MORGAN & CLAYPOOL PUBLISHERS

Generating Plans from Proofs

The Interpolation-based Approach to Query Reformulation

Michael Benedikt
Julien Leblay
Balder ten Cate
Efthymia Tsamoura

SYNTHESIS LECTURES ON DATA MANAGEMENT

Z. Meral Özsoyoğlu, *Series Editor*

Generating Plans from Proofs

The Interpolation-based Approach to Query Reformulation

Synthesis Lectures on Data Management

Editor
Z. Meral Özsoyoğlu, ***Case Western Reserve University***

Synthesis Lectures on Data Management is edited by Meral Özsoyoğlu of Case Western Reserve University. The series publishes 80- to 150-page publications on topics pertaining to data management. Topics include query languages, database system architectures, transaction management, data warehousing, XML and databases, data stream systems, wide scale data distribution, multimedia data management, data mining, and related subjects.

Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation
Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura
2015

Veracity of Data: From Truth Discovery Computation Algorithms to Models of Misinformation Dynamics
Laure Berti-Équille and Javier Borge-Holthoefer
2015

Datalog and Logic Databases
Sergio Greco and Cristina Molinaro
2015

Big Data Integration
Xin Luna Dong and Divesh Srivastava
2015

Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, and Media Restore
Goetz Graefe, Wey Guy, and Caetano Sauer
2014

Similarity Joins in Relational Database Systems
Nikolaus Augsten and Michael H. Böhlen
2013

Information and Influence Propagation in Social Networks
Wei Chen, Laks V.S. Lakshmanan, and Carlos Castillo
2013

Data Cleaning: A Practical Perspective
Venkatesh Ganti and Anish Das Sarma
2013

Data Processing on FPGAs
Jens Teubner and Louis Woods
2013

Perspectives on Business Intelligence
Raymond T. Ng, Patricia C. Arocena, Denilson Barbosa, Giuseppe Carenini, Luiz Gomes, Jr., Stephan Jou, Rock Anthony Leung, Evangelos Milios, Renée J. Miller, John Mylopoulos, Rachel A. Pottinger, Frank Tompa, and Eric Yu
2013

Semantics Empowered Web 3.0: Managing Enterprise, Social, Sensor, and Cloud-based Data and Services for Advanced Applications
Amit Sheth and Krishnaprasad Thirunarayan
2012

Data Management in the Cloud: Challenges and Opportunities
Divyakant Agrawal, Sudipto Das, and Amr El Abbadi
2012

Query Processing over Uncertain Databases
Lei Chen and Xiang Lian
2012

Foundations of Data Quality Management
Wenfei Fan and Floris Geerts
2012

Incomplete Data and Data Dependencies in Relational Databases
Sergio Greco, Cristian Molinaro, and Francesca Spezzano
2012

Business Processes: A Database Perspective
Daniel Deutch and Tova Milo
2012

Data Protection from Insider Threats
Elisa Bertino
2012

Deep Web Query Interface Understanding and Integration
Eduard C. Dragut, Weiyi Meng, and Clement T. Yu
2012

P2P Techniques for Decentralized Applications
Esther Pacitti, Reza Akbarinia, and Manal El-Dick
2012

Query Answer Authentication
HweeHwa Pang and Kian-Lee Tan
2012

Declarative Networking
Boon Thau Loo and Wenchao Zhou
2012

Full-Text (Substring) Indexes in External Memory
Marina Barsky, Ulrike Stege, and Alex Thomo
2011

Spatial Data Management
Nikos Mamoulis
2011

Database Repairing and Consistent Query Answering
Leopoldo Bertossi
2011

Managing Event Information: Modeling, Retrieval, and Applications
Amarnath Gupta and Ramesh Jain
2011

Fundamentals of Physical Design and Query Compilation
David Toman and Grant Weddell
2011

Methods for Mining and Summarizing Text Conversations
Giuseppe Carenini, Gabriel Murray, and Raymond Ng
2011

Probabilistic Databases
Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch
2011

Peer-to-Peer Data Management
Karl Aberer
2011

Probabilistic Ranking Techniques in Relational Databases
Ihab F. Ilyas and Mohamed A. Soliman
2011

Uncertain Schema Matching
Avigdor Gal
2011

Fundamentals of Object Databases: Object-Oriented and Object-Relational Design
Suzanne W. Dietrich and Susan D. Urban
2010

Advanced Metasearch Engine Technology
Weiyi Meng and Clement T. Yu
2010

Web Page Recommendation Models: Theory and Algorithms
Sule Gündüz-Ögüdücü
2010

Multidimensional Databases and Data Warehousing
Christian S. Jensen, Torben Bach Pedersen, and Christian Thomsen
2010

Database Replication
Bettina Kemme, Ricardo Jimenez-Peris, and Marta Patino-Martinez
2010

Relational and XML Data Exchange
Marcelo Arenas, Pablo Barcelo, Leonid Libkin, and Filip Murlak
2010

User-Centered Data Management
Tiziana Catarci, Alan Dix, Stephen Kimani, and Giuseppe Santucci
2010

Data Stream Management
Lukasz Golab and M. Tamer Özsu
2010

Access Control in Data Management Systems
Elena Ferrari
2010

An Introduction to Duplicate Detection
Felix Naumann and Melanie Herschel
2010

Privacy-Preserving Data Publishing: An Overview
Raymond Chi-Wing Wong and Ada Wai-Chee Fu
2010

Keyword Search in Databases
Jeffrey Xu Yu, Lu Qin, and Lijun Chang
2009

Copyright © 2016 by Morgan & Claypool

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews, without the prior permission of the publisher.

Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation

Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura

www.morganclaypool.com

ISBN: 9781627056892 paperback
ISBN: 9781627057202 ebook

DOI 10.2200/S00703ED1V01Y201602DTM043

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON DATA MANAGEMENT

Lecture #43
Series Editor: Z. Meral Özsoyoğlu, *Case Western Reserve University*
Founding Editor: M. Tamer Özsu, *University of Waterloo*
Series ISSN
Print 2153-5418 Electronic 2153-5426

Generating Plans from Proofs

The Interpolation-based Approach to Query Reformulation

Michael Benedikt
Oxford University

Julien Leblay
National Institute of Advanced Industrial Science and Technology (AIST), Tokyo

Balder ten Cate
Google, Inc.

Efthymia Tsamoura
Oxford University

SYNTHESIS LECTURES ON DATA MANAGEMENT #43

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Query reformulation refers to a process of translating a source query—a request for information in some high-level logic-based language—into a target plan that abides by certain interface restrictions. Many practical problems in data management can be seen as instances of the reformulation problem. For example, the problem of translating an SQL query written over a set of base tables into another query written over a set of views; the problem of implementing a query via translating to a program calling a set of database APIs; the problem of implementing a query using a collection of web services.

In this book we approach query reformulation in a very general setting that encompasses all the problems above, by relating it to a line of research within mathematical logic. For many decades logicians have looked at the problem of converting "implicit definitions" into "explicit definitions," using an approach known as interpolation. We will review the theory of interpolation, and explain its close connection with query reformulation. We will give a detailed look at how the interpolation-based approach is used to generate translations between logic-based queries over different vocabularies, and also how it can be used to go from logic-based queries to programs.

KEYWORDS

data integration, query optimization, query reformulation, views, tableau, Craig interpolation, Beth definability

Contents

Preface xv

Acknowledgments xix

1 Introduction 1

1.1 Overview 1
1.2 First-order Logic and Databases 7
1.3 Entailment and Proofs 19
1.4 Summary 28
1.5 Bibliographic Remarks 29

2 Vocabulary-based Target Restrictions 31

2.1 Reformulating Queries Using Interpolation 31
2.1.1 From a Semantic Property to a First-order Reformulation 32
2.1.2 Craig Interpolation and Beth Definability 34
2.1.3 Handling Equality 40
2.2 Relativized-quantifier Interpolation 46
2.3 Positive Existential Reformulation 51
2.4 Existential Reformulation 54
2.5 The Methodology in Action 56
2.6 Safety of Reformulations 59
2.7 Decidable Reformulation 61
2.7.1 Decidable End-to-end Reformulation for Expressive Constraints 62
2.7.2 Reformulation with Inclusion Dependencies 65
2.7.3 TGDs with Terminating Chase: Positive Existential Reformulation . . . 66
2.7.4 TGDs with Terminating Chase: RQFO Reformulation 68
2.8 Finite Instances and Restricted Constraints 69
2.9 Summary 74
2.10 Bibliographic Remarks 74

3 Access Methods and Integrity Constraints . . . 79
3.1 Basics of Target Restrictions Based on Access Methods . . . 79
3.2 Nested Plans . . . 83
3.3 Expressiveness of Plan Languages . . . 85
3.3.1 Relationship of $USPJAD$-plans to $USPJAD$ Queries . . . 92
3.3.2 Relationship of $USPJAD$-plans to Other Formalisms . . . 93
3.4 Semantic Properties and Entailments Related to Plans . . . 94
3.5 Statement of the Main Results on Access Determinacy and Reformulation . . . 102
3.6 Access Interpolation . . . 104
3.7 Proving the Access Interpolation Theorem . . . 110
3.8 Extension to Non-boolean Queries . . . 112
3.9 Decidable Plan-generation . . . 114
3.9.1 The Case of Inclusion Dependencies . . . 114
3.9.2 Constraints with Terminating Chase . . . 115
3.10 Finite Instances and Access Restrictions . . . 116
3.11 Summary . . . 118
3.12 Bibliographic Remarks . . . 118

4 Reformulation Algorithms for TGDs . . . 121
4.1 Finding Plans Through Chase Proofs . . . 122
4.2 Plan Search Algorithms . . . 126
4.3 Properties of SPJ Plan-generation . . . 126
4.4 RA-plans for Schemas with TGDs . . . 130
4.4.1 Proof to RA-plan Algorithm . . . 130
4.4.2 Correctness of the Algorithm . . . 134
4.5 Chase-based and Interpolation-based Plan-generation . . . 139
4.6 Summary . . . 142
4.7 Bibliographic Remarks . . . 142

5 Low-cost Plans Via Proof Search . . . 145
5.1 Cost Functions on Plans . . . 145
5.2 How Good are Proof-based Plans? . . . 148
5.2.1 Optimality in Terms of Methods Used . . . 148
5.2.2 Optimality of Proof-based Plans in Runtime Accesses . . . 151
5.3 Simultaneous Proof and Plan Search . . . 154
5.4 Beyond Prefix Proofs and Left-deep Plans . . . 162

5.5 Summary 166
5.6 Bibliographic Remarks 168

6 Conclusion 171

Bibliography 173

Authors' Biographies 179

Index 181

Preface

Query reformulation. Query reformulation refers to a process of translating a *declarative source query* into a *target plan* that abides by certain *interface restrictions*, restrictions that the source query may not satisfy. By a source query we mean some request for information in a high-level logic-based language. For example a query asking for the names of the advisors of a university student called "Smith" would be written in the standard database language SQL as

```
SELECT profname FROM Professor, Student
WHERE Student.advisorid = Professor.profid
AND Student.lname = "Smith"
```

and in first-order logic as:

$$\{\text{profname} \mid \exists \text{profid}\ \exists \text{dname}\ \exists \text{studid}\ \text{Professor}(\text{profid}, \text{profname}, \text{dname}) \wedge \text{Student}(\text{studid}, \text{"Smith"}, \text{profid})\}$$

Here it is assumed that the user posing the query thinks of the information in terms of two tables, Student and Professor. Student contains the student id and last name of each student along with the id of their advisor, while Professor contains entries for the id, last name, and department of each professor.

What kind of translation might we perform on an expression like the one above? It might be that to answer the source query it is necessary to access information stored in a different format. The stored data may have a table Professor′ where the professor's id attribute is dropped, and a table Student′ where the advisor's id is replaced with an attribute advisorname giving the advisor's last name. In order to retrieve the information over these reformatted sources, the query should be transformed. It is easy to see that in this case the correct transformation is just to get the advisorname attribute of rows corresponding to "Smith" in Student′. In SQL the translation would be:

```
SELECT advisorname FROM Student′ WHERE Student′.lname = "Smith"
```

and in first-order logic it would be:

$$\{\text{advisorname} \mid \exists \text{studid}\ \text{Student}'(\text{studid}, \text{"Smith"}, \text{advisorname})\}$$

In order to say that this represents a correct translation of the source query we need to know something about the semantics of the data. For us this will be captured by *integrity constraints*. In the above example, integrity constraints would describe the relationship between the accessible

tables (Student′ and Professor′) and the tables mentioned in the source query (Student and Professor). Relative to those constraints, the SQL and logic translations above are correct.

Our notion of a target plan is very broad. We could be translating from one high-level query to another, as in the example above. We also consider translations from a high-level query to something operational, like a low-level program that makes calls to data access APIs. A basic function of a database management system is to translate a high-level language (e.g. first-order logic) to a low-level program. The goal there is to produce not just any equivalent program, but an efficient one. We will therefore look at the impact of efficiency considerations on reformulation. How to measure the efficiency of plans will not be our concern here—there is a rich research literature on the subject. We will instead be interested in algorithms that can return low-cost plans without specialized knowledge of the cost functions.

Reformulation via interpolation. Reformulating queries over restricted interfaces may sound very remote from concerns in mathematics. But it turns out that this problem is closely connected to a long line of research within mathematical logic. This book will provide an overview of the connection, explaining how ideas from logic can solve all of the reformulation problems above (and more). For each type of reformulation we will isolate a *semantic property* that any input query Q must have with respect to the target language and integrity constraints in order for the desired reformulation to exist. We then express this property as a *proof goal*: a statement that one logical formula follows from another. We will explain how to translate reformulation tasks into proof goals.

Reformulation proceeds by searching for a proof that witnesses the goal. From the proof we will then extract an *interpolant*, a logical formula that contains "only the necessary information" for the proof. We show that interpolants can be converted into reformulations through a very simple algorithm.

This "recipe" for reformulation dates back to work of the logician William Craig in the late 1950s. We show that it applies to a wide variety of reformulation scenarios. It is not a magic bullet that can always produce practical reformulation algorithms, but it often provides algorithms with optimal worst-case complexity, and it can be coupled with techniques for proof search and minimization of reformulations to become competitive with other reformulation techniques. We will explain the interpolation-based approach first for vocabulary-based restrictions, then for access-method based restrictions, and finally in the presence of cost information. We proceed in each case by explaining how the method is applied, then proving theorems stating that the resulting technique is complete—if a reformulation exists, the method will find it—and finally analyzing the worst-case complexity of the resulting algorithms.

About the book. This book has a number of objectives. It aims to explain formally what the interpolation-based method is, to exhibit the diverse ways in which it can be applied, and to explain the properties of the reformulations produced by the method. We also want to relate

the interpolation-based approach to prior work on generating implementations from high-level queries.

This book has the most obvious interest to theoretically minded computer scientists. The focus throughout is on theorems: characterizations of reformulation (e.g., when does a source query have a reformulation of a certain kind?), expressiveness results (can a source query have a reformulation in one class, but no reformulation in another class?), and complexity bounds (what is the complexity of finding a reformulation in a certain class?). We connect our theorems to lines of research within a number of communities within theoretical computer science, particularly database theory, finite model theory, and knowledge representation. In a few cases, we state a theorem but omit the verification, pointing the reader to a paper where the full proofs appear. But the main results are proven in detail in order to present the theory in a self-contained manner. For many of the results, complete proofs have never appeared in print prior to this work.

A second audience for the book consists of researchers in logic. They will be very familiar with basic results about interpolation, along with the related topic of going from implicit definitions to explicit ones, but perhaps not with either the theory or the practice of databases. We aim to introduce logicians to the application of interpolation in data management. We hope that the results here give a new constructive perspective on the relationship between syntax and semantics, a major theme of research in both theoretical computer science and model theory. This book can be seen as working out more practical consequences of what are called "preservation theorems" in first-order model theory —theorems that characterize subclasses of first-order logic via semantic properties.

Finally, we hope that parts of the text will be of interest to researchers in databases, even those who do not work in theory. Chapter 4 and Chapter 5 are the most accessible parts of the text for researchers in data integration and query optimization with a more applied background. These two chapters deal with algorithms that can be understood without reference to interpolation, and without a background in first-order logic.

In trying to give a comprehensive picture of the theory of reformulation, we have completely omitted a host of issues that are critical in practice. For example:

- We deal only with set semantics for queries, not the bag semantics used in SQL.
- We consider only first-order queries, without considering aggregates like COUNT and SUM that play a crucial role in many database applications.
- Our model of data is "un-typed," assuming every column takes values from a fixed infinite set. We assume this infinite set has no structure that can be referenced in queries or constraints. Thus we do not allow queries and constraints that can mention integer inequality or arithmetic, string concatenation or substring comparisons, all of which appear in constraints and queries in practice.
- We do not cover all the integrity constraints that are important in practice. We present some general results about reformulation with arbitrary first-order logic constraints, which

are applicable to all common SQL schema constraints, including referential constraints and key constraints. We obtain decidability and complexity results for reformulation, for some limited constraint classes. But we omit an analysis of a few classes that are significant for database applications. For example, we do not give any special attention to equality-generating dependencies, which subsume the key constraints that play a fundamental role in SQL.

- We consider the problem of getting low-cost reformulations, but our theoretical results apply only to the case of very simplistic cost functions. We do not analyze realistic cost functions that are used in the database or the web data integration setting.

Many of these pragmatic issues are discussed in an earlier textbook [Toman and Weddell, 2011]. Others (like aggregation) represent difficult open problems for any theory of reformulation.

Although the book is focused on theory, we try to give a sense of how the interpolation-based framework is useful in practice. Thus throughout the book we present examples of the results in (simplified) application scenarios, and give pointers to further work concerning systems based on the theory.

Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura
February 2016

Acknowledgments

The authors are very grateful to the following people for reviewing early copies of the manuscript and providing extensive comments: Antoine Amarilli, Cristian Riveros, Michael Vanden Boom, and James Worrell. We are also thankful to Carsten Lutz, Jan Van den Bussche, and Victor Vianu for their detailed feedback on the first official draft of the book. Benedikt's work was sponsored by the Engineering and Physical Sciences Research Council of the United Kingdom, grants EP/M005852/1 and EP/L012138/1.

Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura
February 2016

CHAPTER 1

Introduction

This chapter starts by explaining the topics discussed in the book through examples. It then outlines the overall structure of the book, explaining how each chapter relates to the motivating examples. The remainder of the chapter reviews the basic definitions concerning the database models, constraint languages, and query languages used in the text. In the process we will go over some "fine print" concerning the relationship between the languages and semantics used in mathematical logic and those used in databases.

1.1 OVERVIEW

This book is about translating a *declarative source query* into a declarative query or procedural plan in a *target language* that abides by certain *interface restrictions*. By a declarative source query we always mean a request for information from a database specified using a formula of first-order logic, or its equivalent in the database query language SQL. We often focus on source queries given in the language of *conjunctive queries*, which correspond to a very simple subset of SQL. In logical terms these correspond to formulas built up using only conjunction and existential quantification. By a plan we may mean either another declarative query, or something more operational, such as a relational algebra expression, possibly using some restricted set of physical operators.

Translating from a declarative source query to a target in this broad sense captures a huge number of activities that take place within data management. One of the basic features of modern database systems is to decouple the vocabulary employed by users to ask questions about a dataset from the data structures that are used to implement data access: the "logical model" and the "physical model." In relational databases, the logical model might be a set of tables given an SQL schema: users interact with the data by sending SQL queries that mention these tables. The database manager performs a translation of the queries written over this high-level vocabulary into a program that interacts with the data using some set of data access functions.

In this "classical query evaluation" scenario, the relationship of the source vocabulary to the target vocabulary is very simple: every source relation corresponds to one or more "physical datasource" relations or functions. A more complicated example comes from data integration systems. The goal there is to mask a diverse set of datasources by providing a single unified schema. Suppose we have a multitude of "local databases" with information about different hotel chains. A data integration system would present to users a "global schema"—an easier-to-understand vocabulary that can model information in all of the sources. The system accepts queries written

over the global schema and translates them into queries over the various local sources. The "global database" is virtual, implicitly defined by its relationship with local data.

In this book we will look at only a few flavors of interface restrictions that represent targets for a translation. The most basic kind of restriction we look at is a *vocabulary-based restriction*. There the target is another declarative query, but we limit the query's vocabulary: we begin with a query Q written using some relations $R_1 \dots R_j$, and want to convert it to a query $Q_{\vec{V}}$ making use of a different set of relations $V_1 \dots V_k$. If the queries Q and $Q_{\vec{V}}$ mention different relations, we cannot expect Q to give the same answers as $Q_{\vec{V}}$ on arbitrary instances. But our schema will come with *integrity constraints* which restrict the possible instances of interest. We will thus be considering equivalence only on instances satisfying the constraints.

An example of vocabulary-based restriction comes from *reformulating queries over views*. We have a collection of view relations $V_1 \dots V_k$, and each V_i is associated with a query Q_i defined using some other set of relations $R_1 \dots R_j$. Given a query Q defined over $R_1 \dots R_j$, the goal is to find an equivalent query $Q_{\vec{V}}$ that mentions only $V_1 \dots V_k$. Generally, additional restrictions will be put on $Q_{\vec{V}}$. For example, it should be a conjunctive query, or a union of conjunctive queries, or a relational algebra query. The view-based query reformulation problem can thus be seen as a special case of vocabulary-based restriction, where the integrity constraints are of the form:

$$\forall x_1 \dots \forall x_n \; V_i(x_1 \dots x_n) \leftrightarrow Q_i(x_1 \dots x_n).$$

Example 1.1. A university database has a table Professor containing ids and last names of professors, along with the name of the professor's department. It also has a table Student listing the id and last name of each student, along with their advisor's id.

The database does not allow users to access the Professor and Student tables directly, but instead exposes a view Professor′ where the id attribute is dropped, and a table Student′ where the advisor's id is replaced with the advisor's last name.

That is, Professor′ is a view defined by the formula:

$$\{\, \mathsf{lname},\ \mathsf{dname} \mid \exists\, \mathsf{profid}\ \mathsf{Professor}(\mathsf{profid},\ \mathsf{lname},\ \mathsf{dname})\}$$

or equivalently by the constraints:

$$\forall \mathsf{profid}\ \forall \mathsf{lname}\ \forall \mathsf{dname}\ \mathsf{Professor}(\mathsf{profid}, \mathsf{lname}, \mathsf{dname}) \rightarrow \mathsf{Professor}'(\mathsf{lname}, \mathsf{dname})$$
$$\forall \mathsf{lname}\ \forall \mathsf{dname}\ \mathsf{Professor}'(\mathsf{lname}, \mathsf{dname}) \rightarrow \exists \mathsf{profid}\ \mathsf{Professor}(\mathsf{profid}, \mathsf{lname}, \mathsf{dname})$$

Student′ is a view defined by the formula:

$$\{\mathsf{studid}, \mathsf{lname}, \mathsf{profname} \mid$$
$$\exists \mathsf{profid}\ \exists \mathsf{dname}\ \mathsf{Student}(\mathsf{studid}, \mathsf{lname}, \mathsf{profid}) \wedge \mathsf{Professor}(\mathsf{profid}, \mathsf{profname}, \mathsf{dname})\}$$

or equivalently by constraints:

$$\forall \mathsf{studid}\ \forall \mathsf{lname}\ \forall \mathsf{dname}\ \forall \mathsf{profid}\ \forall \mathsf{profname}\ [\mathsf{Professor}(\mathsf{profid}, \mathsf{profname}, \mathsf{dname}) \wedge \mathsf{Student}(\mathsf{studid}, \mathsf{lname}, \mathsf{profid}) \rightarrow \mathsf{Student}'(\mathsf{studid}, \mathsf{lname}, \mathsf{profname})]$$

$$\forall \mathsf{studid}\ \forall \mathsf{lname}\ \forall \mathsf{profname}\ [\mathsf{Student}'(\mathsf{studid}, \mathsf{lname}, \mathsf{profname}) \rightarrow \exists \mathsf{profid}\ \exists \mathsf{dname}\ \mathsf{Professor}(\mathsf{profid}, \mathsf{profname}, \mathsf{dname}) \wedge \mathsf{Student}(\mathsf{studid}, \mathsf{lname}, \mathsf{profid})]$$

Consider the query asking for the names of the advisors of a given student. We can answer this by simply using the $\mathsf{Student}'$ view, returning the profname attribute of a tuple in the view. But a query asking for the last names of all students that have an advisor in the history department can *not* be answered using the information in the views: knowing the advisor's name is not enough to identify the department. ◁

Integrity constraints need not be restricted to expressing view definitions. A natural use of constraints is to represent *relationships between sources*, such as overlap in the data. This overlap can be exploited to take a query specified over a source that *a priori* does not have sufficient data, and reformulate it over a source that provides the necessary data.

Example 1.2. We look at an example schema from [Onet, 2013]. It has a relation Employee containing the id, name, and department id of each employee, and also a relation Department, containing the department's id, name, as well as the id of the department's manager. The schema contains the constraints:

$$\forall \mathsf{deptid}\ \forall \mathsf{dname}\ \forall \mathsf{mgrid}\ [\mathsf{Department}(\mathsf{deptid}, \mathsf{dname}, \mathsf{mgrid}) \rightarrow \exists N\ \mathsf{Employee}(\mathsf{mgrid}, N, \mathsf{deptid})]$$
$$\forall \mathsf{eid}\ \forall \mathsf{ename}\ \forall \mathsf{deptid}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid}) \rightarrow \exists D \exists M\ \mathsf{Department}(\mathsf{deptid}, D, M)$$

That is, every manager of a department works in it, and every employee works in a department. Suppose further that only the relation Department is accessible to a certain class of users. Intuitively, it should still be possible to answer some questions that one could ask concerning the relation Employee, making use of the accessible relation Department. Suppose a user poses the query asking for all department ids of employees, writing it like this:

$$Q = \{\mathsf{deptid} \mid \exists \mathsf{eid}\ \exists \mathsf{ename}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid})\}$$

The query can be reformulated as:

$$Q_{\mathsf{Department}} = \{\mathsf{deptid} \mid \exists \mathsf{dname}\ \exists \mathsf{mgrid}\ \mathsf{Department}(\mathsf{deptid}, \mathsf{dname}, \mathsf{mgrid})\}$$

Intuitively, the constraints ensure that the set of department id values in the Department and Employee tables are the same. ◁

Access methods. A finer notion of interface than vocabulary-based restrictions is based on *access methods*, which state that a relation can only be accessed by lookups where certain arguments must be given. One example of a restricted interface based on access methods comes from web forms. Thinking of the form as exposing a virtual table, the mandatory fields must be filled in by the user, while submitting the form returns all tuples that match the entered values. Other examples include web services and legacy databases.

Example 1.3. Consider a Profinfo table available via a web form, containing information about faculty, including their last names, office number, and employee id, but with a restricted interface that requires giving an employee id as an input. The interface could be implemented as a web form that requires entering an employee's id and then pressing a submit button to get the matching records. The query Q asking for ids of faculty named "Smith" cannot be "completely answered" over this schema:[1] that is, there is no function over the available data in this schema which is equivalent to Q.

But suppose another web form provides access to a Udirectory table containing the employee id and last name of every university employee, with an interface that allows one to access the entire contents of the table. Then we can reason that Q is answerable using the information in this schema: a plan would pull tuples from the Udirectory form and check them within the Profinfo form to see if they correspond to a faculty member. ◁

In Example 1.3, reasoning about access considerations was straightforward, but in the presence of more complex schemas, we may have to chain several inferences, resulting in a plan that makes use of several auxiliary accesses.

Example 1.4. Suppose we have two directory data sources with overlapping information. One source exposes information from $\mathsf{Udirectory}_1(\mathsf{uname}, \mathsf{addr}, \mathsf{uid})$ via an access method requiring a uname and uid. There is also a "web table" Ids(uid) with no access restriction, that makes available the set of uids (hence we have a "referential integrity constraint" saying that every uid in $\mathsf{Udirectory}_1$ matches a uid in Ids). The other source exposes $\mathsf{Udirectory}_2(\mathsf{uname}, \mathsf{addr}, \mathsf{phone})$, requiring a uname and addr, and also a web table Names(uname) with no access restriction that reveals all unames in $\mathsf{Udirectory}_2$ (that is, a constraint that each uname in $\mathsf{Udirectory}_2$ appears in Names). There is also a constraint saying that each uname and addr in $\mathsf{Udirectory}_2$ appears in $\mathsf{Udirectory}_1$. The schema is shown in Figure 1.1, with the arrows indicating referential constraints and underlining indicating the input positions of each relation. A query asking for all phone numbers in the second directory could be written:

$$Q = \{\mathsf{phone} \mid \exists \mathsf{uname}\ \exists \mathsf{addr}\ \mathsf{Udirectory}_2(\mathsf{uname}, \mathsf{addr}, \mathsf{phone})\}.$$

There is a plan that implements this query: it gets all the uids from Ids and unames from Names first, puts them into the access on $\mathsf{Udirectory}_1$, then uses the uname and addr of the resulting tuples to get the phone numbers in $\mathsf{Udirectory}_2$. ◁

[1] We will formalize the notion of "completely answerable" in Chapter 3.

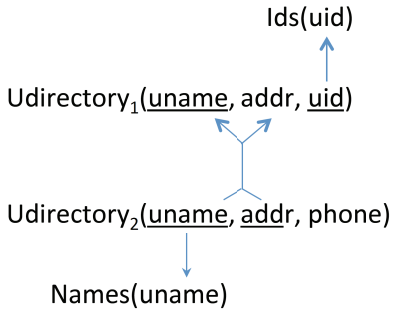

Figure 1.1: The schema for Example 1.4.

We emphasize that our goal in this work is getting plans which give *complete answers to queries*. This means that if we have a query asking for the office number of all professors with last name "Smith," the plan produced should return all tuples in the answer, even if access to the Professor relation is limited. This contrasts with a line of work in data integration that considers the broader question of how to answer any query "as much as possible given the available data:" how to get the *certain answers* or how to compute the *maximally contained query* [Lenzerini, 2002].

In the last part of the book, we look not just at getting any plan in the target language, but one with low cost. In doing this, we are moving closer to the traditional concerns of query optimization in database systems. Examples of access cost include the cost in money of accessing certain services and the cost in time of accessing data through either web service calls, iteratively inputting into web forms, or using particular indices.

Reformulation via interpolation. This text will provide an overview of a general approach that has emerged from the computational logic and database communities. The idea is to go from "semantics to syntax" by means of interpolation algorithms. The "meta-algorithm" for reformulation is as follows:

1. Isolate a *semantic property* that any input query Q must have with respect to the target $\mathcal{T}$ and constraints Σ in order to have a reformulation of the desired type.

2. Express this property as a *proof goal* (in the language we use later on, an *entailment*): a statement that a certain formula follows from another formula.

3. Search for a proof of the entailment within a given proof system. Here we focus on tableau proofs, a well-known proof system within computational logic.

4. From the proof, extract an *interpolant* using an interpolation algorithm. We will review some standard interpolation algorithms and also present new ones.

5. Convert the interpolant to a reformulation.

This approach is very general and can be applied to a variety of proof systems and restrictions on the target, with different target languages corresponding to different entailments. We prove theorems saying that the method is complete: there is a reformulation exactly when there is a proof of the semantic property. These completeness theorems give as a consequence a *definability or preservation theorem*: a query Q has a certain kind of reformulation if and only if it has a certain semantic property. Such theorems are well-known in model theory, and indeed our theorems can be seen as "database versions" of the preservation theorems that are known from classical model theory textbooks, e.g., [Chang and Keisler, 1990].

The fact that interpolation theorems can be used to prove preservation theorems is not new. As early as the 1950s, the logician William Craig used interpolation to prove a definability theorem, and all of our results can be seen as generalizations of Craig's technique. The connection between interpolation and preservation has been explored more recently by Otto [Otto, 2000]. It is also known that interpolation theorems can be adapted from the classical semantics of first-order logic to the setting used in databases, and can be used to characterize when a query can be reformulated over a certain target. This idea goes back to the work of Segoufin and Vianu [Segoufin and Vianu, 2005].

What is less well-known is that this connection between reformulation, interpolation, and preservation can be made effective: *interpolation algorithms yield reformulation algorithms*. We will show that the interpolation technique yields many of the algorithmic results about reformulation mentioned in the database setting. We will also demonstrate that it produces reformulation algorithms in new settings.

Thus our results will link: a semantic property of Q (with respect to $\mathcal{T}$ and Σ), a reformulation property of Q, and a proof goal and interpolation algorithm that achieves the reformulation.

Organization of the book. In the remainder of this chapter we give preliminaries on database schemas, query languages, and logics. Much of it can be found in database textbooks. But we also include some less standard material on the relationship of logic to databases, and present a particular proof system for first-order logic which will play a significant role in our approach.

Chapter 2 begins the exposition of the interpolation-based approach in the setting of vocabulary-based restrictions. It presents theorems concerning the ability to generate reformulations in a variant of first-order logic. It proceeds to refinements of the technique that guarantee that the reformulation will satisfy additional properties. The remainder of the chapter analyzes the decidability and complexity of the reformulation algorithms.

In Chapter 3 we extend interpolation-based reformulation to the setting in which the target is given by access methods. We apply the same approach as in the vocabulary-based case, providing an entailment that captures a property of the input query that is necessary for a reformulation to exist, and then applying interpolation to generate the reformulation from a proof witnessing the entailment. This requires a new interpolation theorem, generalizing one previously stated in [Otto, 2000]. The exposition parallels that in the previous chapter, starting with general

theorems for first-order logic and then analyzing the complexity of reformulation for restricted classes of constraints.

The previous chapters show that one can use interpolation to perform reformulation in theory. But when can we make the reformulation process effective, and when can we make it efficient? We focus on more practical concerns in Chapter 4. We show that for a wide class of constraints used in databases, tuple-generating dependencies, interpolation-based reformulation reduces to a particularly simple algorithm. This setting will also allow us to connect the interpolation-based approach to the main method for reformulation discussed in the prior database literature, based on *the chase* (see Section 1.3).

In Chapter 5 we turn to getting reformulations that are efficient. In the setting of overlapping data sources, there can be many plans with different costs. The chapter presents a method for finding the lowest-cost plan. The main idea is to *explore the full space of proofs, but guiding the search by cost as well as proof structure.* Thus instead of generating a single proof and then sending the corresponding plan on for further optimization, we interleave exploration of proofs with calls to estimate cost (and perhaps further optimize) the corresponding plans.

1.2 FIRST-ORDER LOGIC AND DATABASES

This book applies techniques from mathematical logic to databases. We will thus need to discuss differences between the syntax and semantics of classical mathematical logic and the languages used in databases, and how we can nevertheless apply mathematical logic techniques in the database setting.

Schemas. In order to describe a querying scenario we will use a *basic schema*, which consists of:

- A finite collection of *relation names* (or simply *relations* henceforward), with each relation R associated with an *arity*, denoted $\mathsf{arity}(R)$. A *position* of a relation R is a number between 1 and $\mathsf{arity}(R)$. A relation will also be referred to as a *table*.

- A finite collection of *schema constants* ("Smith", 3, ...). Informally, these represent a fixed set of values that a user might use within accesses, in addition to values that emerge from the database. For example, if the user is performing a query involving the string "Smith," we would assume that "Smith" was a schema constant—but not arbitrary unrelated strings. Throughout this work, *we will assume that all constants used in queries or constraints are schema constants*. Several results, like Theorem 1.4, depend upon this.

- A collection of *integrity constraints*, which will be sentences in some logic.

Syntax of first-order logic. We use standard terminology to describe formulas in first-order logic without function symbols, including the notion of free variable, quantifiers, and connectives [Abiteboul et al., 1995].

First-order logic is built up from *atomic formulas*, which can be either:

- *relational atoms* of the form $R(\vec{t})$, where R is a relation name and each t_i in $\vec{t}$ is either a constant or a variable;
- *equality atoms* of the form $t_i = t_j$, with t_i, t_j either a constant or a variable.

Formulas include atomic formulas and are closed under boolean operations, with formation rules $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2$, and $\neg\varphi$. They are also closed under existential and universal quantifiers, allowing the inductive formation of formulas $\forall x\ \varphi$ and $\exists x\ \varphi$. A formula φ whose relations and constants all come from a given schema Sch is said to be a *formula over* Sch. A formula with no free variables is a *sentence*.

The *equality-free* first-order formulas are built up as above but without equality atoms. Note that the equality atom $x = x$ expresses the formula True that is always true, while the equality formula $\neg x = x$ expresses the formula False that is always false. In equality-free first-order logic, we still want to express these, so we allow the special atomic formulas False and True to be in equality-free first-order logic as well.

If φ is a formula whose free variables include $\vec{x}$, while $\vec{t}$ is a sequence of constants and variables whose length matches $\vec{x}$, then $\varphi[\vec{x} := \vec{t}]$ denotes the formula obtained by simultaneously substituting each x_i with t_i. When the mapping of the free variables x_i to constants or variables t_i is either clear from context or unimportant, we denote this as $\varphi(\vec{t})$. We often omit universal quantifiers from formulas, particularly for formulas where the only quantifiers are universal. For instance, we write $P(x, y) \to Q(x, y)$ as a shorthand for $\forall x\ \forall y\ [P(x, y) \to Q(x, y)]$.

We need to discuss the distinction between the data model and query languages used in databases and the syntax and semantics of first-order logic as used in mathematics. The differences involve at least five issues: 1. the range of universal and existential quantifiers; 2. the treatment of constants; 3. positional notation for tuples vs. named attribute notation; 4. the range of free variables; 5. and finite vs. arbitrary data domains. We will cover the first four issues in the course of introducing our semantic notions below. The discussion of the last issue (finite vs. infinite) will be deferred until later in the text (see, e.g., Section 2.8).

Structures, instances, and range of quantification. A database *instance* I for a schema Sch without constant symbols assigns to every relation R in Sch of arity n a collection of n-tuples $[\![R]\!]_I$, in such a way that all integrity constraints of Sch are satisfied. We call $[\![R]\!]_I$ the *interpretation* of R in I. An association of a database relation R with a tuple $\vec{c}$ of the proper arity will be referred to as a *fact*. A database instance can equivalently be seen as a collection of facts. An instance of a schema that has only a single relation R is a *relation instance*. The *active domain* of an instance I, denoted $\mathsf{adom}(I)$, is the union of the one-dimensional projections of all interpretations of relations: that is, all the elements that participate in some fact of I.

Example 1.5. Suppose our schema consists of only one unary relation UEmployee, containing the ids of university employees. One possible instance I interprets UEmployee by the singleton set $\{e_0\}$. We can alternatively define I by the set of facts $\{\mathsf{UEmployee}(e_0)\}$. In this case we have $\mathsf{adom}(I) = \{e_0\}$. ◁

Classical logic considers *structures* rather than *instances*. A structure consists of a set, the *domain* of the structure, interpretations for each relation as sets of tuples with values in the domain, and an interpretation for each constant as a single element of the domain.

Example 1.6. Returning to the schema in Example 1.5, one structure M that conforms to it consists of a two-element domain $\{e_0, f_0\}$, with UEmployee interpreted as $\{e_0\}$. ◁

In the *classical semantics* of first-order logic, quantifiers range over the domain of the structure. In databases, quantifiers are usually given the *active domain* semantics, in which the quantified variable ranges over the union of the values in the interpretations of relations. The active domain semantics can be used to give a meaning to a sentence in an instance, since the meaning only depends on the instance, not some domain in which the instance sits.

Example 1.7. Let φ be the sentence $\exists x\ \neg\mathsf{UEmployee}(x)$. In the instance I of Example 1.5, φ is intuitively false, since there is only one element mentioned in facts, and it lies inside UEmployee in the instance. The active domain semantics of first-order logic formalizes this intuition, and under it the sentence is false in I.

Now consider the structure M in Example 1.6 which interprets UEmployee as in I, but has a two-element domain $\{e_0, f_0\}$. Under the classical semantics of first-order logic, φ is true in M, since f_0 can be a witness for the existential quantifier. ◁

In order to avoid the confusion that could be caused by using two different semantics for the same syntax, we will not formally define the active domain semantics for a logical formula—the interested reader can refer to [Abiteboul et al., 1995]. Instead, we define a syntax in which quantification is explicitly restricted to data in a certain relation within the instance. Once we restrict to this syntax, *the same semantics for quantifiers can be used both over structures and over instances, there will be no ambiguity about the semantics of quantifiers, and the reader will not have to worry about what quantifiers range over*.

We will use *first-order logic with relativized quantifiers*, RQFO. RQFO is built up from equality and relational atoms via the boolean operators and the quantifications:

$$\exists \vec{x}\ R(\vec{s}, \vec{x}) \wedge \varphi(\vec{s}, \vec{x}, \vec{t})$$

and

$$\forall \vec{x}\ R(\vec{s}, \vec{x}) \rightarrow \varphi(\vec{s}, \vec{x}, \vec{t})$$

for R a relational symbol and φ an RQFO formula. Those familiar with active domain semantics will be able to see that it is subsumed by RQFO, in the sense that an active domain semantics formula can be converted to an equivalent RQFO formula under classical semantics: the active domain interpretation of the quantification $\exists x\ \varphi$ is a disjunction $\bigvee_i \exists \vec{x}\ R_i(\vec{x}) \wedge \varphi$, while a quantification $\forall x\ \varphi$ under active domain semantics translates into a conjunction of relativized universal quantifications.

We can similarly talk about equality-free RQFO formulas, by disallowing equality atoms. By convention, we allow the prefix of existential quantifiers $\exists x_1\ \ldots\ \exists x_n$ to be empty, and similarly

for universal quantifiers. In this way we can capture an atom $R(x_1 \ldots x_n)$ and negated atoms $\neg R(x_1 \ldots x_n)$ as specialized RQFO formulas $R(\vec{x}) \wedge$ True and $R(\vec{x}) \rightarrow$ False, respectively. Thus for equality-free formulas we can in fact assume that the only base formulas are True and False.

The truth value of an RQFO sentence without constants is well-defined over an instance, since from the instance we can determine the range of each relativized quantifier. Similarly the truth value for an RQFO formula without constants is well-defined given an instance and a mapping of the free variables to some values (a *variable binding* or just *binding* for short).

Treatment of constants. In classical first-order logic, constants are *uninterpreted*. They are just symbols used in logical formulas that always represent an individual element. A structure must provide an interpretation of each constant: a mapping of each constant to a value. Using this mapping we can determine, for example, whether $c = d$ holds in the structure. In this work we allow formulas to mention uninterpreted constants, as in classical first-order logic. This will be important in modeling formulas that are generated within first-order proof systems such as tableau, used in Chapters 2 and 3.

In databases, constants usually come with a fixed interpretation, independent of any instance: each constant symbol is associated with a unique value, with distinct constants having distinct values. Thus one can identify the constant with the value. Thus R("Smith,""Jones") holds in an instance if the instance includes the fact R("Smith,""Jones"); no additional mapping is required. Our schema constants, which are the only constants allowed in queries and constraints, are always database-style interpreted constants.

Example 1.8. Returning to the setting of Example 1.5, suppose our schema also includes schema constants "Smith" and "Jones" and the relation Manager. Consider the instance I consisting of the facts {Manager("Jones"), UEmployee("Smith")}. The sentence UEmployee("Smith") $\wedge$ Manager("Jones") is true in this instance.

Now consider the sentence UEmployee(c) where c is an uninterpreted constant. If bind is a mapping taking c to "Smith," then the sentence holds in instance I with mapping bind. If bind′ is a binding taking c to "Jones," then the sentence does not hold in instance I with mapping bind′. ◁

In this text we make use of both interpreted and uninterpreted constants. But for the most part those more comfortable with uninterpreted constants can ignore interpreted ones, as we now explain.

Interpreted constants can be modeled with uninterpreted constants plus additional axioms added to the integrity constraints stating any special properties of the constant symbols.[2] In our setting, it suffices to assume that the integrity constraints include explicit assertions of inequalities between distinct schema constants. With these additional integrity constraints, one can consider all constants to be uninterpreted throughout the text: interpreted constants are used as a kind of "shorthand" for uninterpreted constants supplemented with additional axioms.

[2]This is an idea that goes back to Reiter [1984].

Interpreted constants are a useful shorthand, since they allow us to drop the additional mapping of constants to values in evaluating a formula, and we will make use of them often. In most of the text it will be clear from context whether a constant we are speaking of is uninterpreted or interpreted. Thus we simply refer to something as a "constant" for brevity.

We emphasize the bottom line of this "fine print" discussion of interpreted vs. uninterpreted constants and of instances vs. structures:

> The semantics of our queries and constraints can be stated in terms of instances and interpreted constants, as is common in databases, rather than structures and uninterpreted constants used in classical logic. Since RQFO sentences are syntactically restricted so as to be independent of a surrounding domain, and interpreted constants can be seen as a shorthand for uninterpreted constants plus additional axioms, we are free to make use of results from logic in reasoning about our queries and constraints.

Fragments of relativized-quantifier first-order logic. RQFO formulas that are built up as above but disallowing $\neg$ or $\forall$ will be called *positive existential formulas*, or $\exists^+$ *formulas* for short. Formulas that are built up as above but disallowing $\forall$ and allowing $\neg$ to be applied to only atomic formulas will be called *existential formulas*, or $\exists$ *formulas*.

Although we have defined first-order logic and its variants as including equality, we will usually make explicit when equalities are allowed. The presence or absence of equality in one of our logics will revolve around the ability to express inequalities (e.g., two constants are distinct, there are four distinct elements that satisfy a property). Note that we do not have the inequality symbol $\neq$ as a primitive in our logics. In first-order logic with equality, we can express inequality, since $\neq$ can be seen as an abbreviation for $\neg(x = y)$. But in equality-free first-order logic, we cannot express $\neq$. We can extend the existential fragment to allow negation only in the form $\neg(x = y)$. We refer to such a formula as an *existential formula with inequalities*, or $\exists^{\neq}$ formula. Similarly we can extend the positive existential fragment to *positive existential formulas with inequalities*, or $\exists^{+,\neq}$ *formulas*.

Example 1.9. Consider a schema that includes the relation UEmployee containing ids of each university employee, along with relations Researcher containing the same information but only about researchers, and a relation Lecturer containing the same information about lecturers.

- An RQFO sentence stating that every researcher is a university employee could be written:

$$\forall x\ \mathsf{Researcher}(x) \rightarrow \mathsf{UEmployee}(x).$$

- The $\exists$ sentence below states that there is some university employee who is not a researcher:

$$\exists x\ \mathsf{UEmployee}(x) \wedge \neg\mathsf{Researcher}(x).$$

- This $\exists^+$ sentence states that there is either a researcher or a lecturer:

$$\exists x\ \mathsf{Researcher}(x) \vee \mathsf{Lecturer}(x).$$

The sentence above is not written explicitly in RQFO, but it can be converted to one, namely

$$(\exists x\ \mathsf{Researcher}(x) \wedge \mathsf{True}) \vee (\exists x\ \mathsf{Lecturer}(x) \wedge \mathsf{True}).$$

We have defined the $\exists^+$ formulas as a subclass of RQFO formulas, and similarly for $\exists, \exists^{\neq}, \exists^{+,\neq}$. Since it is often inconvenient to write out formulas in the syntax of RQFO, we will often abuse notation by referring to a formula as being in one of these subclasses when it has an "obvious conversion" into a formula in this subclass, as above.

- This $\exists^{\neq}$ sentence states that there are two university employees who are not researchers:

$$\exists x\ \exists x'\ x' \neq x \wedge \mathsf{UEmployee}(x) \wedge \mathsf{UEmployee}(x') \wedge \neg\mathsf{Researcher}(x) \wedge \neg\mathsf{Researcher}(x').$$

- The following $\exists^{+,\neq}$ sentence states that there are two entities that are either researchers or lecturers:

$$\exists x\ \exists x'\ x' \neq x \wedge (\mathsf{Researcher}(x) \vee \mathsf{Lecturer}(x)) \wedge (\mathsf{Researcher}(x') \vee \mathsf{Lecturer}(x')).$$

◁

Queries via relativized-quantifier first-order logic. Remember that two key inputs to our reformulation problems are a source query and some integrity constraints, while the desired output of reformulation is a declarative query or a plan. By a *query* we mean a mapping from instances of some schema to interpretations of some relation. A *boolean query* is a query where the output is a relation of arity 0. There are only two interpretations for a relation of arity 0, the empty set and the set consisting of the empty tuple. If we denote the former by False and the latter by True, a boolean query is thus a mapping where the output takes one of the values in {True, False}. A query where the output relation has non-zero arity is a *non-boolean query*. Given a query Q and instance I, $[\![Q]\!]_I$ denotes the output of Q on I. The definition of a query as a mapping is very general, but in this text the queries will be described in either a logic-based language or an algebraic language which is equivalent in expressiveness to a logical formula. We now explain how logical formulas define queries, and then give the algebraic formalism for expressing queries.

RQFO sentences clearly can be used to define boolean queries: the result of the query on an instance is True exactly when the sentence holds in the instance. RQFO formulas can similarly be used to define non-boolean queries. Given an RQFO formula with its free variables enumerated as $v_1 \ldots v_n$, we can associate a non-boolean query whose output relation has arity n: the outputs will correspond to variable bindings which make the formula true. Recall that we evaluate RQFO formulas over an instance, not a structure. But we do not restrict the bindings to come from the instance. Instead we consider the free variables to take values within some fixed infinite "universal domain of values"—a set that will contain all elements in the domain of any instance that we

consider. We take the result of a query given by an RQFO formula to be all bindings that take values within that domain and satisfy the formula.

Example 1.10. Consider the RQFO formula $R(x, y) \wedge S(x, y)$. This defines a query that takes as input an instance I consisting of interpretations of R and S, returning as output the relation instance consisting of all pairs in $[\![R]\!]_I \cap [\![S]\!]_I$. ◁

A query given by an RQFO formula will be called an RQFO-*query*. For our query language we often restrict further, to *conjunctive queries* (CQs), logical formulas of the form $Q(\vec{x}) = \exists \vec{y}\ (A_1 \wedge \cdots \wedge A_n)$, where A_i is an atom using a relation of the schema, with arguments that are either variables from $\vec{x}$ and $\vec{y}$ or schema constants. A normalization argument shows that any logical formula built up using $\wedge$ and $\exists$ can be expressed as a CQ. For a conjunctive query Q, a variable binding that witnesses that Q holds in an instance I will also be called a *homomorphism of Q in I*.

In Chapter 3, we also consider *unions of conjunctive queries* (UCQs), which are disjunctions of CQs in which every CQ has the same free variables. Another simple normalization argument, found in [Abiteboul et al., 1995], shows that any $\exists^+$ sentence can be converted to a UCQ.

Example 1.11. In Example 1.9, the $\exists^+$ sentence stating that there is either a researcher or a lecturer can be written as a UCQ: $(\exists x\ \mathsf{Researcher}(x)) \vee (\exists x\ \mathsf{Lecturer}(x))$.

A formula returning the researchers that are also lecturers can be expressed as a CQ:

$$\mathsf{Researcher}(x) \wedge \mathsf{Lecturer}(x)$$

◁

Throughout the text we will state properties of *an* RQFO *query*, *an* $\exists^+$ *query*, and so forth. The reader can freely substitute the word "formula" for "query" in these assertions, but keeping in mind that the formula will likely play the role of either the source or the target in a reformulation problem.

Named vs. positional notation and relational algebra. In our semantics of first-order logic, a relation was associated with an arity n and was interpreted by a set of n-tuples. This is what we refer to as *positional notation*. An alternative notation considers schemas where a relation R is associated not just with an arity n, but with a collection of n attribute names. An instance for such a schema interprets each such relation R by a collection of functions from the attributes of R. A query over such a schema is a function mapping instances of the schema to an interpretation of a relation. There is a correspondence between schemas/instances/queries in the "named attribute" perspective and schemas/instances/queries in the "positional" perspective.

Example 1.12. Recall the schema from Example 1.5, which contained a unary relation UEmployee. The example contained an instance of this schema which consisted of the fact $\{\mathsf{UEmployee}(e_0)\}$. In named notation, we could state that UEmployee has a single attribute eid. Under this notation instance I would be represented by interpreting UEmployee as a single tuple with $\{\mathsf{eid} = e_0\}$. ◁

Relational algebra (RA) is the main "database-style" language we will use for queries and constraints. Relational algebra expressions refer to relations using named attributes. That is, the inputs are schemas where each relation has named attributes, and the output is a relational instance with named attributes. An expression in relational algebra is a term built up from relations by composing the following families of operators:

- selection (denoted σ), which selects tuples satisfying some equality or inequality condition. For example, $\sigma_{\mathsf{lname}=\text{"Smith"}}\mathsf{Employee}$ is an expression that takes as input an instance of the relation Employee, returning the instance containing tuples whose lname attribute is "Smith";
- projection (denoted π) which selects a subset of the attributes in a relation. An example is the relational algebra expression $\pi_{\mathsf{lname}}\mathsf{Employee}$, which takes as input an instance of Employee and returns the projection of each tuple in the instance onto the attribute lname;
- renaming (ρ), which renames the attributes according to some mapping;
- difference ($-$), and union ($\cup$), which have their usual set-theoretic meaning; and
- join ($\bowtie$), which merges tuples from two input relations, where the pair of tuples must satisfy some equality or inequality condition. For example, $\mathsf{Employee} \bowtie_{\mathsf{dname}=\mathsf{dname}} \mathsf{Department}$ takes as input an instance of the Employee relation and an instance of the Department relation, and returns a relation instance that includes each employee tuple in Employee extended with the matching information about the employee's department.

In order to be able to handle constants that are not part of the data, we also include in relational algebra the *constant operators* for each schema constant c: these take no input and produce a single-attribute relation instance containing one row with the attributes value being c. The semantics of these operations is standard (e.g., [Abiteboul et al., 1995]).

Just as we talked about fragments of first-order logic and RQFO, we have corresponding fragments of relational algebra, each defined as compositions of a subset of the operators. The most restricted fragment, SPJ, only allows the constant, selection, projection, renaming, and join operators. We use the prefix "U" to allow the union operator, and the "$\neq$" symbol to allow inequalities in selection and join predicates. We append an "AD" suffix to allow "atomic difference": we can allow subexpressions $E - E'$ only when E' has the form $E \bowtie_\alpha R$ where R is a relation and α is a set of equality conditions that identifies each attribute of R with an attribute of E. We abbreviate this as $E -_\alpha R$. For example, $USPJAD^{\neq}$ supplements SPJ by allowing union, inequality selections, and atomic difference.

Each relational algebra expression has a set of *input relations* and a set of *output attributes*. For instance, the relational algebra expression $\pi_{a,b}(R \bowtie S)$ has input relations $\{R, S\}$ and $\{a, b\}$ as output attributes. A relation with no attributes has only two interpretations, just like a relation with arity 0 under the positional perspective. Thus relational algebra queries with no output attributes define boolean queries, while relational algebra queries with a non-empty set of output

attributes define non-boolean queries. It is well-known that every boolean relational algebra query can be efficiently converted into an RQFO sentence and that every RQFO sentence using only interpreted constants can be converted to a boolean relational algebra query [Abiteboul et al., 1995]. Abiteboul et al. [1995] also presents correspondences between positive existential logical sentences and boolean $USPJ$ queries. These correspondences preserve the semantics of queries, up to the difference between positional and named notation. The argument given there extends in a straightforward way to show that existential sentences correspond to $USPJAD$ boolean queries, and similarly for the counterparts with inequalities (e.g., $\exists^{+,\neq}$ corresponds to $USPJ^{\neq}$). In particular, UCQ sentences, $\exists^{+}$ sentences, and $USPJ$ boolean queries have the same expressiveness. CQ sentences are equivalent in expressiveness to boolean queries defined in the SPJ fragment of relational algebra.

We next present some examples of the correspondence between relational algebra queries and RQFO formulas. Formally, these correspondences rely on a mapping between the positional and named perspectives. We will usually not spell out the mapping, moving back and forth between logic-based notation and relational algebra notation for queries. In the examples, we will omit the equality conditions in joins $E \bowtie F$ when they are the "obvious conditions": those that identify each output attribute of E with the output attribute having the same name in the output of F. This special kind of join is called a "natural join."

Example 1.13. Consider the assertions from Example 1.9. These are all sentences, so to model them in relational algebra, we need to look at queries that return an instance for the relation with no attributes, with the empty instance representing False and the non-empty instance denoting True. We can accomplish this by coming up with queries that return a non-empty output whenever the sentence is true and an empty output whenever the sentence is false, and then projecting those onto the empty set of tuples. We use this tactic in all the examples below.

- The RQFO sentence stating that every researcher is a university employee could be written in relational algebra as:

$$\mathsf{True} - \pi_{\emptyset}(\mathsf{Researcher} - \mathsf{UEmployee})$$

  where True is a query that always returns a non-empty instance with no attributes (such a query can be formed by applying a projection to a constant operator).

- The $\exists$ sentence stating that there is some university employee who is not a researcher could be expressed in $USPJAD$ as:

$$\pi_{\emptyset}[\mathsf{UEmployee} - \mathsf{Researcher}]$$

- The $\exists^{+}$ sentence stating there is either a researcher or a lecturer could be expressed in $USPJ$ as:

$$\pi_{\emptyset}[\mathsf{Researcher} \cup \mathsf{Lecturer}].$$

- The $\exists^{\neq}$ sentence stating that there are two university employees who are not researchers could be written as the $SPJAD^{\neq}$ query:

$$\pi_{\emptyset}[(\mathsf{UEmployee} - \mathsf{Researcher}) \bowtie_{x \neq x'} \rho(x \rightsquigarrow x')(\mathsf{UEmployee} - \mathsf{Researcher})].$$

  where x is the sole attribute of UEmployee and Researcher.

- The $\exists^{+,\neq}$ sentence stating that there are two entities that are either researchers or lecturers could be expressed in $USPJ^{\neq}$ as:

$$\pi_{\emptyset}[(\mathsf{UEmployee} \cup \mathsf{Researcher}) \bowtie_{x \neq x'} \rho(x \rightsquigarrow x')(\mathsf{UEmployee} \cup \mathsf{Researcher})].$$

◁

Example 1.14. Consider the CQ in Example 1.11, asking for researchers that are also lecturers. This could be written as the SPJ query:

$$\mathsf{Researcher} \bowtie \mathsf{Lecturer}.$$

◁

Figure 1.2 shows the fragments of RQFO (left) that we consider in this text, along with the corresponding fragments of relational algebra (right).

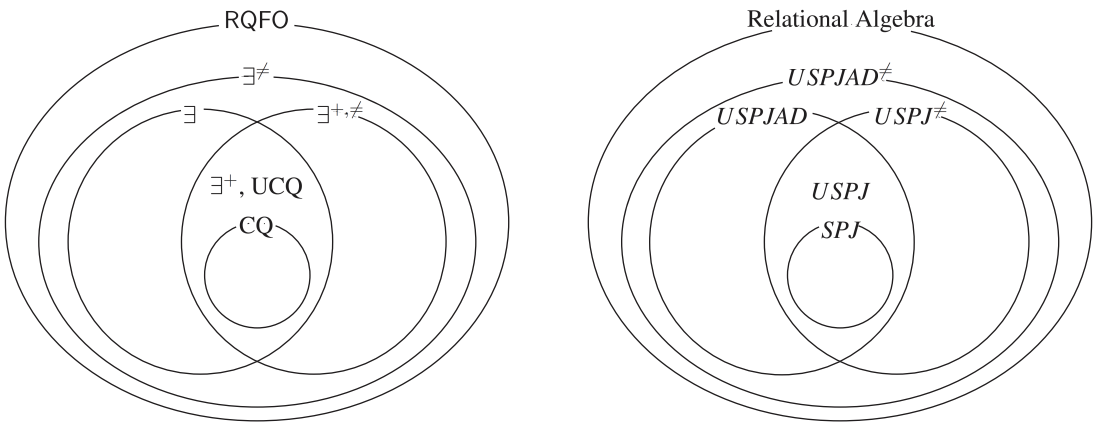

Figure 1.2: Fragments of RQFO and corresponding relational algebra fragments.

Range of free variables in formulas. We have already discussed two issues that need to be addressed in order to talk about an equivalence between first-order logic formulas and relational algebra queries: adjusting the range of quantification in logical formulas to match that of relational algebra, and looking at equivalence up to the distinction between named vs. positional notation. This allowed us to claim a correspondence between *boolean* relational algebra queries and RQFO *sentences*. We next deal with a final distinction between logic-based queries and relational algebra that arises only in the context of non-boolean queries: the range of free variables.

Non-boolean relational algebra queries return only tuples consisting of elements within the active domain of the input instance or schema constants, while first-order logic formulas can hold of arbitrary bindings. To extend this correspondence we could restrict the semantics of logical formulas to bindings whose values are either schema constants or lie in the active domain. With this caveat, the correspondence given in Figure 1.2 extends to non-boolean queries.

Another way to extend the logic/relational algebra correspondence is to deal with RQFO formulas that can only be satisfied by active domain elements or schema constants. A formula is *semantically active domain* if it holds only on bindings for which the values are either in the active domain or are schema constants. Since we assume that our set of schema constants is finite, formulas that are semantically active domain are *safe*, in that for each finite input instance, the corresponding query outputs a finite relation instance. Some of the fragments we have already defined contain only safe formulas. For example, in defining UCQs, we enforced that every CQ had the same free variables, and this suffices to ensure safety. All relational algebra formulas are semantically active domain, and hence safe.

RQFO formulas, even those in fragments such as $\exists^+$, are not necessarily safe: for example the $\exists^+$ formula $A(x) \vee B(y)$ can return infinitely many pairs. However, safety is known to be the only obstacle to converting logical formulas to relational algebra queries:

Theorem 1.1 *The following are equivalent in expressiveness:*

- *safe* RQFO *formulas and relational algebra queries;*
- *safe* $\exists^{\neq}$ *formulas and* $USPJAD^{\neq}$ *queries;*
- *safe* $\exists$ *formulas and* $USPJAD$ *queries;*
- *safe* $\exists^{+,\neq}$ *formulas and* $USPJ^{\neq}$ *queries;*
- *safe* $\exists^+$ *formulas and* $USPJ$ *queries.*

Further, an RQFO *formula is safe if and only if it is semantically active domain. In each case above, the equivalence is up to mappings between named and positional notation, and assumes that the formulas use only interpreted constants.*

The proof of the first item can be found in [Abiteboul et al., 1995], which further provides effective transformations that realize the equivalence. The other items can be proven using the

same transformations. In particular, if we restrict to safe formulas, the correspondences given in Figure 1.2 extend to non-boolean queries. For the majority of this text, we will deal with RQFO formulas which are safe by construction, such as UCQs. By Theorem 1.1, these formulas can all be translated to relational algebra.

Constraints of particular interest. The problems we look at will have as input both a query and a set of constraints. For queries we focus on CQs or on UCQs, as mentioned above. For constraints, we will sometimes consider general RQFO sentences. We also give particular attention to constraints given by *tuple-generating dependencies* (TGDs), given syntactically as

$$\forall \vec{x}\ [\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x}, \vec{y})]$$

where φ and ρ are conjunctions of relational atoms, possibly including constants.

A special subclass consists of *Guarded TGDs* (GTGDs), in which φ is of the form $R(\vec{x}) \wedge \varphi'$ where $R(\vec{x})$ contains all the variables of φ'. These in turn subsume *inclusion dependencies* (IDs): where φ and ρ are single atoms in which no variables are repeated and there are no constants. IDs are also-called "referential constraints."

Example 1.15. Recall the integrity constraints from Example 1.2.

$$\forall \mathsf{deptid}\ \forall \mathsf{dname}\ \forall \mathsf{mgrid}\ \mathsf{Department}(\mathsf{deptid}, \mathsf{dname}, \mathsf{mgrid}) \rightarrow \exists N\ \mathsf{Employee}(\mathsf{mgrid}, N, \mathsf{deptid})$$
$$\forall \mathsf{eid}\ \forall \mathsf{ename}\ \forall \mathsf{deptid}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid}) \rightarrow \exists D \exists M\ \mathsf{Department}(\mathsf{deptid}, D, M).$$

Both of these are IDs. ◁

Relationships between instances. In reformulation problems we often have some "visible" information, and have to consider what underlying structure is consistent with that. The notion of superinstance captures that an instance I' is consistent with the information provided by another instance I. If we have two instances I and I', and for every relation R, $[\![R]\!]_I \subseteq [\![R]\!]_{I'}$, then we say that I is a *subinstance* of I', and I' is a *superinstance* of I. If $\mathsf{adom}(I)$ is the active domain of I, we say that I is an *induced subinstance* of I' if I consists of exactly those facts of I' whose values lie in $\mathsf{adom}(I)$. I being an induced subinstance can be restated as: I is a subinstance of I' and for each fact F of I', if all values in F lie in some fact of I, then F is in I.

Computational problems involving logic. In this work we will make use of basic results about computational problems associated with logical formulas. A first-order logic sentence is *satisfiable* if there is some instance that satisfies it, and *finitely satisfiable* if there is some finite instance that satisfies it. The problem of determining whether an FO sentence is satisfiable is undecidable, and the same is true for determining finite satisfiability [Abiteboul et al., 1995]. Some expressive sublanguages of FO are known to have a decidable satisfiability problem. We will discuss one such, the guarded negation fragment, later in the text. Satisfiability is a special case of the entailment problem, which is discussed in the next section.

1.3 ENTAILMENT AND PROOFS

We will need some results about reasoning with formulas of first-order logic. A basic reasoning problem is to determine whether a sentence λ *entails* another sentence ρ, meaning: in any structure where λ holds, ρ holds. If λ and ρ are RQFO sentences without uninterpreted constants, then we can freely replace "any structure" above with "any instance," since the truth of λ and ρ in a structure only depends on the facts that make up the instance. More generally, we can talk about a formula $\lambda(\vec{x})$ entailing another formula $\rho(\vec{x})$: this means that in any structure and any binding of the variables $\vec{x}$ to elements, if $\lambda(\vec{x})$ holds then $\rho(\vec{x})$ holds.

We write $\lambda \models \rho$ to indicate that λ entails ρ. We often use the following basic fact: when λ and ρ are formulas with free variables, $\lambda \models \rho$ holds exactly when $\lambda' \models \rho'$, where λ' is formed from λ by replacing the free variables with fresh uninterpreted constants, and similarly for ρ'. Thus we can treat free variables in an entailment as if they were uninterpreted constants. In the definition of entailment, we consider *all structures* (in the case of RQFO sentences, *all instances*) whether they are finite or infinite. Throughout the text, by default we use this notion of entailment, and allow our instances or structures to be either finite or infinite. Of course, in databases one is interested only in finite instances. We will use a "trick" to convert results that quantify over arbitrary instances to results that refer only to finite instances later in the text (see, for example, Section 2.8).

We recall that *proof systems* for logics are formal systems for showing that an entailment holds in the logic. A proof system is *complete* if every entailment that is true has a proof. There are many complete proof systems for first-order logic. We now present a sound and complete proof system for first-order logic which is used throughout this work. Following Fitting [1996], we will start with a *tableau proof system*, which is a proof system that assumes classical semantics. We present this for equality-free FO formulas to begin with, discussing the extension to equality later.

Tableau proofs. Our tableau system will work with first-order formulas that are in negation normal form (NNF), i.e., formulas that are built up from atomic formulas and their negations using conjunction, disjunction, existential quantification, and universal quantification. Every first-order formula is equivalent to a formula in NNF that can be obtained by pushing all negations inside as far as possible and applying the rule $\neg\neg\varphi \leftrightarrow \varphi$ to eliminate double negations. Occasionally we will have to talk about "negating" a formula in a tableau, and need to deal with the fact that general negation is not allowed in NNF. If φ is a formula in NNF, we denote by $\sim\varphi$ the formula in NNF obtained by negating φ and pushing the negations inside as far as possible.

Intuitively, a tableau proof is a demonstration that a formula φ is unsatisfiable that consists of a tree of sets of formulas. We start with a root containing φ, and we grow the tree downward, at each stage picking a leaf node and then a formula within the node, generating one or more child nodes which add on consequences of the formula, with the goal of generating a contradiction. Consequences of a formula are generated by breaking down the top-level connective or quantifier in the formula. If the top-level connective is a disjunction, we expand the tableau node containing

the formula with two children, representing a case analysis of both options in the disjunction. For other connectives and the quantifiers, the expansion process generates a single child. We make a node a leaf when it has an explicit contradiction, such as an atomic formula and its negation in the same node. When every path ends in a contradiction, we have proven that the root formula is unsatisfiable.

Formally, a *tableau* is a finite tree in which every node is labeled by a finite set of formulas of equality-free first-order logic, and such that each node other than the root is formed from its parent by applying the replacement given by a *tableau expansion rule*.

- The *conjunction rule* states that if we have a leaf node labeled by a set of formulas that includes a conjunction, we can create a single child of that node, whose set of formulas contains all formulas of the parent and in addition both conjuncts of the conjunction. This rule is abbreviated:

$$\frac{\varphi \wedge \psi}{\begin{array}{c}\varphi \\ \psi\end{array}}$$

- The *existential quantifier rule* states that if a leaf node is labeled by a set of formulas that include a block of existential quantifiers, we can create a single child of that node whose set of formulas adds on a subformula formed by replacing the quantified variables by constants that do not occur in any formula of the node. The rule is represented:

$$\frac{\exists \vec{x}\ \varphi(\vec{x},\vec{t})}{\varphi(\vec{a},\vec{t})}$$

($\vec{a}$ fresh uninterpreted constants) .

- The *disjunction rule* states that if a leaf node is labeled by a set of formulas that includes a disjunction, we can create two children of that node, where each child adds on one of the disjuncts. We represent the rule as:

$$\frac{\varphi \vee \psi}{\varphi \mid \psi}\ .$$

- The *universal quantifier rule* states that if a leaf node is labeled by a set of formulas that includes a universally quantified formula, we can create a child of that node whose set of formulas adds a formula that is an "instantiation" of the universally quantified formula. That is, the new formula is obtained from the universally quantified one by replacing the universally quantified variable by a constant that occurs in some other formula of the node.

$$\frac{\forall \vec{x}\ \varphi(\vec{x},\vec{t})}{\varphi(\vec{v},\vec{t})}$$

($\vec{v}$ constants occurring in the tableau node).

Above, $\varphi(\vec{v},\vec{t})$ indicates substituting any set of constants $\vec{v}$ for $\vec{x}$ in $\varphi(\vec{x},\vec{t})$. Note that *we always interpret the rules as adding formulas to the child*, based on formulas that were present in the parent.[3] Note also that the disjunction rule is the only rule that creates two children.

A branch of a tableau is *closed* if it ends in a leaf node that contains a *clash*, which is either the formula False, or a pair of formulas ψ and $\neg\psi$ for some atomic formula ψ. Otherwise the branch is said to be *open*. A tableau is closed if all of its branches are closed. A *tableau for* $\lambda(\vec{x}) \models \rho(\vec{x})$ is a tableau whose root node is labeled by $\{\lambda(\vec{a}), \sim\rho(\vec{a})\}$ for fresh uninterpreted constant symbols $\vec{a}$. A *closed tableau for* $\lambda \models \rho$ is also known as a *tableau proof of* $\lambda \models \rho$. The basic result that underlies the use of tableau proofs is that they form a complete proof system.

Theorem 1.2 Completeness of tableaux. *If λ, ρ are equality-free first-order formulas in negation normal form, then $\lambda \models \rho$ if and only if there is a tableau proof of $\lambda \models \rho$.*

In particular a first-order formula ρ is valid *(holds in every structure) if and only if there is a closed tableau rooted at $\sim\rho$.*

We sketch the proof only for the second statement, since the first can be reduced to it, proving it only in the case where φ has no free variables.

Proof. For the *soundness* direction: if there is a structure M where ρ does not hold, we claim that every tableau rooted at $\sim\rho$ must have some branch that is not closed, and thus we cannot have a tableau proof of $\models \rho$. The branch in question can be found inductively by starting at the root and traveling downward, mapping the generated constants to elements of M as we do so. The only choice to be made is in nodes where the disjunction rule is applied, and in this case we use the counterexample structure to guide the choice for left-child or right-child, according to which one holds in M.

For the *completeness* direction, we assume a tableau construction process that is *fair*, in the sense that every possible application of a tableau expansion rule is eventually performed. Consider any tableau (possibly infinite) produced by such a fair tableau construction process, and assume that the tableau never closes. Then the tableau would contain a maximal open branch. Take the structure whose facts are precisely the ground facts on the branch in question. Then we can show by induction that every formula on the branch is true in the structure. In particular, the structure satisfies $\neg\rho$ and hence ρ is not valid. □

[3] Copying all formulas from the parent to the child is not necessary in every rule. For example, in the disjunction rule above, we do not need to copy the original disjunction to each child node.

Example 1.16. Figure 1.3 shows a tableau proof witnessing

$$\exists x\ A(x) \wedge \neg B(x) \wedge C(x) \models \neg(\forall y\ [(\neg A(y) \wedge E(y)) \vee B(y)]).$$

In the figure, only a subset of the formulas in non-root nodes are shown, omitting certain formulas that are copied from the parent of the nodes.

The proof creates a tree "top-down," starting with a root node containing the left of the entailment and the negation of the right. In the first step, we proceed by instantiating the existential quantifier in the left to become a new uninterpreted constant c. This creates the node that is the child of the root below. In the next step we break down the conjunctions, resulting in a grandchild of the root.

We can then instantiate the universal quantifier $\forall y$ to any constant. We have only one constant mentioned in the proof at this point, the constant c. Instantiating with c generates a leaf node containing a disjunction, namely $[(\neg A(c) \wedge E(c)) \vee B(c)]$. We can break down the disjunction to get two children, one in which the disjunction is replaced by $\neg A(c) \wedge E(c)$, the other where it is replaced by $B(c)$. Finally in the left hand disjunct we can break down the conjunction.

At this point we have a tree with two leaf nodes. In each node we have a clash showing a contradiction. In the left leaf, this is a clash between $A(c)$ and $\neg A(c)$, while in the rightmost leaf this is a clash between $B(c)$ and $\neg B(c)$. Since every branch has a contradiction, we have a complete proof.

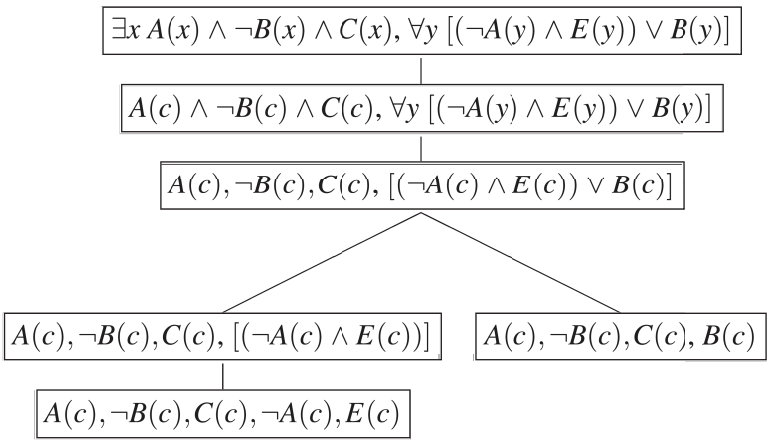

Figure 1.3: A proof of $\exists x\ A(x) \wedge \neg B(x) \wedge C(x) \models \neg(\forall y\ [(\neg A(y) \wedge E(y)) \vee B(y)])$.

◁

Tableaux for relativized-quantifier logic. Recall that in RQFO, quantification is always over elements stored in relations.

$$\exists \vec{x}\ R(\vec{x},\vec{s}) \wedge \varphi$$

and

$$\forall \vec{x}\ R(\vec{x},\vec{s}) \rightarrow \varphi.$$

Like general first-order formulas, RQFO formulas can be put into a negation normal form, where we do not have any negation except on atomic formulas. For equality-free RQFO, NNF formulas could be built up from True and False using quantifiers, $\wedge$ and $\vee$. We will also allow atoms and negated atoms as primitive formulas in NNF for equality-free RQFO, since they are useful in tableau proofs.

We do not need a new proof system to capture entailments between RQFO formulas; we can just consider them as specialized classical formulas. However, it will be convenient to have specialized tableau proof rules just for these new quantifiers. The modified quantifier proof rule for existential quantification is simple:

$$\frac{\exists \vec{x}\ (R(\vec{s},\vec{x}) \wedge \varphi(\vec{s},\vec{x},\vec{t}))}{\begin{array}{c} R(\vec{s},\vec{a}) \\ \varphi(\vec{s},\vec{a},\vec{t}) \end{array}} \quad (\vec{a} \text{ fresh uninterpreted constants})\ .$$

The rule for relativized universal quantification is:

$$\frac{\forall \vec{x}\ (R(\vec{s},\vec{x}) \rightarrow \varphi(\vec{s},\vec{x},\vec{t})) \quad R(\vec{s},\vec{v})}{\varphi(\vec{s},\vec{v},\vec{t})}\ .$$

The reader will see that each of these can be thought of as "macros": they are what we would get if we wrote out the relativized formulas as classical formulas, applied the old rules, and immediately eliminated some nodes that had clashes. However, we can show that if we are interested in entailments involving only relativized formulas, the rules are complete. For convenience, we state the result only for sentences:

Theorem 1.3 *For λ, ρ equality-free* RQFO *sentences in NNF we have $\lambda \models \rho$ if and only if there is a closed tableau starting from $\lambda \wedge \sim\rho$, where the classical quantifier rules are replaced by the relativized-quantifier rules above.*

The proof is a variation of the proof of Theorem 1.2, building a counterexample witness from a non-closed tableau proof by combining formulas along a branch.

Entailment in Theorem 1.3 refers to what holds in all *structures*. But since satisfaction of an RQFO sentence depends only on the underlying instance, when λ and ρ are RQFO sentences without constants we can restate an entailment $\lambda \models \rho$ as: every *instance* I satisfying λ also satisfies ρ. If ρ and λ contain schema constants, the same holds provided that λ and ρ contain the necessary "background axioms": all inequalities between distinct schema constants. If λ and ρ do not contain equality, these additional axioms can be dropped. With this disclaimer about constants in mind,

we can freely apply tableaux to reason about assertions that an RQFO holds on all instances. When we use tableaux to reason about integrity constraints and queries involving schema constants, we will often assume that the integrity constraints are enhanced with the necessary inequalities.

TGDs and the chase. Tableau proofs form a reasoning system to decide arbitrary first-order entailments. In much of this work, we are interested in special kinds of entailments, of the form

$$Q \wedge \Sigma \models Q^*$$

where Q and Q^* are conjunctive queries and Σ is a conjunction of TGDs.

This entailment problem is often called "query containment with constraints" in the database literature. We often say that Q is *contained in* Q^* *w.r.t. constraints* Σ. A specialized method has been developed to prove these entailments, called *the chase* [Abiteboul et al., 1995, Maier et al., 1979]. The idea in the chase is to

1. start with the assumption Q;
2. iteratively derive consequences with the constraints Σ;
3. stop and declare success if we have generated a set of consequences which "match" Q^*.

We will now describe these steps more precisely.

A proof in the chase consists of a sequence of database instances, beginning with the *canonical database* $\mathsf{CanonDB}(Q)$ of conjunctive query Q: the instance whose elements are the schema constants of Q plus distinct elements c_i for each variable x_i in Q and which has a fact $R(c_1 \ldots c_n)$ for each atom $R(x_1 \ldots x_n)$ of Q. These databases evolve by *firing rules*. Given a set of facts I and a TGD $\delta = \forall x_1 \ldots x_j\ \varphi(\vec{x}) \rightarrow \exists y_1 \ldots y_k\ \rho(\vec{x}, \vec{y})$, a *trigger for* δ is a tuple $\vec{e}$ such that $\varphi(\vec{e})$ holds. If there is no $\vec{f}$ such that $\rho(\vec{e}, \vec{f})$ holds in I, the trigger is *active*. A *rule firing* for a trigger adds facts to I that make $\rho(\vec{e}, \vec{f})$ true, where $f_1 \ldots f_k$ are new values ("chase constants") distinct from any values in I and any schema constants. Such a firing is also called a *chase step*. If the trigger was an active trigger, it is a *restricted chase step*. The definition of a valid chase step depends on a set of schema constants (used in defining the notion of newness), but we will usually omit this, since the set can be inferred from context. A *chase sequence* following a set of TGDs Σ is a sequence of instances where an instance is related to its successor by a rule firing of a dependency in Σ. We refer to the instances in the sequence as *chase configurations*.

Recall that a variable binding that witnesses that a CQ Q^* holds in an instance I is also called a homomorphism of Q^* in I. If I has elements corresponding to the free variables of Q^*, then a homomorphism of CQ Q^* into I that maps each free variable to the corresponding element is called a *match* for Q^* in I. Given CQs Q and Q^*, a *chase proof witnessing the entailment* $Q \wedge \Sigma \models Q^*$ is a sequence beginning with the canonical database of Q, proceeding by applying chase steps with Σ, and ending in a configuration having a match for Q^*.

We can utilize chase proofs for a slightly broader set of entailments than query containment with constraints. For a finite instance I and conjunction of constraints Σ, we abbreviate the

sentence $(\bigwedge_{F \in I} F) \wedge \Sigma$ by $I \wedge \Sigma$. Given a CQ Q^* and a finite instance I containing elements for each free variable of Q^*, a *chase proof* for the entailment $I \wedge \Sigma \models Q^*$ is a chase sequence following Σ, beginning with I and ending in a configuration having a match for Q^*.

The following well-known result says that the chase is a complete proof system for these entailments:

Theorem 1.4 *[Fagin et al., 2005, Maier et al., 1979] For any CQ Q^* and finite instance I containing elements for the free variables of Q^*, and any TGD constraints Σ, the entailment $I \wedge \Sigma \models Q^*$ holds if and only if there is a chase proof of this entailment. In particular, chase proofs form a complete proof system for containment of a CQ Q in a CQ Q^* w.r.t. Σ.*

Example 1.17. We recall the schema from Example 1.2, containing information about employees and departments. The constraints Σ were the following two TGDs:

$$\forall \mathsf{deptid}\ \forall \mathsf{dname}\ \forall \mathsf{mgrid}\ \mathsf{Department}(\mathsf{deptid}, \mathsf{dname}, \mathsf{mgrid}) \rightarrow \exists N\ \mathsf{Employee}(\mathsf{mgrid}, N, \mathsf{deptid})$$
$$\forall \mathsf{eid}\ \forall \mathsf{ename}\ \forall \mathsf{deptid}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid}) \rightarrow \exists D \exists M\ \mathsf{Department}(\mathsf{deptid}, D, M).$$

Consider the following two queries:

$$Q = \{\mathsf{deptid} \mid \exists \mathsf{eid}\ \exists \mathsf{ename}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid})\}$$
$$Q^* = \{\mathsf{deptid} \mid \exists \mathsf{dname}\ \exists \mathsf{mgrid}\ \mathsf{Department}(\mathsf{deptid}, \mathsf{dname}, \mathsf{mgrid})\}.$$

We claim that Q is contained in Q^* relative to the constraints of the schema. That is, in the more general logical terminology, that:

$$Q \wedge \Sigma \models Q^*.$$

To do this we perform a chase proof. We begin our proof with the "canonical database" of our assumption query $Q = \exists \mathsf{eid}\ \exists \mathsf{ename}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid})$. That is, we fix constants $\mathsf{eid}_0, \mathsf{ename}_0, \mathsf{deptid}_0$ witnessing the variables to get the "initial database":

$$\mathsf{Employee}(\mathsf{eid}_0, \mathsf{ename}_0, \mathsf{deptid}_0).$$

The second integrity constraint has a trigger on the initial database: the trigger maps variable eid to eid_0, variable ename to constant ename_0, and variable deptid to constant deptid_0. This is an active trigger, since there is no corresponding Department tuple as required by the constraint. We can now perform a "chase step" with this trigger, to derive a new fact:

$$\mathsf{Department}(\mathsf{deptid}_0, D_0, M_0).$$

where D_0, M_0 are new constants.

We can now match Q^* against the set of facts we have produced, with the homomorphism mapping the free variable deptid in Q^* to the corresponding constant deptid_0. This chase proof witnesses that Q is contained in Q^* *w.r.t.* Σ. ◁

Tableaux and the chase. We have presented two proof systems, one applicable to arbitrary first-order entailments (tableau proofs) and one that is defined only for a specific kind of entailment. What is the relationship between these two?

It is easy to see that *every chase proof corresponds to a tableau proof*. This has been pointed out before: see, for example, Section 5.4 of Toman and Weddell [2011]. But since we will need to look more closely at this correspondence in Section 4.5, we explain the relationship here, focusing on the case of containment of CQ Q in Q^* under TGDs Σ.

For conjunctive query Q and set of TGDs Σ, a Q*-rooted chase sequence* for Σ is any sequence starting with the canonical database of Q and proceeding by chase steps for Σ. We can make our claimed correspondence precise using a function that maps every Q-rooted chase sequence $\mathsf{config}_0 \ldots \mathsf{config}_i$ for Σ to a tableau for $Q \wedge \Sigma \vdash Q^*$, where Q^* is another conjunctive query. Each tableau node will include Q, the formula $\sim Q^*$ formed by converting the negation of Q^* into a RQFO formula, and similarly the RQFO equivalents of all constraints in Σ. For example, if Q^* is

$$\exists \vec{y}\ B_1(\vec{x}_1, \vec{y}_1) \wedge B_2(\vec{x}_2, \vec{y}_2).$$

Then $\sim Q^*$ converted to RQFO will be:

$$\forall \vec{y}\ B_1(\vec{x}_1 \vec{y}_1) \rightarrow \neg B_2(\vec{x}_2, \vec{y}_2).$$

For a TGD

$$\forall x_1 x_2\ U(x_1) \wedge V(x_2) \rightarrow W(x_1, x_2)$$

we would convert it to RQFO as

$$\forall x_1\ U(x_1) \rightarrow [\forall x_2\ V(x_2) \rightarrow W(x_1, x_2)].$$

For each configuration config_i in the chase sequence, we will have a tableau node $\mathsf{TabNode}_i$ containing all of the chase facts in config_i, unioned with Σ and $\sim Q^*$ converted as above.

Consider a chase step that takes configuration config_i to config_{i+1}. This involves a TGD

$$\tau = \forall \vec{x}\ \rho_1(\vec{x}) \rightarrow \exists \vec{y}\ \rho_2(\vec{x}, \vec{y})$$

where the left-hand side has a homomorphism h mapping the variables $\vec{x}$ into config_i. We can mimic this step via two sets of tableau evolution rules in the RQFO variant of the tableau proof system. The first set of rules instantiates all universally quantified variables $\vec{x}$ in τ, using repeated applications of the relativized universal quantifier rule, to get $\rho_2' = \exists \vec{y}\ \rho_2[x_1 := h(x_1) \ldots x_n := h(x_n)]$. The next step applies the relativized existential quantification rule repeatedly to instantiate each y_i with a fresh uninterpreted constant c_i.

This process preserves the invariant that the formulas in node $\mathsf{TabNode}_i$ consist of those in config_i unioned with Σ and $\sim Q$. If we have a match h of Q^* in some configuration config_i, we can close this tableau by adding steps that instantiate the universal quantifiers in $\sim Q^*$ using h to get a closed tableau.

Example 1.18. We rephrase the chase proof corresponding to Example 1.17 as a tableau proof. Recall that Q is $\exists$eid $\exists$ename Employee(eid, ename, deptid).
In this case, $\sim Q^*$ is

$$\forall \text{dname } \forall \text{mgrid Department(deptid, dname, mgrid)} \rightarrow \text{False}.$$

The corresponding tableau proof showing that $Q \wedge \Sigma \vdash Q^*$ is shown in Figure 1.4.

$$\boxed{\exists \text{eid } \exists \text{ename Employee(eid, ename, deptid)}, \Sigma, \sim Q^*}$$

$$\boxed{\text{Employee(eid}_0, \text{ename}_0, \text{deptid}_0), \Sigma, \sim Q^*}$$

$$\boxed{\text{Employee(eid}_0, \text{ename}_0, \text{deptid}_0) \text{ , Department(deptid}_0, D, M), \Sigma, \sim Q^*}$$

$$\boxed{\text{Employee(eid}_0, \text{ename}_0, \text{deptid}_0) \text{ , Department(deptid}_0, D, M), \text{False}, \Sigma, \sim Q^*}$$

Figure 1.4: Tableau translation of the chase proof for Example 1.17.

◁

In summary, we see that chase proofs are a kind of tableau proof. The tableaux that correspond to chase proofs are very special, in that universal quantification and existential quantification rules occur in tandem, and disjunction plays only a very limited role.

Chasing until termination. One way to find a chase proof is to "chase an initial instance as much as possible." For any set of TGDs Σ and initial instance I, we could just fire rules in an arbitrary order, making sure that any rule that is triggered fires eventually. The union of all facts generated will give an instance that satisfies the constraints, but it may be infinite. We refer to this as *the result of chasing I with Σ*. There will be many such instances depending on the order of rules fired, but they will all satisfy the same conjunctive queries, by Theorem 1.4.

Sometimes one can fully chase an initial instance and get a finite chase sequence and finite final configuration. A *restricted chase sequence* is one that makes only restricted chase steps (i.e., steps using active triggers). One can show that allowing only restricted chase steps does not jeopardize completeness of the chase. An advantage of restricted chase sequences is that we can get to the point where no steps are applicable, and at this point we can cut off the search. A finite restricted chase sequence *terminates* if in the final configuration there are no active triggers, that is, eventually no rules can fire that add new witnesses. If we have a terminating chase sequence beginning with the canonical database of Q, Theorem 1.4 implies that for any conjunctive query Q^*, Q is contained in Q^* *w.r.t.* the constraints if and only if Q^* has a match in the final configuration of the chase sequence. If a set of TGD constraints have the property that for any initial

instance every maximal restricted chase sequence is terminating, we say that the constraints *have terminating chase*.

1.4 SUMMARY

We have explained the problems that we focus on throughout the book. We have also gone through definitions of the constraints and queries that we consider, and the systems we use for reasoning with them. We summarize some of the major points to keep in mind in reading the remaining chapters:

- The main integrity constraint languages of interest to us are general RQFO sentences and the subclass of these known as TGDs.
- The main query languages of interest are:
  - queries in Relational Algebra, which have the same expressiveness as safe RQFO formulas;
  - $USPJAD$ queries, which have the same expressiveness as safe $\exists$ formulas;
  - $USPJ$ queries, which have the same expressiveness as safe $\exists^+$ formulas.

  There are also variations of these languages in the presence of equality. The main impact of these variations is that they allow us to express inequalities (e.g., there are three distinct elements in relation U).
- Integrity constraints and queries can make use of interpreted constants from the schema. Constants used in tableau proofs are uninterpreted; this implies that distinct constants need not be equal, and we must assign a value to a constant to know whether a formula containing it is true. In contrast, constants mentioned in a schema are interpreted; they are assumed to represent some fixed value, and distinct constants are assumed to represent distinct values.
- We will need to be able to reason about queries and constraints, and in particular to look at entailments between formulas.
  - To reason about general first-order formulas or about RQFO formulas, we can use the tableau proof system.
  - To reason about containment of CQs under TGDs we can use the chase, which can be seen as a special form of tableau.
  - Entailment is always considered over all structures (or all instances, for RQFO) finite or infinite. This is not the same as reasoning over all finite instances, but for some specialized classes of formulas and constraints (discussed later in the text), the two notions will turn out to be equivalent.

1.5 BIBLIOGRAPHIC REMARKS

A fundamental reference for the foundations of databases is Abiteboul, Hull, and Vianu [Abiteboul et al., 1995]. It covers database schemas as well as the use of first-order logic as a query language. The material on first-order languages includes classical first-order logic, fragments of first-order logic, and relational algebra. The text also goes into detail on the key issues arising in applying first-order logic within databases, which we have discussed only briefly here:

- the "notation mismatch": the fact that variable bindings and reference to positions with a tuple are used in logic, while named attributes are used in relational algebra and all practical database query languages;
- the range of quantified variables: as we have mentioned, the classical semantics of first-order logics quantifies over domains, while the common database approach quantifies over the active domain of the database;
- the treatment of constant symbols: classical logic deals with uninterpreted constants, while for database applications interpreted constants are more convenient;
- the range of free variables: the fact that database query languages such as relational algebra always return elements from the active domain unioned with a set of constants that is independent of the input, while first-order formulas can hold of infinitely many bindings.

The material on tableau proofs can be found in a number of textbooks on logic for computer science, such as [Ben-Ari, 2012]. Our presentation of tableau proof systems follows the approach outlined by Smullyan, found in Fitting's textbook [Fitting, 1996]. A full proof of completeness of the method can be found there.

Two standard references for the chase method are [Fagin et al., 2005, Maier et al., 1979], and overviews can be found in Deutsch and Nash [2009], Deutsch et al. [2008]. A longer introduction to the chase method can be found in the survey of [Onet, 2013]. The chase has many applications in data management—see, e.g., [Haas et al., 2005]. The prior texts distinguish several specialized notions of chase sequence, while we focus on the restricted chase here. We have defined the notion of a class of constraints "having terminating chase" to mean that for any initial instance *all* chase sequences terminate. But in all of the decidability results in this text it would suffice to have the weaker requirement that for any initial instance *some* chase sequence terminates. This is a strictly weaker requirement, as shown in [Onet, 2013].

The book by Toman and Weddell [2011] includes an overview of tableau proofs and the relationship of the chase to tableaux.

We refer to the complexity of computational problems throughout the text, in terms of standard complexity classes (for example NP, EXPTIME). The definitions of these classes can be found in a textbook such as [Papadimitriou, 1994]. We do not use any non-trivial theorems from complexity theory.