Scalable Query Evaluation over Complex Probabilistic Databases

Abhay Jha

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2012

Reading Committee:
Dan Suciu, Chair
Magdalena Balazinska
Pedro Domingos

Program authorized to offer degree:
Computer Science and Engineering
Abstract

Scalable Query Evaluation over Complex Probabilistic Databases

Abhay Jha

Chair of the Supervisory Committee:
Professor Dan Suciu
Computer Science and Engineering

The age of *Big Data* has brought with itself datasets which are not just big, but also much more complicated. These datasets are constructed from disparate, unreliable and noisy sources, many times in an ad-hoc way because careful data cleaning and integration is too time consuming and not always necessary anymore. Representing the uncertainty hidden in these datasets is necessary to get meaningful query answers and Probabilistic Databases have come up as arguably the most popular solution to this problem. Their application to practical problems though has been held back because (i) the common models they use are not rich enough to capture the *dependencies* in these problems, and (ii) unlike traditional databases, query evaluation for probabilistic databases can be very expensive and *unpredictable*.

This dissertation addresses these challenges by first proposing a new model for probabilistic databases that is rich enough to capture the dependencies found in most practical applications, while still allowing for a translation to considerably simpler and well-studied models. Our model leverages existing models from AI literature that combine probability theory with logic. The main challenge of query evaluation over probabilistic databases is that it requires solving probabilistic inference which is a notoriously hard problem. This dissertation studies this problem via both (i) foundational results that give new theoretical insights about existing probabilistic inference algorithms, like Read-Once Formulas, Tree-Decompositions, Binary Decision Diagrams, Negation Normal Forms, when applied to the setting of probabilistic databases, which as we will see have their own distinct challenges and expectations, and (ii) building a robust system where the above ideas are leveraged for efficient and reliable query evaluation.
TABLE OF CONTENTS

1 Introduction .................................................... 1
   1.1 Expressing Uncertainty in Databases ......................... 2
      1.1.1 Data Cleaning ........................................ 2
      1.1.2 Information Extraction ................................ 3
   1.2 Overview of Contributions ................................ 5
      1.2.1 Modeling Practical Problems via Simple Probabilistic Models .... 5
      1.2.2 Knowledge Compilation and Database Queries ................ 6
      1.2.3 Treewidth and Query Compilation ....................... 7
      1.2.4 Summary .............................................. 8

2 The Model MVDB .................................................. 9
   2.1 Introduction ............................................... 9
   2.2 Definitions ................................................ 13
      2.2.1 Probabilistic Databases ............................... 13
      2.2.2 Tuple Independent Databases ......................... 14
      2.2.3 Markov Logic Networks (MLNs) ......................... 15
      2.2.4 Our Model MVDB ...................................... 16
      2.2.5 Discussion and Examples .............................. 17
   2.3 Translating From MVDB to INDB ............................. 19
      2.3.1 Main Idea ........................................... 19
      2.3.2 Main Theorem ........................................ 20
      2.3.3 Discussion: Negative Probabilities ................. 22
   2.4 Related Work .............................................. 23

3 Compiling UCQ ................................................... 25
   3.1 Introduction ............................................... 25
   3.2 Background and Definitions ................................. 30
   3.3 Queries with Read-Once Lineage ............................. 39
   3.4 Queries and OBDD ......................................... 43
      3.4.1 Tractable Queries ................................... 45
      3.4.2 Hard Queries ......................................... 47
   3.5 Queries and FBDD ......................................... 50
   3.6 Queries and d-DNNFs ...................................... 59
   3.7 Results on Non-Uniform Classes ............................ 64
   3.8 Conclusion ................................................ 64

4 Query Compilation and Bounded Treewidth ......................... 66
   4.1 Introduction ............................................... 66
   4.2 Results on Treewidth and OBDD ............................. 71
      4.2.1 Treewidth ............................................ 71
      4.2.2 Expression Treewidth ................................. 72
      4.2.3 Background: Some Tractable Functions ............... 73
ACKNOWLEDGEMENTS

First and foremost, I express my gratitude to my advisor Dan Suciu for his guidance and support. This dissertation would not be possible in its current form without his vision and patience with difficult problems. I would also like to thank my reading committee members Magda Balazinska and Pedro Domingos for various fruitful discussions and advice. My co-authors and research collaborators over the years were instrumental in developing my dissertation and I would like to especially thank Wolfgang Gatterbauer, Vibhav Gogate and Vibhor Rastogi. I would be remiss if I didn’t acknowledge the DB group and their frequent seminars and group meetings, in making my stay much more enjoyable and productive. A special thanks to the denizens of Office 502, especially Jonathan Ko and Tyler Robison, because of whom I would come to work even on dreary, rainy days of Seattle. Finally, I want to thank Lindsay Michimoto for all the help and support throughout my PhD program.
Chapter 1

INTRODUCTION

The last decade has seen an explosive growth in the number and size of datasets available for analysis. Unlike the early years of database research, when most of the focus was on enterprise data, today we are swamped with data from sources as diverse as web, sensors and other monitoring tools, physical sciences, social and mobile media, etc. This data deluge, commonly referred to as “Big Data”, has prompted a lot of work on scalability and performance because of the sheer size of these datasets.

But there is another aspect to these datasets that distinguishes them from traditional sources of data, and forms the basis of this dissertation: many of them are imprecise, or uncertain. Sensor readings from a sensornet can be faulty, biased or missing and are often modeled using some probabilistic model [33]. Data extracted from the web has uncertainty that can be modeled by probabilistic models [76] used to extract them. Scientific data, like protein prediction for instance, is inherently uncertain as it is typically done by using some machine learning algorithm that relies on a probabilistic model [65]. A common theme through all these examples is that the existing data is distributed according to a given probabilistic model, unlike traditional datasets where the data was always certain or deterministic.

Even in otherwise precise domains, one can have some noisy, incomplete records. The existing approach, Data Cleaning, that seeks to fix the errors can be expensive, manual, and is known to be error-prone [77]. The general problem of adding data to a warehouse using the Extract-Transform-Load cycle in itself is known to be very expensive because the integration of data in a precise way is a challenging problem. It has been observed [22] that increasingly these expensive pre-processing operations are being skipped because with bigger datasets, they are getting more expensive and infeasible. Moreover, many times users are just interested in statistical queries, hence obviating the need for a precise and clean dataset. Therefore, even in traditional settings, one could benefit from allowing the database to be uncertain.

Many Probabilistic Database Systems [12, 3, 1, 66, 94, 67] that model the uncertainty using probability theory have been developed recently to meet these requirements. A probabilistic
Chapter 1. Introduction

database system returns tuples ordered by their likelihood of being correct. This helps to put the most relevant results at the top; one can also filter out the tuples with very low likelihood to weed out the unlikely results. The main challenge in developing a probabilistic database system is that query evaluation now entails probabilistic inference which is a notoriously hard problem.

Even though this is an old and well-studied problem, in our case, it poses new challenges because of different setting and expectations. Probabilistic inference can be very expensive and even worse the running time can fluctuate wildly with minor changes in the problem instance. Databases on the other hand demand faster and more predictable running time. Moreover, database queries are much more structured – they can be thought of as first order logic sentences – which can be a challenge since they lead to complicated inference problems, but there is also a lot of opportunity here to exploit the structure, as has been done in traditional database settings.

First-order Probabilistic Models (c.f. [42, 34]) aim to combine the expressive power of logic with probability theory; Markov Logic Networks (MLN) [79], a very popular example, express the uncertainty using first-order formulas. MLN have been applied to Information Extraction [74], Record Linkage [87], and Natural Language Processing [75]. These models are capable of expressing relational structure and hence more promising for the probabilistic database setting. In the next section, we will illustrate the promise of expressing the uncertainty using logical formulas in a few database settings. We will then outline the contributions of this dissertation, which broadly speaking are to study probabilistic inference in a similar setting.

1.1 Expressing Uncertainty in Databases

1.1.1 Data Cleaning

Data warehouses have to frequently deal with poor data quality due to manual data entry errors, or because the data has been integrated from disparate sources with different representations. Cleaning this data to perform accurate data analysis is a longstanding research problem in databases. But, increasingly this step is being skipped (c.f. [22]) because it is time-consuming and often unnecessary as users are interested in statistical queries that could work with some uncertainty. The increasing significance of such statistical queries is evidenced by the recent push to support these in the database platforms (see, for e.g [50]). Hence, instead of cleaning the data manually, we show an automated way to approach the same problem with limited effort by transforming the database into a probabilistic database.

One of the main challenges of data cleaning is dealing with records that correspond to the same real-world entity. This is known in the literature by various names like deduplication,
record-linkage, entity resolution, etc. In general, one can resort to various heuristics for merging
duplicate tuples or choosing one over others. Andritsos et. al. [2] propose an approach
to deal with this problem using probabilistic databases. Their focus is on Customer Relationship
Management (CRM) setting, where the data about customers is aggregated from different
sources. We illustrate their approach with a simple example. Suppose we have two different
tables Source1(Name, Address), Source2(Name, Address) that contain data about names and
addresses. We import both into a table CleanSource\textsuperscript{p} using two rules like
\[
\begin{align*}
\text{CleanSource}\textsuperscript{p}(x, y, p) &\leftarrow \text{Source1}(x, y), p = f_1(x, y) \\
\text{CleanSource}\textsuperscript{p}(x, y, p) &\leftarrow \text{Source2}(x, y), p = f_2(x, y)
\end{align*}
\]

$f_1, f_2$ are two procedures that assign the respective probabilities to tuples. Designing these
procedures is non-trivial, but we will just treat it as a black-box for now. Furthermore, we need
to assert that tuples that correspond to same real-world entity are disjoint events, i.e.
\[
\neg (\text{CleanSource}\textsuperscript{p}(x_1, y_1, p_1) \land \text{CleanSource}\textsuperscript{p}(x_2, y_2, p_2) \land y_1 \neq y_2)
\]
The above constraint does require that probability of all tuples with same Name in CleanSource\textsuperscript{p}
sum to at most 1. Procedures $f_1, f_2$ are assumed to guarantee that. The probabilistic relation
CleanSource\textsuperscript{p} now can contain conflicting tuples that would otherwise have to be removed in a
traditional database and if the probabilities are assigned in the right way, the query processor
would take care of putting the more likely results at the top; in other words deduplication is
implicitly done during query evaluation.

1.1.2 Information Extraction

There is a lot of structured data on the web, but most of it can only be accessed in unstructured
form. Many systems [19, 18] have been proposed that attempt to extract this data using Information Extraction (IE) and store it relationally. But these datasets are inherently uncertain and
treating them otherwise as a deterministic table can lead to poor quality of query results (c.f.
[93]). We now illustrate a (simplified) common probabilistic model used for IE and then show
how it can be incorporated into probabilistic databases.

Let us consider a repository of citations. We want to label each citation text with some
label, like author, title, year, etc. First, assume that we have stored citation strings relation-
ally into a table \( \text{tokens}(\text{docID}, \text{pos}, \text{string}) \), which stores for each citation \( \text{docID} \), the string
tokens or \( \text{string} \) and the position, \( \text{pos} \), it appears in. Now we create a probabilistic relation
\( \text{tokens}^P(\text{docID}, \text{pos}, \text{string}, \text{label}, \text{prob}) \) using an existing dictionary \( \text{Dictionary} \) of tokens
and labels.

\[
\text{tokens}^P(x, y, z, l, p) \leftarrow \text{tokens}(x, y, z), \text{Dictionary}(z_1, l), p = f(z, z_1)
\]

The function \( f \) that assigns the probability will once again be just assumed as a black box.
Again, we need to assert that a token can only have one label.

\[
\neg (\text{tokens}^P(x, y, z, l_1, p_1) \land \text{tokens}^P(x, y, z, l_2, p_2) \land l_1 \neq l_2)
\]

But the labels of adjacent tokens are known to be correlated. In particular, we may want
to say ‘author’ is followed by ‘author’; of course its not always true, we want it to be enforced
probabilistically. So we want to assert with some weight \( w \) (whose semantics we need to figure
out) that

\[
\text{tokens}^P(x, y, z_1, \text{‘author’, } p_1) \land \text{tokens}^P(x, y + 1, z_2, \text{‘author’, } p_2)
\]

Similarly, we may want to make other logical assertions with varying weights, but for now we
just end with this one.

The two applications above were meant to illustrate the setting in which probabilistic databases
will be constructed in this dissertation and the constructs we may need to support the proba-
bilistic dependencies they need. The first step is typically constructing a probabilistic relation
and then we need to make some logical assertions on the probabilistic data that may be strict or
\( \text{soft} \). Hence, we need a good way, akin to first-order probabilistic models, to generate probability
distributions given logical formulas that we want to hold with varying weights.
1.2 Overview of Contributions

1.2.1 Modeling Practical Problems via Simple Probabilistic Models

Chapter 2 proposes the probabilistic model we use to model the uncertainty. This model expresses the uncertainty in the database using logical formulas and is based on MLN. An MLN (c.f. [34] for more details) defines a probability distribution over a database given a set of first-order formulas and weights associated to them. If the weight given to a formula is high, then it holds with very high likelihood over the database; conversely, with a negative weight we can penalize every time the formula is satisfied by the database. Besides being expressive and a natural choice in relational setting, they have also been validated in many practical settings and there exist software packages, like Alchemy [98] that one can use to learn them.

We propose a Probabilistic Database model, MVDB, that models the uncertainty in a relational database by defining views called, MarkoViews, over the relations. These views are defined using Union of Conjunctive Queries (UCQ), which are existential positive first-order formulas, or Select-Project-Join-Union SQL queries. Every tuple in a MarkoView has a weight associated to it, that controls the likelihood of that tuple being present in the database, much like in MLN. Since MVDB use UCQ, a subset of first-order formulas that one can use in MLN, they are not as expressive as MLN, but they form a fairly general class that capture many practical settings.

Our result here, as will be shown in Chapter 2, is to show that evaluating any query over this highly expressive model, is equivalent to probabilistic inference over a UCQ formula: computing the probability that the formula is satisfied if each variable is set to true/false with certain probability. More specifically, probability of any boolean UCQ $Q$ over an MVDB $M$: $Pr_M(Q)$, is equivalent to a probabilistic inference problem over formula $Q \lor W$, modulo some constants, where $W$ is independent of $Q$ and depends only on the MarkoViews. The latter problem is a counting problem that has been studied for decades with many results in the form of algorithms and data structures. Moreover, most of the work in probabilistic databases itself, including systems [12, 3] and theoretical results [25, 28] have focused on this model because they are simpler to understand and hence implement and optimize. Our result allows a lot of this literature and existing systems to be applied to real-world settings, where they were deemed hitherto infeasible. This work is going to appear in Proceedings of Very Large Databases (PVLDB), 2012 [56].
1.2.2 Knowledge Compilation and Database Queries

Having reduced our query evaluation to a counting problem over UCQ, we observe that the formula we need to count: \( Q \lor W \), consists of the query \( Q \), and \( W \), which is independent of \( Q \). Furthermore, because \( W \) includes multiple views which are defined over the entire database, it tends to be a lot harder to evaluate than \( Q \) itself. Hence, the strategy we wish to employ is to compile \( W \) offline into a data structure that will enable us to evaluate the probability of \( Q \) over it very efficiently during the online phase. Our contributions listed below can be divided into the theoretical results and the empirical results from the system we developed.

Compiling UCQ

There has been a lot of work in the area of Knowledge Compilation on compiling Boolean formulas into data structures over which various queries, like satisfiability, entailment, model counting, etc, can be answered efficiently. Since the complexity of answering these queries depends on the size of compilation, the theoretical problem here is whether a Boolean Formula admits a compilation of tractable size. In our setting, we study the following data structures: Read-Once Formulas(RO), Ordered Binary Decision Diagrams(OBDD), Free Binary Decision Diagrams(FBDD) and deterministic-Decomposable Negation Normal Forms(d-DNNF) : all of these allow for model counting in the time polynomial in the size of the compilation. However, our problem departs from the normal setting in that instead of a particular Boolean formula, we consider a UCQ(note that \( W \) is a UCQ) and study whether it has polynomial size, or compact, compilation for all databases. This is a standard approach in database theory, called query complexity, and focuses the results on exploiting the query structure and independent of the database. Chapter 3 has the detailed results.

We give complete characterizations for the following two sets of UCQ : (i) always RO independent of database, (ii) admits a compact OBDD for any database. For the two remaining classes, FBDD and d-DNNF, we give a sufficient criteria for a query to have a compact compilation via novel construction algorithms. Furthermore, we show a strict containment hierarchy between the four classes : queries that are RO or admit a compact compilation as OBDD, FBDD, d-DNNF. This result is surprising because over the earlier studied subset of UCQ – Conjunctive Queries without Self Join, these classes were known to coincide [68]. Another significance of this hierarchy result is that as a corollary we get certain Boolean formulas that are simple (monotone with polynomial size DNF) and separate OBDD, FBDD and d-DNNF. This was not known hitherto and in fact finding simple Boolean Formulas that do not have a compact compilation in these languages has been the focus of a few papers [39, 10] ; our results enable one to construct a
family of such Boolean formulas. These results were presented at the International Conference on Database Theory, 2011 [59] and are also going to appear in the Theory of Computing Systems Journal.

**MVDB System**

Chapter 5 presents our approach for evaluating queries in an MVDB. We extend OBDD with some pre-computations and indices to propose an index structure: *MVIndex*. We compile *W* offline into an MVIndex, and when presented with a query *Q*, our system constructs an OBDD for *Q* and evaluates it over the MVIndex for *W*. To this end, we propose a top-down evaluation algorithm that processes queries faster through a limited traversal of the MVIndex by relying on the pre-computations and other indices of the MVIndex. Our work serves, to the best of our knowledge, as the first exploration of implementing OBDD exclusively for probabilistic inference. While earlier works focused on algorithms for constructing OBDD, we explore how to improve the performance of online inference, by computing the probability without actually constructing the OBDD for *Q ∨ W*, or even traversing the complete OBDD for *Q, W*.

Our approach outperforms the existing state-of-the-art software Alchemy [98] by an order of magnitude. Furthermore, our running times, at least for simple queries, closely approximate the query execution times on databases. This work too will be a part of the PVLDB’12 paper [56] from Section 1.2.1.

### 1.2.3 Treewidth and Query Compilation

The *treewidth* of a graph is the measure of a how tree-like a graph is; trees have treewidth 1, while a clique over *n* vertices has treewidth *n*. Many otherwise intractable problems become tractable on graphs of bounded treewidth. In particular, probabilistic inference over graphical models is tractable for graphs of bounded treewidth. There is evidence that converse might also be true\(^1\) [21]; hence bounded treewidth has turned out to be a very important concept in graphical models. To apply this concept to Boolean formulas, two graphs: primal, incidence have been associated to them, so that if any of these graphs has bounded treewidth then inference over the Boolean formula is tractable. But these definitions don’t even capture the class of RO formulas, which are the simplest tractable class of Boolean formulas studied for inference. In Chapter 4, we propose a more robust definition of treewidth for Boolean Formulas and relate it to the Knowledge Compilation hierarchy.

\(^1\)This assumes we do not assume anything about the weight or potential functions involved. For special cases, when these functions come from a particular class, inference could be tractable even for unbounded treewidth. See [55] for an example
Define *expression treewidth* as the minimum treewidth among all circuits that represent a Boolean formula. We show that indeed bounded expression treewidth implies tractable inference. This definition generalizes the existing definitions of treewidth for Boolean Formulas, and in particular RO formulas have treewidth 1. We also relate treewidth to OBDD, by showing that bounded treewidth implies polynomial size OBDD, and also demonstrate a formula with polynomial size OBDD with unbounded expression treewidth. Hence, we show that even with this stronger definition, the notion of treewidth is strictly weaker than OBDD at capturing tractable inference for Boolean formulas. Furthermore, we show an intimate connection between our definition and OBDD: bounded expression pathwidth (defined similarly as expression treewidth) is equivalent to having a constant-width OBDD. We further study these classes and connections for UCQ and UCQ with inequalities (≠) as well. This work was presented at the International Conference on Database Theory, 2012 [58].

1.2.4 Summary

To summarize, we first tackle the gap between modeling practical problems and simple efficient models, by proposing MVDB: they are expressive enough to model many challenging problems, while at the same time query evaluation over this model reduces to a domain very well-understood and studied – model counting over UCQ. We also pursue a theoretical study of various probabilistic inference algorithms from Knowledge Compilation and Tree-decomposition, for our setting specifically. Some of our results, for e.g. on expression treewidth, are actually applicable in general and shed some light on the relationship between treewidth and OBDD that are interesting in their own right. Finally, we propose a framework for query evaluation on probabilistic databases based on knowledge compilation that uses the algorithms from our theoretical study, and demonstrate that we outperform the existing state-of-the-art by orders of magnitude.

**Roadmap:** Chapter 2 presents our model MVDB and illustrates its connection to simpler models. Chapters 3, 4 present the theoretical results on probabilistic inference in our setting, while Chapter 5 evaluates our system that is based on these results. We conclude in Chapter 6.
Chapter 2

THE MODEL MVDB

2.1 Introduction

As we discussed in Chapter 1, the task of analyzing and extracting knowledge from large datasets often requires probabilistic inference over a complex probabilistic model on the data. This step represents a major challenge. Since the complexity of probabilistic inference is determined by the probabilistic model used, a lot of the work has gone into building probabilistic models that are efficient for specific practical scenarios. In this chapter, we present our model MVDB that seeks to find the sweet spot between expressivity and efficiency.

Most of the scalable query processing techniques developed for probabilistic databases assume that the tuples are independent events, or disjoint-independent [12, 3, 71]. For example, MystiQ, MayBMS, and SPROUT report running times of a few seconds on databases of tens of millions of tuples, using a combination of techniques such as safe plans [26], plan re-orderings and functional dependencies [70], Monte-Carlo simulation combined with top-k optimization [78], or approximate confidence computation that tradeoff precision for performance [71]. These systems scale to quite large databases, but are limited to independent probabilistic databases, or disjoint-independent.

Tuple-independent probabilistic databases are insufficient for analyzing and extracting knowledge from practical datasets. As has been shown in the Machine Learning community, modeling correlations is critical in complex knowledge extraction tasks. For example, in Markov Logic Networks (MLN) [79], users can assert arbitrary probabilistic statements over the data, in the form of First Order Logic sentences, and assign a weight. The sentence, called a feature, is expected to hold to a degree indicated by the weight. Each feature may introduce correlations between a large number of base facts, and thus the MLN can express, very concisely, a large Markov Network. MLNs have been demonstrated to be effective at a variety of tasks, such as Information Extraction [74], Record Linkage [87], Natural Language Processing [75]. A benefit
of MLNs is that the same framework can be used both for learning the weights, and for inferring probabilities of new queries.

In this chapter we present a new approach for representing and querying probabilistic databases. Our data model combines probabilistic databases with MLNs: it consists of a collection of probabilistic (tuples are annotated with a probability) and deterministic tables, and a collection of views, called MarkoViews. A MarkoView is expressed by a Union of Conjunctive Queries (UCQ) over the probabilistic and deterministic tables, and associates a weight to each tuple in the answer; intuitively, it asserts a likelihood for that output tuple, and therefore introduces a correlation between all contributing input tuples. A MarkoView can be seen as a set of MLN features, and thus, its weights can be learned as in MLNs; we do not address learning here, but focus solely on inference, or query evaluation. We call a database consisting of probabilistic tables and MarkoViews an MVDB. The data model of MVDBs is significantly richer than that of tuple-independent probabilistic databases, which we denote with INDB.

We show how to translate query evaluation over an MVDB into query evaluation over an INDB. More precisely, we express the probability $P(Q)$ on an MVDB in terms of the probability $P_0(Q \lor W)$, where $W$ is a union of queries, one for each MarkoView; therefore $W$ is a UCQ. To be precise, we show that $P(Q) = P_0(Q \mid \neg W) = (P_0(Q \lor W) - P_0(W))/(1 - P_0(W))$. The probability $P_0$ is on an INDB obtained from the MVDB through a simple, yet quite non-obvious transformation, discussed in Section 2.3. Without going into the transformation details, we would like to mention that $\neg W$ logically stands for the MarkoViews, hence intuitively the translation is computing the probability of the query given that the views hold. The significance of our model is that if $Q$ is a UCQ, so is the translated query; hence, both the query and the probabilistic model are simple and come from well-understood domains. In particular, while there are very few results on tractability of MLNs and none complete, the set of tractable UCQ over INDB is already known [28]. Therefore, the translation moves our problem into a domain that is well-understood and allows for easier detection of tractable instances. We are aware of one more such translation [43] in the literature, but it leads to a more complicated query, which is no longer a UCQ. In contrast, our translation leads to both a simple query (UCQ) and a simple model (tuple independent), where the complexity is well understood and the tractable cases are fully characterized.

Running Example  We illustrate in Figure 2.1 how MarkoViews can be used to add domain knowledge to DBLP [64], a database consisting of a few million tuples. We use an approach developed in the AI community for inferring new relations, such as advisor or affiliation, from a database of citations [79, 86, 62]. Any MVDB has three parts.
Chapter 2. The Model MVDB

### Tables obtained from DBLP

<table>
<thead>
<tr>
<th>Table</th>
<th># Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(aid, name)</td>
<td>1M</td>
</tr>
<tr>
<td>Wrote(aid, pid)</td>
<td>4.5M</td>
</tr>
<tr>
<td>Pub(pid, title, year)</td>
<td>1.7M</td>
</tr>
<tr>
<td>HomePage(aid, url)</td>
<td>18.7K</td>
</tr>
</tbody>
</table>

### Derived tables (standard views)

<table>
<thead>
<tr>
<th>Table</th>
<th># Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>FirstPub(aid, year)</td>
<td>1M</td>
</tr>
<tr>
<td>DBLPAffiliation(aid, inst)</td>
<td>18.7K</td>
</tr>
</tbody>
</table>

Three probabilistic tables Student<sup>P</sup>, Advisor<sup>P</sup>, Affiliation<sup>P</sup>.

#### Possible Tuples

<table>
<thead>
<tr>
<th>Tuples</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student&lt;sup&gt;P&lt;/sup&gt;(aid, year)[year - year' + 1] :- FirstPub(aid, year'), year' - 1 &lt;= year &lt;= year' + 5</td>
<td>aid was a student in that year if his first publication was not long before.</td>
<td>6M</td>
</tr>
<tr>
<td>Advisor&lt;sup&gt;P&lt;/sup&gt;(aid1, aid2)[count(pid)] :- Student(aid1, year), Wrote(aid1, pid), Wrote(aid2, pid), Pub(pid, title, year), ¬ Student(aid2, year), count(pid) &gt; 2</td>
<td>aid2 was aid1’s advisor if they published enough papers together while aid1 was a student and aid2 was not.</td>
<td>.25M</td>
</tr>
<tr>
<td>Affiliation&lt;sup&gt;P&lt;/sup&gt;(aid, inst)[count(pid)] :- Wrote(aid, pid), Wrote(aid2, pid), DBLPAffiliation(aid2, inst), Pub(pid, title, year), aid&lt;&gt;aid2, year&gt;2005, ¬ DBLPAffiliation(aid, inst2)</td>
<td>aid’s affiliation is inst if she published recently with people from inst</td>
<td>.27M</td>
</tr>
</tbody>
</table>

The MarkoViews in the MVDB

<table>
<thead>
<tr>
<th>View definition</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1(aid1, aid2)[count(pid)/2] :- Advisor&lt;sup&gt;P&lt;/sup&gt;(aid1, aid2), Student&lt;sup&gt;P&lt;/sup&gt;(aid1, year), Wrote(aid1, pid), Wrote(aid2, pid), Pub(pid, title, year)</td>
<td>The more they published together while aid2 was a student, the more likely aid1 was his advisor</td>
<td>.25M</td>
</tr>
<tr>
<td>V2(aid1, aid2, aid3)[0] :- Advisor&lt;sup&gt;P&lt;/sup&gt;(aid1, aid2), Advisor&lt;sup&gt;P&lt;/sup&gt;(aid1, aid3), aid2 &lt;&gt; aid3</td>
<td>A person has only one advisor</td>
<td>.38M</td>
</tr>
<tr>
<td>V3(aid1, aid2, inst)[count(pid)/5] :- Affiliation&lt;sup&gt;P&lt;/sup&gt;(aid1, inst), Affiliation&lt;sup&gt;P&lt;/sup&gt;(aid2, inst), Wrote(aid1, pid), Wrote(aid2, pid), Pub(pid, title, year), year &gt; 2004, count(pid)&gt;30</td>
<td>If two people have published a lot together recently, then their affiliations are very likely to be same</td>
<td>1.5K</td>
</tr>
</tbody>
</table>

**Figure 2.1:** An illustration of MarkoViews on the DBLP database

First, the deterministic database. In our example its is described at the top of Figure 2.1, and consists of four base tables, Author, Wrote, Pub, HomePage, and two materialized views (FirstPub(aid, year), which records for each author the year of her first publication, and DBLPAffiliation(aid, inst), which associates an affiliation to some authors<sup>1</sup>).

Second, an MVDB has probabilistic tables, shown in the middle of Figure 2.1: Student<sup>P</sup>(aid, year) stores likely years when an author was a student, Advisor<sup>P</sup>(aid1, aid2) stores likely advisor/advisee relationship, and Affiliation<sup>P</sup>(aid, inst) records inferred affiliations based on

---

<sup>1</sup>We computed the institute from the person’s Webpage, when it was available in DBLP. For example, both Luis Gravano www.cs.columbia.edu/~gravano and Ken Ross www.cs.columbia.edu/~kar have the same institute, www.cs.columbia.edu.
co-authorship. Each probabilistic table is defined by a query, which also associates a weight to every output tuple; for example, \( \text{Student}^p(\text{aid, year})[e^{1-1.15(\text{year}-\text{year}')}]: \ldots \) associates the weight \( w = e^{1-1.15(\text{year}-\text{year}')} \) to its output. Weights are often preferred over probabilities when the probability function is a product of potential functions, as in MLNs and MVDBs. The intuition is that the weight \( w \) represents the odds of a probability \( p, w = p/(1-p) \) (formal definition in Section 2.2).

Third, the MVDB contains a set of MarkoViews, which in our example are shown at the bottom of Figure 2.1. Each MarkoView is a query over probabilistic tables, and its purpose is to define some correlations between the tuples in those tables. It does this by defining a view over the probabilistic tables, then asserting a certain weight for the tuples in the view. Weights < 1 define a negative correlation, weights > 1 define a positive correlation, and a weight = 1 means independence. A weight = 0 means a hard constraint: the view must be empty. For example, the MarkoView \( V_1 \) defines a correlation between a tuple in \( \text{Student}^p \) and a tuple in \( \text{Advisor}^p \): it states that the more papers two people co-author during the years when the second person was a student, the more likely that the first person was his advisor. \( V_2 \) defines a hard constraint: each person can have only one advisor. Finally, \( V_3 \) introduces positive correlations between common affiliations for people who published a lot together.

Consider now the following simple query on the MVDB: find all students advised by Sam Madden. The query, written over the MVDB, is shown in Figure 2.2 (a). If the tuples in \( \text{Student}^p \) and \( \text{Advisor}^p \) were independent random variables, then this query could be computed very efficiently, because it is a safe query [26]. However, MarkoViews introduce correlations between the probabilistic tuples. We show that the probability of an answer \( \text{aid}, P(Q(\text{aid})) \), can be expressed in terms of the probability of a query \( P_0(Q(\text{aid}) \lor w) \) over a tuple independent database. We give the exact formula in Figure 2.2 (c). The new INDB has five tuple-independent probabilistic tables: \( \text{Student}^p \) and \( \text{Advisor}^p \) (which have the same sets of possible tuples as in the MVDB, and with the same weights), and \( NV1^p, NV2^p, NV3^p \), which are three new, tuple-independent probabilistic tables, whose possible tuples are obtained from the MarkoViews \( V_1, V_2, V_3 \), and whose weights are derived from the latter through the formula \( (1 - w)/w \). Note that if \( w > 1 \), then the translated weight is negative, which, in turn, corresponds to a negative probability; thus, the INDB may have some tuples with negative probabilities! However, the expression for \( P(Q) \) is exact, hence its value is guaranteed in \([0,1]\). We discuss the translation from MarkoViews to INDBs in Section 2.3, and, in particular, the implications of having negative probabilities in a database in Section 2.3.3.

Query evaluation on an MVDB reduces to evaluating a formula such as the one in Figure 2.2
Q(aid) := \neg \text{Student}^p(aid), \text{Advisor}^p(aid, aid1), \\
\quad \text{Author}^p(aid, n), \text{Author}^p(aid1, n1), \\
\quad n1 \text{ like } '%\text{Madden}%' 

(a)

W1 := \neg \text{NV1}^p(aid1, aid2), \text{Advisor}^p(aid1, aid2), \\
\quad \text{Student}^p(aid1, year), \text{Wrote}(aid1, pid), \\
\quad \text{Wrote}(aid2, pid), \text{Pub}(pid, title, year) 

W2 := \neg \text{NV2}^p(aid1, aid2, aid3), \text{Advisor}^p(aid1, aid2), \\
\quad \text{Advisor}^p(aid1, aid3), \text{aid2} \neq \text{aid3} 

W3 := \neg \text{NV3}^p(aid1, aid2, inst), \text{Affiliation}^p(aid1, inst), \\
\quad \text{Affiliation}^p(aid2, inst), \text{Wrote}(aid1, pid), \\
\quad \text{Wrote}(aid2, pid), \text{Pub}(pid, title, year), \\
\quad \text{year} > 2004, \text{count}(pid) > 30 

W := W1 \lor W2 \lor W3 

(b)

P(Q(aid)) = \frac{P_0(Q(aid) \lor W) - P_0(W)}{1 - P_0(W)} 

(c)

Figure 2.2: A query Q over the MVDB (a); helper queries on the INDB (b); expressing the probability on the MVDB in terms of the probability on the INDB (c).

(c), on an INDB. The query dependent part of this expression is $P_0(Q(aid) \lor W)$. While this requires standard evaluation of a query over a tuple-independent database, it can still be a major challenge: the reason is that the lineage of W is usually large, because it includes most probabilistic tuples and all tuples in the MarkoViews. By contrast, the lineage of Q is much smaller, because it has a selection predicate (“%Madden%”).

2.2 Definitions

We use R to denote a relational schema with relation names $R_1, R_2, \ldots, R_k$. We assume each relation has a key. A database instance $I$ is a $k$-tuple $(R^I_1, \ldots, R^I_k)$, where $R^I_i$ is an instance of the relation $R_i$; with some abuse of notation we drop the superscript $I$ and write $R_i$ for both the relation name and the instance of that relation.

\footnote{As usual, if there is no natural key, then the set of all attributes constitutes a key.}
2.2.1 Probabilistic Databases

A probabilistic database is a pair $D = (W, P)$, where $W = \{I_1, \ldots, I_N\}$ is a set of instances, called possible worlds, and $P : W \rightarrow [0, 1]$ is a function such that $\sum_{j=1,N} P(I_j) = 1$. Thus, the instance is not known with certainty: every possible world $I_j$ has some probability, $P(I_j)$. A relation $R_i$ is called deterministic if it has the same instance in all possible worlds $R_i^j = \cdots = R_i^N$; otherwise, we say that $R_i$ is probabilistic, and we sometimes add the superscript $p$, writing $R_i^p$ to indicate that $R_i$ is probabilistic. For example in Figure 2.1, the deterministic relations are Author, Wrote, Pub, HomePage, and the probabilistic relations are Student$^p$, Advisor$^p$, Affiliation$^p$.

We denote $\text{Tup}$ the set of possible tuples, i.e. the set of all tuples occurring in all possible worlds $I_1, \ldots, I_N$. The tuples in $\text{Tup}$ include the relation name where they come from, e.g. the tuples $R(a, b)$ and $S(a, b)$ are considered distinct tuples in $\text{Tup}$. We associate to each tuple $t \in \text{Tup}$ a Boolean random variable, denoted $X_t$: given a random world $I_j$, $X_t = 0$ if $t \notin I_j$ and $X_t = 1$ if $t \in I_j$. The probability of the event $\exists j. t \in I_j$ is denoted $P(t)$ or $P(X_t)$, and is also called the marginal probability of the tuple $t$.

A query is denoted $Q(\bar{x})$, where $\bar{x}$ are called free variables, or head variables. The answer to $Q$ on an instance $I$, $Q(I)$, is the set of all tuples $\bar{a}$ s.t. $I \models Q(\bar{a})$, where $Q(\bar{a})$ is the Boolean query obtained by substituting the head variables $\bar{x}$ with the constants $\bar{a}$. The answer on a probabilistic database, $D$, is a set of pairs of the form $(\bar{a}, p)$, where $\bar{a} \in Q(I)$ for some possible world $I$, and:

$$p = P(Q(\bar{a})) = \sum_{t \in Q(I)} P(I_t)$$

The queries we consider are Unions of Conjunctive Queries, denoted UCQ, which are expressions of the form $Q(\bar{x}) = Q_1(\bar{x}) \vee \ldots \vee Q_m(\bar{x})$, where each $Q_i(\bar{x})$ is a conjunctive query, i.e. has the form $\exists \bar{y}_i. \varphi_i(\bar{x}, \bar{y}_i)$ where $\varphi_i$ is a conjunction of positive, relational atoms, and/or inequality predicates, such as $z < 5$. We write queries in datalog notation, indicating the head variables. For example, $Q(x) = R(x), S(x, y)$ denotes the query $\exists y. R(x) \wedge S(x, y)$, while $Q = R(x), S(x, y)$ denotes the Boolean query $\exists x. \exists y. R(x) \wedge S(x, y)$. With some abuse of notation, we allow the use of aggregates and negations, but only on the deterministic tables\(^3\).

\(^3\)For example we used $\text{count}(\text{pid})$ in $\text{V3}$ in Figure 2.1, meaning that the subquery consisting of the last two lines is first computed as a view over the deterministic tables, then the resulting view is used as a single table in $\text{V3}$; after this transformation, $\text{V3}$ becomes a conjunctive query over probabilistic and deterministic tables.
2.2.2 Tuple Independent Databases

A probabilistic database is \textit{tuple-independent} if, for any set of possible tuples \( t_1, t_2, \ldots, t_n \), the events \( X_{t_1}, X_{t_2}, \ldots, X_{t_n} \) are independent. We write \( D_0 \) for a tuple-independent database, and also denote it with \( \text{INDB} \). It is uniquely defined by a pair \((\text{Tup}, p)\), where \( \text{Tup} \) is the set of possible tuples and \( p : \text{Tup} \to [0, 1] \) is any function. The possible worlds are all subsets \( I \subseteq \text{Tup} \), and their probabilities are \( P(I) = \prod_{t \in I} p(t) \cdot \prod_{t \in \text{Tup} \setminus I} (1 - p(t)) \).

Tuple-independent probabilistic databases are the simplest, and most intensively studied types of probabilistic databases [89]. Even though the input tuples are independent, correlations are introduced during query evaluation, and query evaluation is, in general, \#P-complete, e.g. for the Boolean query \( Q = R(x), S(x, y), T(y) \). However, these probabilistic databases are now well understood, and a complete characterization of UCQ queries into \#P-complete and PTIME queries exists [28]. In addition, many practical methods for query evaluation on tuple-independent databases have been proposed in the literature (see Section 2.4).

2.2.3 Markov Logic Networks (MLNs)

A Markov Logic Network is a set \( L = \{(F_1, w_1), \ldots, (F_m, w_m)\} \), where each \( F_i \) is a formula in First Order Logic called a \textit{feature}, and each \( w_i \) is a number called the \textit{weight} [79, 35]. The formulas are over a relational vocabulary \( \text{R} \), and may have free variables, which are interpreted as being universally quantified. Let \( C \) be a finite set of constants. A \textit{grounding} of a formula \( F_i \) is a formula of the form \( F_i[\bar{a}/\bar{x}] \), where the free variables \( \bar{x} \) of \( F_i \) are substituted with some constants \( \bar{a} \) in \( C \); let \( G(F_i) \) denote the set of grounding of \( F_i \), and let \( G(L) = \{(G, w_i) \mid \exists (F_i, w_i) \in L : G \in G(F_i)\} \) be the \textit{grounded MLN}. Let \( \text{Tup} \) be the set of ground tuples with the relation symbols in \( \text{R} \) and constants in \( C \). The \textit{weight} \( \Phi(I) \) of a world \( I \subseteq \text{Tup} \), and the \textit{partition function} \( Z \) are:

\[
\Phi(I) = \prod_{(G, w) \in G(L) : I = G} w \tag{2.1}
\]

\[
Z = \sum_{I \subseteq \text{Tup}} \Phi(I) \tag{2.2}
\]

\textbf{Definition 2.1.} The semantics of an MLN \( L \) is the probabilistic database \( D_L = (W, P) \), where \( W = \{I \mid I \subseteq \text{Tup}\} \) and \( P(I) = \Phi(I)/Z \) for all \( I \subseteq \text{Tup} \).

The intuition is the following. Any subset of tuples is a possible world, and its weight is the product of the weights of all grounded features that are true in that world. The probability is obtained by normalizing with \( Z \). Note that a feature weight \( w > 1 \) means that worlds where the feature holds are more likely; \( w < 1 \) means that worlds were the feature holds are less likely;
and \( w = 1 \) means indifference. A weight \( w = \infty \) is interpreted as a hard constraint: only worlds that satisfy the feature are considered as possible. This can be seen by letting \( w \rightarrow \infty \) in the expression \( P(I) = \Phi(I)/Z \).

MLNs have been used in several applications of Machine Learning [87, 74, 75, 35]. One reason for their popularity is that they use the same formalism for both learning (of the weights \( w \)) and for probabilistic inference. There are two types of inferences within MLNs: MAP (maximum a posteriori) inference, which computes the most likely world, and marginal inference, which sums the probabilities of all worlds satisfying the query. We only address the latter here, but our solutions easily generalize to solve the MAP inference problem as well.

**Tuple-Independent Databases Revisited**  Consider two possible tuples: \( R(a_1), R(a_2) \), and the MLN consisting of features: \( (R(a_1), w_1), (R(a_2), w_2) \). There are four possible worlds: \( \emptyset, \{R(a_1)\}, \{R(a_2)\}, \{R(a_1), R(a_2)\} \), and their weights \( \Phi(I) \) are:

\[
1 \quad w_1 \quad w_2 \quad w_1w_2
\]

The partition function is \( Z = 1 + w_1 + w_2 + w_1w_2 = (1 + w_1)(1 + w_2) \). In this case the MLN defines a tuple-independent database, where the tuples \( R(a_1), R(a_2) \) have probabilities \( p_1 = w_1/(1+w_1) \), and \( p_2 = w_2/(1+w_2) \). Indeed, the four possible worlds have probabilities

\[
(1-p_1)(1-p_2) \quad p_1(1-p_2) \quad (1-p_1)p_2 \quad p_1p_2.
\]

and this distribution is equivalent to the above (up to the multiplicative factor \( Z \)). More generally, we define:

**Definition 2.2.** A tuple-independent database, INDB, is a pair \( D_0 = (\text{Tup}_0, w_0) \) where \( \text{Tup}_0 \) is a set of possible tuples and \( w_0(t) \) associates a real number to each tuple \( t \).

This definition is equivalent to the one given earlier, by setting the tuple probability to \( p(t) = w_0(t)/(1+w_0(t)) \). Note that in a tuple-independent database a weight, \( w \), represents the odds, \( w = p/(1-p) \). In other words, weight values of 0, 1, \( \infty \) correspond to probabilities 0, 1/2, 1 respectively. From now on, unless otherwise stated, we will assume in the rest of this paper that an INDB is given as in **Theorem 2.2**, that is, we are given the weights of the tuples, not their probabilities. The tuple’s probability can always be recovered as \( w/(1+w) \).

**2.2.4 Our Model MVDB**

In this section we introduce Markov Views (MARKOVVIEW) and Markov View Databases (MVDB).
Definition 2.3. A MarkoView is a rule of the form:

\[ V(\bar{x})[w_{expr}] : \neg Q \quad (2.3) \]

where \( V \) is the view name, \( Q \) is a UCQ, \( \bar{x} \) are head variables, and \( w_{expr} \) is an expression representing a non-negative weight.

Let \( R \) be a relational schema. An MVDB is a triple \((\text{Tup}, w, V)\), where \( \text{Tup} \) is a set of possible tuples over the schema \( R \), \( w : \text{Tup} \rightarrow [0, \infty] \) associates a weight to each possible tuple, and \( V \) is a set of MarkoViews.

Let \( I_{\text{poss}} \) denote the deterministic database instance over the schema \( R \) consisting of all possible tuples \( \text{Tup} \) (forgetting their probabilities). For each MarkoView \( V \), denote \( \text{Tup}_V \) the result of evaluating \( V \) on \( I_{\text{poss}} \), and let \( \text{Tup}_V = \bigcup_V \text{Tup}_V \): this is the set of all possible tuples in all views. For each \( t \in \text{Tup}_V \), let \( w_V(t) \) denote its weight, as computed according by the view \( V \).

The semantics of an MVDB is a restricted MLN having one feature \((F_t, w_t)\) for each tuple \( t \in \text{Tup} \cup \text{Tup}_V \), defined as follows. For each possible tuple \( t \in \text{Tup} \) in the probabilistic database we associate the feature where the formula is \( F_t = t \), the grounded atom represented by the tuple \( t \), and the weight is \( w_t = w(t) \). For each possible tuple \( t \in \text{Tup}_V \) in a view \( V \), consider the view definition \( V(\bar{x})[w_{expr}] : \neg Q \): we associate \( t \) with the feature where the formula is \( F_t = Q(t) \), the Boolean query obtained by substituting the head variables \( \bar{x} \) with the tuple \( t \), and the weight is \( w_V(t) \). Thus, the feature \( F_t \) is a ground tuple in the first case, and a Boolean UCQ in the second case. In both cases, \( F_t \) has no free variables, and therefore it is already “grounded” according to the terminology of MLN’s.

Definition 2.4. Let \((\text{Tup}, w, V)\) be an MVDB. Its semantics is given by the probabilistic database \( D_L \) (Theorem 2.1) associated to the MLN \( L = \{ (F_t, w_t) \mid t \in \text{Tup} \cup \text{Tup}_V \} \).

We denote \( P(Q) \) the probability of a Boolean query \( Q \) on the probabilistic database associated to the MVDB. The problem here is to compute \( P(Q) \).

2.2.5 Discussion and Examples

An MVDB generalizes tuple-independent databases. Any INDB, \((\text{Tup}, w)\) is in particular a MVDB, \((\text{Tup}, w, \emptyset)\), without any MarkoViews. But MVDBs are much more powerful than INDBs, because they can impose correlations between arbitrary sets of tuples. We illustrate with several examples.
Example 2.1. Consider the MVDB with two possible tuples, \( \text{Tup} = \{R(a), S(a)\} \), with weights \( w_1, w_2 \) respectively, and a single MarkoView:

\[
V(x)[w] : -R(x), S(x)
\]

Here \( w \) is a constant. Intuitively, the view asserts that the tuples in \( R \) and \( S \) are correlated, by some weight \( w \). We show now the associated MLN, \( L \). There is a single tuple in the MarkoView, \( \text{Tup}_V = \{V(a)\} \), and therefore the MLN has three features:

\[
L = \{ (R(a), w_1), (S(a), w_2), (R(a) \land S(a), w) \}
\]

The probabilistic database \( D_L \) has four possible worlds, \( \emptyset, \{R(a)\}, \{S(a)\}, \{R(a), S(a)\} \), with weights:

\[
1 \quad w_1 \quad w_2 \quad ww_1w_2
\]

Therefore, the two tuples \( R(a), S(a) \) are correlated. When \( w = 0 \), then \( R(a) \) and \( S(a) \) are exclusive events; when \( w = 1 \) then they are independent events; when \( w = \infty \) then both are certain tuples. More generally, when \( w < 1 \) then \( R(a), S(a) \) are negatively correlated, and when \( w > 1 \) they are positively correlated.

Example 2.2. A more complex example is \( V(x)[w] = R(x), S(x,y) \). Each tuple \( t = V(a) \) in the view defines the MLN feature \( F_t = \exists y. R(a), S(a,y) \). The lineage of this Boolean query is \( (R(a) \land S(a,b_1)) \lor (R(a) \land S(a,b_2)) \lor \ldots \), and the MarkoView introduces a correlation between all tuples in the lineage expression. Here the MarkoView introduces a correlation between a large number of tuples, in turn forming a large clique in the associated Markov Network. The view \( V_1 \) in Figure 2.1 is of this type, because the year attribute of \( \text{Student}^p \) is projected out.

Example 2.3. An example of a large MVDB is given in Figure 2.1. The set of possible tuples \( \text{Tup} \) are defined by the deterministic tables at the top, and by the three queries in the middle of Figure 2.1 (which also define the weight function \( w \) for the probabilistic tables). The MarkoViews are \( V_1, V_2, V_3 \) at the bottom of Figure 2.1. The MVDB has over 6M probabilistic tuples (correlated) and over 6M deterministic tuples.

Thus, MarkoViews allow us to express both positive and negative correlations between probabilistic tuples. They are, however, strictly less expressive than MLNs, because they only allow UCQ as features. The advantage of imposing such a restriction is that it allows us to translate query evaluation of UCQs on MVDBs to query evaluation of UCQs on tuple-independent
databases, as we show in the next section. For example, MarkoViews cannot express a feature like “transitively closed”, which would be written in MLN’s like: \(((R(x,y), R(y,z) \Rightarrow R(x,z)), w)\). One can express this in MarkoViews if we extend them to allow negations:

\[
V(x,y,z)[1.0/w] :- R(x,y), R(y,z), \neg R(x,z)
\]

While the MLN rewards every grounding of \(R(x,y), R(y,z) \Rightarrow R(x,z)\) by a factor \(w\), the MarkoView penalizes every violation by a factor \(1/w\): the two features are equivalent. The technique that we describe in the next section applies to this view too, i.e. queries can still be translated to tuple-independent database; the problem is that the new query contains negation, which is less well understood in probabilistic databases. For that reason, we are currently restricting MarkoViews to be UCQs, without negation.

### 2.3 Translating From MVDB to INDB

#### 2.3.1 Main Idea

Consider two possible tuples \(R(a), S(a)\) with weights \(w_1, w_2\) and the MarkoView \(V(x)[w] : \neg R(x), S(x)\), where \(w\) is a constant. The four possible worlds have weights:

\[
\begin{array}{cccc}
1 & w_1 & w_2 & ww_1w_2 \\
\end{array}
\]

We show how to reduce the probability \(P(Q)\) to the probability of a query on a tuple-independent databases. Any Boolean query \(Q\) corresponds to a subset of worlds; there are \(2^4 = 16\) inequivalent Boolean queries. Then \(P(Q) = \Phi(Q)/Z\), where \(\Phi(Q)\) is the sum of the weights corresponding to the worlds that satisfy \(Q\). For a very concrete example, if \(Q = R(a) \lor S(a)\) then \(\Phi(Q) = w_1 + w_2 + ww_1w_2\), and \(P(Q) = (w_1 + w_2 + ww_1w_2)/(1 + w_1 + w_2 + ww_1w_2)\).

Consider now a tuple-independent database over three relations, \(R, S, NV\), with three possible tuples \(R(a), S(a), NV(a)\); their weights are \(w_1, w_2, w_0\), where \(w_1, w_2\) are as above and \(w_0\) will be determined shortly. Consider the hard constraint \(\neg W\), where \(W \equiv R(a) \land S(a) \land NV(a)\). If one defines \(V(a) = \neg NV(a)\), then \(\neg W \equiv (R(a), S(a) \Rightarrow V(a))\). Seven out of the eight possible worlds satisfy \(\neg W\), and their weights are:

<table>
<thead>
<tr>
<th>(\neg NV(a))</th>
<th>1</th>
<th>(w_1)</th>
<th>(w_2)</th>
<th>(w_1w_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NV(a)</td>
<td>(w_0)</td>
<td>(w_0w_1)</td>
<td>(w_0w_2)</td>
<td>-</td>
</tr>
</tbody>
</table>

| Total:          | \(1 + w_0\) | \((1 + w_0)w_1\) | \((1 + w_0)w_2\) | \(w_1w_2\) |

We write \(\Phi_0, Z_0 = \Phi_0(\text{true})\) and \(P_0\) for the weight function, the partition function, and the probability defined by this tuple-independent database. Suppose we want to compute \(P(Q)\) for
some query $Q$ over the schema $R, S$, and consider its weight $\Phi_0(Q \land \neg W)$ in the new database: each column in the table above is either entirely included in the sum or is not included at all, because $Q$ does not refer to $NV$, only to $R$ and $S$. Thus, $\Phi_0(Q \land \neg W)$ is a sum of a subset of the weights in the last row, labeled “Total”. Set $w_0$ such that $w = 1/(1 + w_0)$: then

$\Phi_0(Q \land \neg W) = (1 + w_0) \cdot \Phi(Q)$. For example, if $Q = R(a) \lor S(a)$ then:

$$\Phi_0(Q \land \neg W) = (1 + w_0)w_1 + (1 + w_0)w_2 + w_1w_2$$

$$= (1 + w_0) \cdot (w_1 + w_2 + \frac{1}{1 + w_0}w_1w_2)$$

$$= (1 + w_0) \cdot \Phi(Q)$$

Therefore:

$$P(Q) = \frac{\Phi(Q)}{\Phi(\text{true})} = \frac{\Phi_0(Q \land \neg W)}{\Phi_0(\neg W)} = \frac{P_0(Q \land \neg W)}{P_0(\neg W) - P_0(W)}$$

In summary, we have reduced the problem of evaluating $P(Q)$ on an MVDB to the problem of evaluating the probabilities $P_0(Q \lor W)$ and $P_0(W)$ on a tuple-independent database. Note that we can also express it as a conditional probability, $P(Q) = P_0(Q \mid \neg W)$: we prefer to use the expression above because both probability expressions $P_0(Q \lor W)$ and $P_0(W)$ are for UCQ’s, for which query evaluation on tuple-independent databases is very well understood.

### 2.3.2 Main Theorem

**Definition 2.5.** Consider an MVDB $D = (\text{Tup}, w, V)$ over the relational schema $R$. Let $\text{Tup}_V$ be the set of all possible tuples in all views, and $w_V : \text{Tup}_V \to [0, \infty]$ be their weights (Section 2.2.4).

Let $NV$ denote the relational schema having one relation symbol $NV_i$ for each MarkoView $V_i$.

The tuple-independent database associated to $D$ is the following database over the schema $R \cup NV$: $D_0 = (\text{Tup}_0, w_0)$, where the set of possible tuples and the weight function are defined

$\Phi(Q)$ is a sum of a subset of weights in $1, w_1, w_2, w_1w_2$ while $\Phi_0(Q \land \neg W)$ is a sum of the corresponding subset of weights in $(1 + w_0), (1 + w_0)w_1, (1 + w_0)w_2, w_1w_2$.
Chapter 2. The Model MVDB

by:

\[ \text{Tup}_0 = \text{Tup} \cup \text{Tup}_{NV} \]
\[ \text{Tup}_{NV} = \{ NV_i(\bar{a}) \mid V_i(\bar{a}) \in \text{Tup}_V \} \]
\[ w_0(t) = \begin{cases} w(t) & \text{if } t \in \text{Tup}, \\ \frac{1-w(t)}{w(t)} & \text{if } t \in \text{Tup}_V \end{cases} \]

In other words, to compute the INDB from the MVDB one proceeds as follows. All deterministic or probabilistic tables in MVDB become independent tables in the INDB, with the same weights. (A deterministic table in the MVDB has all weights = \(\infty\), hence it remains a deterministic table in the INDB.) In addition, create a new relation \(NV_i\) for each MarkoView \(V_i\); the possible tuples are all the possible tuples in the view \(V_i\), and their weights are \(w_0 = (1-w)/w\), where \(w\) is the weight defined by the MarkoView for that tuple. (Note that \(w = 1/(1+w_0)\), a fact we will use in the proof.) The next theorem is the main theoretical result of this chapter, and key to our technique. It says that, in order to compute UCQ queries on the MVDB, it suffices to compute UCQ queries over the associated INDB:

**Theorem 2.6.**  Let \(V = \{V_1, \ldots, V_m\}\) be the MarkoView in the MVDB. For \(i = 1, m\) let \(Q_i\) be the UCQ defining the view \(V_i\). Denote \(W_i\) the Boolean query

\[ W_i = \exists \bar{x}_i. NV_i(\bar{x}_i) \land Q_i(\bar{x}_i) \] (2.4)

Further define \(W = \bigvee_i W_i\) (this is also a Boolean UCQ query). Then, for every Boolean query \(Q\), the following holds:

\[ P(Q) = \frac{P_0(Q \lor W) - P_0(W)}{1 - P_0(W)} \] (2.5)

**Proof.** We follow the same steps as in Section 2.3.1. We start by computing \(\Phi(Q)\) according to Theorem 2.4:

\[ \Phi(Q) = \sum_{J \subseteq \text{Tup}} \Phi(J) = \sum_{J \subseteq \text{Tup}} \Phi_0(J) \cdot \prod_{t \in \text{Tup}_V \land J \models F_i} w(t) \]

Recall that the MLN associated to the MVDB has two sets of features \(F_i\); for \(t \in \text{Tup}\) and for \(t \in \text{Tup}_V\). Thus, \(\Phi(J)\) has two parts: \(\prod_{t \in J} w(t)\), which is the same as, \(\Phi_0(J)\), the weight of \(J\) in the INDB; and the product of all the weights of features satisfied by the world \(J\). This justifies \(\Phi(Q)\).
Next, we compute $\Phi_0(Q \land \neg W)$:

$$\Phi_0(Q \land \neg W) = \sum_{I : I \models \neg W} \Phi_0(I) \quad (2.6)$$

The possible world $I$ ranges over all subsets of $\text{Tup} \cup \text{Tup}_{NV}$, and, hence, it can be written as $I = J \cup K$, where $J \subseteq \text{Tup}$ and $K \subseteq \text{Tup}_{NV}$. Fix the $J$ component of $I$. Fix a MarkoView $(V_i, w, Q_i(\bar{x}_i))$ and a possible tuple in this MarkoView, $t = V_i(\bar{a}) \in \text{Tup}_{V_i}$. There are two cases. Case 1: $\bar{a} \notin Q_i(J)$; equivalently, $J \not\models F_t$. In that case, we can satisfy the constraint $Q_i(\bar{x}_i) \Rightarrow \neg NV_i(\bar{x}_i)$, by either including or not including the tuple $t = NV_i(\bar{a})$ in $K$: the sum of the two weights is $1 + w_0(t)$. Case 2: $\bar{a} \notin Q_i(J)$; equivalently $J \models F_t$. In that case, in order to satisfy the constraint we must remove the tuple $t$ in $K$, hence its multiplicative contribution is 1. Thus, we rewrite Equation 2.6 grouping by $J$:

$$\sum_{J,K : J \cup K \models \neg W} \Phi_0(J \cup K) = \sum_{J} \Phi_0(J) \prod_{t : J \not\models F_t} (1 + w_0(t)) \cdot \prod_{t : J \models F_t} 1 = P \cdot \sum_{J} \Phi_0(J) \cdot \prod_{t : J \models F_t} \frac{1}{1 + w_0(t)} = P \cdot \Phi(Q)$$

where $P = \prod_t (1 + w_0(t))$. In the last line we used the fact that $1/(1 + w_0(t)) = w(t)$. The theorem follows now by noting that $P(Q) = \Phi(Q)/\Phi(\text{true})$, and repeating the argument at the end of Section 2.3.1.

We end this section with three observations. First, we note that the lineage of $Q \lor W$ is no larger than the lineages of $Q$ and $W$ combined. In fact, the lineage of $Q \lor W$ is precisely the disjunction of the two lineage expressions of $Q$ and $W$ respectively. Thus, our translation does not add any complexity to the probabilistic inference problem. Second, we note that the translation gives us an immediate tool for identifying tractable cases. The UCQ queries that can be evaluated in PTIME over INDB are fully characterized in [28], and are called safe queries. An immediate corollary of Theorem 2.6 is that query evaluation over MVDB is tractable if both $Q \lor W$ and $W$ are safe. Other tractability results on INDB also carry over immediately to MVDBs, for example the query compilation results in [59]. Finally, a comment on denial views, which are MarkoViews where the weight is 0: for example view $V_2$ in Figure 2.1 is a denial view. In that case $NW$ is a deterministic table, since its weight is $(1 - 0)/0 = \infty$, and can be dropped entirely from the definition of $W_i$, simplifying Equation 2.4.
2.3.3 Discussion: Negative Probabilities

Some of the probabilities in $D_0$ may be negative: if $w > 1$, then $w_0 = (1 - w)/w < 0$, and the probability $p_0 = w_0/(1 + w_0) = 1 - w$ is negative. This may raise questions about the soundness of our approach. However, negative probabilities have already been considered before; it has been proven that probability theory can be consistently extended to allow for negative probabilities [6], and there is interest in applying them to quantum mechanics [16] and financial modeling [49]. In our setting, the negative probabilities have a much more benign role: they are simply numbers that need to be plugged into the right hand side of Equation 2.5 to make the equality hold. Every query answer $P(Q)$ will be a correct probability, in $[0, 1]$, even if the probabilities $P_0$ on the right are negative. All familiar equalities still hold for $P_0$: for example, the rules for negation $P_0(\neg Q) = 1 - P_0(Q)$, and inclusion/exclusion $P_0(Q_1 \lor Q_2) = P_0(Q_1) + P_0(Q_2) - P_0(Q_1 \land Q_2)$ still hold; similarly, if $Q_1, Q_2$ are independent queries (their lineages have no common variables) then the laws of independence continue to hold: $P_0(Q_1 \land Q_2) = P_0(Q_1)P(Q_2)$ and $P_0(Q_1 \lor Q_2) = 1 - (1 - P_0(Q_1))(1 - P_2(Q_2))$; similarly, Shannon’s expansion formula holds. In fact all exact inference methods (Davis-Putnam procedure, tree-width based methods, OBDD constructions) work without any modification on probability spaces with negative probabilities. We take advantage of this fact in the next section.

Approximate methods, however, no longer work out-of-the-box. For example, the inequality $P_0(Q_1 \lor Q_2) \leq P_0(Q_1) + P_0(Q_2)$ must be replaced with the weaker $|P_0(Q_1 \lor Q_2)| \leq |P_0(Q_1)| + |P_0(Q_2)|$; another issue is that, approximation methods may no longer return final values in $[0, 1]$; it is also unclear how to run sampling-based methods. We do not consider any approximation or sampling methods here, instead restrict the discussion only to exact probability computation. We show next how to do this quite effectively.

2.4 Related Work

Query Evaluation on Tuple Independent Databases. This problem reduces to that of evaluating the probability $P(\Phi)$ of a Boolean formula $\Phi$ (over the Boolean variables $X_t$) called the lineage of the query $Q$. There are two lines of research on query evaluation on INDB’s. One aims at identifying classes of queries for which $P(Q)$ can be computed in polynomial time in the size of the database: these are called safe queries. For UCQ’s without inequality predicates (like $x < 5$ or $x \neq y$), there exists a syntactic characterization of safe queries that is complete, i.e. a dichotomy: either $Q$ is safe and then one can compute $P(Q)$ in PTIME, or $Q$ is unsafe and then computing $P(Q)$ is $\#P$-hard. The other line of research aims at developing effective
heuristics for computing $P(\Phi)$. In this formulation the problem is related to model counting, for which several effective heuristics exists. The most popular is the Davis-Putnam procedure, which is based on Shannon expansion, see for example [7], and which has been used in probabilistic databases in [61]; refinements of this procedure also exist [36]. Based on ideas similar to the Davis-Putnman procedure, a number of techniques have been described for using OBBDs for query evaluation on probabilistic databases [68, 69, 59]. Finally, these evaluation methods have been extended with several approximation techniques [71, 40].

Markov and Bayesian Networks in Probabilistic Databases. Several proposals exists for extending probabilistic databases to represent Markov Networks or Bayesian Networks. For example, in [60] the probabilistic database is defined directly as a Markov Network. The system uses a novel indexing techniques for the junction tree decomposition of the network, allowing queries on a large database to be evaluated efficiently at runtime. The method works very well, but only when the tree width of the Markov Network is small, otherwise the method is intractable. Note that the Markov Networks in MVDBs have very large cliques, hence very large tree widths. Tuffy [67] implements MLN’s directly in a relational database system. Its focus is less on developing new algorithms, instead it is on leveraging query processing in the database in order to implement existing MLN algorithms. Two other projects [97, 94] implement MCMC inside a relational database for efficiently answering general SQL queries over a CRF model, commonly used in Information Extraction. The approach taken by these systems is to scale up existing general purpose probabilistic inference methods. Our MVDB approach differs from these in that we do not optimize any existing algorithm for inference over complex probabilistic models, but propose a new approach by which we translate from a complex probabilistic model to a tuple-independent probabilistic model.
Chapter 3

COMPILING UCQ

3.1 Introduction

The goal of Knowledge compilation [17, 32, 96] is to represent a Boolean expression in a language in which it can answer a range of problems, also called “online-queries”, in PTIME. Typical problems are satisfiability, validity, implication, model counting, substitution with constants, substitution with functions. For example, the model counting problem asks for the number of satisfying assignments to a Boolean expression; the more general probability computation problem asks for the probability of that expression being true, if every variable is true/false independently with some probability. If one compiles the Boolean expression into (say) an FBDD, then the model counting problem and the probability computation problem can be solved in linear time in the size of the FBDD. Different compilation languages can solve efficiently different classes of problems, in time polynomial in the size of compiled expression [32]. This motivates the need to know if an expression can be compiled into a small-sized or compact representation in a given language.

The provenance of a query on a relational database is an expression that describes how the answer was derived from the tuples in the database [46]. We are interested in the flavor of provenance called PosBool in [90] (see also [47]), which we will refer to as lineage. The lineage is a Boolean expression over Boolean variables corresponding to tuples in the input database. Our goal in this chapter is to identify queries whose lineage admits a compact compilation. Our main motivation comes from (but is not limited to) probabilistic databases, where the problem is the following: given a query and a probabilistic database (i.e. each tuple has a given probability), compute the probability of each query answer [27]. If the lineage has been compiled into a compact format that supports the probability computation, then one can compute the output probabilities efficiently. We study queries whose lineage always admits a compact compilation, on any database instance. We are only interested in the data complexity i.e. we assume the
Table 3.1: Several representative queries. All queries are hierarchical, and have the additional syntactic properties shown. \( \hat{0} \) denotes the minimal element of the query’s CNF-lattice; \( \mu \) its Mobius function. Queries \( q_2, q_V, q_W \) separate the corresponding classes. We conjecture that \( q_9 \) separates UCQ\((dDNNF)\) from UCQ\((P)\). \( ^\ast \) : assuming \( FP \neq \#P \); \( ? \) : conjectured.

<table>
<thead>
<tr>
<th>Query</th>
<th>Syntactic properties</th>
<th>Membership in UCQ((T)), where ( T ) is</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 = R(x_1), S(x_1, y_1) \lor T(x_2), S(x_2, y_2) )</td>
<td>inversion-free, read-once</td>
<td>yes yes yes yes yes</td>
</tr>
<tr>
<td>( q_2 = R(x_1), S(x_1, y_1), \lor S(x_2, y_2), T(x_2) )</td>
<td>inversion-free</td>
<td>no yes yes yes yes</td>
</tr>
<tr>
<td>( q_V = R(x_1), S(x_1, y_1), \lor S(x_2, y_2), T(y_2) \lor R(x_3), T(y_3) )</td>
<td>has inversion, all lattice points have separators</td>
<td>no no yes yes yes</td>
</tr>
<tr>
<td>( q_W ) (Table 3.2)</td>
<td>lattice point ( \hat{0} ) has no separator but is erasable</td>
<td>no no no yes yes</td>
</tr>
<tr>
<td>( q_9 ) (Table 3.2)</td>
<td>lattice point ( \hat{0} ) has no separator and has ( \mu = 0 ) and is non-erasable</td>
<td>no no no( ^\ast ) no( ^\ast ) yes [28]</td>
</tr>
<tr>
<td>( h_1 = R(x_1), S(x_1, y_1) \lor S(x_2, y_2), T(y_2) )</td>
<td>lattice point ( \hat{0} ) has no separator and has ( \mu \neq 0 )</td>
<td>no no no no( ^\ast ) no( ^\ast ) [28]</td>
</tr>
</tbody>
</table>

Table 3.2: \( Q_9, Q_W \) defined using common components.

\[
\begin{align*}
    h_{30} &= R(x_0), S_1(x_0, y_0) \\
    h_{31} &= S_1(x_1, y_1), S_2(x_1, y_1) \\
    h_{32} &= S_2(x_2, y_2), S_3(x_2, y_2) \\
    h_{33} &= S_3(x_3, y_3), T(y_3) \\
    Q_W &= (h_{30} \lor h_{32}) \land (h_{30} \lor h_{31}) \land (h_{31} \lor h_{33}) \\
    Q_9 &= (h_{30} \lor h_{33}) \land (h_{31} \lor h_{33}) \land (h_{32} \lor h_{33}) \land (h_{30} \lor h_{31} \lor h_{32})
\end{align*}
\]

query to be fixed (and, in particular its size is constant). Our query language is that of unions of conjunctive queries, UCQ, and, as usual, we restrict our discussion to Boolean queries.

We consider four compilation targets. For each target \( T \), we denote by UCQ\((T)\) the class of UCQ queries whose lineage admits a compact compilation in \( T \) for all input databases: the precise definition of the term “compact” depends on the compilation target, but usually means that the target has polynomial size. There are two different ways of defining a compact compilation: (i) **Uniform**: A compact compilation can be found in polynomial time, (ii) **Non-Uniform**: A compact compilation exists, but there are no restrictions on how to find it. The first interpretation is more
strict, but makes more practical sense if one is looking for tractable algorithms for compilation; all our upper bounds are for uniform compilation. The second interpretation is more useful in complexity theory, since the results show the expressibility, and limitation of different models of compilation/computation; all our lower bounds are for non-uniform compilation. Unless stated otherwise, we always assume that the compilation is uniform, except in Sect. 3.7 where we focus exclusively on non-uniform compilation.

Our first target is: Read-Once expression, RO. A Read-once Boolean expression is an expression consisting of $\land$, $\lor$, $\neg$ operators such a way that every input variable is used only once. A read-once Boolean formula is one that can be represented by a read-once expression. Read-Once formulas admit an elegant characterization due to Gurvich [48] (see [44]). Thus, $UCQ(RO)$ is the class of queries $q$ such that for every input database, the lineage of $q$ on that database is a read-once formula. The second and third targets are Ordered and Free BDD. A Binary Decision Diagram\(^1\), BDD, is a rooted DAG where each internal node is labeled with a variable and has two outgoing edges labeled 0 and 1, and each sink node is labeled either 0 or 1. A BDD represents a Boolean function, as follows. Given an assignment to the Boolean variables, the value of the function is obtained by traversing the BDD starting at the root node, and at each internal node following either the 0 or the 1 edge, according to the value of that node’s variable. The unique sink node reached at the end of the traversal gives the value (0 or 1) of the Boolean function under that assignment. A BDD is free (hence FBDD) if any path from the root to a sink node reads every variable at most once. An FBDD is ordered (hence OBDD) if there exists a total order on the Boolean variables s.t. any path from the root to a sink node reads the variables in this order (it may skip some variables). Thus, $UCQ(OBDD)$ and $UCQ(FBDD)$ denote the class of queries $q$ s.t. that for any database instance $D$, one can construct an $OBDD$ ($FBDD$) for the lineage of $q$ on $D$, in time polynomial in $D$; in particular, the resulting OBDD or FBDD has also size polynomial in $D$. Finally, our fourth target are d-DNNF, introduced by Darwiche [30] (see also [32]), which are DAGs whose leaves are labeled with Boolean variables or their negation, and internal nodes are labeled either an independent-$\land$ (where the two children must have distinct sets of Boolean variables), or with disjoint-$\lor$ (where the two children must be exclusive Boolean formulas). We also allow for one more type of internal node : not($\neg$). $UCQ(dDNNF)$ represents the class of queries s.t. one can construct a d-DNNF of its lineage in PTIME for any input database.

In addition to these four classes defined by a compilation target, we also consider $UCQ(P)$, the class of queries $q$ with the property that, for every probabilistic database $D$, the probability

\(^1\)BDD are also known as Branching Program (BP) in the literature
of \( q \) on \( D \) can be computed in PTIME in the size of \( D \). It follows from known results that these five classes form an increasing hierarchy: \( UCQ(RO) \subseteq UCQ(OBDD) \subseteq UCQ(FBDD) \subseteq UCQ(dDNNF) \subseteq UCQ(P) \).

Dalvi and Suciu [26, 27] have studied the evaluation problem over probabilistic databases for conjunctive queries without self-joins, denoted here \( CQ^- \), and showed that the class of queries computable in PTIME, \( CQ^- (P) \), consists precisely of hierarchical queries (reviewed in Section 3.2). Olteanu and Huang [68] have shown a remarkable result: that for any hierarchical query, its lineage is a read-once formula. In other words, they explained that the reason why hierarchical queries can be computed in PTIME is because their lineage is read once. This immediately implies (assuming \( FP \neq \#P \)) that the following five classes collapse: \( CQ^- (RO) = CQ^- (OBDD) = CQ^- (FBDD) = CQ^- (UCQ) = CQ^- (P) \).

We show that, over unions of conjunctive queries (UCQ), these classes no longer collapse. In fact they form a strict hierarchy: \( UCQ(RO) \subseteq UCQ(OBDD) \subseteq UCQ(FBDD) \subseteq UCQ(dDNNF) \subseteq UCQ(P) \). This means that the reason why certain queries can be computed in PTIME over probabilistic database is no longer their read-once-ness, or any other efficient compilation method. (We were not able to separate \( UCQ(dDNNF) \) from \( UCQ(P) \) but we conjecture that they are also separated); instead, each notion of efficiency is distinct. We refer to Table 3.1 to discuss our results.

Our results make use of three syntactic properties of a query, called inversion [25], separator [28], and hierarchical queries [27], reviewed in Section 3.2. The following strict implications hold: inversion-free implies existence of separators at all levels, which implies the query is hierarchical.

We give a complete characterization of \( UCQ(RO) \) and \( UCQ(OBDD) \). First, \( UCQ(OBDD) \) coincides with inversion-free queries. \( UCQ(RO) \) coincides with queries that are both inversion-free and can be written using \( \land, \lor, \exists \) such that every relation symbol occurs only once. For example, consider the query \( q_1 \) in Table 3.1, \( q_1 = \exists x_1. \exists y_1. R(x_1), S(x_1, y_1) \lor \exists x_2. \exists y_2. T(x_2), S(x_2, y_2) \); we drop existential quantifiers when they are clear from the context, and write the query as \( q_1 = R(x_1), S(x_1, y_1) \lor T(x_2), S(x_2, y_2) \). The query can also be written as \( \exists x.(R(x) \lor T(x)) \land \exists y.(S(x, y)) \): here each symbol \( R, S, T \) occurs only once and, since \( q_1 \) is also inversion-free, it follows that it is in \( UCQ(RO) \). Note that our characterization of \( UCQ(RO) \) is unrelated to Gurvich’s characterization of read-once Boolean expressions [48, 44], or to algorithms for checking read-once-ness in [84, 80]: these results apply to the Boolean formula, while our results apply directly to the query.

For \( UCQ(FBDD) \) and \( UCQ(dDNNF) \), we only give sufficient conditions by making use of
the \textit{CNF-lattice} associated to a query (introduced in [28]), where each lattice element $x$ is labeled by a subquery, denoted $\lambda(x)$. A sufficient condition for a query to be in \textit{UCQ(FBDD)} is for every lattice element to have a separator and to satisfy some additional condition. A sufficient condition for \textit{UCQ(dDNNF)} is that every lattice element must have a separator, \textit{except} those lattice elements that can be erased (a notion we define in Section 3.6). For comparison, the necessary and sufficient condition for \textit{UCQ(P)} is that every lattice element must have a separator, \textit{except} those lattice elements where the Mobius function is 0 ($\mu = 0$) [28]. If an element can be erased, then its Mobius function is 0, but the converse is not true, as illustrated by $q_9$ in Table 3.1. We conjecture that $q_9$ is not in \textit{UCQ(dDNNF)}.

The most difficult results of this chapter are the separation results \textit{UCQ(OBDD)} $\subseteq$ \textit{UCQ(FBDD)} $\subseteq$ \textit{UCQ(dDNNF)}; they are separated by the queries $q_V$ and $q_W$ respectively in Table 3.1. In each case we prove that the query does not belong to the smaller class, but that it belongs to the larger class. The first result applies even to the non-uniform definition of the complexity class. More precisely, we show $q_V \notin \textit{UCQ(OBDD)}$, even if one assumes the non-uniform definition of \textit{UCQ(OBDD)}, and that $q_V \in \textit{UCQ(FBDD)}$; similarly $q_W \notin \textit{UCQ(FBDD)}$, even for a non-uniform definition of this class, and $q_W \in \textit{UCQ(dDNNF)}$. These results are significant for the following reason. All lineage expressions for queries in \textit{UCQ} are very simple: they are monotone, and have a DNF expression of polynomial size. This also applies to the lineage expressions of $q_V$ and $q_W$. Thus, our lower bounds make a contribution to the general separation problem of polynomial-size \textit{OBDD}, \textit{FBDD}, and d-DNNF. Early lower bounds for \textit{FBDD} were for non-monotone formulas, with exponential size DNFs. The first “simple” Boolean formula shown to have exponential \textit{FBDD} was given by Gál in [39], followed by a “very simple” formula given by Bollig and Wegener [10]. But it is not very surprising that the “very simple” has no polynomial size FBDDs, since computing the probability of that formula is \#P-hard: the “very simple” formula is precisely the lineage of the non-hierarchical query $R(x), S(x, y), T(y)$, for which computing the probability is \#P-hard. In contrast, for both $q_V$ and $q_W$ one can compute the probability in polynomial time: hence, the fact that they do not admit polynomial size OBDDs or polynomial size FBDDs respectively is more surprising. On the other hand, we can use Bollig and Wegener’s result to prove that, for every non-hierarchical query, its lineage has no polynomial size FBDD.

The lineage of the query $q_V$, that we use for the first major separation \textit{UCQ(OBDD)} $\subseteq$ \textit{UCQ(FBDD)} is, to the best of our knowledge, the first “simple” Boolean formula separating polynomial-size \textit{OBDD} from \textit{FBDD}. Previous Boolean formulas separating the two classes are non-monotone, and do not have polynomial size DNFs. The classic example is the Weighted
Bit Addressing problem (WBA), defined as $F(X_1,\ldots,X_n) = X\sum_{i=1}^n X_i$ (where $X_0 = 0$). Bryant [15] has shown that has it has no polynomial size OBDD, while Gergov and Meinel [41] and independently Sieling and Wegener [85] have shown that WBA has a polynomial sized FBDD. More examples are given in [95]. Our characterization of UCQ(OBDD) and UCQ(FBDD) allows one to give a class of simple boolean expressions that separate polynomial-size OBDD from FBDD.

The lineage of the query $q_W$ that we use for our second major separation $UCQ(FBDD) \subsetneq UCQ(dDNNF)$ is also, to the best of our knowledge, the first “simple” Boolean formula separating polynomial-size FBDD from d-DNNF. The previous separation relies on a result due to Bollig and Wegener [11]: they give an example of two Boolean formulas $\Phi_1, \Phi_2$ that have polynomial size OBDD, $\Phi_1 \land \Phi_2 \equiv \text{false}$, yet $\Phi_1 \lor \Phi_2$ cannot have polynomial size FBDD. Hence $\Phi_1 \lor \Phi_2$ separates d-DNNF from FBDD.

Finally, we note that no formula with exponential lower bound on d-DNNF size is presently known. In particular, we leave open the question whether $UCQ(dDNNF) \subsetneq UCQ(P)$. However, our algorithm in Section 3.6 suggests how d-DNNF may be constructed for general queries, which further suggests that this is not possible for $q_9$. We conjecture that $q_9$ is not in $UCQ(dDNNF)$, and, hence, that its lineage has no polynomial size d-DNNF.

The chapter is organized as follows. We give the basic definitions and review the relevant results in [28] in Section 3.2, then discuss read-once, OBDD, FBDD, and d-DNNF in Section 3.3, Section 3.4, Section 3.5, Section 3.6. We discuss results for non-uniform setting in Section 3.7 and conclude in Section 3.8.

### 3.2 Background and Definitions

We discuss unions of conjunctive queries (UCQ), which are expressions defined by the following grammar:

$$Q ::= R(\bar{x}) \mid \exists x. Q_1 \mid Q_1 \land Q_2 \mid Q_1 \lor Q_2 \quad (3.1)$$

$R(\bar{x})$ is a relational atom with variables and/or constants, whose relation symbol $R$ is from a fixed vocabulary. We replace $\land$ with comma, and drop $\exists$, when no confusion arises: for example we write $R(x), S(x)$ for $\exists x.(R(x) \land S(x))$.

A query is an expression as defined by Equation 3.1, up to logical equivalence. We consider only Boolean queries. A conjunctive query (CQ) is a query that can be written without $\lor$. A
Chapter 3. Compiling UCQ

Boolean variable $X$.

Let $Q$, $R$ be a conjunctive query. The logical implication $q \Rightarrow q'$ holds iff there exists a homomorphism $q' \rightarrow q$ between their representations as sets of atoms [20].

Let $D$ be a database instance. Denote $X_t$ a distinct Boolean variable for each tuple $t \in D$. Let $Q$ be a UCQ. The lineage of $Q$ on $D$ is the Boolean expression $\Phi_Q^D$, or simply $\Phi_Q$ if $D$ is understood from the context, defined inductively as follows, where $\text{ADom}(D)$ denotes the active domain of the database instance:

$$\Phi_{R(a)} = X_{R(a)}$$  \hspace{1cm} (3.2)

$$\Phi_{\exists x.Q} = \bigvee_{a \in \text{ADom}(D)} \Phi_{Q[a/x]}$$

$$\Phi_{Q_1 \land Q_2} = \Phi_{Q_1} \land \Phi_{Q_2}$$  \hspace{1cm} (3.3)

$$\Phi_{Q_1 \lor Q_2} = \Phi_{Q_1} \lor \Phi_{Q_2}$$

Given a probability $p(X_t) \in [0, 1]$ for each Boolean variable $X_t$, we denote $P(\Phi_Q^D)$ the probability that the Boolean formula $\Phi_Q^D$ is true, when each Boolean variable $X_t$ is set to 1 independently, with probability $p(X_t)$.

A probabilistic database is a pair $(D, p)$ where $D$ is a database and $p(t) \in [0, 1]$ assigns a probability to each tuple $t \in D$. Given a Boolean query $Q$ and a probabilistic database $(D, p)$, the query probability $P(Q)$ is defined as $P(Q) = P(\Phi_Q^D)$, where in the latter expression each Boolean variable $X_t$ has a probability equal to that of its corresponding tuple, $p(X_t) = p(t)$.

The query evaluation problem on probabilistic databases is the following: given a query $Q$ and a probabilistic database $(D, t)$, compute $P(Q)$. Usually we are interested in the data complexity of the query evaluation problem: for a fixed $Q$, determine the complexity of computing $P(Q)$ as a function of the input database $(D, p)$.

**Figure 3.1:** CNF Lattices for the queries $q_v$, $q_w$, and $q_9$.

In the lattices for $q_v$ and $q_9$, $\mu(\hat{0}, \hat{1}) = 0$; in all other cases, $\mu(x, \hat{1}) \neq 0$. In $q_w$, the element $\hat{0}$ is erasable; in $q_9$, the element $\hat{0}$ is not erasable.
Definition 3.1. UCQ($P$) is the class of UCQ queries $Q$ s.t. for any probabilistic database $(D, p)$, the probability $P(Q)$ can be computed in PTIME in the size of $D$.

A complete characterization of the class UCQ($P$) was given in [28]. We review here it, since we will reuse some of the same concepts that characterize the class UCQ($P$) to characterize various compilation targets.

We start by discussing connected queries. Consider a conjunctive query $q$ given by the set of its atoms $R_1(\bar{x}_1), \ldots, R_m(\bar{x}_m)$, and assume this representation is minimal, i.e. removing any atom results in an inequivalent query; it is known that this minimal representation is unique up to isomorphism. Define the following undirected graph $G$: there is one node for each atom, and there is an edge from atom $i$ to atom $j$ if $R_i(\bar{x}_i)$ and $R_j(\bar{x}_j)$ share a common variable. We say that the query $q$ is connected if the graph $G$ is connected.

Lemma 3.2. Suppose we restrict all conjunctive queries to be without constants. Let $q$ be a conjunctive query. Then the following conditions are equivalent. (1) The query $q$ is connected. (2) For every two conjunctive queries $q_1, q_2$, if $q_1 \land q_2 \Rightarrow q$ then either $q_1 \Rightarrow q$ or $q_2 \Rightarrow q$. (3) For every two conjunctive queries $q_1, q_2$, if $q \equiv q_1 \land q_2$ then either $q \equiv q_1$ or $q \equiv q_2$.

Proof. (1) implies (2). Assuming $q_1 \land q_2 \Rightarrow q$ we obtain a homomorphism $q \rightarrow q_1 \land q_2$. Since neither $q_1$ nor $q_2$ have constants, the homomorphism must map every variable in $q$ to a variable (i.e. not to a constant). Since $q$ is connected, the image of this homomorphism must be a connected graph, and, therefore, it is included either in $q_1$ or in $q_2$; this means that the homomorphism is either $q \rightarrow q_1$ or $q \rightarrow q_2$, implying either $q_1 \Rightarrow q$ or $q_2 \Rightarrow q$.

(2) implies (3). Assume $q_1 \land q_2 \equiv q$. In particular, $q_1 \land q_2 \Rightarrow q$, and by property (2) we have $q_1 \Rightarrow q$ or $q_2 \Rightarrow q$. Assuming the former, we derive $q_1 \Rightarrow q_1 \land q_2$, which further implies $q_1 \equiv q_1 \land q_2 \equiv q$.

(3) implies (1). Suppose that $q$ is not connected. Suppose its minimal representation has $m$ atoms. Let $G$ be the graph corresponding to its minimal representation. We can partition its nodes into two sets, each with strictly less than $m$ atoms, s.t. they don’t share any variables. Thus, we have written $q = q_1 \land q_2$ where $q_1, q_2$ share no common variables and each has strictly less than $m$ atoms. By condition (3) it follows the either $q_1 \Rightarrow q$ or $q_2 \Rightarrow q$. Assuming the former, we have $q \equiv q_1$, contradicting the fact that the minimal representation of $q$ has $m$ atoms.

The restriction to conjunctive queries without constants is necessary for the lemma to hold. Otherwise, consider the connected query $q = R(x, y), S(y, z)$, and $q_1 = R(x, a)$, $q_2 = S(a, z)$, where $a$ is a constant and $x, y, z$ are variables: we have $q_1 \land q_2 \Rightarrow q$ but neither $q_1 \Rightarrow q$ nor $q_2 \Rightarrow q$ holds.
We now define the key notions needed to characterize $UCQ(P)$, and which we need throughout this chapter:

- A **component**, $c$, is a conjunctive query that is **connected**.

- Every **conjunctive query** can be written as a conjunction of components. That is, $q = c_1, c_2, \ldots, c_k$, s.t. $c_i$ and $c_j$ do not share any common variables, for all $i \neq j$. If $q = c_1, c_2, \ldots$ and $q' = c'_1, c'_2, \ldots$ are two conjunctive queries given as conjunction of components, and if they don’t have any constants, then the logical implication $q \Rightarrow q'$ holds iff $\forall j. \exists i$ s.t. $c_i \Rightarrow c'_j$.

- A **disjunctive query** is a disjunction of components, $d = c_1 \lor \ldots \lor c_k$. Given two disjunctive queries $d = c_1 \lor c_2 \lor \ldots$ and $d' = c'_1 \lor c'_2 \lor \ldots$, the logical implication $d \Rightarrow d'$ holds iff $\forall i. \exists j$ s.t. $c_i \Rightarrow c'_j$.

- A **UCQ** in DNF a disjunction of conjunctive queries, $Q = q_1 \lor \ldots \lor q_m$. Given two queries in DNF, $Q = q_1 \lor q_2 \lor \ldots$ and $Q' = q'_1 \lor q'_2 \lor \ldots$, the logical implication $Q \Rightarrow Q'$ holds iff $\forall i. \exists j$ s.t. $q_i \Rightarrow q'_j$.

- A **UCQ** in CNF is conjunction of disjunctive queries, $Q = d_1 \land \ldots \land d_m$. Given two queries in CNF, $Q = d_1 \land d_2 \land \ldots$ and $Q' = d'_1 \land d'_2 \land \ldots$, if they don’t have any constants, then the implication $Q \Rightarrow Q'$ holds iff $\forall j. \exists i$ s.t. $d_i \Rightarrow d_j$.

Obviously, any component is both a conjunctive query and a disjunctive query; also, every conjunctive query is a UCQ in DNF, and every disjunctive query is a UCQ in CNF.

The containment condition for DNF is due to Sagiv and Yannakakis [82]. The containment condition for CNF is from [28], and only holds if the queries have no constants. To see that this requirement is needed, consider the following three disjunctive queries: $d_1 = R(x,a)$, $d_2 = S(a,z)$, and $d = R(x,a), S(y,z)$, where $a$ is a constant and $x, y, z$ are variables. Define the following two UCQ’s: $Q = d_1, d_2$ and $Q' = d$. Both are in CNF, and $Q \Rightarrow Q'$, yet neither $d_1 \Rightarrow d$ nor $d_2 \Rightarrow d$ holds.

Following [28] we first perform the following transformations on the query. They preserve the lineage of the query and hence membership in $UCQ(P)$ and all the classes considered in this chapter.

**Remove constants** Every query with constants is rewritten into an equivalent query without constants, over an extended vocabulary, by repeatedly substituting a relation $R(A_1, \ldots, A_k)$ with 2 relations, $R_1 = \sigma_{A_i \neq a}(R)$ and $R_2 = \Pi_{A_1 \ldots A_{i-1} A_{i+1} \ldots A_k}(\sigma_{A_i = a}(R))$, for every attribute position $i$ and every constant $a$ that occurs in the query. For example, $R(x,a), S(x) \lor$
Chapter 3. Compiling UCQ

Assume an ordered domain. A query is \( \exists \) not. However, we can write the query equivalently as a disjunctive query due to the definition:

\[ R(x, y), T(x) \text{ is rewritten as } R_2(x), S(x) \lor R_2(x), T(x) \lor R_1(x, y), T(y), \text{ where } R_1(x, y) = \sigma_{y \neq a}(R(x, y)), \text{ and } R_2(x) = \pi_x(\sigma_{y=a}(R(x, y))). \]

**Ranking** Assume an ordered domain. A query is ranked if it remains consistent after adding all predicates of the form \( x < y \), for all pairs of variables \( x, y \) that co-occur in some atom, such that \( x \) occurs before \( y \). For example, \( R(x, y), R(y, z), R(x, z) \) is ranked because \( x < y \land y < z \land x < z \) is consistent, while \( R(x, y), S(y, x) \) is not ranked (\( x < y \land y < x \) is inconsistent), and \( R(x, x, y) \) is not ranked (\( x < x \land x < y \) is inconsistent). Every query is rewritten into an equivalent, ranked query, over an extended vocabulary, by repeatedly substituting a relation \( R(A_1, \ldots, A_k) \) with three relations: \( R_1 = \sigma_{A_i < A_j}(R), R_2 = \Pi_{A_1 \ldots A_{i-1} A_{i+1} \ldots A_k}(\sigma_{A_i = A_j}(R)), R_3 = \Pi_{A_1 \ldots A_{i-1} A_{i+1} \ldots A_k}(\sigma_{A_i > A_j}(R)), \) for every two attributes \( A_i, A_j \) s.t. \( i < j \). We give here the main intuition by illustrating with \( q = R(x, y), R(y, x) \), and refer to [28] for further details. Denoting \( R_2(x, y) = \sigma_{x < y}(R), R_2(x) = \pi_x(\sigma_{x=y}(R(x, y))), \ R_3(y, x) = \pi_y(\sigma_{x > y}(R)) \), we rewrite the query as \( R_2(x_1) \lor R_1(x_2, y_2), R_3(x_2, y_2) \). The new query is ranked.

The reason for the first transformation is to ensure that the implication criteria for CNF expressions holds. As a consequence, every UCQ has a unique, minimal representation in DNF, and a unique, minimal representation in CNF. The reason for the second transformation will become clear below. We will assume throughout the paper that a CNF or DNF expression of a query is minimized.

The first step in characterizing UCQ(P) is to describe a class of disjunctive queries that are hard for \#P, using the notion of a separator. Consider a query, and a subexpression of the form \( \exists w.Q \) (see grammar Equation 3.1): the scope of the variable \( w \) is the subexpression \( Q \).

**Definition 3.3.** A variable \( w \) is called a root variable if it occurs in all atoms in its scope.

For a simple illustration, consider \( \exists x. \exists y R(x) \land S(x, y) \). Then \( x \) is a root variable, but \( y \) is not. However, we can write the query equivalently as \( \exists x. R(x) \land (\exists y S(x, y)) \): now both \( x \) and \( y \) are root variables.

**Definition 3.4.** A disjunctive query \( d \) has a separator if it can be written as \( d \equiv \exists w.Q, \) such that \( w \) is a root variable, and for every two atoms \( g, g' \) using the same relational symbol \( R \), the variable \( w \) occurs in the same position in \( g \) and in \( g' \). In this case the variable \( w \) is called a separator variable in the expression \( \exists w.Q \).

The hardness part of characterizing UCQ(P) consists of showing that, if a disjunctive query has no separator then it is hard for \#P: hence, it cannot be in UCQ(P) unless \( FP = \#P \). Recall that we assume all queries to be without constants, ranked, and minimized.
Theorem 3.5. [28] Let $d$ be a disjunctive query s.t. each component has at least one variable. If $d$ has no separator, then $d$ is hard for #$P$.

If $d$ has any component without variables then it trivially has no separator. For example, consider $d = R() \lor S(x)$: the first component, $R()$, has no variables, and clearly $d$ has no separator, e.g. if we write it as $\exists x. (R() \lor S(x))$ then $x$ is not a root variable. However, it is always easy to get rid of the components without variables, then apply Theorem 3.5. Indeed, write the disjunctive query as $d = d_0 \lor d'$ where $d_0$ contains all components without variables and $d'$ contains all components with variables. Thus, $d_0$ is a disjunction $R_1() \lor R_2() \lor \ldots$ of zero-ary relational symbols, and $d'$ is a disjunction of components $c_1 \lor c_2 \lor \ldots$, each having at least one variable.

None of the symbols $R_i()$ occurs in any component $c_j$, otherwise $c_j$ would not be connected. Thus, $d_0$ and $d'$ are independent probabilistic events, and $P(d) = 1 - (1 - P(d_0))(1 - P(d'))$, in other words computing $P(d)$ reduces to computing $P(d')$, and this is the reason why the theorem focuses only on the latter. Note that the theorem holds only if the query is ranked: for a counter-example, $R(x,y), R(y,x)$ has no separator, yet is in $UCQ(P)$ (this follows from the ranking shown above, and from Theorem 3.7 below); this is the reason why we rank queries.

Conversely, if $d$ has a separator, $d = \exists w.Q$, then its probability can be computed as $P(d) = 1 - \prod (1 - P(Q[a_i/w]))$, where $a_1, \ldots, a_n$ is the active domain of the database, because the events $Q[a_1/w], \ldots, Q[a_n/w]$ are independent. Furthermore, this can be computed efficiently, provided that each query $Q[a_i/w]$ is in $UCQ(P)$. Although we disallowed constants in queries, the expression $Q[a_i/w]$ is OK because all occurrences of a relational symbol have the constant $a$ in the same position; we simply remove $a$ from all atoms, renaming all relational symbols, and decreasing their arity by 1.

Example 3.1. Query $q_1$ in Table 3.1 has a separator, because\(^2\) $q_1 \equiv \exists w. (R(w), S(w,y_1) \lor T(w), S(w,y_2))$. We can compute its probability as $P(q_1) = 1 - \prod (1 - P(R(a_i), S(a_i, y_1) \lor T(a_i), S(a_i, y_2)))$. Query $h_1$, on the other hand, does not have a separator: if we write it as $\exists w. (R(w), S(w,y_1) \lor S(w,y_2), T(y_2))$ then $w$ is not a root variable, and if we write it as $\exists w. (R(w), S(w,y_1) \lor S(x_2, w), T(w))$ then $w$ occurs on different positions in $S(w,y_1)$ and $S(x_2, w)$. Therefore, $h_1$ is hard for #$P$.

Consider a UCQ in CNF: $Q = d_1 \land \ldots \land d_k$. For each subset $s \subseteq [k]$ denote $d_s = \bigvee_{i \in s} d_i$. The inclusion/exclusion formula gives us $P(Q) = -\sum_{s \neq \emptyset}(-1)^{|s|}P(d_s)$ and, therefore, if all $d_s$ are in $UCQ(P)$ (in particular, they have separators), then so is $Q$. The formula is exponential in the size of the query, but this does not affect data complexity. However, the condition $d_s \in UCQ(P)$

\(^2\)We omitted the inner quantifiers $\exists y_1$ and $\exists y_2$.\)
is not necessary for all $s$: some terms in the inclusion/exclusion formula may cancel out, and $Q$ may be in $UCQ(P)$ even if some disjunctive queries $d_s$ are hard.

To characterize precisely when $Q$ is in $UCQ(P)$, [28] defines the CNF lattice $(L, \leq)$ for $Q$. Each element $x \in L$ corresponds to a distinct disjunctive query, denoted $\lambda(x) = d_s$, for some $s \subseteq [k]$, up to logical equivalence; that is, if $d_s \equiv d_{s_2}$ then they correspond to the same element in $x \in L$. The order relation $\leq$ is reversed logical implication: $x \leq y$ iff $\lambda(y) \Rightarrow \lambda(x)$.

The maximal element in the lattice is denoted $\hat{1}$, and corresponds to the lattice $\lambda(\hat{0}) = d_1 \lor \ldots \lor d_k$. Three examples are shown in Figure 3.1.

We say $x$ covers $y$ if $y \leq x$ and there is no $z \in L$ s.t. $y < z < x$. $x$ is an atom if it covers $\hat{0}$; it is a co-atom if it is covered by $\hat{1}$. Denote by $L^*$ the set of co-atoms. The meet closure of $S \subseteq L$ is the lattice $\bar{S} = \{\bigwedge T \mid T \subseteq S\}$. Note that $\bigwedge \emptyset = \hat{1}$, the meet closure of any set $S$ contains the maximal element $\hat{1}$.

The Mobius function of a lattice $(L, \leq)$ is the function $\mu_L : L \times L \rightarrow \mathbb{Z}$ defined by $\mu_L(x, y) = 1$, $\mu_L(x, y) = -\sum_{x < z \leq y} \mu_L(z, y)$. Note that $\mu_L(x, y) = 0$ whenever $x \nleq y$. We will drop the $L$, i.e. denote $\mu_L$ by simply $\mu$, henceforth when it is clear from the context. Mobius’ inversion formula applied to $P(Q)$ is: $P(Q) = -\sum_{x \leq \hat{1}} \mu(x, \hat{1})P(\lambda(x))$. Now it becomes obvious that we only need to compute $P(d_s)$ for those queries for which $\mu(x, \hat{1}) \neq 0$. This justifies:

**Definition 3.6 (Safe queries).** [28] (1) Let $Q = d_1 \land \ldots \land d_k$, and $k \geq 2$. Then $Q$ is safe if for every element $x$ in its CNF lattice, if $\mu(x, \hat{1}) \neq 0$, then the disjunctive query $\lambda(x)$ is safe (recursively). (2) Let $d = d_0 \lor d_1$, be a disjunctive query where $d_0$ contains all components without variables, and $d_1$ contains all components with at least one variable. Then $d$ is safe if $d_1$ has a separator $w$ and $d_1[a/w]$ is safe (recursively), for a constant $a$.

The characterization of $UCQ(P)$ is:

**Theorem 3.7.** [28] Any safe query is in $UCQ(P)$. Any unsafe query is hard for $\#P$.

The first part of the theorem follows from our discussion so far. The second part is proven in [28] by using Theorem 3.5.

This completes the characterization of $UCQ(P)$ from [28]. We still need to introduce two more notions that we use in rest of the chapter: hierarchical queries and inversion-free queries.

**Hierarchical queries** Let $q$ be a conjunctive query, and denote $at(x)$ the set of atoms containing the variable $x \in Vars(q)$. We say that $q$ is hierarchical if for any two variables $x, y$, we have $at(x) \subseteq at(y)$ or $at(x) \supseteq at(y)$, or $at(x) \cap at(y) = \emptyset$. A UCQ query $Q$ is hierarchical if it is the union of hierarchical conjunctive queries. We give an alternative definition next:
Definition 3.8. Let $Q$ be a query expression given by the grammar Equation 3.1. We say that it is a hierarchical expression if every variable is a root variable.

It is easy to check that a query is hierarchical iff it can be written as a hierarchical expression. For example, the query $R(x, y), S(x, z)$ is hierarchical, because it can be written as $\exists x. (\exists y. R(x, y) \land \exists z. S(x, z))$. Examples of non-hierarchical queries are $R(x), S(x, y), T(y)$ and $R(x, y), R(y, z), R(x, z)$. The following is easy to see:

Proposition 3.9. If $Q$ is safe, then it is hierarchical.

Proof. By induction on the structure of $Q$. If $Q = d_1 \land \ldots \land d_k$, then each $d_i$ corresponds to a co-atom $x$ in the CNF lattice, hence $\mu(x, \tilde{1}) = -1 \neq 0$, and therefore $d_i$ must be safe, hence it is hierarchical by induction, hence $Q$ is hierarchical. If $Q = d_1 \lor d_2$ and $d_1$ is a separator, $d_1 = \exists w. Q_1$, then $Q_1[a/w]$ is safe, hence it is hierarchical by induction, hence $\exists w. Q_1$ is hierarchical because $w$ occurs in all atoms of $Q_1$.

The converse is not true: for example $h_1$ in Table 3.1 is hierarchical, but unsafe. Thus, all non-hierarchical queries are \#P-hard, but the converse fails in general.

Inversions Inversions were first defined in in [25]. In this paper we show how to use inversions to characterize UCQ(RO) and UCQ(OBDD). Let $Q = q_1 \land \ldots \land q_k$ be a query in DNF. The unification graph $G$ has as nodes all pairs of variables $(x, y)$ that co-occur in some atom, and has an edge between $(x, y)$ and $(x', y')$ if the following holds: $x, y$ co-occur in some atom $g$, $x', y'$ co-occur in some atom $g'$, the atoms $g$ and $g'$ are over the same relation symbol and $x, y$ appear at the same positions in $g$ as $x', y'$ in $g'$. In other words, $g$ and $g'$ are unifiable, and the unification equates $x = x'$ and $y = y'$. Given $x, y \in \text{Vars}(q_i)$, denote $x \succ y$ if $at(x) \not\subseteq at(y)$.

Definition 3.10 (Inversion). [25] An inversion in $Q$ is a path of length $\geq 0$ in $G$ from a node $(x, y)$ to a node $(x', y')$ s.t. $x \succ y$ and $x' \prec y'$. If no such path exists, we say $Q$ is inversion-free.

If a query is non-hierarchical then it has an inversion. Indeed, let $x, y$ be two variables occurring in the same non-hierarchical conjunctive query, such that $at(x) \cap at(y) \neq \emptyset$ and neither of the two sets $at(x)$, $at(y)$ contains the other. Consider the node $(x, y)$ in the unification graph (such a node exists because $at(x) \cap at(y) \neq \emptyset$). Since we have both $x \succ y$ and $x \prec y$, the empty path staring and ending at $(x, y)$ is an inversion. The converse fails: $h_1$ in Table 3.1 is hierarchical, yet has an inversion, from $(x_1, y_1)$ to $(x_2, y_2)$.

We give now an alternative, syntactic characterization of an inversion-free query, which we need later. Consider a query expression $Q$ given by the grammar Equation 3.1. Let $g$ be an atom in $Q$, over the relation symbol $R$ of arity $k$; thus $g$ contains $k$ distinct variables. Assume
the existential quantifiers of these $k$ variables are in the following order: $\exists x_1, \exists x_2, \ldots, \exists x_k$. In other words, each variable $x_{i+1}$ is within the scope of $x_i$. Define $\pi_g$ to be the permutation for which $g = R(x_{\pi_g(1)}, \ldots, x_{\pi_g(k)})$.

**Definition 3.11.** A query expression $Q$ given by the grammar Equation 3.1 is an inversion-free expression if it is a hierarchical expression, and for any two atoms $g_1, g_2$ with the same relational symbol, $\pi_{g_1} = \pi_{g_2}$.

If $Q$ is a hierarchical expression and $R$ a relational symbol, then we write $\pi_R$ for the common permutation $\pi_g$ of all atoms $g$ with symbol $R$. We have the following equivalence:

**Proposition 3.12.** $Q$ is inversion free iff it can be written as an inversion-free expression.

*Proof.* We first show how to write an inversion-free query as an inversion-free expression. Let $Q$ be inversion free. We will define an order relation $x \gg y$ on $Q$’s variables s.t. (a) for every atom $g$, $\gg$ is total over the set of variables occurring in $g$, and (b) if $g, g'$ are two atoms with the same relation symbol $R$ then the order imposed by $\gg$ on the attributes of $R$ is the same in $g$ and $g'$. The order $\gg$ gives us immediately the permutation $\pi_g$ for every atom $g$; since $Q$ can be written as a union of conjunctive queries, each of which is hierarchical, it follows that $Q$ can be written as an inversion-free expression. To define $\gg$, we first define a weaker relation $\gg: x \gg y$ if there exists a path in the unification graph from a node $(x, y)$ to a node $(x', y')$ s.t. $x' \succ y'$. Clearly $\gg$ is antisymmetric, because if we have both $x \gg y$ and $x \ll y$ then the graph has an inversion. We prove that $\gg$ is transitive. Indeed, suppose $x \gg y$ and $y \gg z$. By definition there exists a unification path $(y, z), (y_1, z_1), (y_2, z_2), \ldots, (y_k, z_k)$ s.t. $y_k \succ z_k$. Consider the first edge of this path: there exists two atoms $g, g_1$ with the same relation name, $g$ contains $y, z$, and $g_1$ contains $y_1, z_1$ on the same position. Then $g$ must contain $x$ as well (otherwise $x \prec y$ contradicting $x \gg y$). Denote $x_1$ the variable on the same position in $g_1$: since $x \gg y$ and $y \gg z$ we have $x_1 \gg y_1$ and $y_1 \gg z_1$. Repeating the same argument we find variables $x_i$ s.t. $x_i \gg y_i$ and $y_i \gg z_i$, for $i = 1, k$. Since $y_k \succ z_k$, it also follows that $x_k \succ z_k$ (because $x_k \gg y_k$ implies $\text{at}(x_k) \supset \text{at}(y_k)$ hence $\text{at}(x_k) \not\subseteq \text{at}(z_k)$), proving that $x \gg z$. Therefore, $\gg$ defines a partial order on the set of variables. It is not a total order yet, because it may leave pairs of variables unordered. To make it a total order, we use the fact that the query $Q$ is ranked, and define $x \ggg y$ to be: $x \gg y$ or ($x \nless y$ and there exists an atom $g$ containing both $x$ and $y$ s.t. $x$ occurs before $y$). It is easy to check that $\ggg$ is a partial order, and for any atom $g$ it is total over its set of variables.

Now, suppose $Q$ can be written as an inversion-free expression and still has an inversion from a node $(x, y)$ to node $(x', y')$ s.t. $x \succ y$ and $x' \prec y'$. Let $\pi$ be the order in which variables are
introduced in the inversion-free expression. Then \( x, y \) appear in the same order in \( \pi \) as \( x', y' \). W.l.o.g., let’s assume \( x \) occurs before \( y \). Then \( x' \) occurs before \( y' \). But we have \( y' \succ x' \), hence \( x' \) couldn’t have been a root variable when it was introduced which violates the fact that every inversion-free expression is also a hierarchical expression, a contradiction. This completes the proof.

For example, consider \( q_1 \) in Table 3.1. On one hand we can write it as a union of conjunctive queries, \( q_1 = R(x_1), S(x_1, y_1) \lor T(x_2), S(x_2, y_2) \). The unification graph has four nodes, \((x_1, y_1), (y_1, x_1), (x_2, y_2), (y_2, x_2)\), and two edges \(((x_1, y_1), (x_2, y_2))\) and \(((y_1, x_1), (y_2, x_2))\). We have both \( x_1 \succ y_1 \) (because \( at(x_1) = \{R(x_1), S(x_1, y_1)\} \not\subseteq at(y_1) = \{S(x_1, y_1)\} \)), and similarly \( x_2 \succ y_2 \). Hence, there is no inversion in the graph, and the query is inversion free. The proposition gives us an alternative way to see that, by writing the query as \( q_1 = \exists x_1. R(x_1), \exists y_1. S(x_1, y_1) \lor \exists x_2. T(x_2), \exists y_2. S(x_2, y_2) \); in both \( S \)-atoms the existential variables \( x_1, y_1 \) are introduced in the same order, for \( i = 1, 2 \).

On the other hand, consider the query \( h_1 = R(x_1), S(x_1, y_1) \lor S(x_2, y_2), T(y_2) \). Here \( x_1 \succ y_1 \) and \( x_2 \prec y_1 \), hence the edge \(((x_1, y_1), (x_2, y_2))\) forms an inversion in the unification graph. One can see that we cannot write \( h_1 \) in a way that satisfies Theorem 3.11: if we write it hierarchically as \( \exists x_1. R(x_1), \exists y_1. S(x_1, y_1) \lor \exists y_2. T(y_2), \exists x_2. S(x_2, y_2) \), then the variables in \( S(x_2, y_2) \) are introduced in a different order from those of \( S(x_1, y_1) \).

We end with a simple remark. If \( d \) is a disjunctive query that is inversion free, then it has a separator. Indeed, write \( d = \bigvee_i c_i \), and write each component as a hierarchical expression, \( c_i = \exists x_i. Q_i \). Re-write \( d \) as \( \exists w. (\bigvee_i Q_i[w/x_i]) \). Then \( w \) is a separator variable: it obviously occurs in all atoms, and in every atom with relation symbol \( R \), it must occur in position \( \pi_R(1) \).

### 3.3 Queries with Read-Once Lineage

A Boolean expression \( \Phi \) is read once (RO) if it can be written using the connectors \( \lor, \land, \neg \) such that every Boolean variable occurs at most once. We consider only positive Boolean expressions, and therefore will use only \( \lor \) and \( \land \). The probability of a read-once Boolean expression can be computed in linear time, because of independence: \( P(\Phi_1 \land \Phi_2) = P(\Phi_1) \cdot P(\Phi_2) \) and \( P(\Phi_1 \lor \Phi_2) = 1 - (1 - P(\Phi_1))(1 - P(\Phi_2)) \); this justifies our interest in this class of expressions. In this section we characterize the queries that have read-once lineages. An elegant characterization of read-once Boolean expressions was given by Gurvich [48] (see [44]), but we will not use that characterization. Note that our characterization is of queries, while Gurvich’s characterization is of Boolean expressions.
**Definition 3.13.** \( UCQ(RO) \) is the class of queries \( Q \) s.t. for every database instance \( D \), the lineage of \( Q \) on \( D \) is a read once Boolean expression.

Recall that \( CQ^- \) denotes the set of conjunctive queries without self-joins. Dalvi and Suciu \cite{DalviSuciu2006,DalviSuciu2007} showed that \( CQ^- (P) \) is precisely the class of hierarchical queries. Olteanu and Huang \cite{OlteanauHuang2008} showed that all hierarchical queries in \( CQ^- \) have read-once lineages, implying \( CQ^- (RO) = CQ^- (P) = \) “hierarchical queries”. In this section we characterize the class \( UCQ(RO) \).

**Definition 3.14.** Let \( Q \) be a query expression given by the grammar Equation 3.1. We say that \( Q \) is hierarchical-read-once if it is hierarchical (see Theorem 3.8), and every relational symbol occurs at most once. A query is hierarchical-read-once if it is equivalent to a hierarchical-read-once expression.

Obviously, every hierarchical \( CQ^- \) query is also hierarchical-read-once; our definition is more interesting when applied to \( UCQ \). The following is a necessary condition for hierarchical-read-once-ness:

**Proposition 3.15.** If \( Q \) is a hierarchical read-once expression then it is also an inversion-free expression.

The proof is immediate, since no two distinct atoms in \( Q \) may refer to the same relational symbol, hence the condition \( \pi_{g_1} = \pi_{g_2} \) is satisfied vacuously.

For a simple example, consider query \( q_1 \) in Table 3.1. It is equivalent to the expression \( \exists x. (R(x) \lor T(x)) \land \exists y. S(x, y) \), which is both hierarchical and read-once. Notice that in the definition we require \( Q \) to be be at the same time hierarchical and read-once. Sometimes we can achieve these two goals separately, but not simultaneously: for example \( h_1 = R(x_1), S(x_1, y_1) \lor S(x_2, y_2), T(y_2) \) is hierarchical, and can also be written as \( \exists x. \exists y. (R(x) \lor T(y)) \land S(x, y) \), which is read-once. Since \( h_1 \) has an inversion, by Theorem 3.15 it cannot be written simultaneously as a hierarchical and read-once expression.

**Theorem 3.16.** \( Q \in UCQ(RO) \) iff it is hierarchical-read-once.

The “if” direction is a straightforward extension of the technique used in \cite{OlteanauHuang2008} to prove that hierarchical queries in \( CQ^- \) are read-once. For the “only-if”, we construct one database instance \( D \) that is “large enough” (depending only on the query), and prove the following: if \( Q \)’s lineage on \( D \) is read-once, then \( Q \) is hierarchical-read-once.

**Proof.** If : We prove by induction on the structure of a hierarchical-read-once expression \( Q \) that its lineage is read-once; this proof extends that of \cite{OlteanauHuang2008}. If \( Q = Q_1 \lor / \land Q_2 \), then \( Q_1, Q_2 \) have no relation symbols in common, and their lineage given by Equation 3.3 is also read-once. If
$Q = \exists x. Q_1$ then $x$ must be a root variable, which implies that the lineages of $Q_1[a_1/x], \ldots, Q_1[a_n/x]$ do not share any Boolean variables; hence, the lineage given by Equation 3.2 is also read-once.

Only if: Assume $Q \in UCQ(RO)$; thus $\Phi_Q^D$ is read-once, for every database instance $D$; we show that $Q$ must be equivalent to a hierarchical-read-once expression. First, we use Theorem 3.18 to argue that, if $Q \in UCQ(RO)$ then $Q \in UCQ(OBDD)$, hence $Q$ is inversion-free. Referring to Theorem 3.11, for every relation symbol $R$ we denote $\pi_R$ the permutation mapping the variable nesting order to the order in which they occur in an atom with relation symbol $R$.

We will construct a special database instance $D$: from the read-once-ness of $\Phi_Q^D$, we will extract a hierarchical-read-once expression for $Q$. Let $m$ be the maximum arity of any relational symbol, and $k$ be the total number of variables plus the total number of atoms in $Q$. We first construct the active domain for $D$. Start by choosing $k$ constants $a_1, a_2, \ldots, a_k$ called “root constants”: these will be used to populate the attribute $\pi_R(1)$ of the relation $R$, for each relation symbol $R$. Next, choose $k^2$ constants $a_{ij}, 1 \leq i, j \leq k$: these will populate the attribute $\pi_R(2)$ of each relation $R$, such that $a_{ij}$ occurs only in those tuples that also contain $a_i$. Next, choose $k^3$ constants for the next level of the hierarchy, etc. This way we construct $k^a$ tuples for a relation of arity $a$: note that the functional dependencies $\pi_R(i + 1) \rightarrow \pi_R(i)$ hold for every relation $R$ and every $i = 1, \ldots, \text{arity}(R) - 1$.

Thus, we have fixed the database $D$. Next, we prove the following statement by induction: Let $Q$ be any inversion-free query where the total number of variables plus atoms is at most $k$, and consider its lineage over our fixed database $D$, $\Phi_Q^D$: if the lineage is read-once, then $Q$ can be written as a hierarchical-read-once expression. Our induction proceeds on the structure of the read-once expression $\Phi_Q^D$, abbreviated $\Phi_Q$.

Case 1: Suppose $\Phi_Q = \phi_1 \land \phi_2$. Then $\phi_1, \phi_2$ have no common Boolean variables. We prove something stronger: that the variables come from disjoint sets of relations. Assume contrary, that both have a Boolean variable over the relational symbol $R$. To simplify the discussion we will assume $R$ is unary; our argument extends in general too. First of all, note that if $m_1$ is a minterm of $\phi_1$ and $m_2$ is a minterm of $\phi_2$, then $m_1m_2$ must be a minterm of $\phi_Q$. This is because $\phi_1, \phi_2$ have no tuples in common, hence if another minterm $m'_1m'_2 \Rightarrow m_1m_2$, then $m'_1 \Rightarrow m_1$ and hence $m'_1$ couldn’t have been a minterm of $\phi_1$.

Now, suppose $X_{R(a_1)}$ occurs in $\phi_1$ and not in $\phi_2$, and $X_{R(a_2)}$ occurs in $\phi_2$ and not in $\phi_1$. Then $X_{R(a_1)}X_{R(a_2)}$ occurs in some minterm in $\phi_Q$. Since the lineage is invariant under permutations of the active domain, for all $1 \leq i < j \leq k$ the term $X_{R(a_i)}X_{R(a_j)}$ occurs in some minterm of $\phi_Q$. Consider a third tuple of $R$, say $R(a_3)$. It must occur in either $\phi_1$ or $\phi_2$: assume w.l.o.g. it
occurs in $\phi_2$, and since $\Phi_Q$ has a minterm that contains $X_{R(a_2)}X_{R(a_3)}$, $\phi_2$ must have a minterm that contains it. Hence, after conjoining with $\phi_1$, we obtain a minterm in $\Phi_Q$ that contains $X_{R(a_1)}X_{R(a_2)}X_{R(a_3)}$, and, therefore, for every $1 \leq i < j < k \leq l$ there exists a minterm in $\Phi_Q$ containing $X_{R(a_i)}X_{R(a_j)}X_{R(a_k)}$. Repeating this argument leads us to conclude that $\Phi_Q$ has a minterm containing $X_{R(a_1)} \ldots X_{R(a_k)}$: this is a contradiction because the minterms of $\Phi_Q$ cannot have more variables than the number of atoms in $Q$.

Thus, $\phi_1$ contains only tuples over the relations $R_1, R_2, \ldots$ and $\phi_2$ contains only tuples over the relations $S_1, S_2, \ldots$. Denote $Q_1 = Q[S_1 = S_2 = \ldots = \text{true}]$ be the query obtained from $Q$ by replacing all atoms referring to $S_i$ with true; similarly denote $Q_2 = Q[R_1 = R_2 = \ldots = \text{true}]$. The lineage of $Q_1$ on $D$ is $\phi_1$: hence, by induction hypothesis, $Q_1$ is equivalent to a hierarchical-read-once expression. Similarly $Q_2$. We prove now that $Q \equiv Q_1 \land Q_2$. Since every atom logically implies true, we obtain immediately $Q \Rightarrow Q_1$ and $Q \Rightarrow Q_2$, hence $Q \Rightarrow Q_1 \land Q_2$. For the converse, write $Q_1 \land Q_2$ as a union of conjunctive queries $\bigvee_i q_i$, and let $D_q$ be the canonical database for $q_i$: it suffices to prove that $Q$ is true on $D_q$. Both $Q_1$ and $Q_2$ are inversion-free, hence $q_i$ is inversion free, and its canonical database $D_q$ satisfies all functional dependencies that hold in $D$: therefore we can find an isomorphic copy of $D_q$ in $D$, since we have choose $D$ “large enough”. Set all Boolean variables corresponding to this copy to true and all others to false: we have $\phi_1 \land \phi_2 = \text{true}$ (because $Q_1 \land Q_2$ is true on $D_q$), which implies $\Phi_Q = \text{true}$, implying that $Q$ is true on $D_q$.

**Case 2** $\Phi_Q = \phi_1 \lor \phi_2$. We distinguish two cases:

**Case 2.1**: Every minterm in $\Phi_Q$ consists of tuples that have the same root constant. Thus, it may contain variables like $X_{R(a_1, a_3)}X_{R(a_1, a_3)}X_{S(a_1)}$ (same root constant $a_1$) but not $X_{R(a_1, a_3)}X_{S(a_2)}$ (distinct root constants $a_1, a_2$). Then we claim $Q$ must be a disjunctive sentence. Indeed, consider the DNF expression for $Q = q_1 \lor q_2 \lor \ldots$ and assume w.l.o.g. that $q_1$ is not connected, hence $q_1 = c \land c'$ where $c, c'$ are two components. Then the lineage of $q_1$ includes minterms with mixed root constants, contradiction. Hence, $Q$ is a disjunctive sentence. Now we use the fact that $Q$ is inversion-free; in particular it has a separator, $Q = \exists v. Q_1$, and its lineage is $\Phi_Q = \bigvee_{i=1}^k \Phi_{Q_1[a_i/x]}$. Since $\Phi_Q$ is read-once, so is each $\Phi_{Q_1[a_i/x]}$ (since the latter is obtained from $\Phi_Q$ by setting to false all tuples with root constant $a_j$, for $j \neq i$). Hence, we apply induction hypothesis to $Q_1[a_i/x]$ and obtain a hierarchical-read-once expressions: this proves that $\exists x. Q_1$ is hierarchical-read-once.

**Case 2.2**: $\Phi_Q = \phi_1 \lor \phi_2$ and there is at least one mixed minterm, containing tuples with two distinct root constants, say $X_{R_1(a_1, b_1)}X_{R_2(a_2, c)}$, and assume w.l.o.g. that this minterm appears in $\phi_1$. We will show that all $R_2$-tuples occur in $\phi_1$, i.e. $\phi_2$ does not have $R_2$ tuples. We
consider the case when $R_1$ and $R_2$ are distinct relational symbols: the case when they are the same symbol is similar. We use again the fact that $Q$ is invariant under permutations of $D$ to argue that $\Phi_Q$ must contain minterms that contain the tuples $X_{R_1(a_1,b)}X_{R_2(a_j,c')}$ for any $j \leq k$. All these minterms must be in $\phi_1$, since $\phi_2$ may not contain $X_{R_1(a_1,b)}$. Thus, $\phi_1$ must contain all tuples over $R_2$. With a similar argument, it must also contain all tuples over $R_1$.

Denote $R_1, R_2, \ldots$ the relation symbols that occur only in $\phi_1$ and $S_1, S_2, \ldots$ the other symbols. By our assumption, at least one symbol is in the first list (because there is a mixed minterm in $\phi_1$) and at least one symbol is in the second list (because $\phi_2$ is not empty). We prove that no minterm contains tuples with relation symbols from both lists. Indeed, if the minterm is mixed, then we have seen that its symbols appear either only in $\phi_1$ (hence they are all $R_i$ symbols), or only in $\phi_2$ (hence they are all $S_j$ symbols). Suppose the minterm contains a unique root constant, say $a_1$, i.e. the minterm contains $X_{R_1(a_1,\ldots)}X_{S_j(a_1,\ldots)}$. Since the minterms are closed under isomorphisms of the domain, there are minterms containing $X_{R_1(a_2,\ldots)}X_{S_j(a_2,\ldots)}$, $X_{R_1(a_3,\ldots)}X_{S_j(a_3,\ldots)}$, etc. All these must belong to $\phi_1$ (because they contain $R_i$); hence $\phi_1$ contains all tuples $S_j$, and therefore $S_j$ must also be in the list $R_1, R_2, \ldots$.

Define $Q_1 = Q[R_1 = R_2 = \ldots = \text{false}]$ i.e. formula obtained from $Q$ by setting all relation symbols $R_i$ to false. Similarly, define $Q_2 = Q[S_1 = S_2 = \ldots = \text{false}]$. We show that $Q = Q_1 \lor Q_2$, and since by induction hypothesis $Q_1, Q_2$ have an read-once expression, so does $Q$. First note that $Q_i \Rightarrow Q$, $i = 1, 2$, since false implies anything, hence $Q_1 \lor Q_2 \Rightarrow Q$. We will now show $Q \Rightarrow Q_1 \lor Q_2$, and here we use an argument similar to the above. Let $Q = \bigvee q_i$ and let $D_{q_i}$ be a canonical database for $q_i$. Since $D$ was chosen large enough, there exists an isomorphic copy of $D_{q_i}$ in $D$, and, consequently, a minterm in $\Phi_Q$ consisting of the conjunction of its tuples. This minterm either consists of $R_i$ tuples, hence $D_{q_i} \models Q_1$, or of $S_j$ tuples, hence $D_{q_i} \not\models Q_2$.

It is decidable if a given query $Q$ is hierarchical-read-once, because for a fixed vocabulary there are only finitely many hierarchical-read-once expressions: simply iterate over all of them and check equivalence to $Q$. This implies that it is decidable whether $Q \in UCQ(RO)$. For example, one can check that $q_2$ in Table 3.1 is not in $UCQ(RO)$, by enumerating all hierarchical-read-once expressions over the vocabulary $R, S, T$; we will return to $q_2$ in the next section.

3.4 Queries and OBDD

OBDD were introduced by Bryant [13] and studied extensively in the context of model checking and knowledge representation. A good survey can be found in [96]; we give here a quick overview.
A BDD, is a rooted DAG with two kinds of nodes. A sink node or output node is a node without any outgoing edges, which is labeled either 0 or 1. An inner node, decision node, or branching node is labeled with a Boolean variable $X$ and has two outgoing edges, labeled 0 and 1 respectively. Every node $u$ uniquely defines a Boolean expression $\Phi_u$ as follows: $\Phi_u = \text{false}$ and $\Phi_u = \text{true}$ for a sink node labeled 0 or 1 respectively, and $\Phi_u = \neg X \land \Phi_{u_0} \lor X \land \Phi_{u_1}$ for an inner node labeled with $X$ and with successors $u_0, u_1$ respectively. The BDD represents a Boolean expression $\Phi : \Phi \equiv \Phi_u$ where $u$ is the root of the BDD. A Free BDD, or FBDD is one in which every path from the root to a sink node contains any variable $X$ at most once. Given an FBDD that represents $\Phi$, one can compute the probability $P(\Phi)$ in time linear in the size of the FBDD: this justifies our interest in FBDD.

While it is trivial to construct a large FBDD for $\Phi$ (e.g. as a tree of size $2^n$ that checks exhaustively all $n$ variables $X_1, \ldots, X_n$), it is not trivial at all to construct a compact FBDD. To simplify the construction problem, Bryant [14] introduced the notion of Ordered BDD, OBDD, which is an FBDD such that there exists a total order $\Pi$ on the set of variables s.t. on each path from the root to a sink, the variables $X_1, \ldots, X_n$ are tested in the order $\Pi$ (variables may be skipped). One also writes $\Pi$-OBDD, to emphasize that the OBDD has order $\Pi$. Therefore, the OBDD construction problem has been reduced to the problem of finding a variable order $\Pi$.

One can construct an OBDD for any read-once formula $\Phi$ in time linear in $\Phi$, by an inductive

---

**Figure 3.2: OBDD for the query $R(x), S(x, y)$ (cf. [68])**
argument: if $\Phi = \Phi_1 \land \Phi_2$ first construct OBDDs for $\Phi_1$ and $\Phi_2$, and replace every sink-node labeled 1 in $\Phi_1$ with (an edge to) the root of $\Phi_2$; for $\Phi_1 \lor \Phi_2$, replace every sink-node labeled 0 in $\Phi_1$ with the root of $\Phi_2$.

**Definition 3.17.** UCQ(\textit{OBDD}) is the class of queries $Q$ s.t. for every database $D$, one can construct an OBDD for $\Phi^D_Q$ in time polynomial in $|D|$.

We show an example in Figure 3.2. In this section we prove the following:

**Theorem 3.18.** $Q \in \text{UCQ}(\text{OBDD})$ iff it is inversion-free.

We have seen that $q_2$ from Table 3.1 is not read-once. However, $q_2 \in \text{UCQ}(\text{OBDD})$, because it is inversion-free, therefore we obtain the following separation:

**Proposition 3.19.** $q_2 \in \text{UCQ}(\text{OBDD}) - \text{UCQ}(\text{RO})$

The significance of this result is the following. Olteanu and Huang [68] showed that for any hierarchical query $Q$, one can construct an OBDD for $\Phi^D_Q$ in time $O(|D|)$, proving that $\text{CQ}^-(\text{RO}) = \text{CQ}^-(\text{OBDD})$. Our proposition shows that these classes no longer collapse over UCQ.

We also note that all inversion-free queries are hierarchical (Section 3.2), therefore any non-hierarchical query is not in UCQ(\textit{OBDD}).

In the remainder of the section we prove Theorem 3.18, in two stages: first showing that one can construct in PTIME an OBDD for inversion-free formulae, and every query with inversion has exponential size OBDD over some database.

### 3.4.1 Tractable Queries

Given an OBDD of $\Phi$ over variables $\bar{x}$ with variable order $\Pi$, the width at level $k$, $k \leq n$ is the number of distinct subformulae that result after checking first $k$ variables in the order $\Pi$, i.e. $|\{\Phi_{x_{\pi(1)}...x_{\pi(k)}} = \bar{b} \mid \bar{b} \in \{0,1\}^k\}|$. The width of an OBDD is the maximum width at any level. If the number of variables is $n$, and width $w$, then a trivial upper bound on the size of the OBDD is $nw$. In what follows, we give a variable ordering for inversion-free queries under which the width is always constant (exponential in query size) and hence the size of OBDD is linear. Note that if the size of OBDD is polynomial, then the construction of the OBDD can be done in PTIME in our setting, since the lineage of a UCQ is always a monotone formula and checking the equivalence of monotone formulas is in PTIME.

We first need to define the notion of a shared BDD. A shared BDD for a set of formulas $\Phi_1, \Phi_2, \ldots, \Phi_m$ is a BDD where the sink nodes are labeled with $\{0,1\}^m$ i.e. they give the
valuation for each of the $\Phi_i, 1 \leq i \leq m$. This means a node reached by following the assignments $\bar{x}$ from the root can be thought of as representing a set of subformulae $\Phi_{1x}, \Phi_{2x}, \ldots, \Phi_{kx}$. Shared BDD evaluate a set of formulae simultaneously: this enables us to compute any combination function of the formulae. So, for instance, one can derive the OBDD of $\Phi_1 \odot \Phi_2$ for any boolean operation $\odot$ from the shared OBDD for $\Phi_1, \Phi_2$

The following is a well-known lemma for OBDD synthesis.

**Lemma 3.20.** (cf. [96]) Let $\Phi_1, \Phi_2$ be two boolean functions and consider a fixed variable order $\Pi$. If there exists $\Pi$-OBDDs of width $w_1, w_2$ for $\Phi_1, \Phi_2$ respectively, then there exists a shared $\Pi$-OBDD of width $w_1w_2$ for $\Phi_1, \Phi_2$.

**Proposition 3.21.** If $Q$ is inversion-free, then for every database $D$ its lineage has an OBDD with width $w = 2^g$, where $g$ is the number of atoms in the query. Therefore, the size of the OBDD is linear in the size of the database.

We give a simple proof, using Theorem 3.20, that constructs the OBDD inductively on the hierarchical expression for $Q$: the resulting OBDD has size $O(|D|)$.

**Proof.** Consider a hierarchical expression for $Q$, and let $\pi_R$ be the permutation associated to the symbol $R$ (Theorem 3.11). Let $D$ be a database, and assume that its active domain $ADom(D)$ is an ordered domain. We start by defining a linear order $\Pi$ on all tuples in $D$. Fix any linear order on the relational symbols, $R_1 < R_2 < \ldots$. We add all relation symbols to $ADom(D)$, placing them at the beginning of the order. We associate to each tuple in $D$ a string in $(ADom(D))^*$, as follows: tuple $R(a_{\pi_R(1)}, a_{\pi_R(2)}, \ldots, a_{\pi_R(k)})$ is associated to the string $a_1a_2\ldots a_kR$. That is, the first element is the constant on the root attribute position; the second element is the constant on the attribute position corresponding to a quantifier depth 2, etc. We add the relation name at the end. Next, we order the Boolean variables in the lineage expression $\Phi_Q$ lexicographically by their string, and denote $\Pi$ the resulting order. We prove that $\Pi$-OBDD has width $w = 2^g$, inductively on the structure of the inversion-free expression $Q$. If $Q = Q_1 \lor / \land Q_2$ then we use Theorem 3.20. If $Q = \exists x. Q_1$, then $\Phi_Q = \bigvee_{a \in ADom(D)} \Phi_{Q[a/x]}$: Let the active domain consists of $a_1 < a_2 < \ldots < a_n$, in this order. The OBDDs for $\Phi_{Q[a_1/x]}, \ldots, \Phi_{Q[a_n/x]}$ are over disjoint sets of Boolean variables (because $x$ is a root variable); assume that their width is $w$. The OBDD for $\Phi_Q$ consists of their union, where we redirect the 0 sink nodes of $\Phi_{Q[a_i/x]}$ to the root node of $\Phi_{Q[a_{i+1}/x]}$: the width is still $w$. The OBDD of a single ground atom, say $R(\bar{a})$, has width only 2. This completes the proof.

**Corollary 3.22.** If a set of components $c_1, c_2, \ldots, c_m$ is inversion-free, then for every database $D$, they have a shared-OBDD with size linear in the size of the database.
3.4.2 Hard Queries

For \( k \geq 1 \), define the following queries (see also Figure 3.1):

\[
\begin{align*}
  h_{k0} &= R(x_0), S_1(x_0, y_0) \\
  h_{ki} &= S_i(x_i, y_i), S_{i+1}(x_i, y_i) & i = 1, k - 1 \\
  h_{kk} &= S_k(x_k, y_k), T(y_k)
\end{align*}
\]

Denote \( h_k = \bigvee_{i=0}^{k} h_{ki} \). The queries \( h_k \) were shown in [25, 28] to be hard for \#P and are used to prove the hardness of a much larger class of unsafe queries. We show here that they have a remarkable property w.r.t. OBD: if the same variable order \( \Pi \) is used to compute all queries \( h_{k0}, h_{k1}, \ldots, h_{kk} \), then at least one of these \( k+1 \) OBDDs has exponential size. Note that each query is inversion-free, hence it admits an efficient OBDD, e.g. Figure 3.2 illustrates \( h_{k0} \): what we prove is that there is no common order under which all have an efficient OBDD. This tool is quite powerful, allowing us to give a rather simple proof that queries with inversion have exponential size OBDD (Theorem 3.24). There is no analogous tool for proving \#P-hardness: all queries \( h_{ki} \) are in PTIME, for \( i = 0, k \), and this tells us nothing about the larger query where they occur.

The complete bipartite graph of size \( n \) is the following database \( D \) over the vocabulary of \( h_k \): relation \( R \) has \( n \) tuples \( R(a_1), \ldots, R(a_n) \), relation \( T \) has \( n \) tuples \( T(b_1), \ldots, T(b_n) \), and each relation \( S_i \) has \( n^2 \) tuples \( S_i(a_j, b_l) \), for \( i = 1, k \), and \( j, l = 1, n \).

**Proposition 3.23.** Let \( D \) be the complete bipartite graph of size \( n \), and fix any ordering \( \Pi \) on the corresponding Boolean variables. For any \( i = 0, k \), let \( n_i \) be the size of some \( \Pi \)-OBDD for the lineage of \( h_{ki} \) on \( D \). Then \( \sum_{i=0}^{k} n_i > k \cdot 2^\frac{n}{k} \).

**Proof.** Denote the Boolean variables associated to the tuples \( R(a_i) \), \( i = 1, n \) with \( X_1, X_2, \ldots \); those associated to the tuples \( S_p(a_i, b_j) \) with \( Z_p \); and those associated to the tuples \( T(b_j) \) with \( Y_j \). We will refer generically to any variable as \( v_i \), and assume the order \( \Pi \) is \( v_1, v_2, \ldots \). Denote \( \Phi_{kp} \) the lineage of \( h_{kp} \) on \( D \); by assumption, we have \( \Pi \)-OBDD for each of them. Assume w.l.o.g. that each OBDD is complete i.e. every path from root to sink contains every variable exactly once.

In any OBDD of a Boolean expression \( \Phi \), the number of nodes at level \( h \) (i.e. after first \( h \) variables \( v_1 \ldots v_h \) have been eliminated) is the size of the set \( \{ \Phi([v_1 \ldots v_h] = b_j) \mid b \in \{0, 1\}^h \} \). This is because every distinct subformula will result in a new separate node. A standard technique in proving lower bounds on the size of OBDD is to find a level where the number of distinct
formulae must be exponential. This immediately gives the same exponential lower bound on the size of \( OBDD \) for that ordering.

For any level \( h \), denote \( h_1, h_2 \) the number of \( X \), and of \( Y \) variables respectively in the initial sequence \( v_1, v_2, \ldots, v_h \) of \( \Pi \). Define \( h \) to be the first level for which \( h_1 + h_2 = n \). Denote \( X^{\text{set}} = \{ X_i \mid X_i \in \{ v_1, \ldots, v_h \} \} \) and \( X^{\text{unset}} = X \setminus X^{\text{set}} \), and similarly \( Y^{\text{set}}, Y^{\text{unset}}, Z^{\text{set}}, Z^{\text{unset}} \).

W.l.o.g. assume \( h_1 \geq n/2 \).

Consider the \( OBDD \) for \( \Phi_{k_0} = \bigvee_{ij} X_i Z_{ij}^1 \). Suppose there exists \( j \) s.t. \( \forall i. X_i \in X^{\text{set}} \Rightarrow Z_{ij}^1 \in Z^{\text{unset}} \); then for each assignment \( \bar{b} \) to \( X^{\text{set}} \), we get a different subformula \( \Phi_{k_0}[X^{\text{set}} = \bar{b}] \). Since the number of such formulae is \( 2^{h_1} \geq 2^{n/2} \), we obtain \( n_0 > 2^{n/2} \), which proves the claim. Hence we can assume there is no such \( j \). This means \( \forall j, \exists i \) s.t. \( X_i \in X^{\text{set}} \) and \( Z_{ij}^1 \in Z^{\text{set}} \).

Define \( S \) to be a set of pairs \((i, j)\) as follows. For each \( j \) s.t. \( Y_j \in Y^{\text{unset}} \), choose some \( i \) s.t. \( Z_{ij}^1 \in Z^{\text{set}} \); then include \((i, j)\) in \( S \). Note that the cardinality of \( S \) is \( n - h_2 = h_1 \).

For each \( p = 1, \ldots, k-1 \), denote \( C_p \) the subset of \( S \) consisting of indices \((i, j)\) s.t. \( Z_{ij}^1, \ldots, Z_{ij}^p \in Z^{\text{set}} \) and \( Z_{ij}^{p+1} \in Z^{\text{unset}} \); and let \( C_k = S - \bigcup_{p=1,k-1} C_p \). Thus, \( C_1, \ldots, C_k \) forms a partition of \( S \). Denoting \( c_1, \ldots, c_k \) their cardinalities we have \( c_1 + \ldots + c_k = h_1 \).

Next, for each \( p = 1, \ldots, k-1 \), consider the \( OBDD \) for \( \Phi_{k_p} = \bigvee_{ij} Z_{ij}^p Z_{ij}^{p+1} \). For all \((i, j) \in C_p \) we have \( Z_{ij}^p \in Z^{\text{set}} \) and \( Z_{ij}^{p+1} \in Z^{\text{unset}} \). Each assignment of the former variables leads to a different expression over the latter variables: hence there are at least \( 2^{c_p} \) distinct expressions, therefore the number of nodes in this \( OBDD \) is \( n_p \geq 2^{c_p} \).

Finally, consider the \( OBDD \) for \( \Phi_{k_k} = \bigvee_{ij} Z_{ij}^k Y_j \). For all \((i, j) \in C_k \) we have \( Z_{ij}^k \in Z^{\text{set}} \) and \( Y_j \in Y^{\text{unset}} \). Using the same argument, we obtain \( n_k \geq 2^{c_k} \).

Putting everything together we obtain:

\[
\sum_{i=1,k} n_i \geq \sum_{i=1,k} 2^{c_i} \geq k 2^{\frac{\sum c_i}{k}} \geq k 2^{\frac{h_1}{k}} > k 2^{\frac{n}{2k}}
\]

Notice that \( n_0 \) does not appear above, but we used it in order to construct the set \( S \). This proves our claim.

\( \square \)

**Proposition 3.24.** Let \( Q \) be a query, and suppose it has an inversion of length \( k > 0 \). Let \( D_0 \) be a complete bipartite graph of size \( n \) (i.e. a database over the vocabulary of \( h_k \)). Then there exists a database \( D \) for \( Q \) s.t. \( |D| = O(|D_0|) \) and any \( OBDD \) for \( Q \) has size \( \Omega(k 2^{n/2k}) \).

We use the inversion of length \( k \) to construct a database \( D \) that mimics the query \( h_k \) over a complete bipartite graph. Assuming an \( OBDD \) for \( Q \) on this database, we show that one can
set the Boolean variables to 0 or 1, to obtain a lineage for each $h_{ki}$. What is interesting is that this construction cannot be used to prove \#P-hardness of $Q$ by reduction from $h_k$: in other words, $Q$ over $D$ is not equivalent to $h_k$ over $D_0$. But we make $Q$ equivalent to each $h_{ki}$, and by Theorem 3.23 this is sufficient to prove that $Q$ has a no compact OBDD.

Proof. Write $Q = \bigvee q_j$ in DNF, and let $(x_0, y_0), (x_1, y_1), \ldots, (x_k, y_k)$ be an inversion in $Q$. Assume w.l.o.g. that the inversion is of minimal length: this implies there exist atoms $r, s_1, s'_1, \ldots, s_k, s'_k, t$ with the following properties: $r \in at(x_0) \setminus at(y_0)$, $t \in at(y_k) \setminus at(x_k)$, and for every $i = 1, k$, $s_i$ contains $x_{i-1}, y_{i-1}$, $s'_i$ contains $x_i, y_i$, they unify, and the unification equates $x_{i-1} = x_i$ and $y_{i-1} = y_i$. In particular, the atoms $s_i$ and $s'_i$ have the same relation symbol. Assume that $x_i, y_i$ are variables in the query $q_{ji}$, for $i = 0, k$. We assume that these $k$ queries are distinct: if not, simply create a fresh copy of the query, creating new copies of its variables. Thus, $q_{j0}$ contains the atoms $r, s_1$, query $q_{ji}$ contains the atoms $s'_i, s_2$ and so on. Next, we perform variable substitutions in the queries $q_{j0}, \ldots, q_{jk}$ in order to equate all variables in $s_i$ and $s'_i$, except for $x_{i-1}, y_{i-1}, x_i, y_i$.

In other words, all atoms along the inversion path have the same variables, except for the variables forming the actual inversion. For example, if the queries were $R(x_0, u_0), S_1(x_0, y_0, u_0)$; $S_1(x_1, y_1, u_1), S_2(x_1, y_1, u_1, v_1)$; $S_2(x_2, y_2, u_2, v_2), \ldots$ then we equate $u_0 = u_1 = u_2 = \ldots$ and $v_1 = v_2 = \ldots$ This is possible in general because $Q$ is ranked: we only equate variables between $q_{ji}$ and $q_{ji}$, but not within the same $q_{ji}$. We now construct the database $D$ as follows. Its active domain consists of all constants $a_1, \ldots, a_n, b_1, \ldots, b_n$ and all variables $z \in Vars(q_{ji})$ s.t. $z \neq x_i$, $z \neq y_i$, for $i = 0, k$. For each $i = 0, k$, and each $j = 1, n$, $l = 1, n$, let $q_{ji}[a_j, b_l]$ denote the set of tuples obtained by substituting $x_i$ with $a_j$ and $y_i$ with $b_l$. Define $D$ to be the union of all these sets: $D = \bigcup_{i,j,l} q_{ji}[a_j, b_l]$. Because of our earlier variable substitutions, $s_i[a_j, b_l]$ and $s'_i[a_j, b_l]$ are the same tuple: this tuple corresponds to the tuple $S_i(a_j, b_l)$ in the bipartite graph. Similarly, $r(a_j)$ and $t(b_l)$ correspond to the tuples $R(a_j)$ and $T(b_l)$ in the bipartite graph. Thus, the bipartite graph $D_0$ is isomorphic to a subset of the database $D$. Consider now any OBDD for $\Phi^D_Q$, over a fixed variable ordering $\Pi$. We can obtain an OBDD for $h_{ki}$ for every $i = 0, k$ as follows. Assume $0 < i < k$. Then we keep unchanged the Boolean variables corresponding to $S_i(a_j, b_l)$ and $S_{i+1}(a_j, b_l)$, (that is, the atoms $s'_i[a_j, b_l]$ and $s_{i+1}[a_j, b_l]$). All other Boolean variables corresponding to tuples in $q_j[a_j, b_l]$ are set to $\text{true}$; all remaining Boolean variables are set to $\text{false}$. Then the lineage $\Phi^D_Q$ becomes the lineage $\Phi^D_{h_{ki}}$. The case $i = 0$ is similar (here we keep unchanged the Boolean variables corresponding to $R(a_j)$ and $S_1(a_j, b_l)$), and so is the case $i = k$. Thus, we obtain $k + 1$ OBDD’s for all queries $h_{ki}$, and all use the same variable order $\Pi$. The claim follows now from Theorem 3.23. \qed
If $Q$ has an inversion of length 0, then it is non-hierarchical and as we discuss later in Theorem 3.40 $Q \not\in UCQ(FBDD)$, and hence $Q \not\in UCQ(OBDD)$ either.

### 3.5 Queries and FBDD

We now turn to $FBDD$, also known as Read-Once Branching Programs. Unlike $OBDD$, here we no longer require the same variable order on different paths. $FBDD$ are known to be strictly more expressive than $OBDD$ over arbitrary (non-monotone) Boolean expressions, for example the Weighted Bit Addressing problem admits polynomial sized $FBDD$, but no polynomial size $OBDD$ \cite{15, 41, 85}. On the other hand, to the best of our knowledge no monotone formula was known to separate these two classes. Moreover, over conjunctive queries without self-joins, $FBDD$ are no more expressive than $OBDD$, since the latter already capture $CQ^-(P)$. In this section we show that $FBDD$ are strictly more expressive than $OBDD$ over $UCQ$. In particular, we give a simple (!) monotone Boolean expression for which one can construct a $FBDD$ in $PTIME$, but no polynomial size $OBDD$ exists.

**Definition 3.25.** $UCQ(FBDD)$ is the class of queries $Q$ s.t. for any database $D$, one can construct an $FBDD$ of $\Phi^D_Q$ in time polynomial in $|D|$.

Clearly $UCQ(OBDD) \subseteq UCQ(FBDD)$: we prove now that the inclusion is strict, using a simple example.
Example 3.2. Consider $q_V$ in Table 3.1. This query has an inversion between $S(x_1,y_1)$ and $S(x_2,y_2)$, hence it does not admit a compact OBDD. We show how to construct a compact FBDD. Write it in CNF:

$$q_V = (R(x_1), S(x_1,y_1) \lor T(y_3)) \land (S(x_2,y_2), T(y_2) \lor R(x_3))$$

$$= d_1 \land d_2$$

Its CNF lattice is shown in Figure 3.1. The minimal element of the lattice is:

$$d_3 = d_1 \lor d_2 = R(x_3) \lor T(x_3)$$

Each of $d_1, d_2, d_3$ is inversion-free, hence they have OBDDs, denote them $F_1, F_2, F_3$. Of course, $F_1$ and $F_2$ use different variable orderings and cannot be combined into an OBDD for $q_V$. Consider the database given by the bipartite graph (Section 3.4) and assume the following order on the active domain: $a_1 < \ldots < a_n < b_1 < \ldots < b_n$. Our FBDD starts by computing $d_3$. If $d_3 = 0$, then $q_V = 0$; this is a sink node. If $d_3 = 1$, then, depending on which sink node in $F_3$ we have reached, either $d_1 = 1$ or $d_2 = 1$, and we need continue with either $F_2$ or $F_1$ respectively. This way, no path goes through both $F_1$ and $F_2$. Note that the FBDD is not ordered, since some paths use the order in $F_1$, others that in $F_2$. Figure 3.3 illustrates the construction. The FBDD inspects the variables in this order: $X_{R(a_1)}, X_{R(a_2)}, \ldots, X_{R(a_n)}, X_{T(b_1)}, \ldots, X_{T(b_n)}$. (This is $F_3$.) Each edge $X_{R(a_i)} = 0$ leads to the next node, $X_{R(a_{i+1})}$, etc. Consider now an edge $X_{R(a_i)} = 1$. Here we know $d_2$ is true, but we still need to evaluate $d_1$. We create a new copy of $F_1$ where we set all variables $X_{R(a_1)}, \ldots, X_{R(a_{i-1})}$ to 0 (i.e. eliminate these nodes and redirect their incoming edge to their 0-child) and set $X_{R(a_i)}$ to 1. Then we connect the edge $X_{R(a_i)} = 1$ to the root node of this copy of $F_1$. Similarly, we connect an edge $X_{T(b_j)} = 1$ to the root of a copy of $F_2$ where we set $X_{R(a_1)} = \ldots = X_{R(a_n)} = X_{T(b_1)} = \ldots = X_{T(b_{i-1})} = 0$ and $X_{T(b_i)} = 1$. The result is an FBDD of size\(^3\) $O(n^3)$ (since $F_1, F_2$ have sizes $O(n^2)$).

Thus:

**Proposition 3.26.** $q_V \in UCQ(FBDD) - UCQ(OBDD)$.

The significance of this result is the following. The lineage of $q_V$ is, to the best of our knowledge, the first "simple" Boolean expression (i.e. monotone, and with polynomial size DNF) that has a polynomial size FBDD but no polynomial size OBDD. Previous examples separating these classes where Weighted Bit Addressing problem (WBA) [15, 41, 85], and other

---

\(^3\)In this particular example one could reduce the size to $O(n^2)$ by sharing nodes among the multiple copies of $F_1$ and similarly for $F_2$. 

examples given in [95], and these were not “simple”. Our result also constrains UCQ to $CQ^-$: for the latter it follows from [68] that $CQ^-(OBDD) = CQ^-(FBDD)$.

In the reminder of this section we will give a partial characterization of $UCQ(FBDD)$, by providing a sufficient condition, and a necessary condition for membership. We start with the sufficient condition.

**Definition 3.27.** Let $d = \lor c_i$ and $d' = \lor c'_j$ be two disjunctive queries, s.t. the logical implication $d' \Rightarrow d$ holds. We say that $d$ dominates $d'$ if for every component $c'_j$ in $d'$ and for every atom $g$ in $c'_j$ one of the following conditions hold: (a) the relation symbol of $g$ does not occur in $d$, or (b) there exists a component $c_i$ and a homomorphism $c_i \rightarrow c'_j$ whose image contains $g$.

In Example 3.2, $d_3$ dominates $d_1$: if one considers the component $R(x_1), S(x_1, y_1)$ in $d_1$, then the atom $R(x_1)$ is the image of a homomorphism, while the atom $S(x_1, y_1)$ does not occur at all in $d_3$. Similarly $d_3$ dominates $d_2$.

In analogy to the definition of safe queries Theorem 3.6 we define here rf-safe queries:

**Definition 3.28.** (1) Let $Q = d_1 \land \ldots \land d_k$, and $k \geq 2$. Then $Q$ is rf-safe if for every element $x$ in its CNF lattice the disjunctive query $\lambda(x)$ is rf-safe, and for every two lattice elements $x \leq y$, $\lambda(x)$ dominates $\lambda(y)$. (2) Let $d = d_0 \lor d_1$, be a disjunctive query, where $d_0$ contains all components $c_i$ without variables, and $d_1$ contains all components $c_i$ with at least one variable. Then $d$ is rf-safe if $d_1$ has a separator $w$ and $d_1[a/w]$ is rf-safe, for a constant $a$.

For example, query $q_V$ is rf-safe, since $d_3$ dominates both $d_1$ and $d_2$. Our sufficient characterization of $UCQ(FBDD)$ is:

**Theorem 3.29.** Every rf-safe query is in $UCQ(FBDD)$.

**Proof.** Of Theorem 3.29.

We start with a definition. Consider a monotone, Boolean expressions, written as disjunction of minterms: $\Phi = \lor_{i=1,n} T_i$. We also view $\Phi$ as a set of minterms, writing $T_i \in \Phi$. Each minterm $T_i$ is a set of variables: thus, $T_i \Rightarrow T_j$ means that, as sets, $T_j \subseteq T_i$.

**Definition 3.30.** Let $\Phi, \Phi'$ be two monotone, Boolean expressions, s.t. $\Phi' \Rightarrow \Phi$. We say that $\Phi$ dominates $\Phi'$ if for every minterm $T' \in \Phi'$, and for every Boolean variable $X \in T'$, one of the following conditions hold: (a) either $X$ does not occur in $\Phi$, or (b) there exists a minterm $T \in \Phi$ s.t. $X \in T$ and $T \subseteq T'$.

The following is easy to check:

---

4rf is for restricted, since we don’t have a full characterization yet
Lemma 3.31. If the disjunctive query \( d \) dominates \( d' \), then for any database \( D \), the lineage \( \Phi^D_d \) dominates \( \Phi^D_{d'} \).

Consider an FBDD for \( \Phi \), and a node \( x \). Any path from the root to \( x \) corresponds to an assignment of a subset of Boolean variables; we call that an assignment at \( x \).

Call an FBDD for \( \Phi \) greedy if each sink node \( x \) labeled 1 has an additional label consisting of (a) a set \( s \subseteq [n] \) and (b) an index \( i \), such that, for any assignment at \( x \), the following properties hold: \( T_i = 1 \) (i.e. all variables in \( T_i \) are set to 1 by the assignment). (2) For any \( j \in s \), \( T_j = 0 \). (3) For any \( k \notin s \), if the assignment sets a value of a variable in \( T_k \), then that variable is in \( T_i \) (hence it is set to 1). A greedy FBDD does exactly what the name says: it evaluates the Boolean DNF expression greedily. If a variable \( X = 0 \) then it skips all minterms that contain \( X \): these minterms now belong to the set \( s \). If a variable \( X = 1 \), then it continues to read only variables \( Y \) that co-occur with \( X \) in some minterm.

Let \( \Phi = \bigwedge_{i=1}^m \Phi_i \), where each \( \Phi_i \) is a monotone, Boolean expression. Let \((L, \leq)\) be the CNF-lattice for \( \Phi \) constructed as follows. Its elements \( x \) are in one-to-one correspondence with Boolean expressions \( \Phi_s = \bigvee_{i \in s} \Phi_i \), where \( s \subseteq [m] \), up to logical equivalence (i.e. if \( \Phi_{s_1} \equiv \Phi_{s_2} \) then they correspond to the same element \( x \)); \( \lambda(x) = \Phi_s \) denotes the Boolean expression associated to \( x \). And \( x \leq y \) if \( \lambda(y) \Rightarrow \lambda(x) \).

Lemma 3.32. Let \( \Phi = \bigwedge_{i=1}^m \Phi_i \), and \((L, \leq)\) be its CNF lattice. Suppose that, for all \( x \leq y \), \( \lambda(x) \) dominates \( \lambda(y) \). Let \( F_x \) be a greedy FBDD for \( \lambda(x) \), for each \( x \in L \). Then there exists a greedy FBDD for \( \Phi \), of size \( O((\prod u_x)^m) \).

Proof. (Sketch) We construct the FBDD by generalizing the idea of Example 3.2. Let \( x_0 = \hat{0} \) be the smallest element in the lattice. The FBDD starts with \( F_{x_0} \). Consider a sink node. If it is labeled 0, then we leave it labeled 0: we know that \( \Phi_1 = \ldots = \Phi_m = 0 \), hence so is \( \Phi \). If it is labeled 1, then we have two additional labels: a minterm \( T_i \) (known to be 1), and a set of minterms, \( TT \) (all known to be 0). Let \( s \subseteq [m] \) be the set s.t. \( i \in s \) iff \( T_i \) does not imply \( \Phi_i \); thus, from that sink node we have to continue to evaluate \( \Phi_s \). We assume, inductively, to have an FBDD for \( \Phi_s \). Then we modify it, by setting all variables in \( T_i \) to 1, and setting all other variables occurring in the set of minterms \( TT \) to 0: dominance ensures that the new FBDD still computes correctly \( \Phi_s \).

The proof of Theorem 3.29 follows now directly from the last two lemmas.

rf-safe is not a complete characterization of UCQ(FBDD). The following query \( q_T \), is not rf-safe, but one can construct a polynomial-size FBDD for it.
Next, we present our separation result. Recall the query \( q_W \) from Table 3.1. We prove here:

**Theorem 3.33.** \( q_W \notin UCQ(FBDD) \).

We will return to this query in the next section.

**Proof.** Consider the following three queries:

\[
\begin{align*}
d_1 &= R(x)S_1(x,y) \lor S_2(x,y)S_3(x,y) \\
d_2 &= R(x)S_1(x,y) \lor S_3(x,y)T(y) \\
d_3 &= S_1(x,y)S_2(x,y) \lor S_3(x,y)T(y)
\end{align*}
\]

Thus, \( q_W = d_1 \land d_2 \land d_3 \). We first prove a surprising fact that given an FBDD for \( q_W \), we can construct a shared FBDD for \( d_1, d_2, d_3 \). Note that in general it is not possible to split an FBDD into a shared FBDD: we exploit the properties of \( d_1, d_2, d_3 \) to do this.

**Lemma 3.34.** Given any FBDD for \( q_W \) of size \( N \), there exists a shared FBDD for \( d_1, d_2, d_3 \) of size polynomial in \( N \) and data instance.

Call a node shared if for any two paths \( \bar{x}, \bar{y} \) from root to the node, \( d_{ix} = d_{iy} \), for \( i=1,2,3 \). Hence if all nodes were shared, then the FBDD would also be shared. To prove the proposition, we show that if a node is not shared then the subformulae represented by that node is actually an inversion-free query, so we could just replace the FBDD below that node with a shared OBDD. Hence, one can make every node shared, and therefore the FBDD shared.

**Proof.** Of Theorem 3.34 Transformation into a shared FBDD. Each node \( x \) in an FBDD for \( F \) represents a Boolean expression \( F_x \), obtained by setting some variables in \( F \). If two paths \( P_1, P_2 \), lead to the same \( x \), then \( F[P_1] = F[P_2] \), where \( F[P_1] \) denotes the partial assignment done by \( P_1 \). Let \( F \) be the lineage of \( q_W = d_1 \land d_2 \land d_3 \), and write it as \( F = G_1 \land G_2 \land G_3 \), where
Chapter 3. Compiling UCQ

55

\(G_i\) is the lineage of \(d_i\). In general, two paths \(P_1, P_2\) in the FBDD\((q_W)\) that lead to the same node do not have to equate \(d_1\), i.e. we may have \(G_i[P_1] \neq G_i[P_2]\).

**Definition 3.35.** An FBDD for \(q_W\) is called shared if for any two path \(P_1, P_2\), if \(F[P_1] = F[P_2]\) then for all \(i = 1, 2, 3\), \(G_i[P_1] = G_i[P_2]\).

This lemma is significant, because it says that we can transform FBDD\((q_W)\) to compute each of the three queries \(d_1, d_2, d_3\) separately. In general, this is is not possible for an arbitrary conjunction of Boolean formulas. What makes this work in our case is that, whenever a node \(x\) fails to keep track separately of the three subqueries then the formula at \(x\) depends only on \(d_1, d_2\) or only on \(d_2, d_3\). Both these queries are inversion-free, hence in both cases we can construct shared OBDD for the remaining computation of the two queries, replacing the rest of the FBDD\((q_W)\)

Call a node \(x\) shared if for any two paths \(P_1, P_2\) leading to \(x\), we have \(G_i[P_1] = G_i[P_2]\), for \(i = 1, 2, 3\). We will leave shared nodes unchanged. If \(x\) is an non-shared node, then we show how to replace it with a shared OBDD for the remaining formulas. (Some of \(x\)'s descendants may become unreachable, and they can be removed later.)

Suppose \(x\) is a non-shared node. Let \(P_1, P_2\) be two paths leading to a node \(x\) in the FBDD\((F)\). Then \(F_x = F[P_1] = F[P_2]\). Suppose \(G_i[P_1] \neq G_i[P_2]\).

**Case 1** For every pair of constants \(a, b\), the path \(P_1\) sets either \(S_i(a, b) = 0\) or \(T(b) = 0\). Then \(G_2[P_1]\) has only terms \(R(a'), S_1(a', b')\) and similarly \(G_3[P_1]\) has only terms \(S_1(a', b'), S_2(a', b')\).

Denote the three queries below:

\[
\begin{align*}
  d_1 &= R(x), S_1(x, y) \lor S_2(x, y), S_3(x, y) \\
  e_2 &= R(x), S_1(x, y) \\
  e_3 &= S_1(x, y), S_2(x, y)
\end{align*}
\]

Let \(D_1, D_2, D_3\) be the set of tuples that are unset in \(G_1[P_1]\), \(G_2[P_2]\), and \(G_3[P_3]\). Then \(F_x\) is the conjunction of the lineages of \(d_1, e_2, e_3\) on these three databases. Since \(d_1, e_2, e_3\) are inversion-free, we can compute a shared OBDD that evaluates all three of them in parallel on \(D_1, D_2, D_3\) (Theorem 3.21) : thus, we have separated the FBDD at \(x\) and below \(x\).

**Case 2** There exists a pair of tuples \(X = S_3(a, b), Y = T(b)\) s.t. \(P_1\) leaves each of them unset, or sets them to 1. Set \(X = Y = 1\). Then \(G_2, G_3\) become true, and therefore \(G_1[P, X = 1, Y = 1] = F[P, X = 1, Y = 1]\), for \(P \in \{P_1, P_2\}\). Since \(F[P_1] = F[P_2]\) we obtain \(G_1[P_1, X = 1, Y = 1] = G_1[P_2, X = 1, Y = 1]\). Hence, the only way in which \(G_1[P_1]\) and \(G_1[P_2]\) may differ
is that either one has the term $S_2(a, b)$ and the other has $S_2(a, b), S_3(a, b)$, or one is true and the other has the term $S_3(a, b)$. The first case is impossible because it means that one of the two paths has set $S_3(a, b)$ to true. The second case implies $F_x = G_2[P1] \land G_3[P1]$ and we repeat the argument above.

This completes the case when $G_1$ differs on two paths. The case when $G_3$ differs is similar and omitted. So suppose $G_1[P1] = G_1[P2]$ and $G_3[P1] = G_3[P2]$. Then, we prove that we also have $G_2[P1] = G_2[P2]$. Indeed, this follows immediately by inspecting the three query lineages: $G_1$ is $\bigvee_{a,b} R(a)S_1(a, b) \lor S_2(a, b), S_3(a, b)$ and $G_2$ is $\bigvee_{a,b} R(a), S_1(a, b) \lor S_3(a, b), T(b)$. Thus, the portion of $R(a), S_1(a, b)$ that remains in $G_1[P1]$ is the same as that in $G_2[P1]$; since $G_1[P1] = G_1[P2]$, it means that this part of $G_2$ is the same in $P1$ and in $P2$. Similarly, from $G_3[P1] = G_3[P2]$ we conclude that the part $S_3(a, b), T(b)$ in $G_2$ is the same in $P1$ and $P2$. Hence, $G_2[P1] = G_2[P2]$.

Let $h'_1 = R(x_1), S(x_1, y_1), S(x_2, y_2), T(y_2)$. We can show that

\textbf{Lemma 3.36.} $h'_1 \notin UCQ(FBDD)$.

We start at the root and choose $2^{n-1}$ paths (our database is bipartite as in Section 3.4) as follows: for each node we go in both directions if the given node isn’t a prime implicant, otherwise we choose only the 0-edge. Then we exploit the fact that each of these paths has only set a limited number of variables to show that any two paths can differ on the assignment of only a few variables, hence most of them must end in distinct subformulas.

\textbf{Proof.} Of Theorem 3.36

Define $\Phi_1$ to be $\bigvee_{i,j=1}^n r_is_{i,j}$, $\Phi_2$ as $\bigvee_{i,j=1}^n s_{i,j}t_j$ and $F_n$ to be $\Phi_1 \land \Phi_2$. Then we show $FBDD(F_n) = n^{\Omega(\log(n))}$. We start at the root and choose $2^{n-1}$ paths $\mathcal{P}$ as follows: for each node we go in both directions if the given node isn’t a prime implicant in either $\Phi_1$ or $\Phi_2$: we call such nodes branching; otherwise we choose only the 0-edge. We ignore redundant (i.e. the two branches for 0,1 point to the same node) nodes. We stop a path after we have branched on $n-1$ nodes. There is a possibility that our function may become 0 before we branched on $n-1$ nodes, but that can be shown to be not possible because each branching variable can set at most $n^3$ minterms to 0. We have $n^4$ minterms to set to 0 and that can’t be done with $n-1$ branching nodes. The set of variables on a branch are denoted by $Vars(p)$ and $p(v)$ denotes the assignment of the variable $v$ in $p$. The boolean formulae $f_p = F_n[p(x)/x]$, $x = Vars(p)$ is obtained by applying the assignments on path $p$ to $F_n$. 

Definition 3.37. Given a boolean formulae $f$, call a variable $v$ determined, if there exists $b \in \{0, 1\}$ s.t. for any $p \in \mathcal{P}$ $f_p = f$ implies $v \in \text{Vars}(p)$ and $p(v) = b$. Conversely any variable $x \notin \text{Vars}(f)$ that is not determined is called undetermined.

Now the essence of the proof is that two paths that result in same function can only differ on the undetermined variables. The following lemma characterizes the variables that can be undetermined.

Lemma 3.38. Given a path $p$ and a variable $x \notin \text{Vars}(f_p)$; $x$ is undetermined for $f_p$ iff $x = s_{ij}$ for some $1 \leq i, j \leq n$ and

1. $r_i = 0$ or $s_{i1,j} = 1$ for some $1 \leq i \leq n$ on $p$ and
2. $t_j = 0$ or $s_{i,j1} = 1$ for some $1 \leq j \leq n$ on $p$

Proof. Follows immediately by doing a simple case analysis.

Note that for any $f_p$, number of paths $q$ s.t. $f_q = f_p$ is bounded above by $2^\#\text{undetermined vars in } f_p$. This is because $q$ and $p$ can only differ on the assignment of undetermined vars. We now bound the number of undetermined vars for any path $p$. Let $n_r, n_t$ be the number of $r, t$ variables set to 0 on path $p$; $n_s$ be the number of $s$ variables set to 1. Then by definition the number of undetermined vars is at most $(n_r + n_s)(n_t + n_s)$. Let $m_r, m_t, m_s$ similarly be the number of branching vars amongst $n_r, n_s, n_t$. Then $m_s = n_s$ and $n_r \leq m_r + m_s$, $n_t \leq m_t + m_s$. $m_s = n_s$ because non-branching variables are always set to 0. Also a non-branching variable from $r, t$ is set to 0 iff it becomes a prime implicant; which can only happen if you set one of the variables from $s$ to 1 and conversely setting one var from $s$ makes at most one prime implicant. This proves the second and third inequality.

Hence number of undetermined vars is at most $(m_r + 2m_s)(m_t + 2m_s) \leq 4(m_r + m_s + m_t)^2$. Now consider all paths $p$ where $l = m_r + m_s + m_t = \frac{\log(n)}{8}$. Consider the complete binary tree of depth $n - 1$ on the branching variables. Now flip the edges coming out of $r, t$ nodes i.e. 0 to 1 and vice-versa. We map any such path $p$ to a path in this tree by choosing opposite assignment to variables from $r, t$. This is a 1-1 mapping. And the set of such paths in our tree is exactly the set of paths where number of nodes tested to be 1 are $l$, which is $\binom{n-1}{l}$. Hence size of FBDD is at least $\frac{\binom{n-1}{l}}{2^{4l}}$ which for $l = \frac{\log(n)}{8}$ gives the required bound.

Now to finish the proof, we finally show a reduction from a shared FBDD of $d_1, d_2, d_3$ to $h'_1$.

Lemma 3.39. Given a shared FBDD for $q_W$ of size $N$, there exists an FBDD for $h'_1$ of size at most $N$. 

This is, perhaps, the most surprising step, because it seems to reduce a query that is hard for \#P \( (h_1') \) to a query that is in PTIME \( (q_W) \). However, what we describe below is not a reduction: instead is a transformation of an FBDD.

The goal of the reduction is to set \( S_1(a,b) = \neg S_2(a,b) = S_3(a,b) \) in \( FBDD(q_W) \): it can be easily seen (by inspecting the definition of the two queries) that the new FBDD computes \( h_1' \). Since we have shown in Theorem 3.36 that \( h_1' \) has no polynomial size FBDD, this completes the proof.

The difficulty of this step is to show that the FBDD has enough memory to remember if any of \( S_1(a,b) \), \( S_2(a,b) \), or \( S_3(a,b) \) was set. We show that it does have enough memory, by using the fact that it is a shared FBDD, i.e. it computes the queries \( d_1, d_2, d_3 \) simultaneously.

**Proof. Of Theorem 3.39**

Start with the shared FBDD for \( q_W \), given by Theorem 3.34. We want to enforce

\[
S_1(a,b) = \neg S_2(a,b) = S_3(a,b)
\]

for every tuple \( S_1(a,b) \).

Modify each node \( x \) as follows. If it is a test for \( R(a) \) or for \( T(b) \), then make no change: it will continue to be a test for the same variable. If it is a test for \( S_i(a,b), i = 1, 3 \), then we will either replace it with a test for \( S(a,b) \), or will “know” the value of \( S_i(a,b) \) and will follow only the 0 or the 1 branch. We give the details next.

Suppose node \( x \) tests for \( S_1(a,b) \). Denote \( G_{i,x} \), \( i = 1, 2, 3 \) the three Boolean expressions at \( x \): the FBDD is shared, so it can keep track of them separately.

**Step 1:** Inspect \( G_{3,x} \). Recall that the original \( G_3 \) contained \( S_1(a,b), S_2(a,b) \lor S_3(a,b), T(b) \). There are a few cases: (a) \( G_{3,x} \) does not contain \( S_1(a,b) \) at all: then we know \( S_2(a,b) = 0 \) on all paths leading to \( x \). (b) \( G_{3,x} \) contains \( S_1(a,b) \) (a prime implicant). Then on all paths to \( x \) must set \( S_2(a,b) = 1 \). In both (a) and (b) we know how to set \( S_1(a,b) \) according to Equation 3.4. (c) \( G_{3,x} \) contains \( S_1(a,b), S_2(a,b) \): then we know \( S_2(a,b) \) is unset, and continue with step 2.

**Step 2.** At this point we know \( S_2(a,b) \) is unset. Inspect \( G_1 \). The original \( G_1 \) was \( R(a), S_1(a,b) \lor S_2(a,b), S_3(a,b) \). We know \( S_2(a,b) \) is unset, hence the cases are: (a) \( G_{1,x} \) does not contain \( S_2(a,b) \): then we know \( S_3(a,b) = 0 \) on all paths to \( x \). (b) \( G_{1,x} \) contains \( S_2(a,b) \) (a prime implicant). Then on all paths to \( x \), \( S_3(a,b) = 1 \). In cases (a) and (b) we know how to set \( S_1(a,b) \) according to the constraint Equation 3.4. (c) \( G_{1,x} \) contains \( S_2(a,b), S_3(a,b) \). Then we know \( S_3(a,b) \) is also unset, and it means we can read \( S(a,b) \).
Chapter 3. Compiling UCQ

So far we have assumed that neither $G_{1,x}$ nor $G_{3,x}$ are true. Suppose $G_{3,x}$ is true. Let $x$ be a maximal node where $G_3$ becomes true: due to our previous construction when we transformed the FBDD into a shared one, the entire subgraph reachable from $x$ is isolated. First, we consider all variables $S_1, S_2, S_3$ already set at $x$, and update the subgraph under $x$ accordingly. We will need to cope with the variables read within this subgraph. Here, we first rewrite the query $d_1, d_2 = R(x, S_1(x,y)) \lor S_2(x_1,y_1) \lor S_3(x_2,y_2), T(y_2)$. Given our constraint Equation 3.4, this query is equivalent to $R(x, S_1(x,y))$. We simply replace the entire subtree at $x$ with an OBDD for $R(x, S(x,y))$. This completes the proof.

Our hardness result for FBDD is more limited in scope than that for OBDD; in particular it says nothing about non-hierarchical queries. This, however, follows from a very strong result by Bollig&Wegener[10]. They showed that, for arbitrary large $n$, there exists a bipartite graph $G$ s.t. the formula $\Phi = \bigvee_{(i,j) \in G} X_{i} Y_{j}$ has no polynomial size FBDD$^5$. This immediately implies that the query $Q = R(x), S(x,y), T(y)$ is not in UCQ(FBDD), because from any FBDD for $Q$ on the complete, bipartite graph one can obtain and FBDD for $\Phi$ by setting all variables $X_{S(i,j)} = 1$ for $(i,j) \in G$ and setting $X_{S(i,j)} = 0$ for $(i,j) \notin G$. In particular, this implies:

Theorem 3.40. (cf. [10]) If $Q$ is non-hierarchical, then $Q \notin UCQ(FBDD)$.

3.6 Queries and d-DNNFs

d-DNNFs were introduced by Darwiche [30]; a good survey is [32], we review them here briefly. A Negation Normal Form is a rooted DAG, internal nodes are labeled with $\lor$ or $\land$, and leaves are labeled with either a Boolean variable $X$ or its negation $\neg X$. Each node $x$ in an NNF represents a Boolean expression $\Phi_x$, and the NNF is said to represent $\Phi_z$, where $z$ is the root node. A Decomposable NNF, or DNNF, is one where for every $\land$ node, the expressions of its children are over disjoint sets of Boolean variables. A Deterministic DNNF, or d-DNNF is a DNNF where for every $\lor$ node, the expressions of its children are mutually exclusive. Given a d-DNNF one can compute its probability in polynomial time, by applying the rules $P(\Phi_x \land \Phi_y) = P(\Phi_x)P(\Phi_y)$ and $P(\Phi_x \lor \Phi_y) = P(\Phi_x) + P(\Phi_y)$ (and similarly for nodes with out-degree greater than 2); this justifies our interest in d-DNNF. Any FBDD of size $n$ can be converted to an d-DNNF of size $5n$ [32]: for any interior node labeled with variable $X$ in the FBDD, write its formula

$^5$Their graph is the following: fix $n = p^2$ where $p$ is a prime number. Then $G = \{(a+bp,c+dp) \mid c \equiv (a+bd) \ mod \ p\).
as \((\neg X) \land \Phi_y \lor X \land \Phi_z\), where \(y\) and \(z\) are the 0-child and the 1-child: obviously, the \(\lor\) is “deterministic”, and the \(\land\)’s are “decomposable”.

It is open whether d-DNNFs are closed under negation [32, pp. 14]; NNFs are obviously closed under negation, but the d-DNNF impose asymmetric restrictions on \(\land\) and \(\lor\), so by switching them during negation, the resulting NNF is no longer a d-DNNF. For that reason, we extend here d-DNNF’s with \(\neg\)-nodes, and denote the result d-DNNF\(^\neg\): probability computation can still be done in polynomial time on a d-DNNF\(^\neg\).

**Definition 3.41.** UCQ\((dDNNF)\) is the class of queries \(Q\) s.t. for any database \(D\), one can construct a d-DNNF for \(\neg \Phi^D_Q\) in time polynomial in \(|D|\). UCQ\((dDNNF)\) is the class of queries \(Q\) s.t. one can construct a d-DNNF for \(\Phi^D_Q\) in time polynomial in \(|D|\).

\[ UCQ(FBDD) \subseteq UCQ(dDNNF) \subseteq UCQ(dDNNF) \subseteq UCQ(P) \]; the first inclusion is can be shown to be strict.

**Proposition 3.42.** \(q_W \in UCQ(dDNNF) \neq UCQ(FBDD)\).

The significance of this result is the following. This is, to the best of our knowledge, the first example of a “simple” Boolean expression (meaning monotone, and with a polynomial size DNF) that has a polynomial size d-DNNF but not FBDD. The previous separation of FBDD and d-DNNF is based on a result by Bollig and Wegener [11], which we review briefly. Consider a Boolean matrix of variables \(X_{ij}\). Let \(\Phi_1\) denote the formula “there are an even number of 1’s and there is a row consisting only of 1’s”. Let \(\Phi_2\) denote the formula “there are an odd number of 1’s and there is a column consisting only of 1’s”; [11] show that \(\Phi_1 \lor \Phi_2\) does not have a polynomial size FBDD. However, this formula has a polynomial size d-DNNF, because each of \(\Phi_1, \Phi_2\) that have polynomial size OBDDs and \(\Phi_1 \land \Phi_2 \equiv \text{false}\). Note, however, that these formulas are non-monotone and have exponential size DNF’s (they are not in AC\(^0^\)). By contrast, the lineage of \(q_W\) is monotone, has polynomial size DNF, and separates FBDD from d-DNNF.

In the rest of the section, we give a sufficient criteria for a query \(Q = d_1 \land \ldots \land d_m\), to be in UCQ\((dDNNF)\), which is quite interesting because it explains the border between d-DNNF and PTIME in terms of lattice-theoretic concepts. We need to define some lattice theoretic concepts first. Let the CNF lattice of \(Q\) be \((L, \leq)\).

We now describe the construction algorithm. If \(m = 1\) then \(Q\) is a disjunctive query; in this case it must have a separator (assuming FP \(\neq \#P\) (Theorem 3.5)), \(d_1 = \exists w.Q_1\) and we write: \(\neg Q = \bigwedge_{a \in ADom(D)} d_1[a/w]\). The \(\land\) operator is “decomposable”, i.e. its children are independent.
If \( m \geq 2 \), we express \( Q = Q_1 \land Q_2 \), and consider the following derivation for \( \neg Q \), where we write \( \lor^d \) to indicate that a \( \lor \) operation is disjoint:

\[
\neg(Q_1 \land Q_2) = \neg Q_1 \lor \neg Q_2 = \neg Q_1 \lor^d [Q_1 \land \neg Q_2] = \neg Q_1 \lor^d [\neg Q_1 \lor Q_2] = \neg Q_1 \lor^d [\neg Q_1 \land \neg Q_2 \lor^d Q_2] = \neg Q_1 \lor^d [\neg(Q_1 \lor Q_2) \lor^d Q_2] \tag{3.5}
\]

The effect of the decomposition above is that it reduces \( Q \) to three subqueries, namely \( Q_1, Q_2, \) and \( Q_1 \lor Q_2 \), whose CNF lattices are meet-sublattices of \( L \), obtained as follows. Let \( Q = Q_1 \land Q_2 \), where \( Q_1 = d_{l1} \land d_{l2} \land \ldots \) and \( Q_2 = d_{o1} \land d_{o2} \land \ldots \). Denote by \( v_1, v_2, \ldots \) are the co-atoms of this lattice, such that \( v_1, v_2, \ldots \) are the co-atoms corresponding to \( d_{l1}, d_{l2}, \ldots \) and \( u_1, u_2, \ldots \) are the co-atoms for \( d_{o1}, d_{o2}, \ldots \).

- The CNF lattice of \( Q_1 \) is \( \overline{M} \), where \( M = \{v_1, \ldots, v_m\} \).
- The CNF lattice of \( Q_2 \) is \( \overline{K} \), where \( K = \{u_1, \ldots, u_k\} \).
- The CNF lattice of \( Q_1 \lor Q_2 \) is \( \bigwedge_{i,j}(d_{li} \lor d_{oj}) \) is \( \overline{N} \), where \( N = \{v_i \land u_j | i = 1, m; j = 1, k\} \).

Here \( v_i \land u_j \) denotes the lattice-meet, and corresponds to the query-union.

Note that each of the three lattices above, \( \overline{M}, \overline{K}, \overline{N} \) is a strict subset of \( L \).

This justifies the following definition of \( d \)-safe queries, analogous to \( d \)-safe queries Theorem 3.6.

**Definition 3.43.** (1) Let \( Q = Q_1 \land Q_2 \). Then \( Q \) is \( d \)-safe if \( Q_1, Q_2, Q_1 \lor Q_2 \) are all \( d \)-safe. (2) Let \( d = c_1 \lor \ldots \lor c_k \) be a disjunctive query, and let \( d = d_0 \cup d_1 \), where \( d_0 \) contains all components \( c_i \) without variables, and \( d_1 \) contains all components \( c_i \) with at least one variable. Then \( d \) is \( d \)-safe if \( d_1 \) has a separator \( w \) and \( d_1[a/w] \) is \( d \)-safe.

**Theorem 3.44.** If \( Q \) is \( d \)-safe, then it is in UCQ\((dDNNF^-)\).

To illustrate this algorithm, we now show how to construct a \( d \)-DNNF for \( q_W \).

**Proof.** Of Theorem 3.42 : Consider \( q_W = d_1 \land d_2 \land d_3 \) in Figure 3.1.
Denote the three lower points in the lattice as:

\[ \begin{align*}
    d_{12} &= d_1 \lor d_2 \\
    d_{23} &= d_2 \lor d_3 \\
    d_{123} &= d_1 \lor d_2 \lor d_3
\end{align*} \]

We express \( Q_W \) as \( d_1 \land (d_2 \land d_3) \), and using Equation 3.5 get

\[-Q_W = \neg d_1 \lor [\neg (d_1 \lor (d_2 \land d_3)) \lor d_2 \land d_3] \]

\( d_1 \) is hierarchical-read-once, and \( d_2 \land d_3 \) is inversion-free; hence they both have compact d-DNNF. \( d_1 \land (d_2 \land d_3) = d_{12} \) is inversion-free and hence it also admits a compact d-DNNNF. \( \square \)

We prove now that every d-safe query is also safe. Fix a lattice \( L \). Every non-empty subset \( S \subseteq L - \{\hat{1}\} \) corresponds to a query, \( \bigwedge_{u \in S} \lambda(u) \). We define a nondeterministic function \( NE \) that maps a non-empty set \( S \subseteq L - \{\hat{1}\} \) to a set of elements \( NE(S) \subseteq \overline{S} \), as follows. If \( S = \{v\} \) is a singleton set, then \( NE(S) = \{v\} \). Otherwise, partition \( S \) non-deterministically into two disjoint, non-empty sets \( S = M \cup K \), define \( N = \{v \land u \mid v \in M, u \in K\} \), and define \( NE(S) = NE(M) \cup NE(K) \cup NE(N) \). Thus, \( NE(S) \) is non-deterministic, because it depends on our choice for partitioning \( S \). The intuition is the following: in order for the query \( \bigwedge_{u \in S} \lambda(u) \) to be d-safe, all lattice points in \( NE(S) \) must also be d-safe: they are “non-erasable”.

Call an element \( z \in L \) erasable if there exists a non-deterministic choice for \( NE(L^*) \) that does not contain \( z \). The intuition is that, if \( z \) is erasable, then there exists a sequence of applications of the first rule, which avoids computing \( z \); in other words, it “erases” \( z \) from the list of queries in the lattice for which it needs to compute the d-DNNF”, and therefore \( Q_z \) is not required to be \( R_d \) safe. We prove that only queries \( Q_z \) where \( \mu_L(z, \hat{1}) = 0 \) can be erased:

**Lemma 3.45.** If \( z \) is erasable in \( L \), then \( \mu_L(z, \hat{1}) = 0 \).

**Proof.** We prove the following claim, by induction on the size of the set \( S \): if \( z \not\in NE(S) \), \( z \neq \hat{1} \), then \( \mu_{\overline{S}}(z, \hat{1}) = 0 \) (if \( z \not\in \overline{S} \), then we define \( \mu_{\overline{S}}(z, \hat{1}) = 0 \)). The lemma follows by taking \( S = L^* \) (the set of all co-atoms in \( L \)).

If \( S = \{v\} \), then \( NE(S) = \{v\} \) and \( \overline{S} = \{v, \hat{1}\} \): therefore, the claim hold vacuously. Otherwise, let \( S = M \cup K \), and define \( N = \{v \land u \mid v \in M, u \in K\} \). We have \( NE(S) = NE(M) \cup NE(K) \cup NE(N) \). If \( z \not\in NE(S) \), then \( z \not\in NE(M) \), \( z \not\in NE(K) \), and \( z \not\in NE(N) \). By induction hypothesis \( \mu_{\overline{M}}(z, \hat{1}) = \mu_{\overline{K}}(z, \hat{1}) = \mu_{\overline{N}}(z, \hat{1}) = 0 \). Next, we notice that (1) \( \overline{M}, \overline{K}, \overline{N} \subseteq \overline{S} \), (2) \( \overline{S} = \overline{M} \cup \overline{K} \cup \overline{N} \) and (3) \( \overline{M} \cap \overline{K} = \overline{N} \). Then, we apply the definition of the Möbius function
Chapter 3. Compiling UCQ

directly, using a simple inclusion-exclusion formula:

\[
\mu_S(z, \hat{1}) = - \sum_{u \in S, z < u \leq \hat{1}} \mu_S(u, \hat{1}) = - (\sum_{u \in M, z < u \leq \hat{1}} \mu_S(u, \hat{1}) + \sum_{u \in K, z < u \leq \hat{1}} \mu_S(u, \hat{1})) = \mu_M(z, \hat{1}) + \mu_K(z, \hat{1}) - \mu_N(z, \hat{1}) = -0 - 0 + 0 = 0
\]

The lemma implies immediately:

**Proposition 3.46.** For any UCQ query \( Q \), if \( Q \) is \( d \)-safe, then it is safe. The converse does not hold in general: query \( q_9 \) is safe but is not \( d \)-safe.

It is conjectured that \( q_9 \notin UCQ(dDNNF^-) \). Note that the proposition only states that \( q_9 \) is not \( d \)-safe, but it is not known whether \( d \)-safety is a complete characterization of \( UCQ(dDNNF^-) \).

**Proof.** We prove the statement by induction on \( Q \). We show only the key induction step, which is when \( Q = \bigwedge_i d_i \), and \( L \) is its CNF lattice. Let \( Z \subseteq L \) denote the nodes corresponding to \( d \)-unsafe queries: if \( Q \) is \( d \)-safe, then all elements in \( Z \) are erasable. This implies that \( \forall z \in Z, \mu(z, \hat{1}) = 0 \). Hence, we can apply Möbius’ inversion formula to the lattice \( L \), and refer only to queries that are \( d \)-safe; by induction hypothesis, these queries are also safe, implying that \( Q \) is safe.

We show that \( q_9 \) is safe, but is not \( d \)-safe. We will denote the lattice points with query \( d_i \lor d_j \lor \ldots \) in Figure 3.1 as \( d_{ij}... \). The query at \( \hat{0} \) is the only hard query (since it is equivalent to \( h_3 \)), and \( \mu(\hat{0}, \hat{1}) = 0 \). On the other hand, we prove that \( \hat{0} \) cannot be erased. Indeed, the co-atoms of the lattice are \( L^* = \{d_1, d_2, d_3, d_4\} \); given the symmetry of \( d_1, d_2, d_3 \), there are only three ways to partition the co-atoms into two disjoint sets \( L^* = M \cup K \):

- \( M = \{d_1, d_2, d_3\}, K = \{d_4\} \). In this case the lattice \( \overline{M} \) is \( \{\hat{0}, d_{12}, d_{13}, d_{23}, d_1, d_2, d_3, \hat{1}\} \), and \( \mu_{\overline{M}}(\hat{0}, \hat{1}) = -1 \), proving that this query is unsafe, and, therefore, \( d \)-unsafe.
- \( M = \{d_1, d_2\}, K = \{d_3, d_4\} \). In this case the lattice \( \overline{K} \) is \( \{\hat{0}, d_3, d_4, \hat{1}\} \), and has \( \mu_{\overline{K}}(\hat{0}, \hat{1}) = 1 \), hence, by the same argument, is \( d \)-unsafe.
- \( M = \{d_1\}, K = \{d_2, d_3, d_4\} \). Here, too, \( \overline{K} = \{\hat{0}, d_{23}, d_2, d_3, d_4, \hat{1}\} \), and \( \mu_{\overline{K}}(\hat{0}, \hat{1}) = 1 \).

\( \square \)
3.7 Results on Non-Uniform Classes

In this section we look at the non-uniform compact classes for different targets $T$.

**Definition 3.47.** For target $T \in \{\text{OBDD}, \text{FBDD}, \text{d-DNNF}\}$, denote by $UCQ^n(T)$ the class of all queries $Q \in UCQ$ s.t. $\Phi^Q_D$ has a compilation in $T$ of size polynomial in $|D|$ for all $D$.

Note that in case of RO, a formula is either RO or not RO. Furthermore, thanks to the result due to Gurvich [48], this can be done in PTIME for monotone formulas that form the lineages of $UCQ$. On the other hand, a formula may have a compact OBDD, FBDD, or d-DNNF, but our algorithm may not be able to construct it. Hence $UCQ^n(T) \supseteq UCQ(T)$.

For OBDD though, it follows from the results of Section 3.4, that

**Proposition 3.48.** $UCQ^n(\text{OBDD}) = UCQ(\text{OBDD})$

The proof follows from Theorem 3.24, which implies that queries not in $UCQ(\text{OBDD})$ do not have a polynomial size OBDD, i.e., $UCQ - UCQ(\text{OBDD}) \subseteq UCQ - UCQ^n(\text{OBDD})$. Hence $UCQ^n(\text{OBDD}) \subseteq UCQ(\text{OBDD})$, which means the two sets must be equal.

We do not have a full characterization for FBDD, d-DNNF, so we don’t know if the same is true for them. The main separation results though still hold for non-uniform classes as well.

**Proposition 3.49.**

\[
UCQ^n(\text{OBDD}) \subset UCQ^n(\text{FBDD}) \\
UCQ^n(\text{FBDD}) \subset UCQ^n(\text{dDNNF}) \\
UCQ^n(\text{FBDD}) \subset UCQ(\text{P})
\]

**Proof.** We can construct an FBDD for $q_V$ in PTIME(Theorem 3.29), but it doesn’t always admit a compact OBDD(Theorem 3.24). This proves the first result. Similarly one can construct a d-DNNF for $q_V$ in PTIME(Theorem 3.42), but it admits no polynomial size FBDD(Theorem 3.33). This proves the last two separations.

3.8 Conclusion

We have studied the problem of compiling the query lineage into compact representations. We considered four compilation targets: read-once, OBDD, FBDD, and d-DNNF. We showed that over the query language of unions of conjunctive queries, these four classes form a strict hierarchy.
For the first two classes we gave a complete characterization based on the query’s syntax. For the last two classes we gave sufficient characterizations.

Our two main separation results, between $\text{UCQ(OBDD)}$ and $\text{UCQ(FBDD)}$, and between $\text{UCQ(FBDD)}$ and $\text{UCQ(dDNNF)}$, are the first examples of “simple” Boolean expressions (meaning: monontone, and with polynomial size DNFs) that separate those two classes.

We leave three open problems: complete characterizations of $\text{FBDD}$ and d-DNNF, and separation of the latter from PTIME. Also, as future work, it would be interesting to investigate compact representations of lineages in other semirings described in [46].
QUERY COMPILATION AND BOUNDED TREEWIDTH

4.1 Introduction

In this chapter we study the connection between the treewidth of a Boolean function, and the size of an OBDD for that function. Our main motivation comes from query evaluation on probabilistic database, which, at its core, consists of computing the probability of a Boolean function (namely, the query’s lineage).

The tree-width of a graph is a measure of how tree-like a graph is. A tree has a tree-width equal to one, while a complete graph, or complete bipartite graph has a large tree-width. Many graph problems that are hard on arbitrary graphs, become tractable over graphs with a bounded tree-width [45, 9]. This also holds for probabilistic inference problem in graphical models. In fact, all exact probabilistic inference algorithms on graphical models known in the literature run in time that is exponential in the tree-width [72, 24, 31], so, for all practical purposes, bounded tree-width characterizes tractability in graphical models.

The probabilistic inference for Boolean functions is a special case of inference in graphical models, and has quite distinct characteristics. The problem asks for the probability that a Boolean expression is true, if each of its variables is set to true independently, with a given probability. It generalizes model counting, and is #P-hard, even for positive 2CNF expressions [91]. The extent to which bounded tree-width can be used in this context is less well understood. In the primal graph of a CNF expression nodes represent variables, and edges represent pairs of variables that co-occur in a clause; it follows (from algorithms on graphical models) that, if the tree-width of the primal graph is bounded, then the probability of the Boolean function can be computed in PTIME. But this notion of tree-width leaves out some very simple tractable cases, like a single clause, $X_1 \lor \ldots \lor X_n$, whose probability can be computed in linear time$^1$ yet its primal graph is a complete graph and has tree-width $n$. Fischer et al. proved a stronger connection [38].

They defined the incidence graph of a CNF expression to be the bipartite graph with one node

\[ 1 - (1 - \Pr(X_1)) \cdot \ldots \cdot (1 - \Pr(X_n)). \]

\[ ^1 \]
Chapter 4. Query Compilation and Bounded Treewidth

Figure 4.1: The lineage of the query \( q_{rs} = R(x, y), S(y, z) \) on a small database instance; \( s_{ij} \) denotes the tuple \( S(i, j) \) and similarly for the others. On any database instance, the lineage of \( q_{rs} \) is a read-once expression \([68]\), hence, the expression tree-width is 1. Its OBDD width is also 1, hence we show that its expression path-width is \( \leq 5 \); on the other hand, the tree width of the incidence graph for either CNF or DNF is unbounded. We also illustrate the query plan that we used to compute the lineage.

Figure 4.2: The lineage of the query \( q_{rsst} = R(x_1), S(x_1, y_1), S(x_2, y_2), T(x_2) \) on a small database. Notice that each variable occurs only once, and as a consequence the expression is a DAG. On any database instance, the lineage of \( q_{rsst} \) has an OBDD of width \( \leq 2^4 = 16 \) \([57]\), hence the expression path-width is \( \leq 80 \). We also show the query plan that we used to compute the lineage.

for each variable and one node for each clause, and edges connect the variables with the clauses where they appear. They proved that, if the tree-width of the incidence graph is bounded, then the probability of the Boolean function can be computed in PTIME\(^2\). The dual result also holds: if the tree-width of the incidence graph defined for the DNF is bounded, then the probability can also be computed in PTIME. However, this notion of tree-width is also insufficient, because it leaves out a large class of Boolean expressions. A read-once Boolean expressions \([48, 44]\) is an expression using connectives \( \land, \lor, \neg \) where each Boolean variable may occur only once. The probability of a read-once Boolean expression can be computed in linear time in its size, because here the laws of independence hold, e.g. \( \Pr (E_1 \land E_2) = \Pr (E_1) \cdot \Pr (E_2) \); an example of a read-once Boolean expression is in Figure 4.1. However, the incidence graphs of both the CNF and the DNF representation of a read-once expression can have arbitrarily large tree-width.

\(^2\)They only proved that model counting can be solved in PTIME, but their algorithm generalizes immediately to probabilistic inference.
Since read-once expressions are of particular interest in probabilistic databases [68, 84, 81], the tree-width of the incidence graph is too weak for identifying tractable cases in these applications.

A concept that captures many tractable instances of model counting and probability computation for Boolean functions are Ordered Binary Decision Diagrams (OBDDs) [96]. An OBDD is a special case of a branching program: it is a rooted DAG where each internal node is labeled with a Boolean variable and has two outgoing edges, labeled 0 and 1, and has two leaves, labeled 0 and 1 respectively; furthermore, it is required that every path from the root to a leaf visits the Boolean variables in the same order. Given an OBDD for a Boolean function, one can compute its probability in linear time in the size of the OBDD. Thus, Boolean functions that have an OBDD of polynomial size are tractable. This class includes all read-once expressions: each read-once expression has an OBDD of width 1 (meaning that each variable occurs at most once, hence the size is linear). A connection between tree-width and OBDD was established by Huang&Darwiche [51] and Ferrara et. al. [37], who proved that if the primal graph of a CNF has a bounded tree-width, then the Boolean function defined by the CNF has an OBDD of polynomial size.

Our contributions In this chapter we introduce a more powerful notion of tree-width for Boolean functions, which captures a more robust class of tractable functions. An expression DAG is a rooted DAG whose internal nodes are labeled with $\land$, $\lor$, $\neg$ and whose leaves are labeled with the Boolean variables, such that each variable occurs at most once. The expression treewidth of a Boolean function is the smallest treewidth of any expression DAG that represents it. This notion generalizes previous notions of tree-width: both the CNF and the DNF representations of a Boolean function correspond to some expression DAG, and if the incidence graph has bounded tree-width, then the expression tree-width is bounded too. It also naturally includes read-once expressions: every read-once Boolean expression has an expression treewidth 1, because it is given by an expression DAG that is a tree. Similarly, we define the expression pathwidth of a Boolean function as the smallest pathwidth of any expression representing it. For example, consider the two Boolean expressions shown in Figure 4.1 and Figure 4.2, which represent the lineages of two conjunctive queries, $q_{rs}$ and $q_{rsst}$ respectively. It is known [68] that the lineage of $q_{rs}$ is always a read-once expression, for any input database; hence its expression tree-width is 1. It is far less obvious (but we will show below) that the lineage of $q_{rs}$ on any database instance has an expression path-width $\leq 5$, and, similarly, the lineage of $q_{rsst}$ has an expression path-width $\leq 80$.

In this chapter we prove several results connecting expression tree- (or path-) width to OBDD, in three different settings: for unrestricted Boolean functions, for Boolean functions that are
FIGURE 4.3: Primal and incidence graphs for the read-once Boolean expression $F_{mnp} = \bigvee_{i=1,n}(\bigwedge_{j=1,m} r_{ij}) \wedge \bigvee_{k=1,p} s_{jk}$, for $m = 4, n = 1, p = 4$. This expression is precisely the lineage of $R(x, y), S(y, z)$ from Figure 4.1 on the database $R = \{(i, j) \mid i \in [m], j \in [n]\}$, $S = \{(j, k) \mid j \in [n], k \in [p]\}$.

FIGURE 4.4: Relationship between tractability w.r.t RO, OBDD and the new notions of expression path-width and treewidth. RO = read-once, EPWD = bounded expression path-width, ETWD = bounded expression tree-width, OBDD = polynomial-size OBDD.

lineages of Unions of Conjunctive Queries with $\not\equiv (UCQ^\#)$, and for Unions of Conjunctive Queries (UCQ).

**Results relating expression-width to OBDD** Our first result is the following: if the expression pathwidth of a Boolean function is $< k$, then there exists an OBDD for it whose width is $\leq 2^{(k+1)^2} k + 1$; in particular, the size of the OBDD is linear in the number of Boolean variables. We also show that, if the expression tree-width is bounded, then there exists an OBDD whose size is polynomial in the number of variables. Note that the former result does not imply the latter, unlike in prior work by Ferrara et al. [37]. They prove that if the primal graph has pathwidth $\leq k$, then there exists an OBDD of size $O(n2^k)$: in any graph of tree-width $t$, the path...
width is bounded by \( k = O(t \log n) \), hence, if the tree-width is bounded, then the OBDD has a polynomial size because \( O(n^{2t \log n}) = n^{O(1)} \). In our setting, however, the path-width occurs in a double exponent, preventing us from applying the same argument. Instead, we prove both results on expression path-width and tree-width together, using a common technical lemma.

We also show the following converse: if a Boolean function has an OBDD of width \( \leq w \), then its expression pathwidth is \( \leq 5w \). In other words, the notions of bounded expression pathwidth and bounded OBDD width coincide. It is known that every read-once Boolean expression has an OBDD of width 1: our result implies that any read-once expression has an expression path-width \( \leq 5 \). For another illustration, it is known from \cite{57} that the lineage of \( q_{rssr} \) in Figure 4.2 on any database has an OBDD of width \( \leq 2^4 = 16 \): therefore its expression path-width is \( \leq 80 \). Our results are summarized in the first diagram of Figure 4.4.

**Application to Query Compilation** While our main motivation was to apply these results to query compilation, it turned out that query compilation actually helped us better understand the relationships between expression path/tree width and OBDDs. We are interested in the following problem: for a fixed query, determine whether, for any input database, the Boolean function representing the lineage of this query on some arbitrary database has a bounded expression path/tree width, or a polynomial size OBDD. In prior work \cite{57} we have shown that a Union of Conjunctive Queries, UCQ, has a polynomial size OBDD iff it is inversion-free, here denoted IF; we also proved that every query IF has constant width OBDD. It follows that, when restricted to lineages of UCQ queries, these three classes collapse: bounded expression path/tree width, and polynomial size OBDD.

Next, we looked at Unions of Conjunctive Queries extended with \( \neq \), UCQ\( ^\neq \), and found that their lineage expressions have more interesting properties. First we proved that a query in UCQ\( ^\neq \) has a polynomial-size OBDD iff, after removing \( \neq \), the query is inversion free. Thus, IF\( ^\neq \) characterizes the class of queries with polynomial size OBDD. However, unlike IF, which have constant-width OBDD, these queries have polynomial-width OBDD, and therefore use the full power of OBDD. In fact, we prove something stronger: we show that a particular query in IF\( ^\neq \), \( q_{nr} \), has no bounded expression treewidth. This is a result of interest beyond query compilation, because it separates Boolean functions of bounded expression treewidth from those with a polynomial size OBDD: by giving this separation through a query, we obtain a very simple formulation of the result. Moving lower in the hierarchy, we seek to separate bounded expression pathwidth and bounded expression treewidth. We give a Boolean function, \( bt_m \), that separates these two classes, but we could not find a query in UCQ\( ^\neq \) whose lineage separates them. We conjecture that, when restricted to query lineages, these classes collapse. We further describe a
syntactic class of queries, $H^\neq$, whose lineage have bounded expression pathwidth.

**Discussion** To the best of our knowledge, ours are the first results connecting the tree(path)-
width of the expression DAG of a Boolean function to the size of the OBDD. Related is a very
general result by Courcelle et. al. [23] that implies that probabilistic inference is tractable
given a tree decomposition of an expression DAG, and a more efficient algorithm for the same
problem [54], with time complexity $16^t|G|$, where $G$ is the expression DAG and $t$ is its tree-width.

Our expression tree-width inherits a general weakness of tree-width based approaches and of
OBDD: it is NP-hard to compute the tree-width of a general graph, and similarly it is NP-hard
to compute an optimal OBDD. To this, expression tree-width adds another layer of difficulty,
since one also needs to find the expression DAG that minimizes the tree width. In fact, we
do not know if computing the expression tree-width of an arbitrary Boolean function is even
decidable. However, when the Boolean function is restricted to the lineage of a $UCQ^\neq$ query,
then in all tractable cases we also give polynomial time algorithms for computing the polynomial
size OBDD.

The class $UCQ^\neq$ has not been studied before in the context of probabilistic databases.
Olteanu and Huang study the join-free fragment of $CQ^<$, and characterize all queries that
have a polynomial size OBDD[69]. The characterization of queries in $CQ^<$ or in $UCQ^<$ with
polynomial size OBDD is open.

The chapter is organized as follows. In Section 4.2 we give our results connecting expression
treewidth/pathwidth with OBDD; in Section 4.3 we give the results on $UCQ^\neq$. The proofs
for Section 4.2 are in Section 4.4, while the proofs for Section 4.3 are in Section 4.5. We conclude
in Section 4.6.

## 4.2 Results on Treewidth and OBDD

In this section, we will define formally the *expression treewidth* and give the results relating it to
OBDD size/width.

### 4.2.1 Treewidth

A graph $G$ is a pair $(V(G), E(G))$, where $E(G) \subseteq V(G) \times V(G)$. We call $V(G)$ the vertices and
$E(G)$ the edges in the graph $G$. A tree-decomposition of $G$ is a tree $T = (V(T), G(T))$, where $V(T) = \bar{Y} = \{Y_1, Y_2, \ldots\}$ is a family of subsets of $V(G)$ s.t.

1. $\bigcup_i Y_i = V(G)$

2. for every edge $(u, v) \in E(G)$, there exists a $Y_i$ s.t. $u \in Y_i$ and $v \in Y_i$
3. For any \( v \in V(G) \), the set \( \{ Y_i \mid v \in Y_i \} \) forms a connected component of \( T \).

The width of the tree-decomposition is defined as \( \max_i |Y_i| - 1 \). The treewidth of \( G \), \( \text{twd}(G) \), is defined as the minimum width over all possible tree-decompositions of \( G \). Note that all trees have treewidth 1. Analogous to treewidth, the pathwidth, \( \text{pwd}(G) \), is the minimum width over all path-decompositions, where a path-decomposition is defined just like a tree-decomposition except that \( T \) is required to be a path instead of a tree. If the graph has \( n \) nodes, then \( \text{twd}(G) \leq \text{pwd}(G) = O(\text{twd}(G) \cdot \log n) \). The last inequality is nearly tight: if \( G \) is a complete binary tree with \( 2^{k+1} - 1 \) nodes then \( \text{pwd}(G) = \lceil \frac{k}{2} \rceil \) [83]. Many problems that are intractable on general graphs become tractable over graphs of bounded treewidth [45, 9]. The problem of determining whether treewidth \( \leq k \) was shown to be NP-complete in [4]. For fixed \( k \) though, Bodlaender [8] showed that one can construct a tree-decomposition of width \( k \) in linear time.

### 4.2.2 Expression Treewidth

Let \( F \) be a Boolean function over Boolean variables \( \bar{X} = X_1, X_2, \ldots, X_n \). The problem of interest to us is:

**Definition 4.1.** The probability inference problem is: Given \( F \) and probability assignments \( p_i \) to each variable \( X_i \), compute the probability of \( F \), \( \Pr(F) \), where

\[
\Pr(F) = \sum_{z: \bar{X} \rightarrow \{0, 1\}, F(z)=1} \prod_{z(X_i)=0} (1 - p_i) \prod_{z(X_i)=1} p_i \tag{4.1}
\]

Inference is \#P-complete, even for positive 2CNF [91].

To define the expression treewidth, we make the usual distinctions between a Boolean function, and an expression using \( \land, \lor, \lnot \) representing that function.

**Definition 4.2.** An expression \( E \) over variables \( \bar{X} \) is defined by the following grammar:

\[
E ::= X_i \mid \lnot E \mid E_1 \lor \ldots \lor E_m \mid E_1 \land \ldots \land E_p \tag{4.2}
\]

CNF and DNF are particular examples of expressions.

**Definition 4.3.** An expression DAG \( G \) is a rooted DAG whose internal nodes are labeled with \( \land, \lor, \lnot \), and whose leaves are labeled with Boolean variables, s.t. each variable occurs at most once. The graph represents the Boolean function given by the expression obtained by unfolding the DAG into a tree.

A simple illustration of an expression DAG is in Figure 4.2. Finally, the expression treewidth is:
Definition 4.4. The expression treewidth of a Boolean function $F$, $etwd(F)$, is defined as $\min twd(G)$ over all possible expression DAGs $G$ representing $F$. Similarly, the expression pathwidth, $epwd(F)$, is defined as $\min pwd(G)$.

Our definition is robust w.r.t. the choice of operators used in the grammar Equation 4.2, in the following sense. Consider a different grammar:

$$E ::= X_1 | \neg E_1 | E_1 \otimes E_2,$$

where $\otimes \in \mathbb{B}^\times \mathbb{B}$

Here $\mathbb{B}^\times \mathbb{B}$ is the set of all $2^4$ Boolean operators over two variables. Define $etwd_*(F)$ to be $\min twd(G)$ where $G$ ranges over all possible DAGs representing $F$ using the extended grammar. We prove:

Proposition 4.5. For any Boolean function $F$ we have $etwd_*(F) \leq etwd(F) \leq etwd_*(F) + 2$.

4.2.3 Background: Some Tractable Functions

We explain now the connection between bounded expression treewidth and some previously known tractable functions, and start with read-once functions.

Definition 4.6. [48, 44] A Boolean expression is called read-once if every variable occurs at most once. A Boolean function is called read-once if it admits a read-once expression.

It is known that the inference problem can be solved in linear time for a read-once expression. They are of particular interest in probabilistic databases, where an important class of queries is known to have lineage expressions that are read-once [68, 57]. Obviously, if $F$ is a read-once function then $etwd(F) = 1$, because the read-once expression is a tree.

Next, assume that $F$ is given as a DNF expression$^3$, $\bigvee_j T_j$, where each $T_j = \bigwedge_i L_i$, and each literal $L_i$ is either a Boolean variable or its negation. We assimilate $T_j$ with the set of Boolean variables occurring in $T_j$. The primal graph $G^P$, and the incidence graph of $F$ are defined as

$$G^P = (\bar{X}, \{(X_i, X_j) | \exists T_k. X_i \in T_k \land X_j \in T_k\})$$

$$G^I = (\bar{T}, \bar{X}, \{(T_k, X_i) | X_i \in T_k\})$$

In the primal graph the nodes are variables and the edges are pairs of co-occurring variables. The incidence graph is bipartite; its nodes are the conjuncts and the variables, and its edges connected each conjunct with the variables it contains. The two graphs are of interest in our setting since bounded treewidth in either case leads to tractable inference.

$^3$In most of the literature, the CNF form is used. We prefer DNF because it arises naturally in probabilistic databases.
Proposition 4.7. [38] The inference problem for $F$ can be solved in time $\min \left(2^{\text{twd}(G^P)}, 4^{\text{twd}(G^I)}\right) \cdot O(n)$.

The result for primal graph follows from the classical complexity results of inference over Markov Networks [72, 24]. The result for incidence graph is due to [38]. The relationship is [63, pp. 327-328]:

Proposition 4.8. Let $m = \max_j |T_j|$, then $\text{twd}(G^I) \leq \text{twd}(G^P) + 1$ and $\text{twd}(G^P) \leq (\text{twd}(G^I) + 1) \cdot m - 1$.

Thus, a bounded treewidth of the primal graph always implies a bounded treewidth of the incidence graph; the converse holds too when the size of the conjuncts is bounded: the latter is indeed the case in query compilation.

A bounded treewidth of the incidence graph also implies a bounded expression treewidth. More precisely, for any Boolean function $F$, $\text{etwd}(F) \leq \text{twd}(G^I) + 1$. Thus, tractability results for bounded expression treewidth are at least as strong as Theorem 4.7. However, the converse fails. In particular, read-once Boolean functions have, in general, incidence graphs with large treewidth, while their expression treewidth is $= 1$. Thus, bounded treewidth is strictly stronger.

Example 4.1. Consider the Boolean function $F_{mnp} = \bigvee_{j=1}^{m} [\bigvee_{i=1}^{n} r_{ij}] \land (\bigvee_{k=1}^{p} s_{jk})$. This Boolean function occurs naturally in query compilation, since it is the lineage of the query $R(x,y), S(y,z)$. Written in DNF it becomes $\bigvee_{i=1}^{m} \bigvee_{j=1}^{n} \bigvee_{k=1}^{p} r_{ij} \land s_{jk}$, thus, in the primal graph all variables $r_{1j}, \ldots, r_{mj}$ are connected to all variables $s_{j1}, \ldots, s_{jp}$: Figure 4.3 shows the primal graph (on the right) and also the incidence graph (left) for the case $m = p = 4, n = 1$.

More generally, if $n = 1$ and $m = p$ are arbitrary, then the treewidth of primal graph is $m$, and that of the incidence graph is $\geq (m + 1)/2$. While it happens that for $n = 1$ the CNF expression has an incidence graph with treewidth 1, as we increase $n$ both CNF and DNF incidence graphs have large treewidth. On the other hand, $\text{etwd}(F_{mnp}) = 1$.

4.2.4 Ordered Binary Decision Diagrams

OBDD were introduced by Bryant [13] and studied extensively in model checking and knowledge representation. A good survey can be found in [96]; we give here a quick overview. A Binary Decision Diagram, BDD, is a rooted DAG with two kinds of nodes. A sink node or output node is a node without any outgoing edges, which is labeled either 0 or 1. An inner node is labeled with a Boolean variable $X_i$ and has two outgoing edges, labeled 0 and 1 respectively. Every node $u$ uniquely defines a Boolean function as follows: $F_u = \text{false}$ and $F_u = \text{true}$ for a sink node labeled 0 or 1 respectively, and $F_u = \neg X_i \land F_{u0} \lor X_i \land F_{u1}$ for an inner node labeled with $X_i$ and
with successors \( u_0, u_1 \) respectively. The BDD represents a Boolean function \( F \equiv F_u \) where \( u \) is the root of the BDD. An Ordered BDD, denoted OBDD, is such that there exists a total order \( \Pi \) on the set of variables, and on any path from the root to a sink every variable appears at most once and in the order \( \Pi \) (variables may be skipped). One also writes \( \text{OBDD}_\Pi \), to emphasize that the order is \( \Pi \). Given \( 1 \leq m \leq p \leq n \), denote \( \Pi(m : p) = (\Pi(m), \ldots, \Pi(p)) \); given \( \bar{a} \in \{0, 1\}^m \) we denote \( F|_{\Pi(1:p)=\bar{a}} \) the Boolean function obtained from \( F \) by substituting the first \( m \) variables in \( \Pi \) with the values \( \bar{a} \).

The size of an OBDD is the number of nodes, and the width at level \( m, m \leq n \) is the number of distinct nodes labeled with the variable \( X_{m+1} \). An upper bound on the width of \( \text{OBDD}_\Pi \) at level \( m \) is given by the number of subfunctions that result after substituting the first \( m \) variables, i.e. \( |\{F|_{\Pi(1:m)=\bar{b}} \mid \bar{b} \in \{0, 1\}^m\}| \). The width of an OBDD is the maximum width at any level. Obviously, the size of an OBDD of width \( w \) with \( n \) variables is \( \leq n \cdot w \).

A shared BDD for a set of function \( \bar{F} = \{F_1, F_2, \ldots, F_m\} \) is defined analogously, except it computes all the functions simultaneously. The sink nodes are labeled with \( \{0, 1\}^m \) and every node \( u \) represents \( m \) subfunctions \( \{F_{u1}, F_{u2}, \ldots, F_{um}\} \), where \( F_{ui} \) can be obtained by applying the assignments until this node to \( F_i \). From a shared OBDD for \( \bar{F} \) one can construct an OBDD for any Boolean function over \( \bar{F} \) of no larger size. This is often used in OBDD synthesis [96], where, instead of computing the OBDD of, say \( F_1 \land F_2 \lor F_3 \), one computes the shared OBDD of \( \{F_1, F_2, F_3\} \).

### 4.2.5 Results on Expression-width and OBDD

We present here the results relating expression-width parameters to OBDD width/size. We define four sets of Boolean functions:

**Definition 4.9.**

\[
\begin{align*}
RO &= \{F \mid F \text{ is read-once}\} \\
\text{EPWD}(k) &= \{F \mid \text{epwd}(F) < k\} \\
\text{ETWD}(k) &= \{F \mid \text{etwd}(F) < k\} \\
\text{OBDD}(w) &= \{F \mid F \text{ has an OBDD of width } \leq w\}
\end{align*}
\]

The size of any OBDD in \( \text{OBDD}(w) \) is \( \leq w \cdot n \); if \( w = O(1) \) then the size of the OBDD is linear, but we also allow \( w = w(n) \) to be a polynomial in \( n \), and in that case the size is polynomial.
Our results are:

\[
RO \subseteq EPWD(O(1)) \equiv OBDD(O(1)) \subseteq \\
\subseteq ETWD(O(1)) \subseteq OBDD(n^{O(1)})
\] (4.4)

We start by describing the containment results. Let \( F \) be a boolean function over \( \bar{X} = X_1, X_2, \ldots, X_n \). Let \( G = (V(G), E(G)) \) be an expression DAG for \( F \), and let \( T = (V(T), E(T)) \) be a tree-decomposition of \( G \). We call \( T \) an expression tree-decomposition of \( F \). Our first, and main result, shows that we can derive a “good” variable order \( \Pi \) from \( T \), such that \( OBDD_{\Pi} \) has a width bounded by the width of \( T \). We will describe now how to construct \( \Pi \). We need to introduce some notations.

Recall that each node \( Y \in V(T) \) is a set of nodes from \( V(G) \), which, in turn, are labeled with a variable \( X_i \) or with an operator \( \land, \lor, \neg \). Denote \( \text{Var}(Y) \) the set of variables \( X_i \) occurring in \( Y \): recall that, for any variable \( X_i \), the set \( \{ Y \mid X_i \in \text{Var}(Y) \} \subseteq V(T) \) is connected. Denote \( \text{Var}(V) = \bigcup_{Y \in V} \text{Var}(Y) \) for a subset \( V \subseteq V(T) \). We say that \( Y \) splits the tree \( T \) into \( V_1, V_2 \) if \( V_1 \cup V_2 = V(T), V_1 \cap V_2 = \{ Y \} \) and every path from \( V_1 - \{ Y \} \) to \( V_2 - \{ Y \} \) goes through \( Y \).

**Definition 4.10.** Let \( T \) be an expression tree decomposition of \( F \), \( \Pi \) be permutation of the variables \( \bar{X} \), and \( m \leq n \). We say that \( \Pi(1 : m) \) is compatible with some tree node \( Y \in V(T) \), if \( Y \) splits the tree into \( V_1, V_2 \) such\( \Pi(1 : m) \subseteq \text{Var}(V_1) \) and \( \Pi(m + 1 : n) \subseteq \text{Var}(V_2) \).

The following is the key technical lemma for our main result; we prove the lemma in Section 4.4.

**Lemma 4.11.** If \( T \) is an expression path-decomposition of \( F \), \( \Pi \) is a permutation of its variables, \( \Pi(1 : m) \) is compatible with \( Y \) for some \( Y \in V(T) \), and \( k = |Y| \), then:

\[
\left| \{ F|_{\Pi(1:m)=\bar{b}} \mid \bar{b} \in \{0,1\}^m \} \right| \leq 2^{(k+1) \times 2^{k+1}}
\] (4.5)

Thus, if we have a tree decomposition of width < \( k \) and \( \Pi(1 : m) \) is compatible with some node \( Y \), then the width of \( OBDD_{\Pi} \) at depth \( m \) is bounded by the lemma. The bound in the lemma is almost tight, even for monotonic Boolean functions. To see this, let \( m = 2^k \), consider \( n = m + k \) variables \( X_1, \ldots, X_m, Z_1, \ldots, Z_k \), and let \( F = \bigvee_i \left( X_i \land \bigwedge_{j \in s_i} Z_j \right) \), where \( s_1, \ldots, s_m \) represent all \( m \) subsets of \( [k] \). Consider the path decomposition \( T \) with \( m \) nodes, \( Y_i = \{ X_i, \land, \lor \} \cup \bar{Z} \), where \( \land \) denotes the \( i \)th conjunct, and \( \lor \) represents the root node in the expression DAG of \( F \): the width is \( k + 2 \). Let \( \Pi \) be the permutation \( X_1, \ldots, X_m, Z_1, \ldots, Z_k \). Then \( \Pi(1 : m) \) is compatible with any node \( Y_i \) in the path \( T \) (by considering the split \( V_1 = V(T) \) and \( V_2 = \{ Y_i \} \)).
yet the set \( \{ F_{\pi(1:m)} = b \mid b \in \{0,1\}^m \} \) contains all monotonic Boolean function in the variables \( \bar{Z} \): the number of such functions is the Dedekind number, \( M(k) \), which is super-exponential, \( M(k) \geq 2^{\binom{k}{2}} \). It is straightforward to extend the example to a non-monotonic Boolean function \( F \), where the number of functions \( F_{\pi(1:m)} = b \) is equal to \( 2^b \).

Ideally, we would like to find a permutation \( \pi \) such that each of its prefix \( \pi(1:m) \) is compatible with some tree node \( Y \): in that case we have a bound on the width of OBDD\( _R \). This is possible if \( T \) is a path; if it is a tree, we can still use the lemma and obtain a polynomial width.

A “good” permutation \( \pi^R \) is defined by any orientation \( T^R \) of \( T \), which is obtained by designating a node \( R \in V(T) \) as the root node. Thus, \( T^R \) is a directed tree, and each \( Y_i \in V(T) \) has a unique parent (except the root \( R \)) and a set of children \( Y_{i_1}, Y_{i_2}, \ldots \). We denote \( T_i \) the subtree rooted at \( Y_i \).

Consider the following in-order traversal of the nodes \( V(T) \), defined recursively, for each subtree. Assuming \( Y_i \) has \( c \) children, we order them such that \( |Var(T_{i_1})| \geq |Var(T_{i_2})| \geq \ldots \geq |Var(T_{i_c})| \); then we traverse \( T_i \) in the order \( T_{i_1}, Y_i, T_{i_2}, T_{i_3}, \ldots, T_{i_c}, \) where each \( T_{i_j} \) is traversed recursively. This defines a total order of the nodes \( V(T) \): \( Y_1, Y_2, \ldots, Y_N \). For each \( i \), consider the set of variables \( X_j \) first encountered at \( Y_i \) during this traversal: \( FVar(Y_i) = Var(Y_i) - \bigcup_{j<i} Var(Y_j) \). Let \( \pi_i \) be an arbitrary permutation of \( FVar(Y_i) \). Then, we define the permutation \( \pi^R = \pi_1, \ldots, \pi_N \).

**Corollary 4.12.** If \( T \) is an expression path decomposition of \( F \) of width \( < k \), then for any node \( R \in V(T) \), \( OBDD_{\pi^R}(F) \) has width at most \( 2^{(k+1)2^k+1} \).

This implies \( EPWD(k) \subseteq OBDD(2^{(k+1)2^k+1}) \).

**Proof.** Notice that, when \( T \) is a path, then the tree traversal \( Y_1, Y_2, \ldots, Y_N \), is simply a traversal of the path, from left to right or right to left (depending on the choice of \( R \)). Thus, \( \pi^R \) lists the variables in the order in which they are first encountered on this path. For any prefix \( \pi^R(1:m) \), let \( Y_j \in V(T) \) be the first node that contains the variable \( \pi^R(m) \), i.e. \( \pi^R(m) \in FVar(Y_j) \). We claim that \( \pi^R(1:m) \) is compatible with \( Y_j \): indeed, \( Y_j \) splits the path into \( V_1 = \{ Y_1, \ldots, Y_{j-1}, Y_j \} \) and \( V_2 = \{ Y_j, Y_{j+1}, \ldots, Y_N \} \), all variables \( X_i \) in \( \pi^R(1:m) \) are in \( Var(V_1) \), and all variables \( X_i \) in \( \pi^R(m+1, n) \) are in \( Var(V_2) \). Thus, the width of the OBDD is given by the lemma.

**Corollary 4.13.** If \( T \) is an expression tree decomposition of width \( < k \) of \( F \), then for any node \( R \in V(T) \), \( OBDD_{\pi^R}(F) \) has width at most \( n2^{(k+1)2^k+1} \).

This implies \( EPWD(k) \subseteq OBDD(n2^{(k+1)2^k+1}) \).

**Proof.** We use an inductive argument. Fix \( Y_1, Y_2, \ldots, Y_N \) the in-order traversal of the tree \( T^R \), denote \( T_i \) be the subtree rooted at \( Y_i \), \( X_i = Var(T_i) \), and \( n_i = |X_i| \), for \( i = 1, N \). Let
Let \( Y \) be the result of substituting in \( F \) all variables encountered before reaching \( Y_i \), with some values \( \bar{b} \). That is, \( F_i \) denotes the result of substituting in \( F \) all variables encountered before reaching \( Y_i \), with some values \( \bar{b} \).

Note that \( T^R \) is also an expression tree-decomposition of \( F_i \), and that the permutation that \( T^R \) defines on the variables of \( F_i \) is precisely \( \Pi^R(m + 1 : n) \). We prove, by induction on the node \( Y_i \), that the width of \( \text{OBDD}_{\Pi^R(m+1:n)} \) for \( F_i \) is \( 2^\log n_i 2^{(k+1)2^{k+1}} \); the corollary follows by applying this claim to the root node \( Y_i \). Let \( M \) be any depth in this OBDD, \( M = 1, n - m \). Let \( Y_i, \ldots, Y_{i_k} \) be the children of \( Y_i \), and let \( T_{i_1}, T_{i_2}, \ldots, T_{i_k} \) be the subtrees rooted at these children. Consider where the variable \( \Pi^R(m + M) \) is first encountered when traversing the tree \( T_i \) in the order \( T_{i_1}, Y_i, T_{i_2}, \ldots, T_{i_k} \). If it is encountered in \( T_{i_1} \), then the claim for \( F_i \) follows inductively from the claim about \( F_{i_1} \), because \( F_i = F_{i_1} \). If it is in \( Y_i \), then \( \Pi^R(m + M) \) is compatible with the node \( Y_i \), because \( Y_i \) splits the tree into \( T_{i_1} \cup \{ Y_i \} \) (which contains all variables in \( \Pi^R(m + 1, m + M) \)) and the rest of the tree \( T \) (which contains all variables \( \Pi^R(m + M + 1, n) \)), thus the claim follows by the lemma. If \( \Pi^R(m + M) \) is first encountered in \( T_{i_j} \), where \( j > 1 \), then let \( \Pi^R(m + L) \) be the last variable not in \( T_{i_j} \) (thus \( L < M \)). Here, we first notice that the width of \( \text{OBDD}_{\Pi^R(m+1:n)} \) for \( F_i \) at depth \( L \) is \( \leq 2^{(k+1)2^{k+1}} \): this follows from the lemma and the fact that \( \Pi^R(m + 1, m + L) \) is compatible with \( Y_i \). Indeed, \( Y_i \) splits the tree into \( T_{i_1} \cup \ldots \cup T_{i_{j-1}} \cup \{ Y_i \} \) (which contains all of \( \Pi^R(m + 1, m + L) \)) and the rest (which contains \( \Pi^R(m + L + 1, n) \)). Thus, at depths \( L \), \( \text{OBDD}_{\Pi^R(m+1:n)} \) has width \( \leq 2^{(k+1)2^{k+1}} \) for each node at that level it has one copy of \( G_{i_j} \). By induction, each such copy has a width \( 2^{\log n_i 2^{(k+1)2^{k+1}}} \).

Now we use the fact that the subtrees \( T_{i_j} \) were ordered in decreasing number of variables. Since \( j > 1 \), its number of variables is \( n_{i_j} \leq n_i/2 \). It follows that, at depth \( M \), \( \text{OBDD}_{\Pi^R(m+1:n)} \) has width \( \leq 2^{(k+1)2^{k+1}} \times 2^{\log n_{i_j} 2^{(k+1)2^{k+1}}} \leq 2^{\log n_i 2^{(k+1)2^{k+1}}} \), proving our inductive claim about \( F_i \).

Next, we state the surprising converse for Theorem 4.12.

**Theorem 4.14.** If there exists an OBDD for \( F \) with width \( w \), then there exists an expression DAG \( G \) representing \( F \) s.t. \( \text{pwd}(G) \leq 5w \).

This implies \( \text{OBDD}(w) \subseteq \text{EPWD}(5w + 1) \).

Finally, we connect to \( RO \). The following is folklore:

**Proposition 4.15.** Every read-once Boolean function has an OBDD of width \( \leq 1 \). Thus, \( RO \subseteq \text{OBDD}(1) \).
This can be shown by induction on the size of the expression. The OBDD for $E_1 \land E_2$ consists of a copy of the OBDDs for $E_1$ and $E_2$, with the 1-labeled sink node of the former replaced by the root node of the latter; similarly for $E_1 \lor E_2$. Thus:

**Corollary 4.16.** $RO \subseteq EPWD(\overline{6})$

This completes our description of the containment results in Equation 4.4. The separation results are as follows. The first separation is given by the lineage of a query in UCQ, and the last separation is given by the lineage of a query in UCQ$^\neq$; they will both be discussed in Section 4.3. We show here the second separation. For that, we define the Boolean functions $bt_m$ over $2^m$ variables, where $m$ is even:

$$bt_m(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4) = (bt_{m-2}(\bar{x}_1) \oplus bt_{m-2}(\bar{x}_2)) \land (bt_{m-2}(\bar{x}_3) \oplus bt_{m-2}(\bar{x}_4))$$

where $\bar{x}_i, i = 1, 4$ are variable vectors of size $2^{m-2}$ and $\oplus$ is the XOR-operator. This is a read-once expression in the extended grammar Equation 4.3: it is not a read-once expression using our regular grammar Equation 4.2. Hence, $etwd_e(bt_m) = 1$, and, by Prop. 4.5, we have $etwd(bt_m) \leq 3$, hence $bt_m \in ETWD(4)$. On the other hand we show the following, which separates $OBDD(O(1)) \nsubseteq ETWD(O(1))$:

**Theorem 4.17.** Any OBDD for $bt_m$ has width $\geq \frac{2^m}{2}$. Thus, $bt_m \notin OBDD(w)$ for any constant $w$.

### 4.3 Results on Query Compilation

In this section we discuss applications to query compilation (the right diagram of fig. 4.4), and also use them to derive separation results.

#### 4.3.1 Background: UCQ and UCQ$^\neq$

A conjunctive query, $q = R_1(\bar{x}_1) \land R_2(\bar{x}_2) \land \ldots \land R_m(\bar{x}_m)$ is a conjunction of relational atoms $R_i(\bar{x}_i)$, where $\bar{x}_i$ consists of variables and constants, and $R_i$ are symbols from a fixed vocabulary. An inequality predicate over $q$ is of the form $x \neq y$, or $x \neq a$, where $x, y$ are variables and $a$ is a constant. A Union of Conjunctive Query with inequalities (UCQ$^\neq$) is defined as $Q = \bigvee_{i=1}^k (q_i \land p_i)$, where $q_1 \ldots q_k$ are conjunctive queries and $p_i$ is a conjunction of pairwise inequality

Notice that number of subfunctions $F_{\Pi(1,m)=\overline{5}}$ of a read-once expression is, in general, unbounded.
predicates over $q_i$. An example is $R(x), S(x, y), x \neq y \lor R(x), T(y)$, where we have used comma for $\land$, a convention we adopt in the rest of the chapter as well. A Union of Conjunctive Query (UCQ) is a query without inequalities. All queries in this chapter are Boolean queries.

Let $D$ be a database instance. Denote $X_t$ a distinct Boolean variable for each tuple $t \in D$. Let $Q$ be a UCQ. The lineage of $Q$ on $D$ is a Boolean expression $F(Q, D)$ over $\bar{X}$ s.t. for any $D' \subseteq D$, $D' \models Q$ iff the assignment $X_t = \text{true}$, if $t \in D'$ and $\text{false}$ otherwise, satisfies $F(Q, D)$. Figures 4.1, 4.2 have examples where lineage is represented as an expression DAG.

Green at al. [46] describe a general algorithm for computing the semiring annotation of a query output, by using a relational algebra plan for the query: this can be used to compute the lineage $F(Q, D)$, and also to derive an expression DAG for it.

We are only interested in the data complexity, hence we assume query to be fixed, and the database to be variable. Thus, each query defines a set of Boolean functions, and we denote:

**Definition 4.18.** For any $C \in \{\text{UCQ, UCQ}^\neq\}$, define

\[
C(\text{RO}) = \{Q \in C \mid \forall D.F(Q, D) \in \text{RO}\}
\]
\[
C(\text{EPWD}) = \{Q \in C \mid \exists k \forall D.F(Q, D) \in \text{EPWD}(k)\}
\]
\[
C(\text{ETWD}) = \{Q \in C \mid \exists k \forall D.F(Q, D) \in \text{ETWD}(k)\}
\]
\[
C(\text{OBDD}) = \{Q \in C \mid \exists w \forall D.F(Q, D) \in \text{OBDD}(w)\}
\]
\[
C(\text{OBDD}_{\text{poly}}) = \{Q \in C \mid \exists k \forall D.F(Q, D) \in \text{OBDD}(|D|^k)\}
\]

We assume our queries to be ranked, [29, 89], which means that the query has no constants (and, hence, no predicates of the form $x \neq a$), and there exists a global order $\prec$ on the variables such that, whenever $x, y$ occur in a common relational atom and $x$ precedes $y$, then $x \prec y$. Every query is equivalent to (meaning that it has the same lineage as) a ranked query over some different relational vocabulary; for example, $R(x, y), R(y, x)$ is equivalent to $R_1(x, y), R_2(x, y) \lor R_3(z)$, where $R_1 = \sigma_{x<y}(R), R_2 = \Pi_{yx}(\sigma_{x>y}(R))$, and $R_3 = \Pi_x(\sigma_{z=y}(R))$ form a partition on $R$; we refer to [89] for details.

### 4.3.2 Background: Inversion-Free Queries, IF

Given a conjunctive query $q$, its Gaifman graph is a graph with nodes as the variables in query and two variables are connected if they are present together in some atom in the query. A component $c$ is a conjunctive query whose Gaifman graph is connected. Hence, every conjunctive query $q$ can be expressed as $q = c_1 \land c_2 \land \ldots \land c_k$, where each $c_i$ is a component, and for all $i \neq j$,
Chapter 4. Query Compilation and Bounded Treewidth

c_i, c_j do not share common variables. We denote the set of components \( C(q) = \{c_1, c_2, \ldots, c_k \} \).

Given a UCQ\(\exists\) query \( Q = \bigvee_i (q_i \land p_i) \), we define its components as \( C(Q) = \bigcup_{i=1}^n C(q_i) \).

**Definition 4.19.** Let \( c \) be a component. A variable \( x \) is a root variable if it occurs in all atoms of \( c \).

Let \( \bar{c} = \{c_1, c_2, \ldots, c_m \} \) be a set of components. A set of variables \( \bar{x} = \{x_1, x_2, \ldots, x_m \} \) is a separator if for each relational symbol \( R \) there exists a number \( i_R \) such that for all \( j = 1, m \), every relational atom in \( c_j \) with symbol \( R \) has the variable \( x_j \) on position \( i_R \). In particular, \( x_j \) is a root variable in \( c_j \).

**Example 4.2.** The query \( R(x), S(x, y) \) has root variable \( x \); the query \( R(x), S(x, y), T(y) \) has no root variables.

The set of components \( \{[R(x_1), S(x_1, y_1)], [S(x_2, y_2), T(x_2)]\} \) has separator \( x_1, x_2 \). Indeed, \( i_R = i_S = i_T = 1 \); note that both S-atoms have the separator variable on position 1. On the other hand, \( \{[R(x_1), S(x_1, y_1)], [S(x_2, y_2), T(y_2)]\} \) has no separators: the set \( x_1, y_2 \) is not a separator because we cannot take either \( i_S = 1 \) or \( i_S = 2 \): \( x_1 \) occurs on the first position in \( S(x_1, y_1) \), while \( y_2 \) occurs on the second position in \( S(x_2, y_2) \).

A ground atom is an atom without variables: since the query is ranked, this must be a nullary relation symbol \( R() \) (a ground tuple with constants, like \( R(a, b) \), is assimilated with a nullary symbol while ranking the query [89]). A ground atom is a component by itself. If \( \bar{c} \) is a set of components, denote \( \bar{c}^+ \subseteq \bar{c} \), the subset of components that have at least one variable, i.e., we remove ground atoms. Let \( \bar{x} \) be a separator for \( \bar{c}^+ \). Define a new vocabulary where each relation \( R \) has the arity decreased by one, and is obtained by removing the attribute \( i_R \); denote \( c_{i,-x} \) the conjunctive query obtained from the component \( c_i \) by removing from each atom \( R \) the attribute \( i_R \). Notice that \( c_{i,-x} \) is not necessarily connected. Let \( \bar{c}^+_{-x} = \bigcup_i C(c_{i,-x}) \), be the new set of components, where \( c_i \) ranges over \( \bar{c}^+ \).

**Definition 4.20.** A set of components \( \bar{c} \) is consistently hierarchical if either \( \bar{c}^+ \) is empty or has a separator \( \bar{x} \) s.t. \( \bar{c}^+_{-x} \) is consistently hierarchical.

**Definition 4.21.** \( IF^\exists \) is the set of all UCQ\(\exists\) queries \( Q \), s.t. \( C(Q) \) is consistently hierarchical.

We call queries in \( IF^\exists \) inversion-free. Denote \( IF \) the set of inversion-free queries that do not use \( \exists \).

**Example 4.3.** Consider the query shown in Figure 4.2, \( q_{rsst} = R(x_1), S(x_1, y_1), T(x_2), S(x_2, y_2) \).
We prove that it is inversion-free. \( C(q_{rsst}) = \bar{c} = \{[R(x_1), S(x_1, y_1)], [S(x_2, y_2), T(x_2)]\} \) has separator \( x_1, x_2 \). Then \( \bar{c}^+_{-x} = \{[R()], [S(y_1)], [S(y_2)], [T()]\} \). After removing the ground atoms, we
obtain \([\{S(y_1), S(y_2)\}] \equiv \{S(y_1)\}\) which has separator \(y_1\), proving that \(c\) is consistently hierarchical. Thus, \(q_{rsst} \in \text{IF}\).

### 4.3.3 Results for \(UCQ\)

The following result is known from [57]:

**Theorem 4.22.** [57] If \(Q \in \text{IF}\), then for any database \(D\), \(F(Q, D)\) admits an OBDD of width \(2^g\), where \(g\) is the number of atoms in \(Q\). Furthermore, if \(Q \in \text{UCQ} - \text{IF}\), then there exists a database \(D\) over which no OBDD for \(F(Q, D)\) has size polynomial in \(D\).

From here we derive immediately the collapse of the following classes:

**Corollary 4.23.** The following hold: \(\text{UCQ}(\text{EPWD}) = \text{UCQ}(\text{ETWD}) = \text{UCQ}(\text{OBDD}) = \text{UCQ}(\text{OBDD}_{\text{poly}}) = \text{IF}\).

For example, for any input database, the lineage of \(q_{rsst}\) has an OBDD of width 15 and, from here, it follows from Theorem 4.14 that it has an expression DAG of path width \(\leq 80\). Notice that this is not obvious at all from examining the lineage expression for \(q_{rsst}\) in Fig. 4.2, since the “natural” lineage has an unbounded tree-width: we need the detour through OBDD to obtain a bounded path-width expression.

It is also known from [57] that \(q_{rsst} \not\in \text{UCQ}(\text{RO})\), thus \(\text{UCQ}(\text{RO}) \subsetneq \text{UCQ}(\text{OBDD})\), proving the first separation in Equation 4.4. As discussed in Section 4.1, \(q_{rs} \in \text{UCQ}(\text{RO})\).

### 4.3.4 Results for \(UCQ^\neq\)

We present here our main result on query compilation. All proofs are in Sect. 4.5.

**Theorem 4.24.** \(\text{IF}^\neq = \text{UCQ}^\neq (\text{OBDD}_{\text{poly}})\).

Unlike for \(\text{IF}\), the OBDD are no longer of constant width though. Therefore, queries in \(\text{IF}^\neq\) are excellent candidates for separating constant-width OBDD from polynomial-size OBDD. We do this with the following simple query \(q_{nr} = R(x_1, y_1), R(x_2, y_2), x_1 \neq x_2, y_1 \neq y_2\). Since \(q_{nr}\) is inversion-free, we have \(q_{nr} \in \text{UCQ}^\neq(\text{OBDD}_{\text{poly}})\). We show:

**Theorem 4.25.** For any \(c\), there exists a database \(D\), s.t. \(\text{etwd}(F(q_{nr}, D)) > c\). This implies that \(\text{UCQ}^\neq(\text{ETWD}) \subsetneq \text{UCQ}^\neq(\text{OBDD}_{\text{poly}})\).

The theorem immediately implies the last separation result in Equation 4.4, \(\text{ETWD}(O(1)) \subsetneq \text{OBDD}(n^{O(1)})\). It turns out that there exists a matching upper bound for \(q_{nr}\): for any database \(D\), the lineage of \(q_{nr}\) on \(D\) has an OBDD of width linear in the number of tuples of \(D\).
Since \( UCQ^\#(OBDD_{poly}) \) has a syntactic characterization, a natural question arises if there exists a syntactic characterization for \( UCQ^\#(ETWD) \). We define a language, \( H^\# \), as a fragment of \( IF^\# \), and prove that \( H^\# \subseteq UCQ(EPWD) \), which implies that membership in \( H^\# \) is a sufficient condition for \( UCQ^\#(ETWD) \).

**Definition 4.26.** Given a set of components \( \vec{c} = c_1 \ldots c_k \) and a set of inequality predicates \( P \), we say \( (\vec{c},P) \) is consistently hierarchical if \( \vec{c^+} \) is empty or there exists a separator \( \vec{x} \) of \( \vec{c^+} \) s.t. (i) for every predicate \( z \neq y \) in \( P \) s.t. \( z \notin \vec{x} \) and \( y \notin \vec{x} \), and any component \( c_i \) : if \( z \) appears in \( c_i \), then so does \( y \) and vice-versa, and (ii) let \( P_{-x} \) be the set of predicates from \( P \) that do not involve any variable from \( \vec{x} \), then \( (\vec{c^+_x},P_{-x}) \) is consistently hierarchical.

Define \( H^\# \), a subset of \( IF^\# \), as the set of all queries of the form \( Q = \bigvee_{i=1}^{k} (q_i \land p_i) \) s.t. \( (\bigcup_i C(q_i), \bigcup_j p_i) \) is consistently hierarchical. Clearly, \( IF \subseteq H^\# \) by its definition. We have the result

**Theorem 4.27.** \( H^\# \subseteq UCQ(EPWD) \)

**Example 4.4.** \( R(x_1), S(x_1,y_1), S(x_2,y_2), T(x_2), x_1 \neq x_2 \) is in \( H^\# \) because it only compares two separator variables. For another example, \( R(x,y), S(x,z), y \neq z \) is also in \( H^\# \): the inequality is from within the same component, and, after removing the separator variable \( x \), the inequality \( y \neq z \) is between separator variables.

\( R(x_1), S(x_1,y_1), S(x_2,y_2), T(x_2), x_1 \neq x_2, y_1 \neq y_2 \) is not in \( H^\# \) since \( y_1, y_2 \) belong to two different components.

The query \( q_{nr} \) is not in \( H^\# \). There are two components, \( R(x_1,y_1) \) and \( R(x_2,y_2) \) and we have two choices of separator variables: either \( x_1, x_2 \) or \( y_1, y_2 \). But with either choice, one of the inequality conditions violates the condition of \( H^\# \).

We conjecture that \( UCQ^\#(EPWD) = UCQ^\#(ETWD) \).

### 4.4 Proofs on OBDD vs Treewidth

**Proof Of Theorem 4.11.** We call the variables from \( \Pi(1:m) \) set and the rest as \( unset \). Let \( Y = \{ w_1, w_2, \ldots, w_k \} \) where \( w \) are vertices of the expression DAG \( G \) for \( F \). Now we associate a formula \( g_u \) to each node \( u \) in \( G \) as follows : for all \( w_j \), it is a new variable \( f_j \); else either \( u = v \odot w \), then \( g_u = g_v \odot g_w \), or \( u \) is a leaf \( X_i \), then \( g_u = X_i \). Suppose \( F = F_1 \odot F_2 \), then \( F_1, F_2 \) can be defined as formulae over \( \bar{f} \) and some subset of variables \( \bar{X}_1, \bar{X}_2 \) respectively from \( \bar{X} \). We claim that either every variable in \( \bar{X}_1 \) is \( set \) or they are all \( unset \) and same for \( \bar{X}_2 \). Suppose variables \( X_{i1}, X_{i2} \) are in \( \bar{X}_1 \) and \( X_{i1} \in Var(V_1), X_{i2} \in Var(V_2) \), where \( V_1, V_2 \) are as
in Theorem 4.10. But both $X_{i3}, X_{i2}$ must be connected to $F_1$ by some path since $F_1$ depends on these variables, hence $F_1$ must belong in $Y$. But then $F_1 = f_j$ for some $j$ and it shouldn’t depend on any $X_i$: a contradiction. We could argue similarly for $F_2$.

Now we will bound the number of possible values $F_1, F_2$ can take as variables from $\Pi[1 : m]$ are set to 0/1. We will prove for $F_1$ and symmetrically the same can be shown for $F_2$. Suppose $\bar{X}_1$ are all set. Then $F_1$ is a Boolean function over $k$ variables $\bar{f}$. There are only $2^{2^k}$ possible Boolean functions over $k$ variables. Now, each variable $f_i$ stands for the function represented at the gate $w_i$. Let assume each $f_i$ take at most $N$ possible values too. Then, we have a bound of $2^{2^k}N^k$ on the number of possible values of $F_1$. Similarly, if all $\bar{X}_1$ were unset, the bound would be $N^k$. We will now bound $N$.

For any $w_j$ which has $l$ nodes from $\bar{w}$ as its descendants, we claim that $f_j$ takes at most $2^{2^l}$ values. If $w_j$ doesn’t have any nodes from $\bar{w}$ as its descendant, then $N$ is either 1 or $2 = 2^0$. Suppose this holds for all nodes below $w_j$. We use the same argument as above again. Consider the set of nodes $\bar{u} = \{u_1, u_2, \ldots, u_l\}$ from $\bar{w}$ directly below $w_j$, i.e., there exists no other node $w_i$ which is a descendant of $w_j$ and the ancestor of the said node. Then $f_j = f_{j1} \Rightarrow f_{j2}$, where $f_{j1}, f_{j2}$ are boolean formulae over $\bar{u}$ and other nodes, all of which must be set or unset. Hence the number of possible values of $f_{j1}, f_{j2}$ is at most $2^{2^k} \prod_i 2^{2^l_i}$, where $l_i$ is the number of nodes from $\bar{w}$ below $u_i$. Since $l + \sum_i l_i \leq k - 1$, we get $N \leq 2^{2^k}$. Hence both $F_1, F_2$ take at most $2^{(k+1)2^k}$ values, hence the number of possible valuations of $F$ can’t be more than $2^{(k+1)2^k} = 2^{(k+1)2^{k+1}}$.

Proof Of Th. 4.14. We construct an expression $G$ for $F$ s.t. $pwd(G) = 5w$. We do this using the concept of a shared expression DAG: Given formulas $f_1, f_2, \ldots, f_w$, a shared expression DAG is an expression DAG which represents all the $\bar{f}$, i.e., it has $w$ root or output nodes and the formula obtained by evaluating the expression below these nodes corresponds exactly to $\bar{f}$. The construction follows from the following more general lemma.

Lemma 4.28. If $\bar{f} = f_1, f_2, \ldots, f_w$ have a shared OBDD with width $w$, then there exists a shared expression for them having path width $5w$, s.t. all root nodes $\bar{f}$ occur on a leaf of the path decomposition.

Proof. We prove by induction on the number of variables $n$. Let the first variable in the variable order of OBDD be $X_1$ and denote the formulae at the first level by $g_1, g_2, \ldots, g_w$. Then every $f_i$ can be written as $\neg X_1 \land f_j \lor X_1 \land f_k$ for some $j, k$. Denote the nodes corresponding to new $\land, \lor, \neg$ operators by $\overline{mp}$. Now by induction hypothesis, $\bar{g}$ have a path-decomposition with width $5w$ one of whose leaves contains $\bar{g}$. We connect that leaf to a new node which contains $\bar{g}, \bar{f}, X_1, \overline{mp}$. The resulting path-decomposition of $\bar{f}$ has width $5w$. \qed
Proof Of Th. 4.17. We will prove that an OBDD of \{bt_m, \neg bt_m\} has width \( \geq 2^{m/2} \). Since the width of a function and its negation are the same, we get that the width of \( bt_m \) is at least \( 2^{m/2} / 2 \).

We will proceed by induction. In the base case \( m = 0 \), width of \( \{x, \neg x\} \) is \( 2 \geq 2^0 / 2 \). Let \( bt_m = f = (f_1 \oplus f_2) \land (f_3 \oplus f_4) \), where \( f_i, i = 1 \ldots 4 \) is \( bt_{m-2} \), and hence width of \( \{f_i, \neg f_i\} \) is at least \( 2^{m/2-1} \).

Consider the first level where all variables of one of the \( f \) have been set to 0/1. : say \( f_1 \). We first consider the case that \( X \), a variable of one of \( f_2, f_3, f_4 \) has been set before this level. By induction hypothesis(IH), we know that there exists a level, where width of \( f_1 \) is \( w = 2^{m/2-1} / 2 \). Let \( \{u_1, u_2, \ldots, u_w\} \) be the corresponding \( w \) subformulae of \( f_1 \). We have two cases depending on whether \( X \) is a variable of \( f_2 \) or \( f_3, f_4 \).

Case I : \( X \) is a variable of \( f_2 \). Let \( g_1, g_2 \) be two distinct subformulae of \( f_2 \) at this level. Pick a subformula \( h \) of \( f_3 \oplus f_4 \) from this level, s.t. \( h \neq \text{false} \). We define 4 sets of formula at this level :

\[
S_i = \{(u_p \oplus g_i) \land h) \mid 1 \leq p \leq w\}, i = 1, 2
\]

\[
T_i = \{\neg((u_p \oplus g_i) \land h) \mid 1 \leq p \leq w\}, i = 1, 2
\]

So \( S_1, S_2 \) contain the subformula from \( f \) and \( T_1, T_2 \) from \( \neg f \). First, we claim that no formula from \( S \) can equal any from \( T \). This is simple to see since setting \( h = 0 \) sets any formula in \( S \) to 0 while in \( T \) to 1. Hence they can’t be equal. Now, we compare the functions in \( S_1, S_2 \). Clearly \( (u_i \oplus g_1) \land h \neq (u_i \oplus g_2) \land h \), since setting \( g_1, g_2 \) to 0,1 gives two different functions \( u_i \land h, \neg u_i \land h \).

Now, suppose \( (u_i \oplus g_1) \land h = (u_j \oplus g_2) \land h \), then if we set \( g_1 = g_2 \) to 0 or 1, we get \( u_i = u_j \). We can do this because one of \( g_i, g_j \) must not be 0/1, as not all variables of \( f_2 \) have been set. Hence we get functions in \( S_1, S_2 \) are also distinct and similarly the same holds for \( T \). Therefore, total width \( = 4w = 4 \cdot 2^{m/2-1}/2 = 2^{m/2} \).

Case II : \( X \) is a variables of \( f_3 \) or \( f_4 \). This case is also similar and easy to verify as all 4 classes lead to different formulae.

So, now we can assume that all variables of \( f_1 \) are set and no other variables have been. Again, we need to consider two cases depending on whether we start with \( f_2 \) or \( f_3, f_4 \) next. Note that, by the same argument as above, we will be setting all variables of one of them.

Case III : Suppose we set all variables of \( f_2 \) next. Let \( h = f_3 \oplus f_4 \). After exhausting variables of \( f_1 \), we have 4 different formulae in the OBDD of \( \{f, \neg f\} \) which we group as : \( \{f \land h, \neg f \land h\} \) and \( \{\neg(f \land h), \neg(\neg f \land h)\} \). Now lets consider a level where the width of an OBDD for \( \{f_2, \neg f_2\} \)
is \( w = 2^{m/2-1} \). Then clearly we get 2 sets of subformulae of cardinality \( w \) each from the two groups mentioned above. And as argued in the previous cases, the two cannot have any formula in common, since setting \( h = 0 \) leads to different value in two cases. Hence, we get width at this level \( = 2w = 2 \cdot 2^{m/2-1} = 2^{m/2} \).

Case IV: In this case we set variables of \( f_3 \) next. It’s easy to see that in this case, all 4 resulting formulae must lead to distinct subformulae. And the width for each is at least \( 2^{m/2-1}/2 \). Hence width is at least \( 4 \cdot 2^{m/2-1}/2 = 2^{m/2} \).

This exhausts all possible cases, hence proved. \( \Box \)

### 4.5 Proofs on OBDD size of \( IF^\# \)

First, we describe how to get rid of all predicates of the form \( x \neq a \), where \( a \) is a constant. This can be accomplished by rewriting the query and suitably changing the vocabulary just as we did for removing constants. We just illustrate this with an example:

\[
R(x_1), S(x_1, y_1), S(x_2, y_2), R(x_2), x_1 \neq x_2 \land x_2 \neq a.
\]

We can rewrite this to:

\[
R^\neq(x_1), S^\neq(x_1, y_1), S^\neq(y_2), R^\neq(), x_1 \neq x_2 \lor
R^\neq(x_1), S^\neq(x_1, y_1), S^\neq(x_2, y_2), R^\neq(x_2), x_1 \neq x_2.
\]

The resulting query is still inversion-free with the same inequalities.

Now, we show that all queries in \( H^\# \) have an OBDD of width \( O(1) \).

**Proof of Th. 4.27.** Let \( P = \{p_1, p_2, \ldots, p_k\} \) be the set of pairwise inequality predicates in \( Q \) and \( C(Q) = \bar{c} = c_1, c_2, \ldots, c_l \) be the set of components. For any \( s_1 \subseteq [k], s_2 \subseteq [l] \), let \( q_{s_1, s_2} = \bigwedge_{i \in s_1} c_i \land \bigwedge_{j \in s_2} p_j \). We will construct a shared OBDD for all \( 2^{k+l} \) queries \( q_{s_1, s_2} \). One can derive the OBDD for \( Q \) from this shared OBDD.

Let \( \bar{x} = x_1, x_2, \ldots, x_l \) be a separator for \( \bar{c} \). Let the active domain of \( \bar{x} \) be \( \{a_1, a_2, \ldots, a_n\} \). Assume inductively that the width of each of \( q_{s_1, s_2}[a_i/x] \) depends only on the query. Now, note that since we have no inequality predicates between non-root variables, the OBDD for \( \bar{x} = a_i \) and \( \bar{x} = a_j \) can be constructed independently for \( i \neq j \). Suppose we have constructed the shared OBDD for all \( q_{s_1, s_2}^{i-1} = q_{s_1, s_2}, \bar{x} \in \{a_1, a_2, \ldots, a_{i-1}\} \). We show how to extend it to get the shared OBDD of all \( q_{s_1, s_2}^i \). Note that \( q_{s_1, s_2}^i \) is true iff there exists some \( r_1 \subseteq s_1, r_2 \subseteq s_2 \), s.t. \( q_{r_1, r_2}^{i-1} \) is true and for all \( j \in s_1 - r_1, c_j[a_i/x] \) is true, and for any predicate \( x_o \neq x_p \), in \( s_2 - r_2, c_o[a_i/x] \) is true and \( p \in r_1 \) or vice-versa \( c_p[a_i/x] \) is true and \( p \in r_1 \). One can hence construct the shared OBDD for all \( q_{s_1, s_2}^i \) by just following the above logic. The width is at most \( 2^{2k+l} = O(1) \). \( \Box \)
Now we show that all queries in \( IF^\ne \) have an OBDD of polynomial size. The converse is a straightforward generalization of the proof from [57] and we discuss that in the Appendix.

**Proof Of Th. 4.24.** Let \( q^\ne = q, P \), where \( q \) is an inversion-free conjunctive query and \( P \) is a conjunction of inequality predicates. We claim that if for a variable order \( \Pi \), \( OBDD_\Pi(q) \) has polynomial size, then so does \( OBDD_\Pi(q^\ne) \). By the classical result on synthesis of OBDD, we have that if \( OBDD_\Pi(F_1) \) has size \( s_1 \) and \( OBDD_\Pi(F_2) \) is of size \( s_2 \), then \( OBDD_\Pi(F_1 \otimes F_2) \) is of size at most \( s_1s_2 \), where \( \otimes \) is any boolean connective viz. OR, XOR. So if \( Q = \bigvee_{i=1}^{k} (q_i \land p_i) \) is inversion-free: we know there exists \( \Pi \) under which all \( q_i \) have polynomial size OBDD, and hence by our claim so do \( q_i, p_i \) and by the synthesis result \( Q \) has a polynomial size OBDD. So it suffices to prove the claim that \( q^\ne \) has a polynomial size OBDD.

Given any tuple \( t \), we define \( P(t) \) as the formulae obtained by setting variables associated with tuple \( t \) in \( P \). For example if \( P \) is \( x \neq y \) over \( R(x), S(x, y) \), then \( P(R(1)) = y \neq 1 \). Given a vector of tuples \( \bar{t} \) and boolean assignments \( \bar{a} \) on them, we similarly define \( P[\bar{a}/\bar{t}] = \bigvee_{a_i = \text{true}} P(t_i) \).

We already know that there exists an OBDD of constant width for \( q \). Now consider any branch of tuples \( \bar{t} \) with assignment \( \bar{a} \) on this OBDD. Given a database \( D \), if the subformulae/lineage of \( q \) represented by this branch was \( f \), then for \( q^\ne \), it is \( F(q^\ne, D - \bar{t}) \lor (f, P[\bar{a}/\bar{t}]) \), where \( F(q^\ne, D - \bar{t}) \) denotes the lineage obtained by evaluating \( q^\ne \) over \( D \) without the tuples \( \bar{t} \). Since our OBDD has constant width, for most of the assignments \( \bar{a} \) on \( \bar{t} \), the resulting formula \( f \) will be the same.

We will further show that as we iterate over exponentially many assignments \( \bar{a} \), the number of distinct predicates \( P[\bar{a}/\bar{t}] \) generated is only polynomial. Hence, since the width of the original OBDD was constant, the new width only increases polynomially.

Suppose \( P = \bigwedge_{j=1}^{k} x_{ij} \neq y_{j1} \). Suppose there are \( m \) variables in \( \bar{x} = x_1, x_2, \ldots, x_m \) and w.l.o.g. assume the tuples that we have seen only influence the variables \( \bar{y} \). Consider \( DR = \neg P[\bar{a}/\bar{t}] \). It looks like:

\[
\begin{align*}
(x_{i1} = a_{1,j1} \lor x_{i2} = a_{1,j2} \lor \ldots x_{ik} = a_{1,jk}) \land \\
(x_{i1} = a_{2,j1} \lor x_{i2} = a_{2,j2} \lor \ldots x_{ik} = a_{2,jk}) \land \\
\ldots \ldots \ldots \land \\
(x_{i1} = a_{n,j1} \lor x_{i2} = a_{n,j2} \lor \ldots x_{ik} = a_{n,jk})
\end{align*}
\]

where \( n \) tuples were observed to be true. At a quick glance it would seem that the number of minterms in \( DR \) could be \( nk \). But many of them are actually inconsistent, since we can’t have two terms like \( x_1 = a_p \) and \( x_2 = a_q \) in the same minterm. In particular, every minterm is of length at most \( m \). We show that there are at most \( 2^{mk} \) minterms in \( DR \). Now, there are at
most \((1 + |D|)^m\) possible minterms of size less than \(m\) and hence we get number of possible \(DR\) is bounded by \((1 + |D|)^{m2^m}\). We prove the claim next.

**Lemma 4.29.** The number of minterms in \(DR\) is at most \(2^{mk}\).

**Proof.** We prove by induction on \(k\). Consider the following minterm \(T\) of \(DR\): \(x_1 = a_1 \land x_2 = a_2 \land \ldots \land x_m = a_m\).

Each clause in \(DR\) must contain one of \(x_i = a_i\), hence we can write \(DR\) as:

\[
(x_1 = a_1 \lor C_{11}) \land (x_1 = a_1 \lor C_{12}) \land \ldots \land \\
(x_2 = a_2 \lor C_{21}) \land (x_2 = a_2 \lor C_{22}) \land \ldots \land \\
\ldots \land \\
(x_m = a_m \lor C_{m1}) \land (x_m = a_m \lor C_{m2}) \land \ldots 
\tag{4.7}
\]

Consider an arbitrary minterm \(T'\) of \(DR\). We write \(T'\) as \(T' = T_0 \land T_1\), where \(T_0\) contains all inequalities \((x_i = a_i)\) that are common between \(T'\) and \(T\), and \(T_1\) contains the others, i.e. not occurring in \(T\). We will count the minterms \(T'\) by grouping by \(T_0\): there are \(2^m\) possibilities for \(T_0\), and for each of them we will count the number of distinct minterms \(T_1\).

We know that \(T' \Rightarrow DR\), hence if we substitute \((x_i = a_i)\) to be true for every \(x_i = a_i\) term in \(T_0\), we obtain \(T_1 \Rightarrow DR'\), where \(DR'\) is obtained from Eq. (4.7) by removing all rows with index \(i\) s.t. \(x_i = a_i\) is in \(T_0\). Now we know that \(T_1\) doesn’t contain any other minterm of the form \(x_j = a_j\) that is left in \(DR'\), hence we can remove them from \(DR\) to get \(DR''\) and the implication \(T_1 \Rightarrow DR''\) would still hold. So we get that \(T_1\) is a minterm of an expression, \(DR''\), which by induction hypothesis has at most \(2^{m(k-1)}\) minterms. Therefore number of possible choices for \(T_1\) are at most \(2^m2^{m(k-1)} = 2^{mk}\). Hence proved.

**Proof Of Theorem 4.25.** We will first study the OBDD for \(q_{\lor \land}\). In particular, we will show that it has an OBDD of linear size and show a matching lower bound too. Then we will use the lower bound proof to prove the theorem.

Consider \(R = \{(i, j) \mid 1 \leq i, j \leq n\}\) and the order \(\Pi\) on tuples which just sorts them in natural increasing order. We will show that over this database, the OBDD with this order has width \(O(n)\). Note that the OBDD for \(Q\) on any other database where the domain size of attributes is \(n\) or less can be obtained by just setting some tuples in this OBDD to 0; the resulting OBDD would still have width \(O(n)\). This proves the upper bound. For the lower bound, we’ll show
a level where the width has to be \( n/2 \), regardless of the order used. We associate a random variable \( r_{ij} \) with the tuple \( R(i, j) \) to represent lineage of \( Q \), which we call \( \phi \).

Note that
\[
\phi_{|r_{pq}=1} = \left( \bigvee_{j \neq p} r_{iq} \land \bigvee_{j \neq q} r_{pj} \right) \lor \bigvee_{i \neq p, j \neq q} r_{i,j} \quad (4.8)
\]

After setting any subset of the rest of the variables to 0/1, there are only \( O(n) \) possible formula from \( \phi_{|r_{pq}=1} \). Now at any level, there is only one branch where all variables are set to 0. All other branches have at least one variables set to 1, and as we saw there are only \( O(n) \) possibilities for the formula that result from these assignments. Hence the width of the OBDD is \( O(n) \).

Now, for the lower bound consider the \( n/2 \) level. We consider all assignments where exactly one of the variables is set to 1. There are \( n/2 \) such assignments. For each of these assignments, by inspection of Eq. 4.8, we see that we get a different formula that still depends on every variable not yet set. Hence the width is at least \( n/2 \).

Now we prove the theorem. Consider the same data instance of \( R \) as above. Note that the argument for the lower bound proof above actually proves that the width at any level \( l \), s.t., \( cn \leq l \leq n - 1 \), for any \( c < 1 \), is \( l \). Suppose there is an expression DAG for \( \phi \), the lineage of \( q_{nr} \), with constant width tree-decomposition \( T^R \). For the purpose of this proof we will assume the tree-decomposition is normal[45]. This means \( T \) is binary and if \( Y_i \in V(T) \) has two children \( Y_j, Y_k \) then \( Y_i = Y_j = Y_k \) and if \( Y_i \) has only one child \( Y_j \), then \( |Y_i - Y_j| = 1 \). Now suppose we start with \( R \) as in the construction from Theorem 4.13. Consider the neighbor \( Y_1 \) : the number of \( \bar{r} \) variables in descendants of \( Y_1 \) is at least \( n^2/2 \). We similarly pick a child of \( Y_1 \) with at least \( n^2/4 \) \( \bar{r} \) variables in its descendant. We can keep iterating and each time the number of variables in the descendants of the chosen node can decrease by at most half. Hence at one point, we must reach a node \( Y \) s.t. the number of \( \bar{r} \) variables in its descendants is between \( n/4 \) and \( n/2 \). Then, after setting all variables in the descendants of \( Y \), we must get constant number of subformulae according to Theorem 4.11, while our lower bound proof suggests the number of subformulae is at least \( n/4 \); a contradiction.

\[ \square \]

### 4.6 Conclusion and Future Work

We have presented a new notion of treewidth for Boolean formula, called expression treewidth, that captures many of the tractable cases known in probabilistic databases. We have shown that bounded expression treewidth implies polynomial size OBDD. Furthermore, bounded expression pathwidth is equivalent to constant-width OBDD. Since both parameters : treewidth, OBDD,
are widely used in areas like SAT, CSP, Verification, Graphical Models, etc., we think these connections can have wide-ranging applications.

The computability of this parameter though is still an open problem and presents interesting avenues for future research. Also, with all these structural parameters, we can only capture the set of queries which have an OBDD. The set of queries for which model counting is possible, is known to be much larger than this set [57] and there are other compilation languages, like FBDD, d-DNNF, which can solve more queries than OBDD. This motivates the need to find a structural parameter that could at least capture FBDD.

We would also like to investigate other width-parameters beyond treewidth which are more general and can solve problems unsolvable by using tree-decompositions. Clique-width are known to be another parameter under which model counting is tractable. In fact [38] showed that if the clique-width of the incidence graph is tractable, then model counting is tractable. A detailed comparison between expression treewidth and clique-width is left as future work.
In this chapter, we devise an efficient query evaluation method for MVDB. The task is to compute $P_0(Q \lor W)$, where both $Q$ and $W$ are UCQs. Note that $W$ depends only on the MarkoViews, and does not depend on the query $Q$. We describe a new index structure for $W$, called an MV-index, and show how to use it to compute $P_0(Q \lor W)$. The MV-index consists of an Ordered Binary Decision Diagram (OBDD) \cite{13}, extended with additional information that is critical for computing $P_0(Q \lor W)$ efficiently. We extend the construction algorithm for inversion-free queries to construct an OBDD for any $W$: in the particular case when $W$ is inversion-free, the resulting OBDD is guaranteed to be linear in the size of the database. The OBDD itself is not sufficient for computing $P_0(Q \lor W)$ efficiently: state of the art synthesis algorithms require a time proportional to the product of the sizes of the two OBDDs for $Q$ and $W$ respectively. We describe here how to extend the OBDD to an MV-index, then present a top-down evaluation algorithm for computing $P(Q \lor W)$, which traverses only a small fraction of the MV-index.

5.1 Compiling MarkoViews

We have translated the problem $P(Q)$ on an MVDB into the problem $P_0(Q \lor W)$ on a tuple-independent database. Thus, from now on we will only consider tuple-independent databases.

Even though query evaluation over INDB has been well-studied, there is an important distinction in our setting. While the lineage of $Q$ is typically small, the lineage of $W$ is usually large, especially as we try to capture more correlations. For example, the DBLP data in Figure 2.1 has over 6M probabilistic tuples, and many of them (if not all) will be included in the lineage of $W$. Since $W$ is defined offline, we employ the strategy of pre-compiling $W$, in order to speed up query evaluation of $P_0(Q \lor W)$ online. In this section, we will discuss the data structure MV-Index into which we compile $W$ along with the algorithm for compilation. MV-Index has been designed to speed-up the online evaluation of $P_0(Q \lor W)$, which we discuss next.
\[ \Phi_Q = X_1Y_1 \lor X_1Y_2 \lor X_2Y_3 \lor X_2Y_4 \]

**Figure 5.1**: A query \( Q \) on a probabilistic database, its lineage \( \Phi_Q \), and an OBDD for \( \Phi_Q \).

Throughout this section we assume a tuple-independent probabilistic database \( D_0 \). We associate to each tuple a Boolean variable \( X_1, X_2, \ldots, X_n \), Section 2.2.1. We consider only Boolean queries in this section. \( W \) is already a Boolean query (Equation 2.4); the user query \( Q \) is typically not a Boolean query, but for the purpose of query evaluation we substitute its head variables with an answer tuple, thus transforming it into a Boolean query. We denote \( \Phi_Q \) the **lineage** of the query \( Q \) on the probabilistic database. \( \Phi_Q \) is a Boolean formula using variables \( X_1, \ldots, X_n \) (see, e.g. [89]). Figure 5.1 shows a probabilistic database, a query \( Q \), and its lineage expression \( \Phi_Q \). Notice that the lineage of a disjunction is the disjunction of the lineages, \( \Phi_{Q \lor W} = \Phi_Q \lor \Phi_W \).

### 5.1.1 MV Index

An MV-Index consists of a set of OBDD augmented with certain pre-computations and indices that we describe below. But first we briefly review OBDD.

**OBDD**: An Ordered Binary Decision Diagrams, OBDD [13] is a rooted DAG, where internal nodes are labeled with Boolean variables and have two outgoing edges, labeled 0 and 1; sink nodes (leaves) are labeled 0 or 1. There are two constraints: every path from the root to a leaf must visit each variable at most once, and any two paths must visit the variables in the same order (missing variables are OK), see an example in Figure 5.1. Given an OBDD for \( \Phi \) one can compute the probability \( P_0(\Phi) \) in linear time. Denote \( p(u) \) the probability of the Boolean formula encoded by the OBDD rooted at node \( u \). If \( u \) is a leaf node, set \( p(u) = 0 \) or \( p(u) = 1 \) (depending on its label); otherwise it is labeled with some variable, say \( X_i \), and has
two children, say \( u_0, u_1 \) corresponding to the outgoing edges labeled with 0,1 respectively. We set \( p(u) = (1 - P_0(X_i)) \cdot p(u_0) + P_0(X_i) \cdot p(u_1) \). This formula (Shannon expansion) also holds when \( P_0(X_i) \) is negative. The size of an OBDD is the total number of nodes in it. The width at level \( i \) is the number of nodes labeled with variable \( X_i \) and width is the maximum width at any level. Note that the size of OBDD is at most the width times the number of variables.

**The MV-Index** An augmented OBDD for \( \Phi \) is an OBDD where each node \( u \) is annotated with two quantities. First, \( u.\text{probUnder} \) stores \( p(u) \). Second, \( u.\text{reachability} \) stores the sum of the probability of all paths starting from root to \( u \). The probability of a path is defined as a product of the following factors: for an edge \( X_i = 1 \) we include the factor \( P_0(X_i) \), and similarly for \( X_i = 0 \) we include \( (1 - P_0(X_i)) \). To see the intuition, suppose we want to compute \( P_0(\varphi \land \Phi) \) for some “small” expression \( \varphi \), and assume \( \varphi = X_1 \), the root variable of the OBDD. Then we just return \( p \ast u.\text{probUnder} \), where \( p \) is the probability of \( X_1 \) and \( u \) is its 1-child. Assume now \( \varphi = X_i \), and that there are \( c \) nodes in the OBDD labeled \( X_i, u_1, u_2, \ldots, u_c \). Assume every path from the root to a sink contains one of these nodes. Let \( v_1, v_2, \ldots, v_c \) be their 1-children. Then \( P_0(X_i \land \Phi) = p \ast \sum_{j=1}^{c} u_j.\text{reachability} \ast v_j.\text{probUnder} \). When the width \( c \) is small, then the augmented OBDD allows us to compute these probabilities efficiently.

Finally, an MV-Index consists of a set of augmented OBDD, each of them associated with a particular key. These OBDD are over disjoint set of variables. Besides the OBDD, it keeps two indices. *InterBddIndex*, given a tuple, returns the key of the OBDD where the tuple is located. *IntraBddIndex* returns all the nodes in the OBDD which correspond to that tuple. For e.g., in our previous example where we were computing \( P(X_1) \), we still needed to locate all the nodes labeled with \( X_1 \). These indices enable us to do that in constant time.

### 5.1.2 Constructing the MV-Index

In this section, we will describe our algorithm to construct an OBDD for a UCQ. Given an OBDD, adding the pre-computations and indices needed to make it an MV-Index are straightforward and hence skipped in this section.

Let \( \Pi \) be the order in which an OBDD visits the variables on each path. (E.g. \( \Pi = X_1, Y_1, Y_2, X_2, Y_3, Y_4 \) in Figure 5.1). The order \( \Pi \) uniquely determines the OBDD [96], up to merging of equivalent nodes, therefore the problem of finding a small OBDD is equivalent to finding a good order \( \Pi \) (meaning one that leads to a small OBDD). Denote \( OBDD_\Pi(\Phi) \) the OBDD of \( \Phi \) with order \( \Pi \). If \( \Phi = \Phi_1 \lor \Phi_2 \), or \( \Phi = \Phi_1 \land \Phi_2 \), and we are given OBDDs \( G_1, G_2 \) for \( \Phi_1, \Phi_2 \) with the same order \( \Pi \), then one can compute \( OBDD_\Pi(\Phi) \) in time \( O(|G_1||G_2|) \) by a
procedure called synthesis. CUDD [88], a widely popular package for OBDDs, uses this synthesis procedure. It starts with some order Π and synthesizes the OBDD traversing Φ recursively.

We do the following improvement over the synthesis procedure in CUDD. Suppose Φ1, Φ2 are independent, i.e. they use disjoint sets of Boolean variables, then one can synthesize Φ1 ∨ Φ2 more efficiently: stack the OBDD's G1, G2 on top of each other, and redirect every 0-labeled leaf of G1 to the root of G2 (for Φ1 ∧ Φ1 one would redirect the 1-labeled leaves). We call this concatenation. The size of the new OBDD is only |G1| + |G2|, and, unlike synthesis, concatenation is a constant time operation. Thus, concatenation represents a major improvement over synthesis. We explain next where we can use concatenation when computing the OBDD of a UCQ.

Consider a Conjunctive Query(CQ) Q. A root variable in Q is a variable x that appears in all atoms of Q. One can write Q ≡ Q1(a1/x) ∨ Q2(a2/x) ∨ ... , where a1, a2, ... form the active domain of x. It is not hard to see that if Q has no self-joins then Q1(a1/x) and Q2(a2/x) have no tuples in common; hence the OBDD of Q can be obtained by concatenating the OBDD of Q1(a1/x). This has been illustrated for Q = R(x), S(x,y) in Figure 5.1.

In general for a UCQ Q = Q1 ∨ Q2 ∨ ... , where Qi are CQ, let xi be a root variables of Qi. We write Q = ∃x1.Q1 ∨ ∃x2.Q2 ∨ ... = ∃z.(Q1[z/x1] ∨ Q2[z/x2] ∨ ...). The new variable z occurs in all atoms of all conjunctive queries. We call z a separator variable if any two atoms with the same symbol contain z on the same attribute position. Let a1, a2, ... be the active domain for z. Then once again the same property holds : Q ≡ Q1[a1/z] ∨ Q2[a2/z] ∨ ..., and the queries on the right are independent, i.e. they do not have any common tuples. For e.g., let Q = R(x1), S(x1,y1) ∨ T(x2), S(x2,y2). Both x1 and x2 are root variables, and we write the query as follows (showing quantifiers explicitly now):

Q = ∃z.[∃y1.(R(z) ∧ S(z,y1)) ∨ ∃y2.(T(z) ∧ S(z,y2))]

Thus, Q = Q1[a1/z] ∨ Q2[a2/z] ∨ ... and the OBDD for Q has a size that is the sum of the OBDD for Q1[a1/z]. In general

**Proposition 5.1.** If z is a separator in Q, then there exists an OBDD for Q of size at most the sum of OBDD of Q[a1/z], i = 1, n, where a1, a2, ..., an is the active domain.

Finally, consider Q = R(x1), S(x1,y1) ∨ S(x2,y2), T(y2): this query does not have a separator variable. For such queries we fall back on general tools like CUDD. Then there are hybrid cases like Q = Q1 ∨ Q2, Q1 = R(x), S(x,y), R(y), Q2 = S(x,y)T(x). Here only Q2 has a root variable, so we will choose Π such that we can compute $OBDD_Π(Q_2)$ by concatenation, then use synthesis
to compute $\text{OBDD}_\Pi(Q_1)$ using the same order $\Pi$. Having reviewed the concepts from our prior work, we are now ready to describe the construction, which is the novel part of this section.

Fix a relational schema $\mathbf{R} = \{R_1, \ldots, R_k\}$. Order the relation names from smaller to larger arities. Let $\pi = \{\pi_{R_1}, \ldots, \pi_{R_k}\}$ be set where each $\pi_i$ is a permutation on the attributes of the relation $R_i$. That is, if $\text{arity}(R_i) = m$ then $\pi_i$ is any permutation on $1, \ldots, m$. Consider a database instance $I$, over an ordered active domain $a_1 < a_2 < \ldots < a_n$. Then $\pi$ defines an order $\Pi$ on all tuples in $I$, as follows. $\Pi = \Pi_1, \Pi_2, \ldots, \Pi_n$, where each $\Pi_j$ is the order obtained recursively as follows: for each relation $R_i$, retain from $D$ only those tuples where the first attribute in $R_i$ (according to $\pi_{R_i}$) is $a_j$, then project out that attribute, and compute $\Pi_j$ recursively on the smaller database. For example, consider the schema $R(A), S(A,B)$, and the permutation $\pi_R = (A)$ and $\pi_S = (A, B)$. Consider the database instance in Figure 5.1), where the active domain is ordered by $a_1 < a_2 < b_1 < b_2 < b_3 < b_4$. Then $\Pi = X_1, Y_1, Y_2, X_2, Y_3, Y_4$.

Let $\pi$ be given and let $Q$ be a query. If $x, y$ are two variables then we write $x >_\pi y$ if whenever $y$ occurs in an atom on some attribute $B$, then $x$ also occurs in that atom on some attribute $A$, and $A$ comes before $B$ in the permutation $\pi_{R_i}$.

Given $\pi$ and a query $Q$, the following recursive procedure $\text{ConOBDD}(\pi, Q)$ constructs $\text{OBDD}_\Pi(Q)$, where $\Pi$ is the permutation on the tuples associated to $\pi$:

- **R1** $Q = Q_1 \lor Q_2$ : if $Q_1, Q_2$ have no relations in common, concatenate, else synthesize.
- **R2** $Q = Q_1 \land Q_2$ : if $Q_1, Q_2$ have no relations in common, concatenate, else synthesize.
- **R3** $Q = \exists x . Q_1$ : if $x >_\pi y$ for all variables $y$ in $Q_1$, then concatenate, else synthesize.
- **R4** $Q = R(\bar{a})$ : trivial

We choose heuristically $\pi$ such as to minimize the number of synthesis steps in **R3**. In particular, if $Q$ has a separator variable, then we always choose $\pi$ such that every attribute holding a separator variable occurs first in the permutation $\pi$. Call a query $Q$ inversion-free if there exists $\pi$ such that only concatenations are performed in **R3**. (This is equivalent to the definition of inversion-free queries in [59].) Let $\{a_1, \ldots, a_n\}$ be the active domain. The following proposition, based on [59], gives guarantees on the size of $\text{ConOBDD}(\pi, Q)$ in certain cases.

**Proposition 5.2.** Let $N$ be the size of the OBDD returned by $\text{ConOBDD}(\pi, Q)$. (a) If $Q$ admits a separator variable $z$, then $N = \sum_j N_j$, where $N_j$ is the size of the OBDD of $Q[a_j/z]$, $j = 1, n$. (b) If $Q$ is inversion-free, then the OBDD has constant width; hence $N = O(n)$. [59].
5.1.3 Querying an MV-index

Our goal is to compute efficiently Eq.2.5, whose numerator is $P_0(Q \lor W) - P_0(W) = P_0(Q \land \neg W)$. The OBDD for $\neg W$ is obtained immediately from the OBDD for $W$, by switching the 0 and 1 sink nodes. In this section we will show how to use an MV-Index for $\neg W$, to compute efficiently $P_0(Q \land \neg W)$: we call this operation intersection. Denote $G_W = OBDD_{\Pi}(\neg W)$. Given $Q$, we first construct $G_Q = OBDD_{\Pi}(Q)$. Note that, although $\Pi$ is imposed by $W$, constructing $G_Q$ is usually quite efficient, because lineage of $Q$ is typically small. A naive next step would be to compute $G_Q \land G_W$, but this requires traversing the entire index. We briefly review our improvements over the naive algorithm next.

**MVIntersect** MVIntersect uses $G_Q = OBDD_{\Pi}(Q)$ to guide a search through $G_W = OBDD_{\Pi}(\neg W)$ and sum the probability of only those worlds that satisfy $Q$ via a top-down algorithm. One of the main challenges in doing a top-down intersection is maintaining a cache for memoization. This is well-studied (c.f. [52] for a top-down algorithm), so we don’t discuss it here. Since $G_W, G_Q$ have same variable order, our algorithm just traverses $G_W$ and prunes out branches where $G_Q$ is false. To do this we could implement a DFS traversal of $G_W$ and in the stack, we maintain the nodes from both $G_W$ and $G_Q$. Whenever we pop false from $G_Q$, we don’t traverse the subtree below. The stack here though can become very big and the node from $G_Q$ is almost always the same, since $G_Q$ is very small. MVIntersect exploits this by not adding the same node from $G_Q$ consecutively, but keeping a count of how many times the same query node has been pushed.

**CC-MVIntersect** Since our algorithm is main-memory and considering the increasing gap in the access time of cache and memory, we optimized our data structure to be more cache-conscious and minimize random accesses. A typical BDD data structure, for instance used in CUDD, is to store bdd nodes as pointers and each node contains the pointers to its neighbors. We improve upon it by keeping the bdd nodes in a vector sorted according to the DFS traversal of the obdd. Querying the obdd now can be done by a sequential traversal of this vector. The new bdd nodes though may need to store some additional information about their neighbors now. We call this approach CC-MVIntersect.

**Proposition 5.3.** Let $G_W$ be the OBDD for $\neg W$. Let $X_i$ and $X_j$ be the first and last Boolean variable (according to the permutation $\Pi$) occurring in the lineage of $Q$, and let $m = j - i + 1$. The CC-MVIntersect procedure runs in time $O(m \cdot w)$, where $w$ is the width of $G_W$.

Finally, we would like to point out that if $W$ is inversion-free then since its OBDD is of constant width, the runtime of CC-MVIntersect is linear in the span of the query $Q$, i.e. the
distance between the first and last variable in its lineage.

5.2 Experimental Evaluation

We report here our experimental evaluation of MVDBs, on real data obtained from [64]. We addressed four questions. How do MarkoViews, and indexed MarkoViews compare to other approaches for probabilistic inference on large Markov Networks? How effective is the MV-index construction algorithm compared to the standard approach for constructing OBDDs? How effective is the MV-index-based query evaluation method, how significant is the improvement of the CC-MVIntersect algorithm over MVIntersect? And, finally, how do MVDBs scale to the entire DBLP dataset?

Set Up For probabilistic inference, we compare our approach with Alchemy [98], the de-facto standard inference engine for MLN. For OBDD construction, we have extended CUDD [88], a widey-used OBDD package; for the OBDD experiments we compare native CUDD with our obdd construction algorithm. Our implementation is written in C++. We used Postgres 9.0 for our experiments which are run on a 2.66 GHZ Intel Core 2 Duo, with 4GB RAM, running Mac OS X 10.6. The dataset used is DBLP [64]. Figure 2.1 describes the DBLP data, the probabilistic table and the MarkoViews used to define the MVDB.

5.2.1 Comparison with Alchemy

To compare against Alchemy, we construct an MLN using features over Advisor$^p$ and Student$^p$ i.e., $V_1, V_2$, Figure 2.1. We did not use Affiliations$^p$, because the characters in the affiliation string violated Alchemy's requirement for constants. MLN do not allow features to have parameterized weights like in MVDBs, instead MLNs require a constant weight for each feature. One alternative is to materialize each feature into multiple features, one for each value of the weight; this would have increased the number of features. We opted instead to use constant weight in Alchemy, for simplicity. In MVDB we continued to use the weights exactly as described in Fig. 2.1.

Not surprisingly, Alchemy did not scale to the entire dblp dataset; we could scale it only up to $aid = 10,000$, in Student$^p(aid,\text{year})$, where $aid$ ranges upto 1M. In our experiments, we generate 10 datasets, with domain of $aid$ from 1 to $i \times 1000, i = 1 \ldots 10$. The lineage size of the MarkoView, i.e. tuples involved in the constraints (in other words the size of $\Phi$, Chapter 5) is plotted in Fig. 5.2. Over each such dataset we ran two queries of the form find the advisor of some student $X$, and find all students of some advisor $Y$. Figure 5.3 and Figure 5.4 show the
running time comparison of Alchemy vs MarkoViews. We show both the total execution time and also the time Alchemy reports it spent in just sampling. It is known that Alchemy is very inefficient during the grounding phase, and that database techniques can speed up this phase considerably [67], therefore the interesting line in Figure 5.3 is the lower line, \textit{Alchemy-sampling}, which is Alchemy’s reported sampling time, and is a better measure (likely a lower bound) on the total probabilistic inference time. The sampling method used is MC-Sat[73].

The results in Figure 5.3 and Figure 5.4 show that Alchemy’s MC-Sat algorithm is comparable (within a factor of 5, up or down) with evaluating the MarkoViews directly by constructing the OBDD. Note that OBDDs are an exact inference method while MC-Sat is an approximate method; and, also, OBDDs are not ideal for pure probabilistic inference, but are more appropriate for compilation. Once we use them for this purpose, by constructing an MV-index, the performance of the MarkoViews increases dramatically, and remains mostly constant as we increase the data size.

5.2.2 Comparison with CUDD

Here we studied the OBDD construction time, and compared it to the OBDD construction time as performed by native CUDD. The running time depends on the size of the resulting OBDD, and therefore we needed a feature that allowed us both to increase the size of the OBDD linearly, and for which the OBDD constructed by native CUDD was the same as that resulting from our optimization. The MarkoView \texttt{v2} in Figure 2.1 had both properties. As we varied linearly the domain of \texttt{aid1} in \texttt{Advisor}^p(\texttt{aid1,aid2}) from [1000] to [10000], the size of the resulting OBDD varied linearly, as shown in Figure 5.5. Furthermore, we checked that the size of the OBDDs returned by the two methods were indeed the same. Figure 5.6 shows that our algorithm was two orders of magnitude more efficient than CUDD’s OBDD construction. Since \texttt{aid1} is a separator, our approach exploits concatenation, which is much more efficient than standard OBDD synthesis used by CUDD. Since CUDD is effective at detecting equivalent nodes in the OBDD, it constructs the same OBDD, but it takes a lot of time to achieve the same result. We could not use CUDD to construct an OBDD for the entire DBLP dataset, even for \texttt{v2}: we estimate it would have taken several hours.

5.2.3 MV-index-based Query Evaluation

Recall that our query evaluation \( P_0(Q \lor W) \) is based on optimized intersection of the OBDDs for \( Q \) with the MV-index for \( W \). Here we compare two algorithms: MVIntersect and CC-MVIntersect (recall that that CC stands for cache-conscious). We used the same setting as in
Figure 5.2: Lineage Size of MV for each dataset

Figure 5.3: Alchemy vs MV for querying advisor of a student

Figure 5.4: Alchemy vs MV for querying all students of an advisor
Chapter 5. Empirical Evaluation of MVDB

**Figure 5.5: OBDD Size**

**Figure 5.6: Cudd vs MV : OBDD construction time**

**Figure 5.7: Querying time : MVIntersect vs CC-MVIntersect**
previous section. We used a simple query $Q$ whose lineage consisted of 20 tuples chosen as a worst case scenario: it forced the system to traverse entire MV-index, rendering all pre-computations and indices useless. Thus, query evaluation requires a complete intersection of the two OBDDs. Figure 5.7 shows the running time of the two algorithms. As expected, the running time varies linearly in the size of the MV-index (which, recall Figure 5.5, we designed carefully to be linear in the size of the data), because the entire OBDD must be traversed. The cache-conscious improves by a factor of 2 over the plain algorithm. We note that this is the worst case scenario; in typical cases, the query needs to traverse only a fraction of the MV-index, as will become clear next.

5.2.4 Scalability to a Large Dataset

We finally report our scalability results on the entire DBLP dataset, as described in Figure 2.1. The MarkoViews have a separator, hence the MV-index is obtained by concatenating many small OBDDs; their total size is 1.38M. Note that not all probabilistic tuples ended up in the MV-index, because some did not participate in any views. It took under one hour to construct the OBDD and index.

We evaluated 10 queries, of form find all students of an advisor X, and find the affiliation of a person Y. The running times are reported in Figure 5.8, Figure 5.9 respectively. In all cases we used the CC-MVIntersect. As one can see, the running times are below 5ms for all queries, and many are below 1ms. Query evaluation time includes the round trip call to Postgres, to compute the query’s lineage, then the time to access the OBDD index, which is a main memory data structure. Since each query includes a constant (the name of the advisor X, or the name of the person Y), only a small portion of the full OBDD had to accessed at runtime, which explains the performance of query evaluation. Note that all probability computations are exact: this is unlike Alchemy’s which are approximate.

5.2.5 Discussion

We have demonstrated how MV-indexes can be used to dramatically speed up query evaluation. The key ingredient that makes the index construction possible is the translation of the query evaluation from a Markov Network to a tuple-independent database, since OBDDs are only possible over independent variables. The OBDD construction is non-trivial. However, once the index is constructed, the performance of query evaluation becomes comparable to evaluating that query in postgres.
Future work is needed to assess the applicability of MVDBs, both in terms of modeling power and in terms of performance. Typical applications of MLN’s [74, 87, 75] use 4-10 features, most of which can already be expressed as MarkoViews, but they use much smaller data sets.
In this dissertation, we set out to address the challenge of uncertainty in modern databases. Probabilistic Databases that have been proposed to this end, so far, had to choose between representation power and query performance, because complex probabilistic models needed for practical applications required more involved probabilistic inference algorithms for query evaluation. Moreover, because most of these models were not specified relationally, it was tricky to apply existing relational database theory to understand them. We addressed this by proposing a probabilistic database, MVDB, that expresses correlations using views over the database. MVDB are also carefully designed so that query evaluation over them translates to evaluating a UCQ over independent tuples databases, INDB. The translation itself, on the other hand, is non-trivial; some of the resulting probabilities are negative! We hope that this translation will lead to better understanding of more complex real-world problems in the future.

Our second focus in this dissertation was a theoretical study of evaluating UCQ over INDB. This problem, also known as weighted model counting, had actually been well-studied in the literature. But, we approached this problem from the perspective of exploiting query structure. Note that we can do this because our translation shifted the complexity from the data model to query. In prior work, a lot of the focus was on evaluating very simple queries just because even doing that is challenging when dealing with a complicated model. We looked at a range of probabilistic inference tools: Read-Once Formulas (RO), OBDD, FBDD, d-DNNF, tree-decomposition. For RO, OBDD we gave a complete characterization of queries that are RO or have a compact OBDD. For FBDD, d-DNNF, we gave sufficient characterizations via novel non-trivial algorithms. In our empirical evaluation, we demonstrate how our algorithm for OBDD construction easily outperforms CUDD, the state-of-the-art software for OBDD by orders of magnitude. Our results have implications beyond our setting too: we showed that even over monotone formulas with polynomial size DNF: FBDD is separated from d-DNNF (i.e., there exists a formula with compact d-DNNF and no compact FBDD) and OBDD is separated from
FBDD. Although these separations were already known, the formulas separating these classes hitherto were non-monotone and did not have a small DNF either.

Because of the ubiquity of treewidth-based algorithms in probabilistic inference and AI literature in general, we investigated their application to our problem and their relation to other knowledge complication approaches. We have given a more general notion of treewidth for Boolean formulas that captures tractable classes for inference and is more general than the existing definitions. But in spite of that, we observed that they are strictly weaker than OBDD, when it comes to solving probabilistic inference over Boolean formulas in PTIME. We also studied the relationship between treewidth and OBDD for query classes UCQ and UCQ with inequalities (≠), and discovered that for former, bounded treewidth and polynomial size OBDD are actually equivalent concepts.

We have also proposed a framework for a query processor for probabilistic databases, based on our OBDD results. The empirical evaluation clearly demonstrates the advantage of pushing the complexity to the offline compilation phase. With suitable pre-computations and indices developed offline, one can considerably speedup the query evaluation. Our system vastly outperforms Alchemy, the de-facto inference engine for MLN. Another advantage of our approach is that because the data structure is built offline, for simple queries at least, the execution time is not completely unpredictable.

**Future Work**

We leave many avenues open for further future research. The first problem is inspired from our framework based on compiling queries offline. Even though we have used OBDD for their simplicity, there are other possible languages one could have used. Although, there has been a lot of work on compilation in each of these languages, unfortunately the focus of these work has been to find compact compilation. Another problem, which is arguably more important in our setting is: how does one evaluate probability over a compilation in such a language. This means exploring the right implementation, with indices, pre-computations and the comparing their trade-offs over practical scenarios. A good example would be the case of OBDD vs ADD (Algebraic Decision Diagram) [5]. An ADD is more suitable for the case of probabilistic inference, but they also have considerably bigger compilations. So, which method would work well, especially after OBDD have been extended with right pre-computations that make it more suitable for probabilistic inference?

Our theoretical results also pose many interesting open problems. A complete characterization of UCQ queries which admit a compact FBDD or d-DNNF is still open. The latter problem
is of special significance since it is still open whether there are formulas for which probabilistic inference is tractable, but which do not admit a compact d-DNNF. We believe there are such formulas and in particular, we have even conjectured a query in UCQ that achieves this. These results would follow as a corollary of such a characterization. But, even resolving the separation between d-DNNF and tractable inference would be significant. Our results on treewidth also leave open some problems, the most significant of which would be characterizing the complexity of determining expression treewidth. We also leave open the investigation of expression treewidth vis-a-vis other width-parameters such as clique-width and hypertree-width.

Even though our focus has been on probabilistic databases, our results have also found application in lifted inference. Lifted Inference seeks to come up with more efficient algorithms for probabilistic models that are defined using logic, by exploiting the logical structure of such models. Clearly, that has a lot of overlap with our theoretical results. In fact, we have already extended some of our results to propose a tractable subset of MLN [53] and to come up with more structure-cognizant sampling scheme for MLN [92]. Since MLN use a more general class of formulas(first-order), the problem here is to extend our exact algorithms to work over complicated MLN with some approximation.
BIBLIOGRAPHY


Abhay Jha was born in Bokaro Steel City, India. He earned his B.Tech. from IIT Bombay in 2006 and M.S. from University of Washington in 2008. In 2012, he could finally place Dr. in front of his name by earning a Doctor of Philosophy degree from University of Washington in Computer Science and Engineering.