28 in time $\tilde{\mathcal{O}}((\mathsf{poly}(\mathcal{H}(Q))|\mathbf{D}|)^k)$. □

Similarly, our result can directly be applied to #SAT, the problem of counting the number of satisfying assignments of a CNF formula. Therefore, Theorem 6.44 generalises both [Cap17] and [BCM15] by providing a compilation algorithm for β-acyclic queries to any domain size and to the more general measure of β-hyperorder width. It also shows that not only counting is tractable but also the more general direct access problem.

Figure 6.56 summarises our contributions for join queries with negations and summarises how our contribution is located in the landscape of known tractability results. The two left-most columns of the figure are the contributions presented in this chapter (originally from [Cap+25]) derived from Theorem 6.44. The right-most column is known from [BCM22a] but can be recovered in our framework. A complete presentation of the results stated in this figure can be found in [Cap16].

6.6 Conclusion and Future Work

This chapter has been devoted to proving new tractability results concerning direct access for answers of signed conjunctive queries. In particular, we have introduced a framework unifying the positive and the signed case using a factorised representation of the answer set of the query that we first introduced in Chapter 5. Our complexity bounds, when restricted to the positive case, match the existing optimal ones and we have proved that our algorithm remains optimal for signed join queries without self-joins (in terms of data complexity). This approach opens many new research avenues.

In particular, we believe that the circuit representation that we use is also promising for answering different kinds of aggregation tasks and hence generalising existing results on conjunctive queries to the case of signed conjunctive queries. For example, we believe that FAQ and AJAR queries [ANR16; JPR16] could be solved using this data structure. Indeed, it seems possible to annotate the circuit with semi-ring elements and then project them out in a similar fashion as Theorem 6.46. Similarly, we believe that the framework of [ECK24] for solving direct access tasks on conjunctive queries with aggregation operators may be generalised to the class of ordered $\{\times, \mathsf{dec}\}$-circuits.

Finally, contrary to the positive case, we do not yet know what is the complexity of solving direct access tasks on signed join queries with self-joins. We know that self-joins can only make things easier and have exhibited an example at the end of Section 6.2.3 where having a self-join between the positive and the negative part leads to drastic improvement in the preprocessing complexity. However, it is still unknown whether there are other more subtle cases where self-joins help, as it was shown for enumeration [CS23].

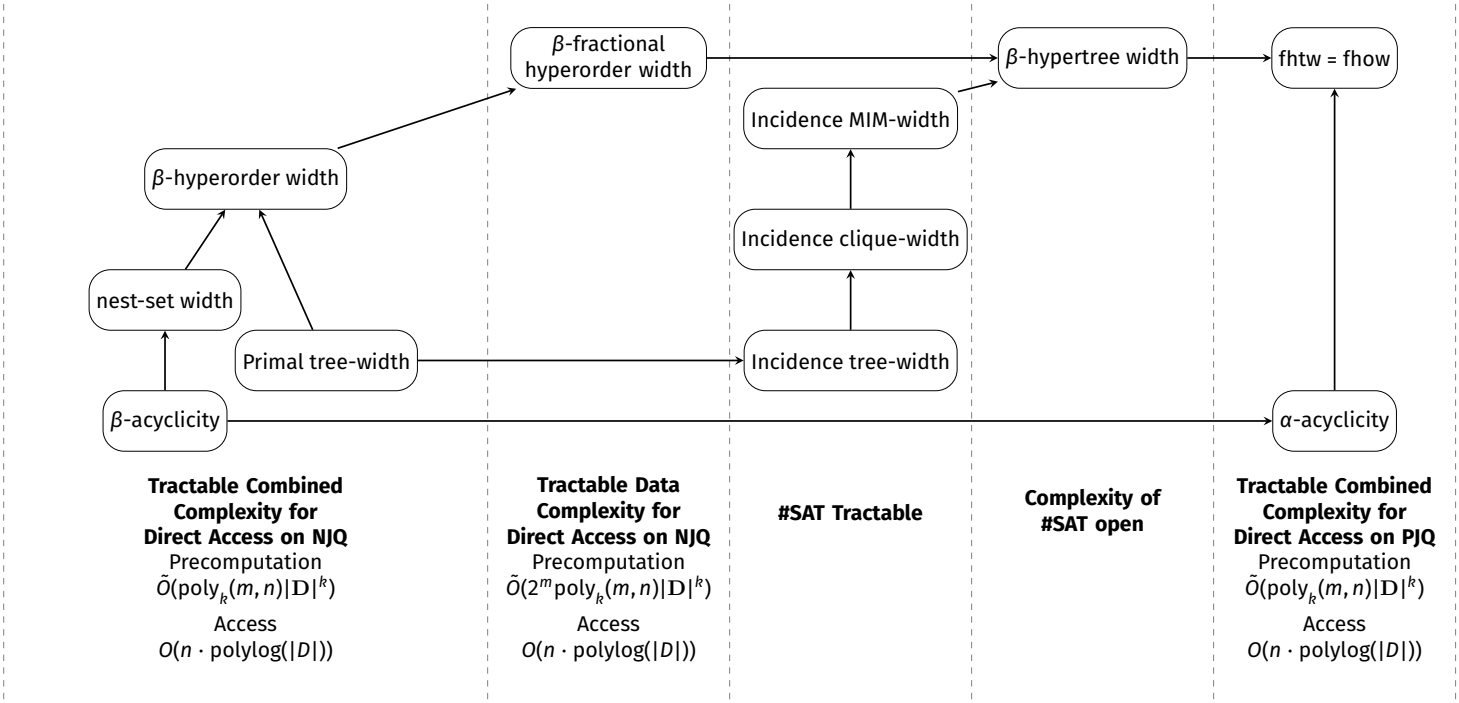


Figure 6.56: Landscape of hypergraph measures and known inclusions with tractability results for direct access on negative join queries (NJQ), direct access on positive join queries (PJQ), and #SAT on CNF formulas.

Here, n is the number of variables, m the number of atoms, $\mathbf{D}$ the database, D the domain and k the width measure ($k = 1$ for α- and β-acyclicity). In the case of CNF formulas, m stands for the number of clauses, the size of the database is at most m and the domain is $\{0, 1\}$. An arrow between two classes indicates inclusion for a fixed k.

All tractability results for direct access are consequences of Theorem 6.48 applied to either fractional hyperorder width in the positive case or β-hyperorder width in the negative case. The β-fractional hyperorder width is a consequence of Theorems 6.16 and 6.18. Comparison between β-hyperorder width, nest-set width and β-hypertree width is a consequence of Theorem 6.51.

Tractability of #SAT for MIM-width is from [STV14], where most inclusions below can be found.

The inclusion between MIM-width and β-hypertree width can be found in [Cap16].

Current chapter references

[ANR16] Mahmoud **Abo Khamis**, Hung Q **Ngo**, and Atri **Rudra**. *FAQ: questions asked frequently*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2016, pp. 13–28. doi 10.1145/2902251.2902280.

[ANS17a] Mahmoud **Abo Khamis**, Hung Q **Ngo**, and Dan **Suciu**. *What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another?* In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. May 2017, pp. 429–444. doi 10.1145/3034786.3056105.

[BCM15] Johann **Brault-Baron**, Florent **Capelli**, and Stefan **Mengel**. *Understanding Model Counting for beta-acyclic CNF-formulas*. In *32nd International Symposium on Theoretical Aspects of Computer Science*. Vol. 30. LIPIcs. Schloss Dagstuhl, Feb. 2015, pp. 143–156. doi 10.4230/LIPIcs.STACS.2015.143.

[BCM22a] Karl **Bringmann**, Nofar **Carmeli**, and Stefan **Mengel**. *Tight Fine-Grained Bounds for Direct Access on Join Queries*. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid **Libkin** and Pablo **Barceló**. PODS '22. Association for Computing Machinery, June 2022, pp. 427–436. doi 10.1145/3517804.3526234.

[BCM22b] Karl **Bringmann**, Nofar **Carmeli**, and Stefan **Mengel**. *Tight Fine-Grained Bounds for Direct Access on Join Queries*. Jan. 2022. arXiv 2201.02401.

[Bod93] Hans L. **Bodlaender**. *A Tourist Guide through Treewidth*. In *Acta Cybernetica* 11.1-2 (Jan. 1993), pp. 1–21. https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3417.

[Bov+15] Simone **Bova**, Florent **Capelli**, Stefan **Mengel**, and Friedrich **Slivovsky**. *On Compiling CNFs into Structured Deterministic DNNFs*. In *Theory and Applications of Satisfiability Testing*. Lecture Notes in Computer Science. Springer International Publishing, Sept. 2015, pp. 199–214. doi 10.1007/978-3-319-24318-4_15.

[Bra12] Johann **Brault-Baron**. *A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic*. In *Computer Science Logic - 26th International Workshop/21st Annual Conference of the EACSL, September 3-6, 2012, Fontainebleau, France*. Ed. by Patrick **Cégielski** and Arnaud **Durand**. Vol. 16. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Sept. 2012, pp. 137–151. doi 10.4230/LIPIcs.CSL.2012.137.

[Bra13] Johann **Brault-Baron**. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis. Université de Caen, Apr. 2013.

[Bra17] Johann **Brault-Baron**. *Hypergraph Acyclicity Revisited*. In *ACM Computing Surveys* 49.3 (Sept. 2017), pp. 1–26. doi 10.1145/2983573.

[Cap+25] Florent **Capelli**, Nofar **Carmeli**, Oliver **Irwin**, and Sylvain **Salvati**. *Direct Access for Conjunctive Queries with Negations*. Oct. 2025. arXiv 2310.15800.

[Cap16] Florent **Capelli**. *Structural restrictions of CNF-formulas: applications to model counting and knowledge compilation*. PhD thesis. Université Paris Diderot, Sorbonne Paris Cité, June 2016. https://florent.capelli.me/publi/these_capelli.pdf.

[Cap17] Florent **Capelli**. *Understanding the complexity of #SAT using knowledge compilation*. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, June 2017, pp. 1–10. doi 10.1109/LICS.2017.8005121.

[Car+23] Nofar **Carmeli**, Nikolaos **Tziavelis**, Wolfgang **Gatterbauer**, Benny **Kimelfeld**, and Mirek **Riedewald**. *Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries*. In *ACM Transactions on Database Systems* (Jan. 2023). doi 10.1145/3578517.

[CI24] Florent **Capelli** and Oliver **Irwin**. *Direct Access for Conjunctive Queries with Negations*. In *27th International Conference on Database Theory, ICDT 2024, March 24 to March 28, 2024, Paestum, Italy*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2024, 13:1–13:20. doi 10.4230/LIPICS.ICDT.2024.13.

[Cor+22] Thomas H. **Cormen**, Charles Eric **Leiserson**, Ronald Linn **Rivest**, and Clifford **Stein**. *Introduction to Algorithms*. Fourth edition. Cambridge, Massachusetts London: The MIT Press, Apr. 2022. 1312 pp.

[CS23] Nofar **Carmeli** and Luc **Segoufin**. *Conjunctive queries with self-joins, towards a fine-grained enumeration complexity analysis*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2023, pp. 277–289. doi 10.1145/3584372.3588667.

[ECK24] Idan **Eldar**, Nofar **Carmeli**, and Benny **Kimelfeld**. *Direct Access for Answers to Conjunctive Queries with Aggregation*. In *27th International Conference on Database Theory, ICDT 2024, March 24 to March 28, 2024, Paestum, Italy*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2024, 20 pages. doi 10.4230/LIPICS.ICDT.2024.4.

[FGP18] Wolfgang **Fischl**, Georg **Gottlob**, and Reinhard **Pichler**. *General and fractional hypertree decompositions: Hard and easy cases*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. May 2018, pp. 17–32. doi 10.1145/3196959.3196962.

[GR04] **Georg Gottlob** and **Reinhard Pichler**. *Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width*. In *SIAM Journal on Computing* 33.2 (Jan. 2004), pp. 351–378. doi 10.1137/S0097539701396807.

[JPR16] Manas R **Joglekar**, Rohan **Puttagunta**, and Christopher **Ré**. *Ajar: Aggregations and joins over annotated relations*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2016, pp. 91–106. doi 10.1145/2902251.2902293.

[Lan23] Matthias **Lanzinger**. *Tractability beyond β-acyclicity for conjunctive queries with negation and SAT*. In *Theoretical Computer Science* 942 (Jan. 2023), pp. 276–296. doi 10.1016/j.tcs.2022.12.002.

[OPS13] Sebastian **Ordyniak**, Daniel **Paulusma**, and Stefan **Szeider**. *Satisfiability of acyclic and almost acyclic CNF formulas*. In *Theoretical Computer Science* 481 (Apr. 2013), pp. 85–99. doi 10.1016/j.tcs.2012.12.039.

[PSS16] Daniël **Paulusma**, Friedrich **Slivovsky**, and Stefan **Szeider**. *Model counting for CNF formulas of bounded modular treewidth*. In *Algorithmica* 76.1 (July 2016), pp. 168–194. doi 10.1007/s00453-015-0030-x.

[SS10] M. **Samer** and S. **Szeider**. *Algorithms for propositional model counting*. In *Journal of Discrete Algorithms* 8.1 (Mar. 2010), pp. 50–64. doi 10.1016/j.jda.2009.06.002.

[SS13] Friedrich **Slivovsky** and Stefan **Szeider**. *Model Counting for Formulas of Bounded Clique-Width*. In *Algorithms and Computation - 24th International Symposium, ISAAC*. Dec. 2013, pp. 677–687. doi 10.1007/978-3-642-45030-3_63.

[STV14] S. Hortemo **Sæther**, J.A. **Telle**, and M. **Vatshelle**. *Solving MaxSAT and #SAT on Structured CNF Formulas*. In *Theory and Applications of Satisfiability Testing*. July 2014, pp. 16–31. doi 10.1007/978-3-319-09284-3_3.

[van75] Peter **van Emde Boas**. *Preserving order in a forest in less than logarithmic time*. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1975, pp. 75–84. doi 10.1109/SFCS.1975.26.

[Zha+24] Hangdong **Zhao**, Austen Z. **Fan**, Xiating **Ouyang**, and Paraschos **Koutris**. *Conjunctive Queries with Negation and Aggregation: A Linear Time Characterization*. In *Proc. ACM Manag. Data* 2.2 (May 2024). doi 10.1145/3651138.

Conclusion

This manuscript has investigated the problem of efficiently accessing the answers of queries, with a particular focus on tasks such as direct access and uniform sampling. A recurring theme throughout the thesis has been to prioritise the structural properties of queries over the explicit enumeration of their answer sets. The central contribution of this work is to show that factorised representations, in particular relational circuit constructions, provide a unifying framework for addressing these tasks across a broad range of queries, including for classes that are known to be challenging, such as queries with negative atoms. Taken as a whole, the results presented in this thesis emphasize the fact that the correct representation of the query answers leads to efficient solving of different tasks.

This thesis advocates the use of branching algorithms and relational circuits as a principled approach to query evaluation, uniform sampling, and direct access. The contributions naturally decompose into two complementary directions: the construction of compact factorised representations of query answer sets, and the exploitation of these representations to efficiently support a variety of computational tasks.

Factorised Representations

Two main factorised representations are discussed in this work. In Chapter 3, we presented a conceptually simple algorithm (Algorithm 3.2) to answer join queries in a worst-case optimal way. Both our algorithm and *Leapfrog TrieJoin* [Vel14] can be seen as simplified forms of *Generic Join* [Ngo+12]. The complexity analysis of this algorithm is also made more simple by being based on easily verifiable properties of the considered query and database classes: prefix-closedness.

The result is a simple branching algorithm, where, for each variable, we branch on the different domain values. To eliminate the overhead incurred by naive value enumeration, we introduced a binarisation technique reducing arbitrary domains to the Boolean case. This allows the algorithm to branch on bits rather than values, thereby pruning the search space efficiently while preserving optimality.

Moreover, a notable by-product of running this algorithm is that its trace has a tree-like structure. This structure is the first factorised representation we introduced in this manuscript. It is then extended to full-fledged relational circuits in Chapter 5, where we considered join query evaluation as a *compilation* task. Instead of enumerating the answers naively, we built on the work we introduced in Chapter 3 to introduce a new data structure as a factorised representation of the answer set of the query : ordered $\{\times, \mathsf{dec}\}$-circuits. These circuits exploit gate factorisation and the independence of subqueries to achieve succinct representations.

Crucially, this circuit-based representation extends to signed join queries: even in the presence of negative atoms, it is possible to compile a query and a database instance into an ordered $\{\times, \mathsf{dec}\}$-circuit that faithfully represents its answer set.

By definition, both these structures do not fully materialise the complete answer set of the considered queries. Therefore, we have been able to use these structures to efficiently answer tasks such as uniform sampling and direct access.

Using Factorised Representations to Answer Queries

The second direction of the contributions from this thesis are related to using our factorised representations in multiple different ways. In Chapter 4, we studied the problem of uniformly sampling answers to join queries. Our result consists in an adaptation of a classical algorithm, that allows uniform sampling of the leaves of a tree without a full exploration. We adapted this algorithm to a more general setting where only a subset of the leaves are of interest. This led us to introduce a new notion of estimator: a tree-superadditive function that allows us to bound the number of interesting leaves in any subtree. To sample efficiently, we used the structure implicitly defined by a run of our worst-case optimal join algorithm from Chapter 3. Leaves of this trace tree correspond to either inconsistencies or valid answers. This provides a clean reduction from uniform answer sampling for a query over a database to leaf sampling in a tree.

We have also shown how we can use known worst-case bounds over query answers as estimators for this sampling algorithm. The essential property here is the *superadditivity* of the estimator, which guarantees the correctness of the sampling procedure. We established more general results by proving that both the AGM bound for cardinality constraints and the polymatroid bound for acyclic degree constraints are tree-superadditive, thereby yielding efficient uniform samplers. Combined with the binarisation technique presented in Chapter 3, this allows us to match recent results in the literature, while relying on a simpler and more modular analysis.

A second use of our factorised representations was proving new tractability results concerning direct access for answers of signed queries. We re-established that the case of signed join queries could not be handled in the same way as positive queries. In an effort to unify the positive and the signed case using the factorised representation of the answer set of the query that we first introduced in Chapter 5, we introduced a new framework in Chapter 6.

The main idea in this framework is to first annotate the relational circuits with counts of the number of answers for each subcircuit, in a bottom-up fashion. This preprocessing phase is followed by an access phase where we go through the circuit from top to bottom, choosing branches as we go according to the index we are accessing and the annotations on the gates of the circuit.

We also showed that our complexity bounds, when restricted to the positive case, match the existing optimal ones and we have proved that our algorithm remains optimal for signed join queries without self-joins (in terms of data complexity).

Open Research Directions & Future Work

The work presented in this thesis leaves several natural extensions open, both from a theoretical and a practical perspective.

The approach of Chapters 5 and 6 open many new research avenues. In particular, we believe that the circuit representation that we use is also promising for answering different kinds of aggregation tasks and hence generalising existing results on conjunctive queries to the case of signed conjunctive queries. For example, we believe that FAQ and AJAR queries [ANR16; JPR16] could be solved using this data structure. Indeed, it seems possible to annotate the circuit with semi-ring elements and then project them out in a similar fashion as Theorem 6.46. Similarly, we believe that the framework of [ECK24] for solving direct access tasks on conjunctive queries with aggregation operators may be generalised to the class of ordered $\{\times, \mathsf{dec}\}$-circuits.

Finally, contrary to the positive case, we do not yet know what is the complexity of solving direct access tasks on signed join queries with self-joins. We know that self-joins can only make things easier and have exhibited an example Chapter 6 where having a self-join between the

positive and the negative part leads to drastic improvement in the preprocessing complexity. However, it is still unknown whether there are other more subtle cases where self-joins help, as it was shown for enumeration [CS23]. Actually, our algorithm to answer direct access tasks on the circuit is completely independent from the way the circuit has been constructed, as long as it has the necessary syntactic properties. Hence, we have hope that larger classes of queries and databases will be tractable using this approach.

A different direction would be evaluating how our factorised representations could be leveraged to adapt our contributions to the context of "dynamic databases", that is, databases that evolve with time. The Yannakakis approach has already been studied in the case of dynamic databases, notably in [IUV17]. This opens the way to also using different representations, as we have done in the case of static databases. While the naive approach in this case could be to rebuild the factorised representation at each update, this would prove costly, especially in the case of large databases or answer sets. Characterising the tractable classes of queries and databases for both direct access or uniform sampling without rebuilding the full factorised representation is an interesting avenue. For direct access, the independence between the access algorithm and the compilation of the relational circuit would also prove useful. Recent work on the tractability of enumeration of query answers under updates provides a natural point of comparison [BKS17; Kar+25].

Another major direction to extend this work in would consist in practical implementations of the different algorithms and the study of complementary optimisations. This would allow to exhaustively evaluate the performance of the solutions we propose against the state of the art algorithms currently used in database management systems.

In Chapter 3 for instance, we reduce the search on the domain values to a search on bits. This facilitates the understanding of the algorithm and its underlying complexity. A practical implementation may nevertheless find it more convenient to use larger domains such as bytes to take advantage of bit-vector operations. Both the compilation algorithm from Chapter 5 and the direct access algorithm from Chapter 6 could be implemented and have their efficiency evaluated against the current state of the art algorithms from real-life database management systems.

Current chapter references

[ANR16] Mahmoud **Abo Khamis**, Hung Q **Ngo**, and Atri **Rudra**. *FAQ: questions asked frequently*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2016, pp. 13–28. doi 10.1145/2902251.2902280.

[BKS17] Christoph **Berkholz**, Jens **Keppeler**, and Nicole **Schweikardt**. *Answering Conjunctive Queries under Updates*. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '17. Chicago, Illinois, USA: Association for Computing Machinery, May 2017, pp. 303–318. doi 10.1145/3034786.3034789.

[CS23] Nofar **Carmeli** and Luc **Segoufin**. *Conjunctive queries with self-joins, towards a fine-grained enumeration complexity analysis*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2023, pp. 277–289. doi 10.1145/3584372.3588667.

[ECK24] Idan **Eldar**, Nofar **Carmeli**, and Benny **Kimelfeld**. *Direct Access for Answers to Conjunctive Queries with Aggregation*. In *27th International Conference on Database Theory, ICDT 2024, March 24 to March 28, 2024, Paestum, Italy*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2024, 20 pages. doi 10.4230/LIPICS.ICDT.2024.4.

[IUV17] Muhammad **Idris**, Martin **Ugarte**, and Stijn **Vansummeren**. *The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates*. In *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, May 2017, pp. 1259–1274. doi 10.1145/3035918.3064027.

[JPR16] Manas R **Joglekar**, Rohan **Puttagunta**, and Christopher **Ré**. *Ajar: Aggregations and joins over annotated relations*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2016, pp. 91–106. doi 10.1145/2902251.2902293.

[Kar+25] Ahmet **Kara**, Zheng **Luo**, Milos **Nikolic**, Dan **Olteanu**, and Haozhe **Zhang**. *Tractable Conjunctive Queries over Static and Dynamic Relations*. In *28th International Conference on Database Theory (ICDT 2025)*. Ed. by Sudeepa **Roy** and Ahmet **Kara**. Vol. 328. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2025, 12:1–12:21. doi 10.4230/LIPIcs.ICDT.2025.12.

[Ngo+12] Hung Q. **Ngo**, Ely **Porat**, Christopher **Ré**, and Atri **Rudra**. *Worst-Case Optimal Join Algorithms: [Extended Abstract]*. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '12: International Conference on Management of Data. Scottsdale Arizona USA: ACM, May 2012, pp. 37–48. doi 10.1145/2213556.2213565.

[Vel14] Todd L. **Veldhuizen**. *Triejoin: A Simple, Worst-Case Optimal Join Algorithm*. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. Ed. by Nicole **Schweikardt**, Vassilis **Christophides**, and Vincent **Leroy**. OpenProceedings.org, Mar. 2014, pp. 96–106. doi 10.5441/002/icdt.2014.13.

Bibliography

[AB16] Sanjeev **Arora** and Boaz **Barak**. *Computational Complexity: A Modern Approach*. 4th printing 2016. New York: Cambridge University Press, 2016. 579 pp.

[Ach+99] Swarup **Acharya**, Phillip B. **Gibbons**, Viswanath **Poosala**, and Sridhar **Ramaswamy**. *Join Synopses for Approximate Query Answering*. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS99: International Conference on Management of Data and Symposium on Principles of Database Systems. Philadelphia Pennsylvania USA: ACM, June 1999, pp. 275–286. doi 10.1145/304182.304207.

[AGM13] Albert **Atserias**, Martin **Grohe**, and Dániel **Marx**. *Size bounds and query plans for relational joins*. In *SIAM Journal on Computing* 42.4 (Jan. 2013), pp. 1737–1767. doi 10.1137/110859440.

[ANR15] Mahmoud **Abo Khamis**, Hung Q **Ngo**, and Atri **Rudra**. *FAQ: questions asked frequently*. Apr. 2015. arXiv 1504.04044.

[ANR16] Mahmoud **Abo Khamis**, Hung Q **Ngo**, and Atri **Rudra**. *FAQ: questions asked frequently*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2016, pp. 13–28. doi 10.1145/2902251.2902280.

[ANS16] Mahmoud **Abo Khamis**, Hung Q. **Ngo**, and Dan **Suciu**. *Computing Join Queries with Functional Dependencies*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS'16: International Conference on Management of Data. San Francisco California USA: ACM, June 2016, pp. 327–342. doi 10.1145/2902251.2902289.

[ANS17a] Mahmoud **Abo Khamis**, Hung Q **Ngo**, and Dan **Suciu**. *What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another?* In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. May 2017, pp. 429–444. doi 10.1145/3034786.3056105.

[ANS17b] Mahmoud **Abo Khamis**, Hung Q. **Ngo**, and Dan **Suciu**. *What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?* In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS'17: International Conference on Management of Data. Chicago Illinois USA: ACM, May 2017, pp. 429–444. doi 10.1145/3034786.3056105.

[Are+21] Marcelo **Arenas**, Luis Alberto **Croquevielle**, Rajesh **Jayaram**, and Cristian **Riveros**. *When Is Approximate Counting for Conjunctive Queries Tractable?* In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing. Virtual, Italy: ACM, June 2021, pp. 1015–1027. doi 10.1145/3406325.3451014.

[Are+22] Marcelo **Arenas**, Pablo **Barcelo**, Leonid **Libkin**, Wim **Martens**, and Andreas **Pieris**. *Database Theory*. Open Source. 2022. https://github.com/pdm-book/community.

[Bag+08] Guillaume **Bagan**, Arnaud **Durand**, Etienne **Grandjean**, and Frédéric **Olive**. *Computing the jth solution of a first-order query*. In *RAIRO-Theoretical Informatics and Applications* 42.1 (Jan. 2008), pp. 147–164. doi 10.1051/ita:2007046.

[Bag09] Guillaume **Bagan**. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques. (Algorithms and complexity of enumeration problems for the evaluation of logical queries)*. PhD thesis. University of Caen Normandy, France, Mar. 2009.

[Bak+13] Nurzhan **Bakibayev**, Tomáš **Kočiský**, Dan **Olteanu**, and Jakub **Závodný**. *Aggregation and Ordering in Factorised Databases*. In *Proceedings of the VLDB Endowment* 6.14 (Sept. 2013), pp. 1990–2001. doi 10.14778/2556549.2556579.

[Bau+05] Michael **Bauland**, Philippe **Chapdelaine**, Nadia **Creignou**, Miki **Hermann**, and Heribert **Vollmer**. *An Algebraic Approach to the Complexity of Generalized Conjunctive Queries*. In *Theory and Applications of Satisfiability Testing*. Ed. by Holger H. **Hoos** and David G. **Mitchell**. Vol. 3542. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 30–45. doi 10.1007/11527695_3.

[BCM15] Johann **Brault-Baron**, Florent **Capelli**, and Stefan **Mengel**. *Understanding Model Counting for beta-acyclic CNF-formulas*. In *32nd International Symposium on Theoretical Aspects of Computer Science*. Vol. 30. LIPIcs. Schloss Dagstuhl, Feb. 2015, pp. 143–156. doi 10.4230/LIPIcs.STACS.2015.143.

[BCM22a] Karl **Bringmann**, Nofar **Carmeli**, and Stefan **Mengel**. *Tight Fine-Grained Bounds for Direct Access on Join Queries*. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid **Libkin** and Pablo **Barceló**. PODS '22. Association for Computing Machinery, June 2022, pp. 427–436. doi 10.1145/3517804.3526234.

[BCM22b] Karl **Bringmann**, Nofar **Carmeli**, and Stefan **Mengel**. *Tight Fine-Grained Bounds for Direct Access on Join Queries*. Jan. 2022. arXiv 2201.02401.

[BCM25] Karl **Bringmann**, Nofar **Carmeli**, and Stefan **Mengel**. *Tight Fine-Grained Bounds for Direct Access on Join Queries*. In *ACM Trans. Database Syst.* 50.1 (Jan. 2025). doi 10.1145/3707448.

[BDG07] Guillaume **Bagan**, Arnaud **Durand**, and Étienne **Grandjean**. *On Acyclic Conjunctive Queries and Constant Delay Enumeration*. In *Proceedings of the 21st International Conference, and Proceedings of the 16th Annual Conference on Computer Science Logic*. CSL'07/EACSL'07. Lausanne, Switzerland: Springer-Verlag, Sept. 2007, pp. 208–222. doi 10.1007/978-3-540-74915-8_18.

[Bee+83] Catriel **Beeri**, Ronald **Fagin**, David **Maier**, and Mihalis **Yannakakis**. *On the Desirability of Acyclic Database Schemes*. In *Journal of the ACM* 30.3 (July 1983), pp. 479–513. doi 10.1145/2402.322389.

[Ber73] Claude **Berge**. *Graphes et Hypergraphes*. 2. éd. Dunod Université, 604. Série Violette; Mathématiques. Paris: Dunod, 1973. 516 pp.

[Ber87] Claude **Berge**. *Hypergraphes: combinatoire des ensembles finis*. Bordas. Paris: Gauthier-Villars [u.a.], May 1987. 240 pp.

[BGS20] Christoph **Berkholz**, Fabian **Gerhardt**, and Nicole **Schweikardt**. *Constant Delay Enumeration for Conjunctive Queries: A Tutorial*. In *ACM SIGLOG News* 7.1 (Feb. 2020), pp. 4–33. doi 10.1145/3385634.3385636.

[BKS17] Christoph **Berkholz**, Jens **Keppeler**, and Nicole **Schweikardt**. *Answering Conjunctive Queries under Updates*. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '17. Chicago, Illinois, USA: Association for Computing Machinery, May 2017, pp. 303–318. doi 10.1145/3034786.3034789.

[Bod06] Hans L. **Bodlaender**. *Treewidth: Characterizations, Applications, and Computations*. In *Graph-Theoretic Concepts in Computer Science*. Ed. by Fedor V. **Fomin**. Vol. 4271. Berlin, Heidelberg: Springer Berlin Heidelberg, June 2006, pp. 1–14. doi 10.1007/11917496_1.

[Bod93] Hans L. **Bodlaender**. *A Tourist Guide through Treewidth*. In *Acta Cybernetica* 11.1-2 (Jan. 1993), pp. 1–21. https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3417.

[Bov+15] Simone **Bova**, Florent **Capelli**, Stefan **Mengel**, and Friedrich **Slivovsky**. *On Compiling CNFs into Structured Deterministic DNNFs*. In *Theory and Applications of Satisfiability Testing*. Lecture Notes in Computer Science. Springer International Publishing, Sept. 2015, pp. 199–214. doi 10.1007/978-3-319-24318-4_15.

[Bra12] Johann **Brault-Baron**. *A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic*. In *Computer Science Logic - 26th International Workshop/21st Annual Conference of the EACSL, September 3-6, 2012, Fontainebleau, France*. Ed. by Patrick **Cégielski** and Arnaud **Durand**. Vol. 16. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Sept. 2012, pp. 137–151. doi 10.4230/LIPIcs.CSL.2012.137.

[Bra13] Johann **Brault-Baron**. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis. Université de Caen, Apr. 2013.

[Bra17] Johann **Brault-Baron**. *Hypergraph Acyclicity Revisited*. In *ACM Computing Surveys* 49.3 (Sept. 2017), pp. 1–26. doi 10.1145/2983573.

[Cap+25] Florent **Capelli**, Nofar **Carmeli**, Oliver **Irwin**, and Sylvain **Salvati**. *Direct Access for Conjunctive Queries with Negations*. Oct. 2025. arXiv 2310.15800.

[Cap16] Florent **Capelli**. *Structural restrictions of CNF-formulas: applications to model counting and knowledge compilation*. PhD thesis. Université Paris Diderot, Sorbonne Paris Cité, June 2016. https://florent.capelli.me/publi/these_capelli.pdf.

[Cap17] Florent **Capelli**. *Understanding the complexity of #SAT using knowledge compilation*. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, June 2017, pp. 1–10. doi 10.1109/LICS.2017.8005121.

[Car+20] Nofar **Carmeli**, Shai **Zeevi**, Christoph **Berkholz**, Benny **Kimelfeld**, and Nicole **Schweikardt**. *Answering (unions of) conjunctive queries using random access and random-order enumeration*. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2020, pp. 393–409. doi 10.1145/3375395.3387662.

[Car+23] Nofar **Carmeli**, Nikolaos **Tziavelis**, Wolfgang **Gatterbauer**, Benny **Kimelfeld**, and Mirek **Riedewald**. *Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries*. In *ACM Transactions on Database Systems* (Jan. 2023). doi 10.1145/3578517.

[CI24] Florent **Capelli** and Oliver **Irwin**. *Direct Access for Conjunctive Queries with Negations*. In *27th International Conference on Database Theory, ICDT 2024, March 24 to March 28, 2024, Paestum, Italy*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2024, 13:1–13:20. doi 10.4230/LIPICS.ICDT.2024.13.

[CIS25] Florent **Capelli**, Oliver **Irwin**, and Sylvain **Salvati**. *A Simple Algorithm for Worst Case Optimal Join and Sampling*. In *28th International Conference on Database Theory (ICDT 2025), March 25 to March 28, 2025, Barcelona, Spain* (ICDT '25). Ed. by Sudeepa **Roy** and Ahmet **Kara**. Vol. 328. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2025, 23:1–23:19. doi 10.4230/LIPIcs.ICDT.2025.23.

[CM77] Ashok K. **Chandra** and Philip M. **Merlin**. *Optimal Implementation of Conjunctive Queries in Relational Data Bases*. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, May 1977, pp. 77–90. doi 10.1145/800105.803397.

[CMN99] Surajit **Chaudhuri**, Rajeev **Motwani**, and Vivek **Narasayya**. *On Random Sampling over Joins*. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS99: International Conference on Management of Data and Symposium on Principles of Database Systems. Philadelphia Pennsylvania USA: ACM, June 1999, pp. 263–274. doi 10.1145/304182.304206.

[Cod70] E. F. **Codd**. *A relational model of data for large shared data banks*. In *Communications of the ACM* 13.6 (June 1970), pp. 377–387. doi 10.1145/362384.362685.

[Cor+22] Thomas H. **Cormen**, Charles Eric **Leiserson**, Ronald Linn **Rivest**, and Clifford **Stein**. *Introduction to Algorithms*. Fourth edition. Cambridge, Massachusetts London: The MIT Press, Apr. 2022. 1312 pp.

[CR00] Chandra **Chekuri** and Anand **Rajaraman**. *Conjunctive Query Containment Revisited*. In *Theoretical Computer Science* 239.2 (May 2000), pp. 211–229. doi 10.1016/S0304-3975(99)00220-0.

[CS23] Nofar **Carmeli** and Luc **Segoufin**. *Conjunctive queries with self-joins, towards a fine-grained enumeration complexity analysis*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2023, pp. 277–289. doi 10.1145/3584372.3588667.

[CY20a] Yu **Chen** and Ke **Yi**. *Random Sampling and Size Estimation Over Cyclic Joins*. In *LIPIcs, Volume 155, ICDT 2020* 155 (Mar. 2020). Ed. by Carsten **Lutz** and Jean Christoph **Jung**, 7:1–7:18. doi 10.4230/LIPICS.ICDT.2020.7.

[CY20b] Yu **Chen** and Ke **Yi**. *Random sampling and size estimation over cyclic joins*. In *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*. Ed. by Carsten **Lutz** and Jean Christoph **Jung**. Vol. 155. LIPIcs. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2020, 7:1–7:18. doi 10.4230/LIPIcs.ICDT.2020.7.

[DJ04] Víctor **Dalmau** and Peter **Jonsson**. *The Complexity of Counting Homomorphisms Seen from the Other Side*. In *Theoretical Computer Science* 329.1-3 (Dec. 2004), pp. 315–323. doi 10.1016/j.tcs.2004.08.008.

[DLT23] Shiyuan **Deng**, Shangqi **Lu**, and Yufei **Tao**. *On Join Sampling and the Hardness of Combinatorial Output-Sensitive Join Algorithms*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '23: International Conference on Management of Data. Seattle WA USA: ACM, June 2023, pp. 99–111. doi 10.1145/3584372.3588666.

[DM02] A. **Darwiche** and P. **Marquis**. *A Knowledge Compilation Map*. In *Journal of Artificial Intelligence Research* 17 (Sept. 2002), pp. 229–264. doi 10.1613/jair.989.

[Dur09] David **Duris**. *Acyclicité des hypergraphes et liens avec la logique sur les structures relationnelles finies*. PhD thesis. Université Paris Diderot, Nov. 2009. https://www.imj-prg.fr/theses/pdf/david_duris.pdf.

[ECK23] Idan **Eldar**, Nofar **Carmeli**, and Benny **Kimelfeld**. *Direct Access for Answers to Conjunctive Queries with Aggregation*. Mar. 2023. 2303.05327. preprint.

[ECK24] Idan **Eldar**, Nofar **Carmeli**, and Benny **Kimelfeld**. *Direct Access for Answers to Conjunctive Queries with Aggregation*. In *27th International Conference on Database Theory, ICDT 2024, March 24 to March 28, 2024, Paestum, Italy*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Mar. 2024, 20 pages. doi 10.4230/LIPICS.ICDT.2024.4.

[Fag83] Ronald **Fagin**. *Degrees of Acyclicity for Hypergraphs and Relational Database Schemes*. In *Journal of the ACM* 30.3 (July 1983), pp. 514–550. doi 10.1145/2402.322390.

[FFG01] Jörg **Flum**, Markus **Frick**, and Martin **Grohe**. *Query Evaluation via Tree-Decompositions: Extended Abstract*. In *Database Theory — ICDT 2001*. Ed. by Jan **Van Den Bussche** and Victor **Vianu**. Red. by Gerhard **Goos**, Juris **Hartmanis**, and Jan **Van Leeuwen**. Vol. 1973. Berlin, Heidelberg: Springer Berlin Heidelberg, Oct. 2001, pp. 22–38. doi 10.1007/3-540-44503-X_2.

[FGP18] Wolfgang **Fischl**, Georg **Gottlob**, and Reinhard **Pichler**. *General and fractional hypertree decompositions: Hard and easy cases*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. May 2018, pp. 17–32. doi 10.1145/3196959.3196962.

[Fic+18] Johannes K **Fichte**, Markus **Hecher**, Neha **Lodha**, and Stefan **Szeider**. *An SMT approach to fractional hypertree width*. In *Principles and Practice of Constraint Programming: 24th International Conference, Lille, France, August 27-31, 2018, Proceedings 24*. Springer. Aug. 2018, pp. 109–127. doi 10.1007/978-3-319-98334-9_8.

[Foc+25] Jacob **Focke**, Leslie Ann **Goldberg**, Marc **Roth**, and Stanislav **Živný**. *Approximately Counting Answers to Conjunctive Queries with Disequalities and Negations*. In *ACM Transactions on Algorithms* 21.1 (Jan. 2025), pp. 1–29. doi 10.1145/3689634.

[Fri04] Ehud **Friedgut**. *Hypergraphs, Entropy, and Inequalities*. In *The American Mathematical Monthly* 111.9 (June 2004), pp. 749–760. doi 10.2307/4145187.

[FW90] Michael L. **Fredman** and Dan E. **Willard**. *BLASTING through the information theoretic barrier with FUSION TREES*. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: Association for Computing Machinery, Apr. 1990, pp. 1–7. doi 10.1145/100216.100217.

[Gan+22] Robert **Ganian**, André **Schidler**, Manuel **Sorge**, and Stefan **Szeider**. *Threshold treewidth and hypertree width*. In *Journal of Artificial Intelligence Research* 74 (Aug. 2022), pp. 1687–1713. doi 10.1613/JAIR.1.13661.

[GLS00] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *A comparison of structural CSP decomposition methods*. In *Artificial Intelligence* 124.2 (Dec. 2000), pp. 243–282. doi 10.1016/S0004-3702(00)00078-3.

[GLS01] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *Hypertree Decompositions: A Survey*. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science*. MFCS '01. Berlin, Heidelberg: Springer-Verlag, Jan. 2001, pp. 37–57. doi 10.1007/3-540-44683-4_5.

[GLS02] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *Hypertree Decompositions and Tractable Queries*. In *Journal of Computer and System Sciences* 64.3 (May 2002), pp. 579–627. doi 10.1006/jcss.2001.1809.

[GLS99] Georg **Gottlob**, Nicola **Leone**, and Francesco **Scarcello**. *Hypertree Decompositions and Tractable Queries*. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. SIGMOD/PODS99: International Conference on Management of Data and Symposium on Principles of Database Systems. Philadelphia Pennsylvania USA: ACM, May 1999, pp. 21–32. doi 10.1145/303976.303979.

[GM14] Martin **Grohe** and Dániel **Marx**. *Constraint solving via fractional edge covers*. In *ACM Transactions on Algorithms (TALG)* 11.1 (Aug. 2014), p. 4. doi 10.1145/2636918.

[GR04] **Georg Gottlob** and **Reinhard Pichler**. *Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width*. In *SIAM Journal on Computing* 33.2 (Jan. 2004), pp. 351–378. doi 10.1137/S0097539701396807.

[GSS01] Martin **Grohe**, Thomas **Schwentick**, and Luc **Segoufin**. *When Is the Evaluation of Conjunctive Queries Tractable?* In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*. STOC01: 33rd ACM Symposium on Theory of Computing. Hersonissos Greece: ACM, July 2001, pp. 657–666. doi 10.1145/380752.380867.

[HD05] Jinbo **Huang** and Adnan **Darwiche**. *DPLL with a trace: from SAT to knowledge compilation*. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. IJCAI'05. Edinburgh, Scotland: Morgan Kaufmann Publishers Inc., July 2005, pp. 156–162. doi 10.5555/1642293.1642318.

[HV21] David **Harvey** and Joris **Van Der Hoeven**. *Integer multiplication in time* $\mathcal{O}(n \log n)$. In *Annals of Mathematics* 193.2 (Mar. 2021), pp. 563–617. doi 10.4007/annals.2021.193.2.4.

[IUV17] Muhammad **Idris**, Martin **Ugarte**, and Stijn **Vansummeren**. *The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates*. In *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, May 2017, pp. 1259–1274. doi 10.1145/3035918.3064027.

[JPR16] Manas R **Joglekar**, Rohan **Puttagunta**, and Christopher **Ré**. *Ajar: Aggregations and joins over annotated relations*. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. June 2016, pp. 91–106. doi 10.1145/2902251.2902293.

[JVV86] Mark R. **Jerrum**, Leslie G. **Valiant**, and Vijay V. **Vazirani**. *Random generation of combinatorial structures from a uniform distribution*. In *Theoretical Computer Science* 43 (Nov. 1986), pp. 169–188. doi https://doi.org/10.1016/0304-3975(86)90174-X.

[Kar+25] Ahmet **Kara**, Zheng **Luo**, Milos **Nikolic**, Dan **Olteanu**, and Haozhe **Zhang**. *Tractable Conjunctive Queries over Static and Dynamic Relations*. In *28th International Conference on Database Theory (ICDT 2025)*. Ed. by Sudeepa **Roy** and Ahmet **Kara**. Vol. 328. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2025, 12:1–12:21. doi 10.4230/LIPIcs.ICDT.2025.12.

[Kep20] Jens **Keppeler**. *Answering Conjunctive Queries and FO+MOD Queries under Updates*. PhD thesis. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät, June 2020. doi 10.18452/21483.

[Kim+23] Kyoungmin **Kim**, Jaehyun **Ha**, George **Fletcher**, and Wook-Shin **Han**. *Guaranteeing the Õ(AGM/OUT) Runtime for Uniform Sampling and Size Estimation over Joins*. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '23: International Conference on Management of Data. Seattle WA USA: ACM, June 2023, pp. 113–125. doi 10.1145/3584372.3588676.

[KV00] Phokion G. **Kolaitis** and Moshe Y. **Vardi**. *Conjunctive-Query Containment and Constraint Satisfaction*. In *Journal of Computer and System Sciences* 61.2 (Oct. 2000), pp. 302–332. doi 10.1006/jcss.2000.1713.

[Lan23] Matthias **Lanzinger**. *Tractability beyond β-acyclicity for conjunctive queries with negation and SAT*. In *Theoretical Computer Science* 942 (Jan. 2023), pp. 276–296. doi 10.1016/j.tcs.2022.12.002.

[Mar10] Dániel **Marx**. *Approximating fractional hypertree width*. In *ACM Trans. Algorithms* 6.2 (Apr. 2010). doi 10.1145/1721837.1721845.

[Ngo+12] Hung Q. **Ngo**, Ely **Porat**, Christopher **Ré**, and Atri **Rudra**. *Worst-Case Optimal Join Algorithms: [Extended Abstract]*. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '12: International Conference on Management of Data. Scottsdale Arizona USA: ACM, May 2012, pp. 37–48. doi 10.1145/2213556.2213565.

[Ngo18] Hung Q. **Ngo**. *Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems*. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '18: International Conference on Management of Data. Houston TX USA: ACM, May 2018, pp. 111–124. doi 10.1145/3196959.3196990.

[NRR14] Hung Q. **Ngo**, Christopher **Ré**, and Atri **Rudra**. *Skew strikes back: new developments in the theory of join algorithms*. In *SIGMOD Rec.* 42.4 (Feb. 2014), pp. 5–16. doi 10.1145/2590989.2590991.

[Olt16] Dan **Olteanu**. *Factorized Databases: A Knowledge Compilation Perspective*. In *AAAI Workshop: Beyond NP*. Feb. 2016.

[OPS13] Sebastian **Ordyniak**, Daniel **Paulusma**, and Stefan **Szeider**. *Satisfiability of acyclic and almost acyclic CNF formulas*. In *Theoretical Computer Science* 481 (Apr. 2013), pp. 85–99. doi 10.1016/j.tcs.2012.12.039.

[OZ12] Dan **Olteanu** and Jakub **Závodnỳ**. *Factorised Representations of Query Results: Size Bounds and Readability*. In *Proceedings of the 15th International Conference on Database Theory*. ACM. Mar. 2012, pp. 285–298. doi 10.1145/2274576.2274607.

[OZ15] Dan **Olteanu** and Jakub **Závodný**. *Size Bounds for Factorised Representations of Query Results*. en. In *ACM Transactions on Database Systems* 40.1 (Mar. 2015), pp. 1–44. doi 10.1145/2656335.

[Per14] Sylvain **Perifel**. *Complexité algorithmique*. Références sciences. Paris: Ellipses, 2014.

[PS13] Reinhard **Pichler** and Sebastian **Skritek**. *Tractable Counting of the Answers to Conjunctive Queries*. In *Journal of Computer and System Sciences* 79.6 (Sept. 2013), pp. 984–1001. doi 10.1016/j.jcss.2013.01.012.

[PSS16] Daniël **Paulusma**, Friedrich **Slivovsky**, and Stefan **Szeider**. *Model counting for CNF formulas of bounded modular treewidth*. In *Algorithmica* 76.1 (July 2016), pp. 168–194. doi 10.1007/s00453-015-0030-x.

[Ros93] Paul R. **Rosenbaum**. *Sampling the Leaves of a Tree with Equal Probabilities*. In *Journal of the American Statistical Association* 88.424 (Dec. 1993), pp. 1455–1457. doi 10.1080/01621459.1993.10476433.

[RS83] Neil **Robertson** and P.D. **Seymour**. *Graph Minors. I. Excluding a Forest*. In *Journal of Combinatorial Theory, Series B* 35.1 (Aug. 1983), pp. 39–61. doi 10.1016/0095-8956(83)90079-5.

[RS86] Neil **Robertson** and P.D **Seymour**. *Graph Minors. II. Algorithmic Aspects of Tree-Width*. In *Journal of Algorithms* 7.3 (Sept. 1986), pp. 309–322. doi 10.1016/0196-6774(86)90023-4.

[San+04] Tian **Sang**, Fahiem **Bacchus**, Paul **Beame**, Henry A **Kautz**, and Toniann **Pitassi**. *Combining Component Caching and Clause Learning for Effective Model Counting*. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*. May 2004. http://www.satisfiability.org/SAT04/programme/21.pdf.

[SS10] M. **Samer** and S. **Szeider**. *Algorithms for propositional model counting*. In *Journal of Discrete Algorithms* 8.1 (Mar. 2010), pp. 50–64. doi 10.1016/j.jda.2009.06.002.

[SS13] Friedrich **Slivovsky** and Stefan **Szeider**. *Model Counting for Formulas of Bounded Clique-Width*. In *Algorithms and Computation - 24th International Symposium, ISAAC*. Dec. 2013, pp. 677–687. doi 10.1007/978-3-642-45030-3_63.

[STV14] S. Hortemo **Sæther**, J.A. **Telle**, and M. **Vatshelle**. *Solving MaxSAT and #SAT on Structured CNF Formulas*. In *Theory and Applications of Satisfiability Testing*. July 2014, pp. 16–31. doi 10.1007/978-3-319-09284-3_3.

[Suc23] Dan **Suciu**. *Applications of Information Inequalities to Database Theory Problems*. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023, Boston, MA, USA, June 26-29, 2023*. IEEE, June 2023, pp. 1–30. doi 10.1109/LICS56636.2023.10175769.

[Val79] Leslie G. **Valiant**. *The Complexity of Enumeration and Reliability Problems*. In *SIAM Journal on Computing* 8.3 (Aug. 1979), pp. 410–421. doi 10.1137/0208032.

[van75] Peter **van Emde Boas**. *Preserving order in a forest in less than logarithmic time*. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1975, pp. 75–84. doi 10.1109/SFCS.1975.26.

[Var82] Moshe Y. **Vardi**. *The Complexity of Relational Query Languages (Extended Abstract)*. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing - STOC '82*. The Fourteenth Annual ACM Symposium. San Francisco, California, United States: ACM Press, May 1982, pp. 137–146. doi 10.1145/800070.802186.

[Vel14] Todd L. **Veldhuizen**. *Triejoin: A Simple, Worst-Case Optimal Join Algorithm*. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. Ed. by Nicole **Schweikardt**, Vassilis **Christophides**, and Vincent **Leroy**. OpenProceedings.org, Mar. 2014, pp. 96–106. doi 10.5441/002/icdt.2014.13.

[WT24] Ru **Wang** and Yufei **Tao**. *Join Sampling Under Acyclic Degree Constraints and (Cyclic) Subgraph Sampling*. In *27th International Conference on Database Theory (ICDT 2024)*. Ed. by Graham **Cormode** and Michael **Shekelyan**. Vol. 290. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2024, 23:1–23:20. doi 10.4230/LIPIcs.ICDT.2024.23.

[Yan81] Mihalis **Yannakakis**. *Algorithms for Acyclic Database Schemes*. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. Sept. 1981, pp. 82–94. doi 10.5555/1286831.1286840.

[Zan25] Bruno **Zanuttini**. *Introduction à la compilation de connaissances*. In *Informatique fondamentale et ses Mathématiques : Une photographie en 2025*. Ed. by Pascal **Vanier**. CNRS Alpha. CNRS Editions, July 2025, pp. 255–302.

[Zha+18] Zhuoyue **Zhao**, Robert **Christensen**, Feifei **Li**, Xiao **Hu**, and Ke **Yi**. *Random Sampling over Joins Revisited*. In *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD/PODS '18: International Conference on Management of Data. Houston TX USA: ACM, May 2018, pp. 1525–1539. doi 10.1145/3183713.3183739.

[Zha+24] Hangdong **Zhao**, Austen Z. **Fan**, Xiating **Ouyang**, and Paraschos **Koutris**. *Conjunctive Queries with Negation and Aggregation: A Linear Time Characterization*. In *Proc. ACM Manag. Data* 2.2 (May 2024). doi 10.1145/3651138.

Branching and Circuits: Algorithmic Techniques for Efficient Query Evaluation

Abstract

Efficiently evaluating and accessing the answers of queries over databases is a central problem in data management, particularly as data volumes continue to grow and queries become more complex. In the relational database model, query evaluation is complicated by the presence of joins, which may induce large intermediate results and lead to prohibitive computational costs. This thesis explores several aspects of query evaluation and their respective computational complexity.

The first aspect we consider is that of worst-case optimal join algorithms. These algorithms aim to return the answer set of a query over a database, while providing strong guarantees ensuring that the evaluation runs in time proportional to the maximum possible output size. In this manuscript, we present a join algorithm that can be shown to be worst-case optimal in a simple and modular way. One of the advantages of our approach is the implicit data representation it induces. By improving on this representation of the answers of a query, we are able to work on finer algorithmic problems.

The two main evaluation tasks we consider in this thesis are direct access and uniform sampling. For uniform sampling, we show how the execution trace of our worst-case optimal join algorithm can be leveraged to sample query answers according to a uniform distribution without enumerating the full result set. For direct access, we exploit relational circuits to navigate the answer space efficiently and retrieve answers at arbitrary positions in a fixed order, extending existing approaches to broader classes of queries, including queries with negation.

In both cases, the aim of our algorithms is to build the answer set of queries incrementally. We also use the structure of relational circuits to improve on the complexity of these methods.

Keywords: databases, algorithms, aggregation, compilation, join queries, conjunctive queries

---

Branchements et Circuits : techniques algorithmiques pour l'évaluation efficace de requêtes

Résumé

L'évaluation et l'accès efficaces aux réponses des requêtes sur les bases de données constituent un enjeu majeur dans le cadre de la gestion des données. Cette problématique revêt une importance croissante avec l'augmentation constante des volumes de données et la complexité croissante des requêtes plus complexes. Dans le modèle des bases de données relationnelles, l'évaluation des requêtes s'avère complexe en raison de la présence des jointures. Ces dernières peuvent engendrer des résultats intermédiaires de grande taille et occasionner des frais de calcul substantiels. Dans le cadre de cette thèse, une exploration approfondie de divers aspects inhérents à l'évaluation des requêtes et à leur complexité computationnelle respective est entreprise.

L'analyse se concentre en premier lieu sur les algorithmes de jointure optimaux dans le pire des cas. Ces algorithmes ont pour objectif de renvoyer l'ensemble des réponses à une requête sur une base de données, tout en offrant des garanties solides quant au temps d'évaluation, qui doit rester proportionnel à la taille maximale de la sortie. Dans le présent manuscrit, nous proposons une exposition d'un algorithme de jointure simple, qui peut être démontré comme étant optimal dans le pire des cas, de manière simple et modulaire.

Dans cette thèse, nous avons exploré deux tâches d'évaluation principales : l'accès direct et l'échantillonnage uniforme. Dans le cadre de l'échantillonnage uniforme, nous montrons comment utiliser la trace d'exécution de notre algorithme de jointure pour échantilloner uniformément, sans pour autant énumérer tous les résultats. Afin d'assurer l'accès direct, nous avons exploité une structure de circuits relationnels. Cette méthode permet une navigation efficace dans l'espace des réponses et la récupération des réponses à des positions arbitraires pour un ordre fixé. Nous montrons également que notre méthode permet d'étendre les résultats présents dans la littérature à des classes plus larges de requêtes, et notamment les requêtes avec négation.

Dans les deux cas, l'objectif des algorithmes développés est de construire l'ensemble des réponses aux requêtes de manière incrémentale. Les structures employées, à base de circuits et de branchements, nous permettent de réduire la complexité des méthodes employées.

Mots clés : bases de données, algorithmes, agrégation, compilation, requêtes de jointure, requêtes conjonctives

---

CRIStAL

Université de Lille - Campus scientifique – Bâtiment ESPRIT – Avenue Henri Poincaré – 59655 Villeneuve d'Ascq - France